Real-World Algorithms

A BEGINNER'S GUIDE

PANOS LOURIDAS

Panos Louridas

Real-World Algorithms
A Beginner's Guide

The MIT Press
Cambridge, Massachusetts London, England

Copyrighted material

©2017 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

This book was set in LaTeX by the author using the Linux Libertine and Inconsolata fonts. The figures were drawn with Ti*k*Z, except for a few that were made with Python. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Names: Louridas, Panos, author.
Title: Real-world algorithms : a beginner's guide / Panos Louridas.
Description: Cambridge, MA : The MIT Press, [2017] | Includes bibliographical references and index.
Identifiers: LCCN 2016025660 | ISBN 9780262035705 (hardcover : alk. paper)
Subjects: LCSH: Computer algorithms—Popular works. | Computer programming—Popular works.
Classification: LCC QA76.9.A43 L67 2017 | DDC 005.1–dc23
LC record available at `https://lccn.loc.gov/2016025660`.

10 9 8 7 6 5 4 3 2 1

Copyrighted material

Contents

Preface ix

1 Stock Spans 1
- 1.1 Algorithms 3
- 1.2 Running Times and Complexity 7
- 1.3 Stock Span Using a Stack 15
- Notes 21
- Exercises 22

2 Exploring the Labyrinth 25
- 2.1 Graphs 27
- 2.2 Graph Representation 33
- 2.3 Depth-first Graph Traversal 40
- 2.4 Breadth-first Search 50
- Notes 55
- Exercises 56

3 Compressing 59
- 3.1 Compression 62
- 3.2 Trees and Priority Queues 65
- 3.3 Huffman Coding 69
- 3.4 Lempel-Ziv-Welch Compression 77
- Notes 90
- Exercises 90

4 Secrets 93
- 4.1 A Decryption Challenge 94
- 4.2 One-time Pad 99
- 4.3 The AES Cipher 104
- 4.4 Diffie-Hellman Key Exchange 112
- 4.5 Fast and Modular Exponentiation 117
- Notes 122
- Exercises 123

5 Split Secrets 125
- 5.1 Public Key Cryptography 126
- 5.2 The RSA Cryptosystem 129
- 5.3 Message Hashing 139

Copyrighted material

5.4 Internet Traffic Anonymization 141
Notes 147
Exercises 148

6 **Tasks in Order** 149
6.1 Topological Sort 150
6.2 Weighted Graphs 156
6.3 Critical Paths 158
Notes 168
Exercises 168

7 **Lines, Paragraphs, Paths** 169
7.1 Shortest Paths 172
7.2 Dijkstra's Algorithm 175
Notes 183
Exercises 184

8 **Routing, Arbitrage** 185
8.1 Internet Routing 189
8.2 The Bellman-Ford(-Moore) Algorithm 194
8.3 Negative Weights and Cycles 201
8.4 Arbitrage 205
Notes 210

9 **What's Most Important** 211
9.1 The PageRank Idea 212
9.2 The Hyperlink Matrix 214
9.3 The Power Method 216
9.4 The Google Matrix 220
Notes 225

10 **Voting Strengths** 227
10.1 Voting Systems 228
10.2 The Shulze Method 231
10.3 The Floyd-Warshall Algorithm 243
Notes 245

11 **Brute Forces, Secretaries, and Dichotomies** 247
11.1 Sequential Search 248

Copyrighted material

11.2 Matching, Comparing, Records, Keys 250
11.3 The Matthew Effect and Power Laws 252
11.4 Self-Organizing Search 259
11.5 The Secretary Problem 263
11.6 Binary Search 266
11.7 Representing Integers in Computers 271
11.8 Binary Search Revisited 276
11.9 Comparison Trees 278
Notes 283

12 A Menagerie of Sorts 285
12.1 Selection Sort 286
12.2 Insertion Sort 290
12.3 Heapsort 295
12.4 Merge Sort 302
12.5 Quicksort 316
12.6 Spoilt for Choice 324
Notes 327
Exercises 327

13 The Cloakroom, the Pigeon, and the Bucket 329
13.1 Mapping Keys to Values 330
13.2 Hashing 335
13.3 Hashing Functions 337
13.4 Floating Point Representation and Hashing 345
13.5 Collisions 348
13.6 Digital Fingerprints 357
13.7 Bloom Filters 362
Notes 374
Exercises 375

14 Bits and Trees 377
14.1 Divination as a Communications Problem 378
14.2 Information and Entropy 381
14.3 Classification 386
14.4 Decision Trees 388
14.5 Attribute Selection 391

Copyrighted material

14.6 The ID3 Algorithm 398
14.7 The Underlying Machinery 405
14.8 Occam's Razor 411
14.9 Cost, Problems, Improvements 413
Notes 417
Exercises 418

15 Stringing Along 421
15.1 Brute Force String Matching 424
15.2 The Knuth-Morris-Pratt Algorithm 427
15.3 The Boyer-Moore-Horspool Algorithm 439
Notes 447
Exercises 448

16 Leave to Chance 449
16.1 Random Numbers 451
16.2 Random Sampling 459
16.3 Power Games 465
16.4 Searching for Primes 477
Notes 486
Exercises 487

Bibliography 489

Index 501

Copyrighted material

Preface

> Like most of my generation, I was brought up on the saying: "Satan finds some mischief for idle hands to do". Being a highly virtuous child, I believed all that I was told, and acquired a conscience which has kept me working hard down to the present moment. But although my conscience has controlled my actions, my opinions have undergone a revolution. I think that there is far too much work done in the world, that immense harm is caused by the belief that work is virtuous, and that what needs to be preached in modern industrial countries is quite different from what always has been preached.
>
> Bertrand Russell, *In Praise of Idleness* (1932)

This book is about *algorithms*; and algorithms are what we do in order *not to have to do something*. It is the work we do to avoid work. By virtue of our inventions we have always been good at using brain for brawn. With algorithms we are using brain for brain.

Reducing human effort is a noble task. It is well ingrained in our minds that we should use machines to reduce our toil whenever possible and this has allowed us to reduce back-breaking work that was the norm for centuries. That is a wonderful thing, and there is no reason to stop at avoiding physical effort when we can also avoid mental labor. Drudgery, dull, repetitive work are the bane of human creativity, and we should do our best to avoid it; and algorithms allow us to do that.

Besides, digital technology today can accomplish feats that do not seem to be mind-numbing, but the essence of human nature. Machines recognize and produce speech, translate texts, categorize and summarize documents, predict the weather, find patterns in mounts of stuff with uncanny accuracy, run other machines, do mathematics, beat us at games, and help us to invent yet other machines. All these they do with algorithms, and by doing them they allow us to do less, they give us time to pursue our interests, and they even give us time and opportunity to discover yet better algorithms that will reduce our daily grind even more.

Algorithms did not start with computers; they have been with us from ancient times; nor are they limited to computer science. It is difficult to come up with a discipline that has not be transformed in some way by algorithms. In this way, many people encounter algorithms through the back door, as it were: they discover that they have become an important part of their discipline, no

Copyrighted material

matter how distant from computers it might appear to be. It behooves them then to learn about algorithms, to be able to reason with and use them.

Even with simple things and everyday tasks, it is amazing how much effort is squandered every day because some modicum of right thinking is not applied. Every time you find yourself doing something repetitive, chances are you should not be doing it. The author has encountered, time and again, that in the course of their daily office jobs, people perform sequences of operations that could be done in a blink of time, if only they knew how to apply themselves on how to avoid doing their work; not by shirking (some people are very adept at that), but getting a computer to do the job for them (more people should be more adept at that).

Intended Audience

This book was written to serve as a first encounter with algorithms. If you study computer science, you can benefit from it for an initial approach and then you can delve into more advanced texts; algorithms are the core of computing and an introduction such as this one can only skim the surface.

There are many others, though, that while pursuing other careers, or studying for them, become aware that algorithms have also become an essential part of their tool-chest. In many disciplines, it is pretty much impossible not to work with algorithms. This book intends to bring algorithms to this audience: those, and there are many, that need to use and understand algorithms as part, even though not as the center, of their job or studies.

And then there are all those who could use some algorithms, no matter how small or trivial, to simplify their work and avoid wasting time on chores. A task that would take hours for a diligent worker can be performed in virtually no time by using just a few lines of computer code in a modern scripting language. Sometimes this can come as an epiphany to the uninitiated, which is a pity because algorithmic thinking is not the prerogative of some illuminated elite.

In the same way that nobody could seriously argue today that a basic knowledge of mathematics and science is not essential to engage meaningfully with the modern world, it is no longer possible to be a productive member of contemporary society without a grasp of algorithms. They underlie your daily human experience.

Copyrighted material

years ago, and different languages are more appropriate for certain things than others. Language wars are silly and counterproductive. Also, a happy outcome of all the wonderful things that computers can do for us now is that people seek actively new ways to work with computers, and this effort leads to new programming languages being invented and older ones evolving.

The author does prefer some computer languages over others, but it is perhaps unfair to impose on the reader his own preferences. Moreover, computer languages go in and out of fashion, and yesterday's darling is frumpy today. Hoping to make the book as widely usable as possible and with an eye to longevity, there are no examples in an actual programming language in these pages. Algorithms are described using pseudocode. The pseudocode can be understood more easily than actual computer code, as it can glide over the foibles that real programming languages invariably have. It is often also easier to reason with pseudocode; when you try to develop a deep understanding of an algorithm, you must write down parts of it, and this is easier in pseudocode than in real code, where you need to attend carefully to syntax.

That said, it is difficult to work with an algorithm unless you do write computer code that implements it. The adoption of pseudocode in this book does not mean that the reader should also adopt a cavalier attitude towards computer code in general. Whenever possible, the algorithms presented should be implemented in a language of choice. Do not underestimate the sense of accomplishment you will get when you manage to create a computer program that implements an algorithm *correctly*.

How to Read this Book

The best way to read the book is sequentially, as earlier chapters provide instruction on concepts that are used later on. In the beginning, you will encounter basic data structures that all kinds of algorithms use, and that are indeed taken up in later chapters. However, once the foundation has been laid down, you may choose later chapters as you wish, if you find some more interesting than others.

You should therefore start with chapter 1 where you will see the way the rest of the chapters are structured: they begin with a description of a problem and then present algorithms that can solve it. Chapter 1 also introduces the pseudocode conventions used in the book and basic terminology and the first data structures you will encounter: arrays and stacks.

Copyrighted material

Chapter 2 gives a first glimpse of graphs and ways to explore them. It also covers recursion, so even if you have seen graphs before but you are not entirely sure about your grasp of recursion, you should not skip it. Chapter 2 presents additional data structures that we will see time and again in algorithms in other chapters. Then, in chapter 3, we turn to the problem of compression and how two different compressing schemes work: this allows us to introduce some further important data structures.

Chapters 4 and 5 treat cryptography. This is different from graphs and compression, but it is an important application of algorithms, especially in recent years where personal data can be found in all sort of places and devices, and all sort of entities are willing to peek into them. These two chapters can be read more or less independently from the rest, although some important pieces, such as how to pick large prime numbers, are left for chapter 16.

Chapters 6–10 describe problems related to graphs: ordering tasks, finding your way in a maze, how to decide the importance of things linked to other things (such as pages in the web), how graphs can be used in elections. Finding your way in a maze has more applications than you might initially think, from typesetting paragraphs, to Internet routing and financial arbitrage; a variant of it appears in the context of elections, so chapters 7, 8, and 10 can be treated as a unit.

Chapters 11 and 12 deal with two of the most fundamental problems in computing: searching and sorting. These two topics can fill entire volumes, and they have. We present some important algorithms that are in common use. When dealing with searching, we take the opportunity to deal with some additional material, such as online searching (searching for something among items that come streaming to you, without being able to revise your decision afterwards) and scale-free distributions, which researchers have found pretty much everywhere they have cared to look. Chapter 13 gives another way of storing and retrieving data, that of hashing, which is extremely useful, common, and elegant.

Chapter 14 covers a classification algorithm: the algorithm learns how to classify data, based on a set of examples, and then we can use it to classify new, unseen instances. This is an example of Machine Learning, a field whose importance has increased immensely as computers have become more and more powerful. The chapter also covers basic ideas on Information Theory, another beautiful field related to algorithms. Chapter 14 is different from the other chapters in the book because it also presents how an algorithm works by calling on smaller algorithms to do part of its work, in the same way that

Copyrighted material

computer programs are composed of small building parts, each one of which does a particular job. It also shows how data structures that have been introduced elsewhere into the book play an essential role in implementing the classification algorithm. The chapter should appeal particularly to readers who would like to see how the details of a high-level algorithm are worked out—an important step in the process of turning algorithms into programs.

Chapter 15 goes into sequences of symbols, called strings, and how we can find things inside them. It is an operation we call our computers to do every time we look for something inside a text, yet it is not obvious how to do it efficiently. Fortunately, there are ways to do it fast and gracefully. Moreover, sequences of symbols can represent many other kinds of things, so that string matching has applications in many areas, for example, in biology.

Finally, chapter 16 treats algorithms that work with chance. It is surprising how many applications of such randomized algorithms exist, so it is only possible to include a smattering of them here. Among other things, they provide answers to problems we have encountered previously in the book, such as how to find large prime numbers, required in cryptography. Or, again related to voting, how you count the impact of your vote.

Course Use

The material in the book can be used for a full semester course that covers algorithms and focuses on understanding the main ideas without going deep into a technical treatment of the subject. Students in diverse disciplines such as business and economics; life, social, and applied sciences; or formal sciences like mathematics and statistics, can use it as the main text in an introductory course, supplemented with programming assignments in which they are called to implement working instances of real, practical algorithms. Those studying computer science per se could use it as an informal introduction that would spur them to appreciate the full depth and beauty of algorithms as presented in a more technical book on the subject.

Copyrighted material

Acknowledgments

When I first broached the idea about this book to MIT Press, little did I know what it would take to realize it, and it would not exist without the support of the wonderful people there. Marie Lufkin Lee guided me through the whole process, ever gently, even when I was treading over deadlines. Virginia Crossman, Jim Mitchell, Kate Hensley, Nancy Wolfe Kotary, Susan Clark, Janice Miller, Marc Lowenthal, and Justin Kehoe all helped at various stages, as did the anonymous reviewers. Amy Hendrickson assisted when I was having fun with LaTeX arcana.

Marios Fragkoulis provided detailed feedback on parts of the manuscript and Diomidis Spinellis found the time to give me great suggestions on how to improve it. Stephanos Androutsellis-Theotokis, George Theodorou, Stephanos Chaliasos, Christina Chaniotaki, and George Pantelis were kind enough to point out errors. The embarassment of any remaining slips and oversights is entirely my own.

And of course my respect to Eleni, Adrian, and Hector, *who really bore the brunt of this book.*

Last First Words

If you write *algorhythm* instead of algorithm, you get a portmanteau word that means "the rhythm of pain," as *algos* is Greek for pain. In reality, the word algorithm comes from al-Khwārizmī, the name of a Persian mathematician, astronomer, and geographer (c. 780–c. 850 CE). Hoping that this book will engage you and not pain you, let's get on with algorithms.

Copyrighted material

1 Stock Spans

Imagine that you are given daily price quotes for a stock. That is, you have a series of numbers, each one representing the closing price of a given stock at a given day. The days are in chronological order. No quote is given for the days on which the stock market is closed.

The *span* of a stock's price on a given day is the number of consecutive days, from the given day going backwards, on which its price was less than or equal to its price on the day we are considering. The Stock Span Problem then is, given a series of daily price quotes for a stock, to find the span of the stock on each day of the series. So, for instance, consider figure 1.1. The first day is day zero. On day six of our data the span is five days, on day five it is four days, and on day four it is one day.

In reality the series can contain thousands of days, and we may want to compute the span for many different series, each one describing the evolution of a different stock price. We therefore want to use a computer to give us the solution.

In many problems that we use computers to solve, there is usually more than one way to arrive at a solution, more than one way to solve it. Usually some of them are better than others. Now, "better" by itself does not really mean anything; when we say better, we actually mean better in terms of something. This can be in terms of speed, memory, or something else that impacts on a resource such as time or space. We will have more to say on this in a bit, but it is important to keep that in mind from the outset because a solution to a problem may be simple but may not be optimal according to some constraint or criterion we have placed.

Suppose you are on day m of the series. One way you could find the span of the stock on day m is to go back one day, so you will be on day $m - 1$. If the price on day $m - 1$ is greater than the price on day m, then you know that the span of the stock on day m is just one. But if the price on day $m - 1$ is less

Copyrighted material

Algorithm 1.1 shows how we will be describing algorithms. Instead of using a computer language, which would force us to deal with implementation details that are not relevant to the logic of the algorithm, we will be using a form of *pseudocode*. Pseudocode is something between real programming code and an informal description. It employs a structured format and adopts a set of words that it endows with specific meaning. However, pseudocode is not real computer code. It is not meant to be executed by computers, but to be understood by humans. By the way, programs should also be understood by humans, but not all programs are—there are a lot of running, badly written, incomprehensible computer programs out there.

Each algorithm has a name, takes some input, and produces some output. We will write the name of the algorithm in `CamelCase` and its input in parentheses. Then we will indicate the output with a →. In the lines that follow, we will be describing the algorithm's inputs and outputs. The name of the algorithm followed by its input in parentheses can be used to *call* the algorithm. Once an algorithm has been written, we can treat it as a black box that we can use by feeding it with some input; the black box will return the algorithm's output. When implemented in a programming language, an algorithm is a named piece of computer code, a *function*. In a computer program, we call the function that implements the algorithm.

Some algorithms do not produce an output that they explicitly return. Instead, their actions impact some part of their context. For example, we may provide the algorithm with some space where to write its results. In this case, the algorithm does not return its output in the conventional sense, but there is an output anyway, the changes it effects on its context. Some programming languages make the distinction between pieces of named program code that explicitly return something, calling them *functions*, and pieces of named program code that do not return something but have nevertheless other side effects, calling them *procedures*. The distinction comes from mathematics, where a function is something that must return a value. For us an algorithm, when coded as an actual program, may turn out to be either a function or a procedure.

Our pseudocode will use a set of keywords in **bold** whose meaning should be self-explanatory if you have some acquaintance with how computers and programming languages work. We will be using the character ← for assignment, and the equals sign (=) for equality. The usual five symbols are adopted for the four mathematical operations (+, −, /, ×, ·); hence we have two signs for multiplication; we will be using both, basing our choice on aesthetics.

Copyrighted material

We will not be using any keywords or symbols to demarcate blocks of pseudocode; we will rely on indentation.

In this algorithm, we use *arrays*. An array is a structure holding data that allows us to manipulate its data in specific ways. A structure holding data that allows specific operations on the data it contains is called a *data structure*. An array is therefore a data structure.

Arrays are to computers what series of objects are to people. They are ordered sequences of elements. These elements are stored in the computer's memory. To obtain the space required for holding the elements and create an array that can hold n elements, we call an algorithm `CreateArray` in line 1 of algorithm 1.1. If you are familiar with arrays, then you may think it strange that the creation of an array requires an algorithm. And yet it does. To get a block of memory to hold data, you must at least search for the available memory inside the computer and mark it for use by the array. The `CreateArray(n)` call does all that is required. It returns an array with space for n elements; initially there are no elements in there, just the space that can hold them. It is the responsibility of the algorithm that calls `CreateArray(n)` to fill the array with the actual data.

For the array A, we denote and access its ith element by $A[i]$. The position of an element in the array, such as i in $A[i]$, is called its *index*. An array of n elements contains elements $A[0], A[1], \dots, A[n-1]$. This may strike you as strange because its first element is the zeroth, and its last element is the $(n-1)$th. You may have expected them to be first and nth instead. However, this is how arrays work in most computer languages, so you had better get used to it now. Because it is so common, when we iterate over an array of size n, we iterate from place 0 to place $n-1$. In our algorithms, when we say that something will take the values from a number x to a number y (assuming that x is less than y), we mean all the values from x up to but not including y; check out line 2 of the algorithm.

We assume that accessing the ith element takes the same time, no matter what i actually is. So accessing $A[0]$ requires the same time as $A[n-1]$. That is an important feature of arrays: elements are uniformly accessible at a constant time; the array does not have to search for an element when we want to access it by its index.

Concerning notation, when describing algorithms we will be using lowercase letters for the variables that appear in them; but when a variable refers to a data structure we may be using uppercase characters, such as the array A, to help them stand out; but this will not always be necessary. When we want

Copyrighted material

to give to a variable a name consisting of many words, we will be using an underscore (_) as `a_connector`; that is necessary because computers do not understand that a set of words separated by spaces constitute a single variable name.

Algorithm 1.1 uses arrays that store numbers. Arrays can hold any type of item, although each array can hold items of a single type in our pseudocode. This is also the case in most programming languages. For example you may have an array of decimal numbers, an array of fractions, an array of items that represent people, and another array of items that represent addresses. You may not have an array that contains both decimal numbers and items representing people. As to what the "items that represent people" may be, that is down to the specific programming language used in a program. All programming languages provides the means to represent meaningful stuff.

A particularly useful kind of array is an array that contains characters. An array of characters represents a *string*, which is a sequence of letters, numbers, words, sentences, or whatever. As in all arrays, the individual characters that the array contains can be referenced individually by the index. If we have the string s = "Hello, World", then $s[0]$ is the letter "H" and $s[11]$ is the letter "d".

Summing up, an array is a data structure that holds a sequence of items of the same type. There are two operations on arrays:

- `CreateArray(`n`)` creates an array that can hold n elements. The array is not initialized, that is, it does not hold any actual elements, but the required space for them has been reserved and can be used to store them.
- As we have seen, for an array A and its ith element, $A[i]$ accesses the element, and accessing any element in the array takes the same time. It is an error to try to access $A[i]$ when $i < 0$.

Back to algorithm 1.1. Following the above, the algorithm contains a loop, a block of code that executes repeatedly, in lines 2–10. The loop is executed n times, once for the calculation of a span, if we have prices for n days. The current day whose span we are considering is given by variable i. Initially, we are at day zero, the earliest point in time; each time we go through line 2 of the loop, we will be moving to day $1, 2, \ldots, n-1$.

We use a *variable* k to indicate the length of the current span; a variable is a name that refers to some piece of data in our pseudocode. The contents of those data, to be precise, the *value* of the variable may change during the execution of the algorithm; hence its name. The value of k when we start to calculate a span is always 1, which we set in line 3. We also use an *indicator*

Copyrighted material

variable, *span_end*. Indicator variables take the values TRUE and FALSE and indicate that something holds or does not hold. The variable *span_end* will be true when we reach the end of a span.

At the start of each span's calculation *span_end* will be false, as in line 4. The length of the span is calculated in the inner loop of lines 5–9. Line 5 tells us to go backwards in time as far as we can, and as long as the span has not ended. As far as we can is determined by the condition $i - k \geq 0$: $i - k$ is the index of the day to which we go back to check if the span ends, and the index cannot be zero, as this corresponds to the first day. The check for the end of a span is at line 6. If the span does not end, then we increase it in line 7. Otherwise we note the fact that the span ends in line 9 so that the loop will stop when execution goes back to line 5. At the end of each iteration of the outer loop of lines 2–10, we store the value of k in the appropriate place in the array *span* in line 10. We return *spans*, which contains the results of the algorithm, in line 11 after exiting the loop.

Note that when we start we have $i = 0$ and $k = 1$. That means that the condition in line 5 will certainly fail for the earliest point in time. That is as it should, as its span can only be equal to 1.

At this point, remember what we just said about algorithms, pen, and paper. The proper way to understand an algorithm is to execute it yourself, manually. If at any time an algorithm seems complicated, or you are not sure you have grasped it entirely, then write down what it does on some example. It will save you a lot of time, old-fashioned though it may seem. If you are unsure about algorithm 1.1 go and do it now, then return here when the algorithm is clear.

1.2 Running Times and Complexity

Algorithm 1.1 is a solution to the Stock Span Problem, but we can do better. Here better means that we can do faster. When we talk about speed in algorithms, we are really talking about the number of steps the algorithm will execute. No matter how fast computers get, although they will execute computational steps faster and faster, the number of steps will remain the same, so evaluating the performance of an algorithm in terms of the steps it requires makes sense. We call the number of steps the *running time* of the algorithm, although this is a pure number, not measured in any time units. Using time units would make any running time estimate relative to a specific computer model, which is not useful.

Copyrighted material

Consider how long it takes to calculate the spans of n stock quotes. The algorithm consists of a loop, starting at line 2, that will execute n times, one for each quote. Then there is an inner loop, starting in line 5, that for each iteration of the outer loop will try to find the quote's span. For each quote it will compare the price of the quote with all previous quotes. In the worst case, if the quote is the highest price yet, then it will examine all previous quotes. If quote k is the highest of all previous quotes, then the inner loop will execute k times. Therefore, in the worst case, which is if the quotes are in ascending order, line 7 will execute the following number of times:

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

If the equation is not clear, then you can easily see that this is indeed so if you add the numbers $1, 2, \ldots, n$ twice:

$$\begin{array}{ccccccc} & 1 & + & 2 & + \cdots + & n & \\ + & n & + & n-1 & + \cdots + & 1 & \\ \hline & n+1 & + & n+1 & + \cdots + & n+1 & = n(n+1) \end{array}$$

Because line 6 is the step of the algorithm that will execute most times, $n(n + 1)/2$ is the worst case running time of the algorithm.

When we talk about algorithm running times, we are really interested in the running time when our input data is large (in our case, the number n). That is the *asymptotic* running time of an algorithm because it deals with the behavior of the algorithm when the input data increase without bounds. There is some special notation that we use for this purpose. For any function $f(n)$, if for all values of n greater than some initial positive value the function $f(n)$ is less than or equal to another function $g(n)$ scaled by some positive constant value c, that is, $cg(n)$, we say that $O(f(n)) = g(n)$. In a more precise way, we say that $O(f(n)) = g(n)$ if there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

The notation $O(f(n))$ is called "big-Oh notation." Keep in mind that we are interested in *big values* of our input, because that is where we will have the biggest savings. Take a look at figure 1.2, where we plot two functions, $f_1(n) = 20n + 1000$ and $f_2(n) = n^2$. For small values of n it is $f_1(n)$ that takes biggest values, but the situation changes drastically quite early, after which n^2 grows much faster.

The big-Oh notation allows us to simplify functions. If we have a function like $f(n) = 3n^3 + 5n^2 + 2n + 1000$, then we have simply $O(f(n)) = n^3$. Why?

Copyrighted material

under the same complexity family, which we usually denote by $O(\log(n))$, although the more specific $O(\lg(n))$ is also used a lot. Algorithms with $O(\lg(n))$ complexity arise when the algorithm repeatedly divides a problem in two because if you divide something repeatedly by two, you are essentially applying the logarithmic function to it. Important logarithmic time algorithms are algorithms related to searching: the fastest searching algorithms run on base two logarithmic time.

More time consuming than logarithmic time algorithms are *linear time algorithms* that run in time $f(n) = n$, that is, in time proportional to their input. For these algorithms, the complexity is $O(n)$. These algorithms may have to scan their whole input in order to find an answer. For example, if we search a random set of items that are not ordered in any way, then we may have to go through all of them to find the one we want. Therefore, such a search runs in linear time.

Slower than linear time are the *loglinear time algorithms,* where $f(n) = n\log(n)$, and we therefore write $O(n\log(n))$. As before, the logarithm can be to any base, although in practice algorithms to base two are common. These algorithms in some way are a combination of a linear time algorithm and a logarithmic time algorithm. That may involve repeatedly dividing a problem and applying a linear time algorithm to each of the divided parts. Good sorting algorithms have a loglinear time complexity.

When the function describing the running time of the algorithm is a polynomial $f(n) = (a_1 n^k + a_2 n^{k-1} + \cdots + a_n n + b)$ we have, as we saw, a complexity of $O(n^k)$ and the algorithm is a *polynomial time algorithm.* Many algorithms run in polynomial time; an important sub-family are algorithms that run in $O(n^2)$ time, which we call *quadratic time algorithms.* Some not efficient sorting methods run in quadratic time, as does the standard way to multiply two numbers with n digits each—note that there are actually more efficient ways to multiply numbers, and we use these more efficient ways in applications where we want high performance arithmetic calculations.

Slower than polynomial time algorithms are *exponential time algorithms,* where $f(n) = c^n$, with c a constant value, so $O(c^n)$. Be sure to notice the difference between n^c and c^n. Although we swapped the place of n and the exponent, it makes for a huge difference in the resulting function. As we said, exponentiation is the reverse of the logarithmic function and is simply raising a constant to a variable number. Careful: exponentiation is c^n; the *exponential function* is the special case where $c = e$, that is, $f(n) = e^x$, where e is the Euler number we met before. Exponentiation occurs when we have to handle

Copyrighted material

Table 1.1
Growth of functions.

Function	Input size				
	1	10	100	1000	1,000,000
$\lg(n)$	0	3.32	6.64	9.97	19.93
n	1	10	100	1000	1,000,000
$n \ln(n)$	0	33.22	664.39	9965.78	1.9×10^7
n^2	1	100	10,000	1,000,000	10^{12}
n^3	1	1000	1,000,000	10^9	10^{18}
2^n	2	1024	1.3×10^{30}	10^{301}	$10^{10^{5.5}}$
$n!$	1	3,628,800	9.33×10^{157}	4×10^{2567}	$10^{10^{6.7}}$

a problem of input n, where each of the n inputs can take a number of c different values and we must try all possible cases. We have c values for the first input, and for each of these we have c values for the second input; in total $c \times c = c^2$. For each of these c^2 cases, we have c possible values for the third input, which makes it $c^2 \times c = c^3$; and so on until the last input that gives c^n.

Still slower than exponential time algorithms are *factorial time algorithms* with $O(n!)$, where the factorial number is defined as $n! = 1 \times 2 \times \cdots \times n$ and the degenerate case $0! = 1$. The factorial comes into play when in order to solve a problem we need to try all possible *permutations* of input. A permutation is a different arrangement of a sequence of values. For example, if we have the values $[1, 2, 3]$, then we have the following permutations: $[1, 2, 3]$, $[1, 3, 2]$, $[2, 1, 3]$, $[2, 3, 1]$, $[3, 1, 2]$, and $[3, 2, 1]$. There are n possible values in the first position; then because we have used one value, there are $n - 1$ possible values in the second position; this makes $n \times (n - 1)$ different permutations for the first two positions. We go on like this for the remaining positions until the last position where there is only one possible value. In all, we have $n \times (n - 1) \cdots \times 1 = n!$. In this way the factorial number arises in shuffles: the number of possible shuffles of a deck of cards is 52!; that's an astronomical number.

A rule of thumb is that algorithms with up to polynomial time complexity are good, so our challenge is often to find algorithms with such performance. Unfortunately, for a whole class of important problems, we know of no polynomial time algorithms! Take a look at table 1.1; you should realize that if for a problem we have only an algorithm with a running time of $O(2^n)$, then the algorithm is pretty much worthless for anything apart from toy problems,

Copyrighted material

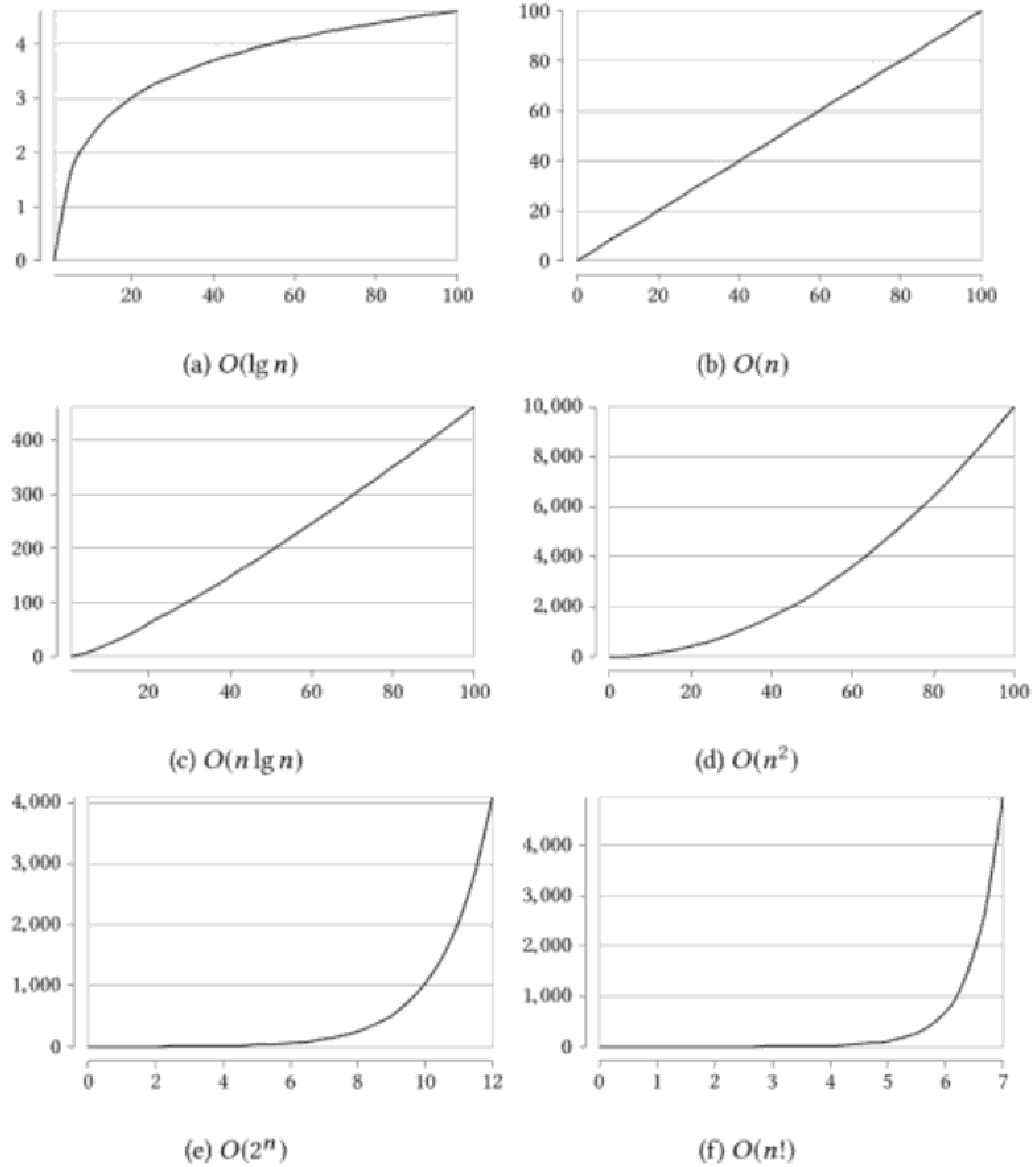
with very small values of input. You can also check that with figure 1.3; in the bottom row, $O(2^n)$ and $O(n!)$ start skyrocketing for small values of n.

In figure 1.3, we show the plots of functions as lines, although in reality the number n when we study algorithms is a natural number, so we would expect to see scatter plots, showing points instead of lines. Logarithmic, linear, loglinear, and polynomial functions are of course directly defined for real numbers, so there is no problem in plotting them with lines by using the normal functional definitions. The usual interpretation of exponentiation is for integers, but powers with rational exponents are also possible because $x^{(a/b)} = (x^a)^{(1/b)} = \sqrt[b]{x^a}$. Then powers with exponents that are real numbers are defined as $b^x = (e^{\ln b})^x = e^{x \ln b}$. Concerning factorials, with some more advanced mathematics, it turns out that they can also be defined for all real numbers (negative factorials are taken to be infinite). So we are justified in drawing the complexity functions with lines.

Lest you think that complexity $O(2^n)$ or $O(n!)$ rarely occurs in practice, consider the famous (or infamous) Traveling Salesman Problem. In this problem, a traveling salesman must travel to a number of cities, visiting each one of them only once. Every city is directly connected to every other city (perhaps the salesman travels by plane). The twist is that the salesman must do that while traveling as few kilometers as possible. A direct solution is to try all possible permutations of the cities. For n cities, this is $O(n!)$. There is a better algorithm that solves the problem in $O(n^2 2^n)$—a bit better, but not much of a practical difference. Then how do we solve this (and other similar problems)? It turns out that, although we may not know a good algorithm that will give us a precise answer, we may know good algorithms that will give us approximate results.

The big-Oh provides an *upper bound* on the performance of an algorithm. The converse is a *lower bound*, when we know that its complexity will be always no better than a certain function, after some initial values. This is called "big-Omega," or $\Omega(n)$, and the precise definition is that $\Omega(f(n)) = g(n)$ if there exist positive constants c and n_0 such that $f(n) \geq cg(n) \geq 0$ for all $n \geq n_0$. Having defined big-Oh and big-Omega, we can also define the situation when we have both an upper and a lower bound. This is "big-Theta," and we say that $\Theta(f(n)) = g(n)$ if and only if $O(f(n)) = g(n)$ and $\Omega(f(n)) = g(n)$. Then we know that the algorithm has a running time that is bounded both from below and from above by the same function, scaled by a constant. You can think of it as the algorithm running time lying in a band around that function.

Copyrighted material



(a) $O(\lg n)$ (b) $O(n)$

(c) $O(n \lg n)$ (d) $O(n^2)$

(e) $O(2^n)$ (f) $O(n!)$

Figure 1.3
Different complexity families.

Copyrighted material

1.3 Stock Span Using a Stack

Let's return to the Stock Span Problem now. We have found an algorithm with a complexity of $O(n(n + 1/2))$. According to what we have been saying, this is equivalent to $O(n^2)$. Can we do better? Go back to figure 1.1. Notice that when we are at day six, we do not need to compare with all the previous days until day one. Because we have gone through all days up to day six, we already "know" that days two, three, four, and five all have quotes less than or equal to that of day six. If we somehow keep that knowledge, then instead of doing all these comparisons, we only need to compare with the quote on day one.

This is a general pattern. Imagine you are on day k. If the stock price quote on day $k - 1$ is less than or equal to the stock price on day k, so that we have $quotes[k-1] \leq quotes[k]$ or equivalently $quotes[k] \geq quotes[k-1]$, then there is no reason to even compare with $k - 1$ again. Why? Take a future day $k + j$. If the quote on $k + j$ is less than the quote on day k, $quotes[k + j] < quotes[k]$, then we do not have to compare with $k - 1$ because the span starting from $k + j$ ends at k. If the quote on $k + j$ is greater than the quote on k, then we know already that it must be $quotes[k + j] \geq quotes[k-1]$ because $quotes[k + j] \geq quotes[k]$ and $quotes[k] \geq quotes[k-1]$. So each time that we are searching backwards for the end of the span, we may throw away all days with values less than or equal to the day whose span we are examining and we may exclude the thrown away days from consideration in any future span.

The following metaphor may help: Imagine you are sitting on top of the column for day six in figure 1.4. You look straight back, not below. You see the column for day one only. That is the only column with which you need to compare the stock value of day six. In general, at each day you only need to compare what is directly in your line-of-sight.

That means that we waste our time in algorithm 1.1 when in the inner loop starting in line 5 we start comparing with each and every previous day. We can save the waste by using some mechanism by which we have at hand the limits of the highest established spans.

We can do that by using a special data structure for holding data called a *stack*. A stack is a simple data structure. It is something on which we can put data, one after the other, and retrieve them. Each time we can retrieve the last one we have put. The stack works like a stack of trays in a restaurant, piled on top of each other. We can only take the top tray, and we can only add trays on top of the pile. Because the last tray to be added to the stack is the first one to be removed, we call a stack a Last In First Out (LIFO) structure. You

Copyrighted material

Algorithm 1.2: Stack Stock Span algorithm.

```
StackStockSpan(quotes) → spans
    Input: quotes, an array with n stock price quotes
    Output: spans, an array with n stock price spans

1  spans ← CreateArray(n)
2  spans[0] ← 1
3  S ← CreateStack()
4  Push(S, 0)
5  for i ← 1 to n do
6      while not IsStackEmpty(S) and quotes[Top(S)] ≤ quotes[i] do
7          Pop(S)
8      if IsStackEmpty(S) then
9          spans[i] ← i + 1
10     else
11         spans[i] ← i − Top(S)
12     Push(S, i)
13 return spans
```

Table 1.2
Boolean short circuit evaluation.

operator	a	b	result
	T	T	T
and	T	F	F
	F	T/F	F
	T	T/F	T
or	F	T	T
	F	F	F

There is a detail in line 6 of the algorithm that merits our attention. It is an error to evaluate Top(S) if S is empty. This will not happen, thanks to an important property about how conditions are evaluated, called *short circuit evaluation*. The property means that when we evaluate an expression involving logical boolean operators, the evaluation of the expression stops as soon as we know its final result, without bothering to evaluate any remaining parts of the expression. Take for example the expression **if** $x > 0$ **and** $y > 0$. If we know that $x \leq 0$, then the whole expression is false, regardless of the value of

Copyrighted material

y; we do not need to evaluate the second part of the expression at all. Similarly, in the expression **if** $x > 0$ **or** $y > 0$, if we know that $x > 0$, then we do not need to evaluate the second part of expression, the one involving y, because we already know that the whole expression is true having established that the first part is true. Table 1.2 shows the general situation for any two-part boolean expression with an **and** or an **or** operator. The shaded rows indicate that the result of the expression does not depend on the second part and therefore the evaluation can stop as soon as we know the value of the first part. With short circuit evaluation, when `IsStackEmpty`(S) returns TRUE, which means that **not** `IsStackEmpty`(S) is FALSE, we will not try to evaluate the right hand of **and** containing `Top`(S), thereby avoiding an error.

You can see how the algorithm works and the line-of-sight metaphor in figure 1.6. At each panel of the figure we show, on the right, the stack at the start of each loop iteration; we also indicate the days in the stack with filled bars, whereas the days we have not handled yet are in dashed bars. The current day we are handling is in the black circle below the panel.

In the first panel we have $i = 1$, and we have to check the value of the current day with the values of the other days in the stack, which is just day zero. Day one has a higher price than day zero. That means that from now on there is no need to compare with the days before day one; our line-of-sight will stop there; so in the next iteration, with $i = 2$, the stack contains the number 1. Day two has a lower price than day one. That means that any span starting from day three may end on day two, if the value on day three is lower than the value on day two, or it may end on day one, if the value on day three is no less than the value on day two. There is no way it can end on day zero, though, as the price on day zero is less than on day one. A similar situation occurs with $i = 3$ and $i = 4$. But when we arrive at $i = 5$, we realize that we no longer need to compare with days two, three, and four in the future. These days lie in the shadow, as it where, of day five. Or, with the line-of-sight metaphor, our view is unobstructed way back until day one. Everything in-between can be popped from the stack and the stack will contain 5 and 1, so that at $i = 6$ we only need to compare at most with these two days. If a day has a value greater than or equal to that of day five, it will certainly surpass the values of days four, three, and two; what we cannot be certain about is whether it reaches the value of day one. When we are done with day six, the stack will contain the numbers 6 and 1.

Is this better than before? In algorithm 1.2 the loop starting in line 5 is executed $n - 1$ times. For each one of these times, say at the ith iteration,

Copyrighted material

Copyrighted image

Lines-of-sight of stock spans.

the Pop operation in the inner loop that starts in line 6 is executed p_i times. That means that in total the Pop operation will be executed $p_1 + p_2 + \cdots + p_{n-1}$ times, p_i times for each iteration of the outer loop. We do not know what the number p_i is. But if you pay close attention to the algorithm, you will see that each day is pushed on the stack only once, the first day at line 3 and the subsequent days at line 11. Therefore each day can be popped from the stack in line 7 at most once. So, throughout the whole execution of the algorithm, in all iterations of the outer loop, we cannot execute line 6 more than n times. In

Copyrighted material

other words, $p_1 + p_2 + \cdots + p_{n-1} = n$, which means that the whole algorithm is $O(n)$; line 7 is the operation that will be executed most times because it is in an inner loop, whereas the rest of the code in lines 5–12 is not.

We can also proceed in our analysis and see that, in contrast to algorithm 1.1, where we could only arrive at a worst-case estimate, here our estimate is also a lower bound on the algorithm's performance—there is no way the algorithm can complete with less than n steps because we need to go through n days. So the computational complexity of the algorithm is also $\Omega(n)$, and so it follows that it is $\Theta(n)$.

Stacks, like all the other data structures we will encounter, have many uses. A LIFO behavior is common in computers, so you will find stacks from low-level programs written in machine language, to the biggest problems running in supercomputers. That is why data structures in general exist in the first place. They are nothing but the essence of years of experience with problem solving with computers. It turns out, time and again, that algorithms use similar ways to organize the data they process. People have codified these ways so that, when looking our way around a problem, we reach to them, availing ourselves of their functionality to develop algorithms.

Notes

The definitive text on algorithms is the multi-volume work by Donald Knuth [112, 113, 114, 115]. The work is 50 years in the making and some volumes are yet to be written, which the existing books do not cover all areas of algorithms, but what they cover they treat with rigor and unsurpassed style. These books are not for the faint-hearted, yet they reward their reader many times over.

A thorough, classic introduction to algorithms is the book by Cormen, Leiserson, Rivest, and Stein [42]. Thomas Cormen has also written another popular book [41] that gives a shorter and gentler introduction to important algorithms. A lay person's introduction to algorithms is MacCormick's book [130]. Other popular introductions to algorithms include the books by Kleinberg and Tardos [107], Dasgupta, Papadimitriou, and Vazirani [47], Harel and Feldman [86], and Levitin [129].

There are also many fine books that deal with algorithms and their implementation in a particular programming language [180, 176, 178, 177, 179, 188, 82].

Copyrighted material

Stacks are about as old as computers. According to Knuth [112, pp. 229 and 459], Alan M. Turing proposed it in a design for an Automatic Computing Engine (ACE) written in 1945 and presented in 1946; the stack operations were called BURY and UNBURY instead of "push" and "pop" [205, pp. 11–12 and 30].

Algorithms are much older than computers, hailing at least since ancient Babylonian times [110].

Exercises

1. A stack is a simple data structure to implement. A straightforward implementation is using an array; go ahead and write an array-based stack implementation. In the text we mentioned that in practice a stack has more operations than the five we mentioned: operations returning its size and checking whether it is full. Make sure you also implement them.

2. We showed two solutions for the Stock Span Problem, one using a stack and one without a stack. We argued that the solution with the stack is faster. Check that it is indeed so by implementing the two algorithms in your programming language of choice and timing how long it takes for each approach to solve the problem. Note that to time a program's execution, you must feed it with enough data so that it does take a reasonable amount of time to finish; then, because lots of things happen to a computer at once and each run may be affected by different factors, you need to to run it repeatedly to get a stable measurement. So this is a good opportunity to look around and read on how programs are benchmarked.

3. Stacks are used for implementing arithmetic calculations written in *Reverse Polish Notation* (RPN), also known as *postfix notation.* In RPN every operator follows all its operands, in contrast to the usual infix notation where the operator is placed between its operands. So, instead of writing $1 + 2$, we write $1\ 2+$. "Polish" refers to the nationality of Jan Łukasiewicz who invented the *Polish* or *prefix notation* in 1924 and in which we write $+1\ 2$. The advantage of RPN is that there is no need for parentheses: $1 + (2 \times 3)$ in infix notation becomes $1\ 2\ 3 * +$ in postfix notation. To evaluate it, we read it from left to right. We push numbers on a stack. When we encounter an operator, we pop from the stack as many items as it needs as operands, we perform the operation, and we push the result in the stack. At the end we get the result at the top (and only element of the stack). For example, when evaluating $1\ 2\ 3\ *\ +\ 2\ -$ the stack, written horizontally in brackets, becomes [ ], [1], [1 2], [1 2 3], [1 6], [7] [7 2], [5]. Write a calculator that evaluates arithmetic expressions given by the user in RPN.

Copyrighted material

2 Exploring the Labyrinth

Finding your way in a labyrinth is an ancient problem. The story goes that king Minos of Crete had forced Athens to send to him seven youths and seven maidens every seven years. They would be thrown into the dungeons of Minos's palace, where the Minotaur lived, a monster with the body of a man and the head of a bull. The dungeons formed a maze, and the hapless sacrificial offerings would be devoured by the Minotaur. The third time that this tribute was due, Theseus volunteered to be among the youths to be sacrificed. When he got to Crete, he charmed the daughter of Minos, Ariadne, who gave him a ball of thread. He unwound the thread as he went along in the maze, found the Minotaur, slaughtered him, and then used the thread to find his way to the exit, instead of getting lost and perishing.

Maze exploration is not of interest just because it appears in an ancient myth or because it gives us amusement in beautifully landscaped parks. A maze is not different from any situation where we have to explore a set of spaces connected with specific paths. A road network is an obvious example; but the problem becomes more interesting if we realize that there are cases when we want to explore more abstract things. We may have a network of computers, connected to each other, and want to find out whether one computer is connected to some other computer. We may have a network of acquaintances, that is, people somehow connected to each other, and want to find out whether we can get from one person to another.

The myth suggests that to find our way in a maze, we must somehow know where we have already been. Otherwise a maze exploration strategy will fail. Let's take an example maze and think of a strategy. Figure 2.1 shows a maze, where we depict rooms as circles and the corridors between them as lines connecting the circles.

In figure 2.2 you can see what happens when we explore the maze systematically, following a specific strategy called "hand on the wall." We indicate

Copyrighted material

Copyrighted image

Figure 2.1
The maze.

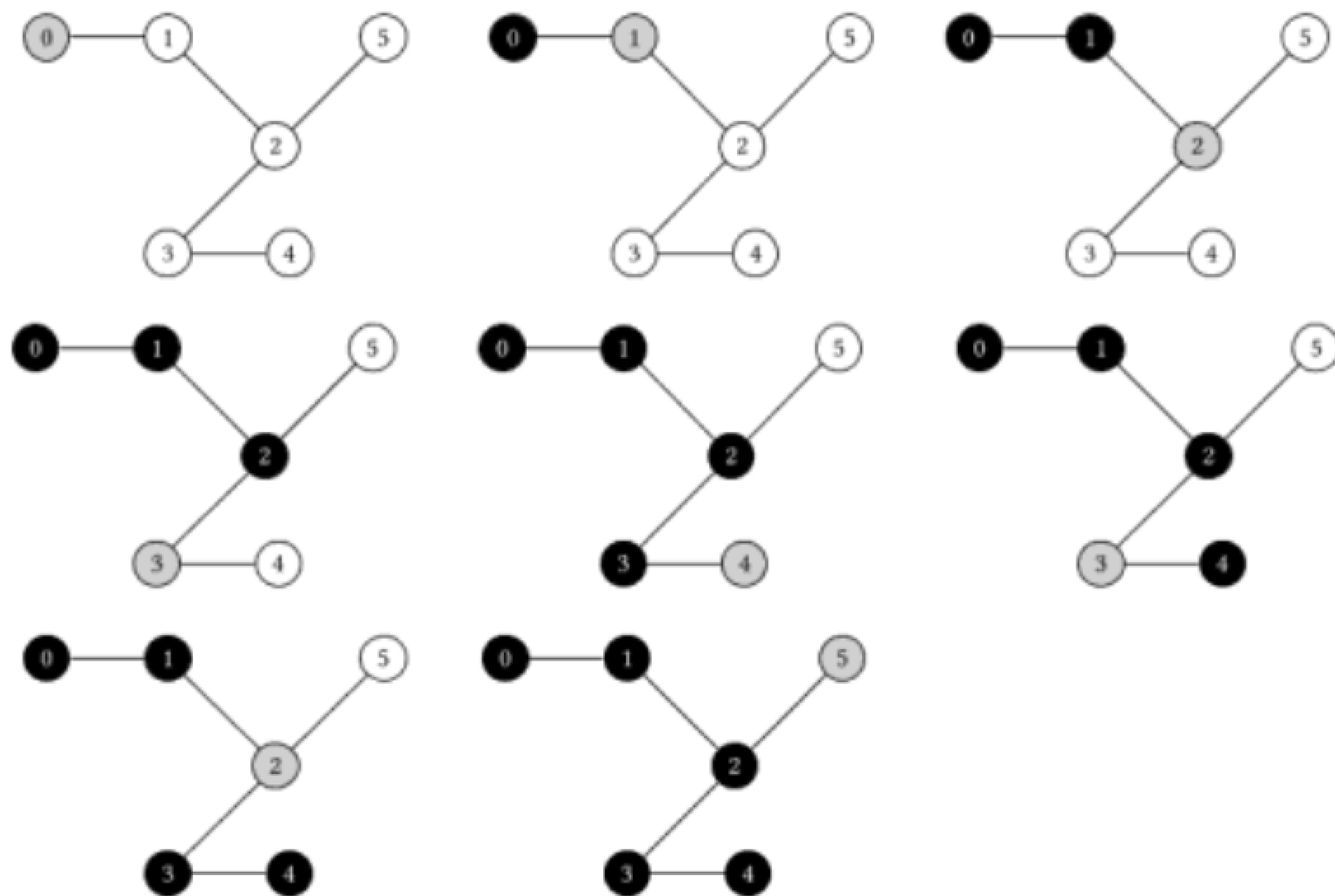


Figure 2.2
Keep the hand on the wall strategy: it works!

the current room as gray and the visited rooms as black. It is a simple strategy. You place your hand on a wall and you never lift your hand from the wall. As you proceed from one room to another, you take care to keep the hand touching the wall as you go. Apparently, the strategy works. But then see the maze in figure 2.3. By following the strategy, you will visit the rooms on the periphery of the maze, and you will miss the room on the interior, as you can verify in figure 2.4.

Copyrighted material

Copyrighted image

Figure 2.3
Another maze.

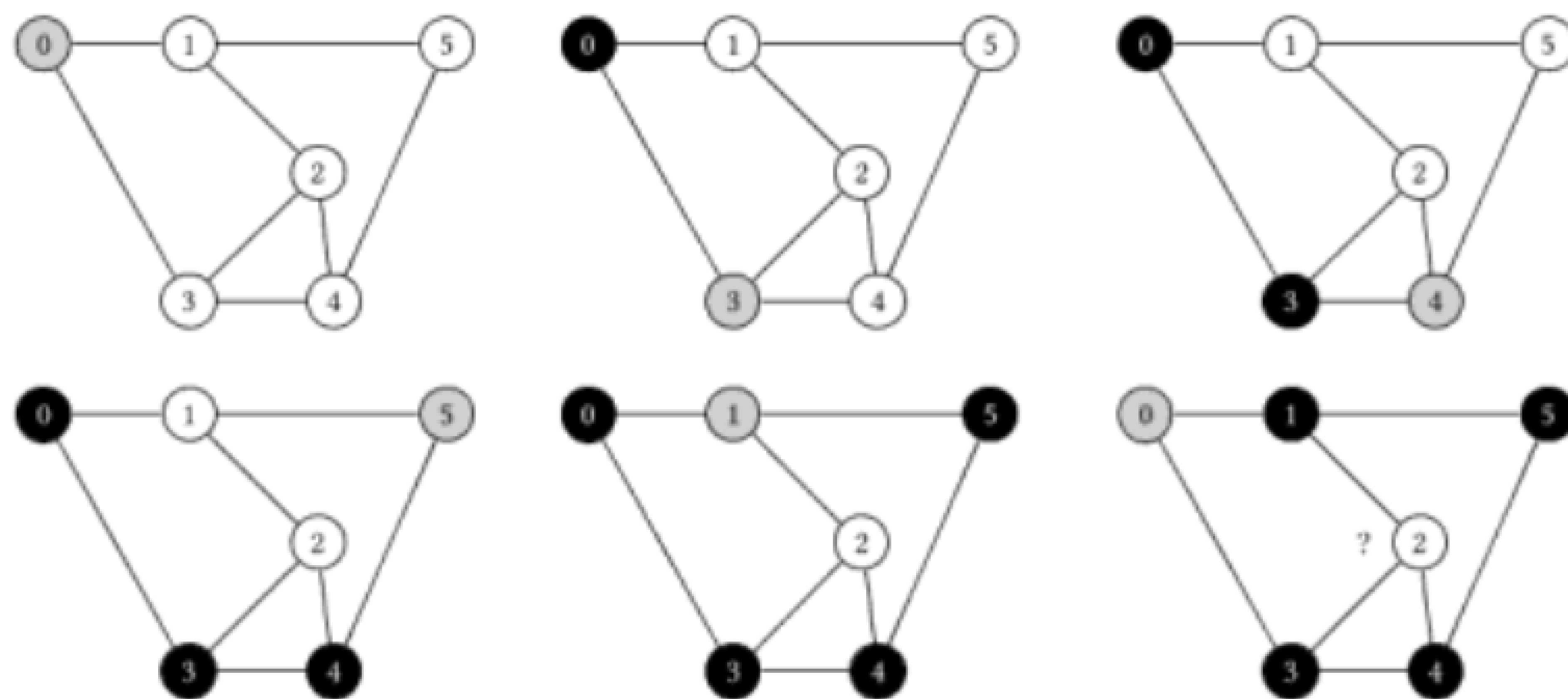


Figure 2.4
Keep the hand on the wall strategy: it fails…

2.1 Graphs

Before we proceed on how we can solve the problem, we have to deal with how we are going to represent mazes in general. The way we have described them, mazes consist of rooms and corridors among them. We hinted that they become more interesting when we realize that they are similar to other structures; in fact they are similar to anything consisting of objects and connections among these objects. This is a fundamental data structure, perhaps the most fundamental of all, because many things in the real world can be represented as objects and connections among objects. Such structures are called *graphs*. A succinct definition is that a graph is a set of *nodes* and *links* among them. Alternatively, we may speak of *vertices*, in singular *vertex*, and *edges*. An edge connects exactly two vertices. A series of edges, in which every two

Copyrighted material

Copyrighted image

Figure 2.5
Directed and undirected graphs.

edges share a node in common, is called a *path*. So, in figure 2.2 there is a path connecting nodes 0 and 2 passing through node 1. The number of edges in a path is called its *length*. An edge is a path with length 1. If a path exists between two nodes, then we say that the two nodes are *connected*. In certain graphs we may want the edges to be directed; these graphs are *directed graphs*, or *digraphs* for short. Otherwise, we deal with *undirected graphs*. Figure 2.5 shows an undirected graph on the left and a directed graph on the right. As you can see, there may be a number of different edges starting or ending on a single node. The number of edges adjacent to a node is called its *degree*. In directed graphs we have the *in-degree*, for incoming edges, and the *out-degree*, for outgoing edges. In figure 2.5a it so happens that all edges have degree 3. In figure 2.5b the rightmost node has in-degree 2 and out-degree 1.

The applications of graphs can fill whole volumes: it is amazing how many things can be represented as graphs, how many problems can be rendered in graph terms, and how many algorithms exist for solving graph-related problems. This is because many things consist of objects and connections among them, as we just observed; this deserves some further attention.

Perhaps the most obvious graph application, at which we already hinted, is in representing *networks*. Points in a network are nodes in the graph, and links are edges between them. There are many different kinds of networks; we have *computer networks*, of course, with computers connected to each other, but there are *transport networks* as well, with cities linked by roads, airplane routes, or railway lines. In computer networks, the *Internet* is the biggest example of all, and the *web* is also a network with pages as nodes connected

Copyrighted material

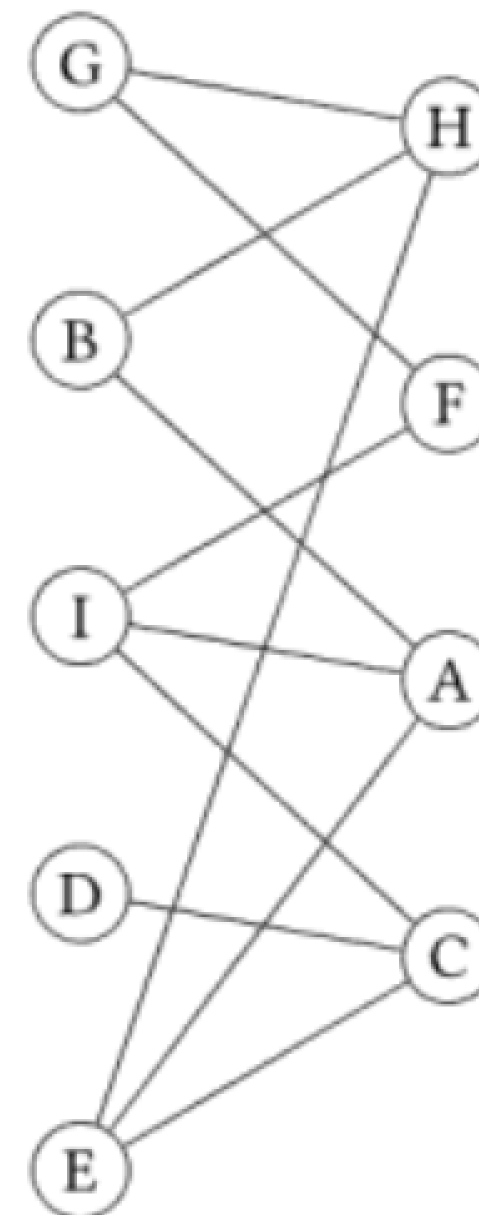
by hyperlinks between them. The *Wikipedia* is an especially large network, a subset of the web network. In the domain of electronics, *circuit boards* consist of electrical components, such as transistors, connected via circuits. In biology we encounter *metabolic networks* that contain, among other things, metabolic pathways: chemicals are connected through chemical reactions. *Social networks* are modeled as graphs, with people as nodes and the relationships among them as edges. *Scheduling* of jobs and tasks among people or machines can also be modeled via graphs, with the tasks as nodes and the dependencies among them, such as which task should precede which other tasks, represented by the edges.

For all the above applications, and more, there exist different kinds of graphs that are suitable for representing varied situations. If there is a path from any node to any other node in a graph, the graph is called *connected*. Otherwise it is called *disconnected*. Figure 2.6 shows a connected and a disconnected graph, both undirected. Note that in a directed graph we have to take into account the direction of the edges to determine whether it is connected. A directed graph in which there is a *directed path* between any two nodes is called *strongly connected*. If we somehow forget about directions and we are interested in whether it is an *undirected path* between any two nodes, then the graph is called *weakly connected*. If a directed graph is not strongly connected nor weakly connected, then it is simply disconnected. Figure 2.7 shows the possibilities. The question of graph connectivity arises whenever we want to determine whether something, which is modeled as a graph, represents a whole entity, or is composed from separate sub-entities. Connected sub-entities in undirected graphs and strongly connected sub-entities in directed graphs are called *connected components*. Therefore, a graph is connected, or strongly connected if it is a directed graph, when it consists of a single connected component. A related question is that of *reachability*, which is whether it is possible to reach some node from some other node.

In a directed graph, or digraph, it may be possible to start from a node, jump from edge to edge, and come back to the node we started from. When this happens, we have traveled in a circle, and the path we have made is a *cycle*. In an undirected graph, we may always return to where we started from by going backward, so we say that we travel in a circle if we can get back to the node we started without going backward on an edge. Graphs with cycles are called *cyclic*; graphs without cycles are called *acyclic*. Figure 2.8 shows two directed graphs with several cycles. Note that the graph on the right has some edges that start and end on the same node. These are cycles of length one, and

Copyrighted material

Copyrighted image



(b) The same graph.

Figure 2.10
A bipartite graph.

edges. For n nodes, because every node is connected to all other $n-1$ nodes, we have $n(n-1)/2$ edges. In general we may say that a graph with n nodes is dense if it has close to n^2 number of edges, and sparse if it has about n edges. This leaves a fuzzy area between n and n^2, but usually we know from the context of an application if we are dealing with a sparse or dense graph; most of the applications use sparse graphs in fact. For example, imagine that we have a graph representing friendship relationships between people. We take the graph to contain 7 billion nodes, or $n = 7 \times 10^9$, assuming that pretty much everybody on this planet is in it. We also assume that everybody is connected to 1,000 friends, which is probably impossible. Then the number of edges is 7×10^{12}, or 7 trillion. The number $n(n-1)/2$ for $n = 7 \times 10^9$ is about 2.5×10^{25}, or 7 septillion. That is much bigger than 7 trillion. The graph would have a very large number of edges, but would still be sparse.

Copyrighted material

Copyrighted image

Figure 2.11
Complete and sparse graphs.

2.2 Graph Representation

Before we can do any work with graphs in computers, we need to see how graphs are represented in computer programs. But before that, a brief excursion into how graphs are actually defined in mathematics is necessary. Usually we call the set of vertices V and the set of edges E. A graph G is then $G = (V, E)$. In undirected graphs, the set E is a set consisting of two element sets $\{x, y\}$ for each edge between two nodes x and y of the graph. We usually write (x, y) instead of $\{x, y\}$. In both cases, the order of x and y does not matter. The graph in figure 2.5a is then defined as:

$$\begin{aligned} V &= \{0,1,2,3\} \\ E &= \{\{0,1\},\{0,2\},\{0,3\},\{1,2\},\{1,3\},\{2,3\}\} \\ &= \{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\} \end{aligned}$$

In directed graphs the set E is a set consisting of two element tuples (x, y) for each edge between two nodes x and y of the graph. This time the order of x and y is important and corresponds to the order of the edge it represents. The graph in figure 2.5b is then defined as:

$$\begin{aligned} V &= \{0,1,2,3\} \\ E &= \{(0,1),(0,2),(0,3),(1,0),(1,2),(1,3),(2,0),(3,2)\} \end{aligned}$$

Copyrighted material

Table 2.1
Adjacency matrix for the graph in figure 2.3.

	0	1	2	3	4	5
0	0	1	0	1	0	0
1	1	0	1	0	0	1
2	0	1	0	1	1	0
3	1	0	1	0	1	0
4	0	0	1	1	0	1
5	0	1	0	0	1	0

The mathematical definition of a graph shows that to represent it we need somehow to represent the vertices and the edges. A straightforward way to represent G is with a matrix. This matrix, called an *adjacency matrix*, is a square matrix with a row and a column for each vertex. The contents of the matrix are 0 or 1. If the vertex represented by the ith row is connected to the vertex connected by the jth row, then the (i, j) element of the matrix will be 1, otherwise it will be 0. In an adjacency matrix, vertices are represented by row and column indices, and vertices are represented by the contents of the matrix.

Following these rules, the adjacency matrix for the graph in figure 2.2 is shown in table 2.1. You can check that the adjacency matrix is symmetric. Also, the diagonal will be all zero, except if there are loops in the graph. If we call the matrix A, then $A_{ij} = A_{ji}$ for any two nodes i and j. This is true for all undirected graphs, but it is not true for all directed graphs (unless for every edge from node i to node j there exists an edge from node j to node i). You can also see that many of the values in the matrix are zero. This is typical of sparse graphs.

Even if a graph is not sparse, we may be wary of the space wasted for all those 0 entries in the adjacency matrix. To get over it, there is an alternative representation for graphs that uses less space. Because real-world graphs can have millions of edges, most of the times we use the alternative representation to save what we can. In this representation, we use an array to represent the vertices of the graph. Each element of the array stands for one vertex and is the start of a *list* that contains the vertices that are neighbors of a given vertex. This list is called the *adjacency list* of the vertex in the graph.

Now what exactly is a list? A list is a data structure that contains elements. Each element in the list, called a *node*, has two parts. The first part contains

Copyrighted material

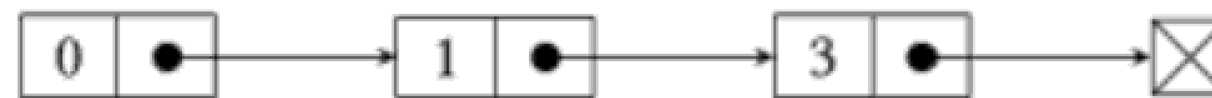


Figure 2.12
A linked list.

some data that describe the element. The second part contains a link to the next element in the list. The second part is usually a *pointer*, as it points to the next element. A pointer in computers is something that points to a location in the computer's memory; it is also called a *reference*, as it refers to that location. So the second part of a list element is usually a pointer holding an address where the next node in the list is located. A list has a *head*, its first element. We follow the elements in the list as if following the links in a chain. When an element has no next element, we say that it points to nowhere, or *null*; we use the term null to refer to nothingness in computers; because it is a special value, we'll denote it by NULL in text and pseudocode. A list constructed this way is more accurately called a *linked list*, and you can see one in figure 2.12. We use a crossed square to show NULL in the figure.

The basic operations that we need to be able to perform with lists are:

- `CreateList()`, creates and returns a new, empty list.
- `InsertListNode`(L, p, n), adds node n after node p in list L. If p is NULL, then we insert n as the new head of the list. The function returns a pointer to n. We assume that the node n has already been created with some data that we want to add in the list. We will not get into the details on how nodes are actually created. Briefly, some memory must be allocated and initialized, so that the node contains the data we want and a pointer. `InsertListNode` then needs only change pointers. It must make the pointer of n point to the next node, or to NULL, if p was the last node of the list. It must also change the pointer of p to point to n, if p is not NULL.
- `InsertInList`(L, p, d), adds a node containing d after node p in list L. If p is NULL, then we insert the new node as the new head of the list. The function returns a pointer to the newly inserted node. The difference with `InsertListNode` is that `InsertInList` creates the node that will contain d, whereas `InsertListNode` takes an already created node and inserts it in the list. `InsertListNode` inserts *nodes*, whereas `InsertInList` inserts *data* contained in nodes it creates. That means that `InsertInList` can use `InsertListNode` to insert in the list the node it creates.

Copyrighted material

- `RemoveListNode`(L, p, r), removes node r from the list and returns that node; p points to the node preceding r in the list, or NULL if r is the head. We will see that we need to know p in order to remove the item pointed by r efficiently. If r is not in the list, it returns NULL.
- `RemoveFromList`(L, d), removes the first node containing d from the list and returns the node. The difference with `RemoveListNode` is that it will search the list for the node containing d, find it, and remove it; d does not point to the node itself; it is the data contained inside the node. If there is no node containing d in the list, `RemoveFromList` returns NULL.
- `GetNextListNode`(L, p), returns the node following p in list L. If p is the last node in the list, then it returns NULL. If p is NULL, then it returns the first node of L, the head. The returned node is not removed from the list.
- `SearchInList`(L, d), searches the list L for the first node containing d. It returns the node, or NULL if no such node exists; the node is not removed from the list.

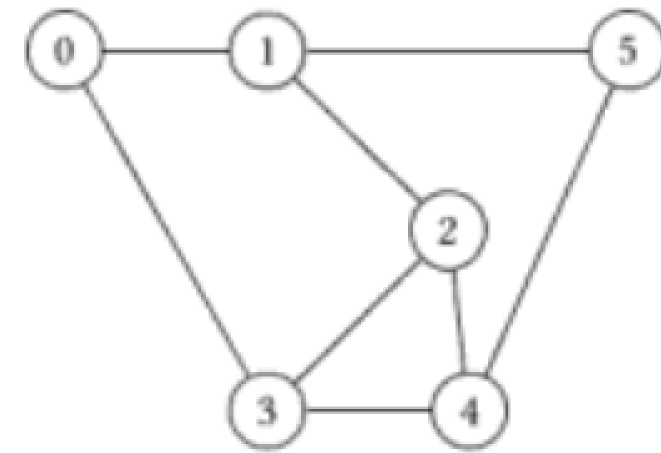
To create a representation of graph using adjacency lists, only `CreateList` and `InsertInList` are essential. To go through the elements of a list L, we need to call $n \leftarrow$ `GetNextListNode`(L, NULL) to get the first element; then, as long as $n \neq$ NULL, we call repeatedly $n \leftarrow$ `GetNextListNode`(L, n). Note that we need a way to access the data inside a node, for example, a function `GetData`(n) that returns the data d stored inside the node n.

To see how insertion works, suppose we have an empty list and we insert into it three nodes, each one of which contains a number. When we write a list in the text, we will enumerate its elements inside brackets, like this [3, 1, 4, 1, 5, 9]. An empty list will be simply [ ]. If the numbers are 3, 1, and 0 and we insert them in this order in the beginning of the list, then the list will grow as in figure 2.13 from [ ] to [0, 1, 3]. Insertion at the front of the list works by passing NULL repeatedly as the second argument to `InsertInList`.

Alternatively, if we want to append a series of nodes at the end of the list, we make a series of calls to `InsertInList` passing as second argument the return value of the previous call; you can see the pattern in figure 2.14.

Removal of a node from the list entails taking the node out and making the previous node, if it exists, point to the node following the node to be removed. If we remove the head of the list, then there is no previous node, and the next node becomes the new head of the list. Figure 2.15 shows what happens when we remove the nodes with data 3, 0, 1 from the list we created in figure 2.13. If we do not know the previous node, we have to start from the head of the

Copyrighted material



Copyrighted image

The adjacency list representation of a graph.

If the array is A, then item $A[i]$ of the array points to the head of the adjacency list of node i of the graph. If node i has no neighbors, $A[i]$ will point to NULL.

The adjacency list representation of the graph of figure 2.4 is in figure 2.18. For convenience a minimized version of the graph is on the upper right corner of figure 2.18. The figure shows on the left the array that contains the heads of the adjacency lists of the graph; each item of the array corresponds to one vertex. The adjacency lists contain the edges of the vertex in their head. So the third element of the array contains the head of the adjacency list for node 2 of the graph. Each adjacency list was constructed by taking the neighbors of each node in their numerical order. For example, to create the adjacency list of node 1, we called `Insert` three times, for nodes 0, 2, and 5, in this order. This explains why the nodes appear in reverse order in this and in every other adjacency list in the figure: nodes are inserted in the head of each list, so inserting them in the order 0, 2, 5 will result in the list [5, 2, 0].

It is straightforward to compare the space requirements of the adjacency matrix and the adjacency list representation of a graph $G = (V, E)$. If $|V|$ is the number of vertices in a graph, the adjacency matrix will have $|V|^2$ elements. Similarly, if $|E|$ is the number of edges in the graph, then the adjacency list representation of it will contain an array of size $|V|$ and then $|V|$ lists that all of them will contain $|E|$ edges. So the adjacency list representation will require $|V| + |E|$ items, much less than $|V|^2$, except if the graph is dense and many vertices are connected to each other.

Copyrighted material

You may think then that there is no reason to bother with adjacency matrices at all. There are two reasons. First, adjacency matrices are simpler. You only need to know about matrices and nothing else; no bother with lists. Second, adjacency matrices are faster. We take for granted that accessing an element in a matrix is an operation that takes constant time, that is, we can retrieve every edge, or element, as fast as any other—it does not matter whether it is near, say, the upper left corner of the matrix or the bottom right one. When we use adjacency lists, we have to access the right element on the array of vertices on the left side of figure 2.18 *and* find the required edge by traversing the list headed by the vertex. So, to see whether nodes 4 and 5 are connected, we need to go first at node 4 on the vertices matrix and then jump to 2, 3, and finally 5. To see whether nodes 4 and 0 are connected, we need to go through all the list headed by 4 until its end and then report that we have not found 0, so they are not connected by link. You may counter that it would be faster if we had searched the list headed by 0 because it is shorter, but we had no way of knowing that.

Using big-Oh notation, determining whether a vertex is connected to another vertex takes constant time, so the complexity is $\Theta(1)$ if we use an adjacency matrix. The same operation using adjacency lists takes $O(|V|)$ time because it is possible that in a graph a vertex is connected to all other vertices, and we may have to search the whole of its adjacency list for a neighbor. As you know, there is no such thing as a free lunch. In computers this translates to trade-offs: we exchange space for speed. This is something we do often, and it even has a name: *space-time tradeoff*.

2.3 Depth-first Graph Traversal

Now back to maze exploration. To explore a maze fully we need two things: some way to keep track of where we have been, and some systematic way to visit all unvisited rooms. Suppose that the rooms are somehow ordered. In the graphs we have seen, we can assume that the order is numeric. Then to visit all rooms we can go to the first room and mark it as visited. We then go to the *first* of the rooms connected to it. We mark it as visited. We go again to the first unvisited room connected to it, and we repeat the procedure: we mark it as visited, and we go to the first unvisited room connected to the room we are. If there are no unvisited rooms connected to a room, then we go back to where we came from, and we try to see whether any unvisited rooms remain there. If yes, we visit the first unvisited room there, and so on. If no, then we

Copyrighted material

Copyrighted image

Figure 2.19
Maze to be explored depth-first.

go retrace our steps back another room. We do that until we come back to the room we started in and find that we have visited all rooms connected to it.

It is easier to see that in practice. The procedure is called *depth-first search*, or DFS for short, because we are exploring the maze in depth rather than in breadth. An example maze is in figure 2.19, along with its adjacency list representation on the right. Note that again we inserted nodes in the list in reverse order. For example, in the adjacency list for node 0, we inserted the neighboring nodes in order 3, 2, 1, so the list is [0, 1, 2, 3].

Tracing depth-first search in figure 2.20, we start at node, or room, 0. We indicate with gray the current node and with black the nodes we have already visited. A double indicate shows a virtual thread that we are holding in our hand during our exploration. Much like Theseus, we use the thread to go back and retrace our steps when we cannot or should not go any farther.

The first unvisited room there is room 1, so we go to room 1. The first unvisited room from room 1 is 4. Then the first unvisited room from 4 is 5. In the same way we get from 5 to 3. At that point we find no unvisited room, so we go back by picking up the thread. We go back to 5. There we find room 6 that is still unvisited. We visit it, and we return to 5. Room 6 still has one unvisited room, 7. We visit that room, and we go back to 5 again. Now 5 has no unvisited rooms next to it, so we go back to room 4. In turn, 4 has no unvisited rooms next to it, so we trace our steps back once more to 1, and

Copyrighted material

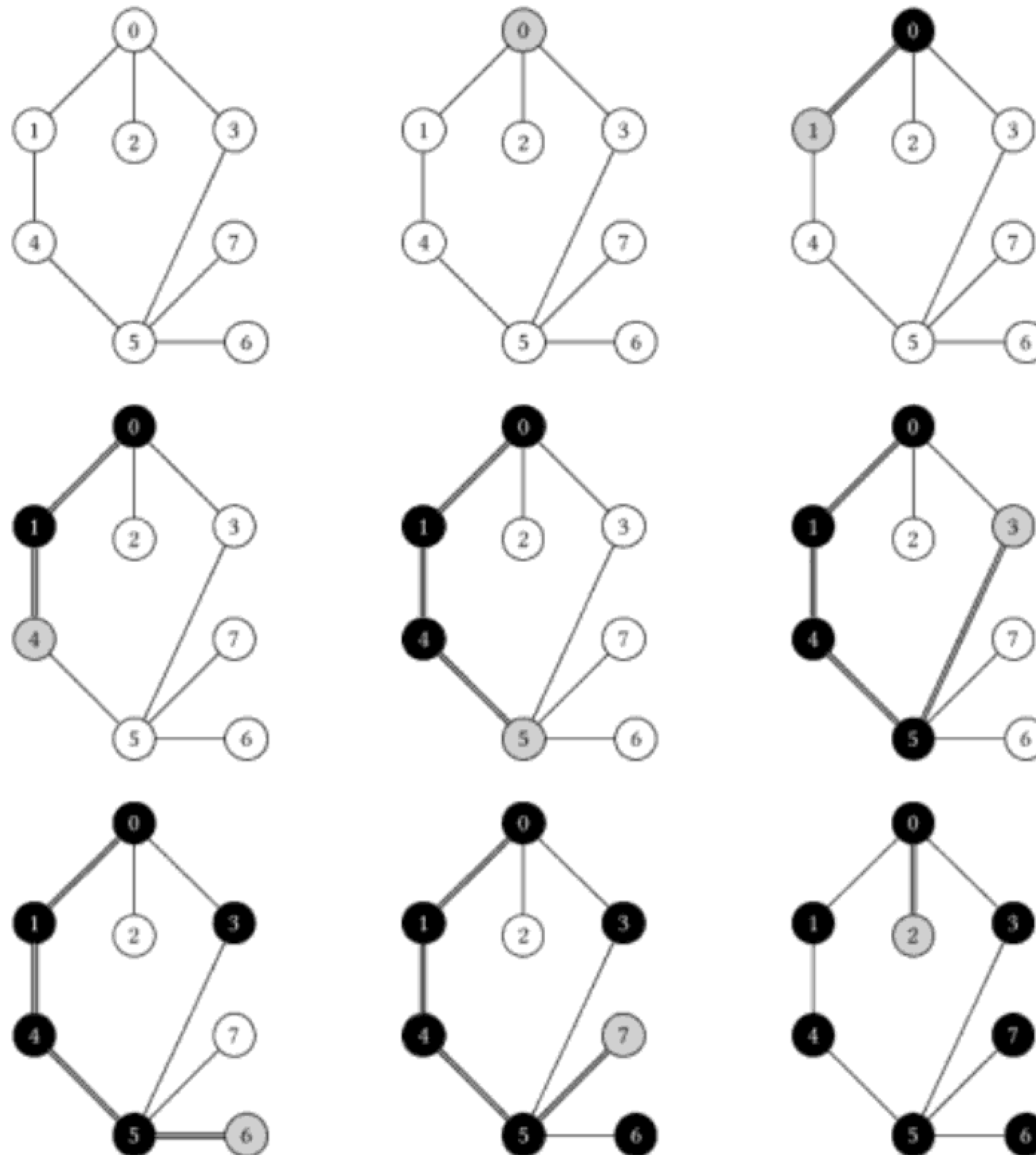


Figure 2.20
Depth-first maze exploration.

then for the same reason we return to 0. There we see that 2 is unvisited; we visit it, and we go back to 0. Now that 1, 2, and 3 are visited, we are done. If you have followed the path we described, you may have verified that we are going deep rather than wide. When in room 0, we went to 1 and then 4, instead of going to 2 and 3. Room 2, although it was next to us when we started, was the last room we visited. So indeed we go as deep as we can before we have to consider any alternatives.

Algorithm 2.1 implements depth-first search. The algorithm takes as input a graph G and a node from where it will start exploring the graph. It also uses an array *visited* that indicates for each node whether it has been visited.

Copyrighted material

Algorithm 2.1: Recursive graph depth-first search.

```
DFS(G, node)
    Input: G = (V, E), a graph
           node, a node in G
    Data: visited, an array of size |V|
    Result: visited[i] is TRUE if we have visited node i, FALSE otherwise
1   visited[node] ← TRUE
2   foreach v in AdjacencyList(G, node) do
3       if not visited[v] then
4           DFS(G, v)
```

In the beginning we have visited no node, so *visited* is all FALSE. Although *visited* is required by the algorithm, we do not include it in its inputs, as it is not something that we pass to the algorithm when we call it; it is an array, created and initialized outside the algorithm, which the algorithm can access, read, and modify. Because *visited* is modified by the algorithm it is really its output, even if we do not really say so. We do not specify any output for the algorithm, because it does not return any; yet, it communicates its results to its environment through the *visited* array, so the changes in *visited* are the algorithm's results. You can think of *visited* as a newly cleaned blackboard to which the algorithm writes its progress.

Algorithm `DFS`(G, *node*) is *recursive*. Recursive algorithms are algorithms that call themselves. `DFS`(G, *node*) marks the current vertex as visited, in line 1, and then calls itself for every unvisited vertex linked to it by walking down the adjacency list; we assume we have a function `AdjacencyList`(G, *node*) that given a graph and a node returns its adjacency list. In line 2 we go through the nodes in the adjacency list; that is easy to do because from any node we can go directly to the next one, by the definition of a list, because each node in a list is linked to the next. If we have not visited that neighboring node (line 3), then we call `DFS`(G, v) for *the neighboring node* v.

The tricky part in understanding the algorithm is understanding the recursion it performs. Let's take a trivial graph, consisting of four nodes, shown in figure 2.21a. We start with node 0. If the graph is called G, then we call `DFS`(G, 0). The function gets the adjacency list of node 0, which is [1, 3]. The loop in lines 2–4 will be executed twice: the first time for node 1 and the second time for node 3. In the first execution of the loop, as node 1 is not visited,

Copyrighted material

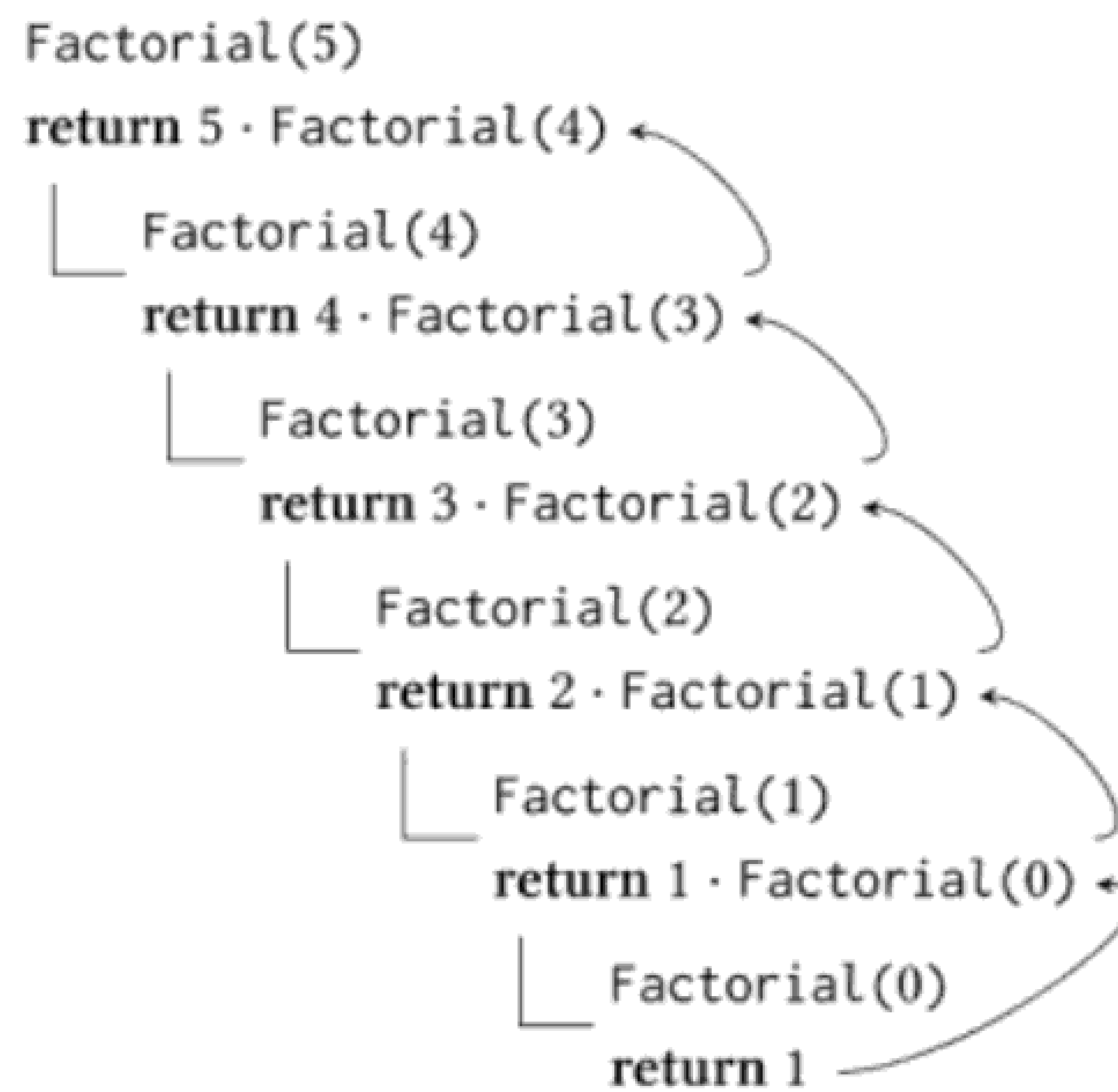


Figure 2.22
Factorial call trace.

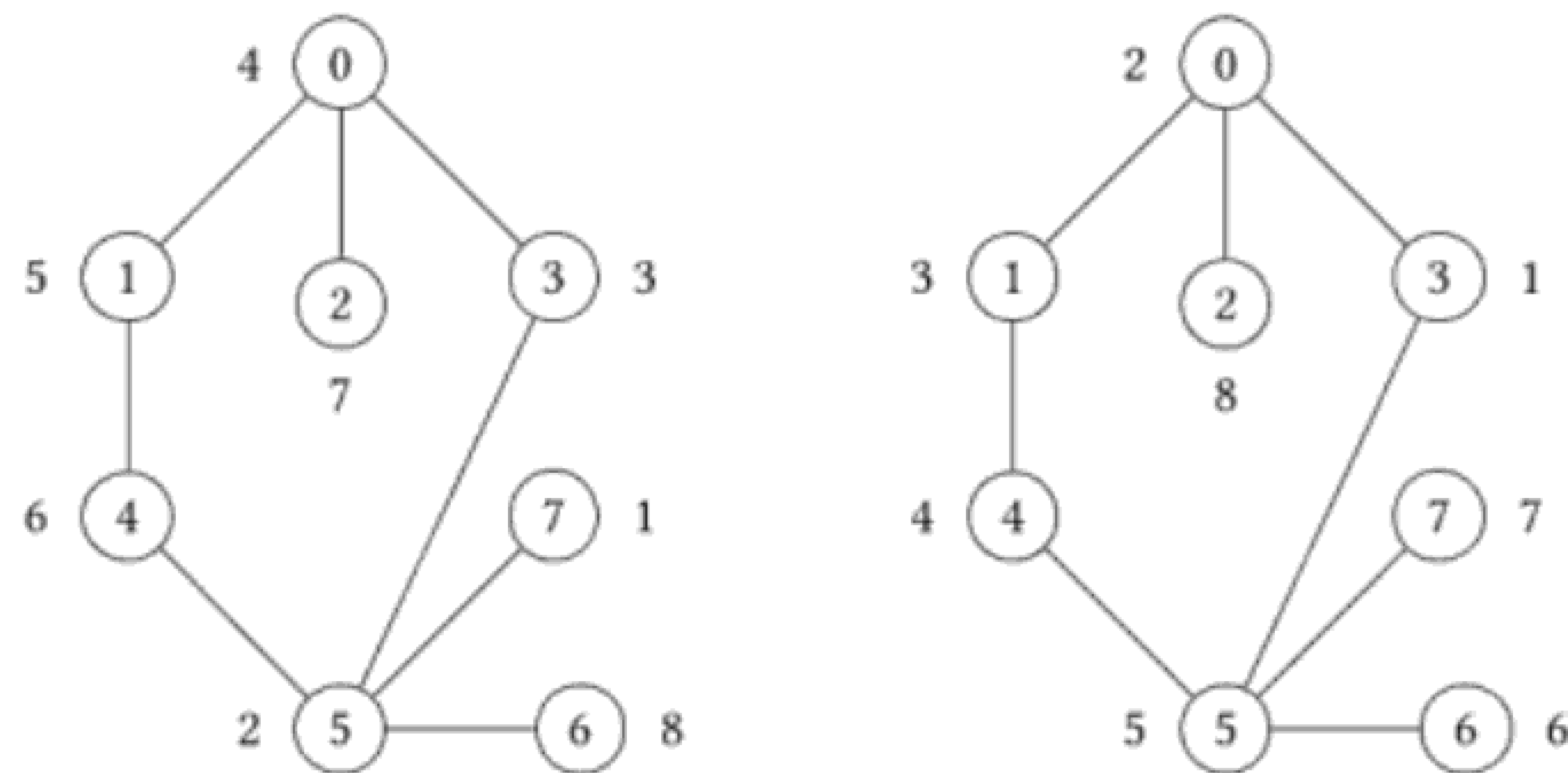
recursive calls for that node; it's time to go back. *This is very important.* Forgetting to specify when to end a recursion is a recipe for disaster. A recursive function without a stopping condition will go on calling itself, and do so ad infinitum. Programmers who forget that are in for nasty bugs. The computer will keep calling and calling the function, until it runs out of memory and the program crashes. You are left with a message that talks about a "stack overflow" or something similar, for reasons we'll explain in a little bit.

Hoping that recursion is clear now (if it is not, give it another go), note that the depth-first search algorithm works from whatever node we start. We used node 0 in our example simply because it is on the top. We can start the algorithm from node 7, or node 3, as you can see in figure 2.23, where we put the order we visit a node next to it. The order that we visit the nodes is different, but we still explore the graph fully.

This would not happen if the graph were undirected and disconnected, or if it were directed and not strongly connected. In such graphs, we must call algorithm 2.1 for each node in turn if it has not been visited. In this way, even nodes that are unreachable from one node will have their turn as start nodes.

How does recursion work in a computer? How is all this housekeeping done, putting functions on hold, calling other functions, and then knowing

Copyrighted material



(a) Depth-first search starting from node 7. (b) Depth-first search starting from node 3.

Figure 2.23
Depth-first exploration from different starting nodes.

where to return? The way a computer knows where to return from a function is by using an internal stack, called *call stack*. The stack holds the current function on top. Below it there is the function which called it with all the information to resume execution where we left it. Below that, the function that called that function, if any, and so on. That's why a recursion gone awry will cause a crash: we cannot have an infinite stack. You can see snapshots of the stack when running algorithm 2.1 in figure 2.24. We show the contents of a stack when the algorithm visits a node, painted gray below the stack. When it hits a dead end, which is a node with no unvisited neighbors to visit, the function returns, or *backtracks* to its caller; the process of retracing our steps is called *backtracking*. We call popping the top of the call stack *unwinding* (although in the case of maze exploration, this corresponds to winding our thread). So from node 3 we go to node 5, painted black to show that we have already visited it. A series of unwinding actions happens in the second row of the figure, when we go from node 7 all the way to node 0 to visit node 2.

All this stack activity happens automatically. However, there is no reason that you cannot do the same magic yourself. Instead of using recursion to perform depth-first search, which uses a stack implicitly, you can use a stack explicitly, in which case you do away with recursion altogether. The idea is that at every node we add the unvisited nodes in the stack, and when we search for nodes to visit, we simply pop elements from the stack. Algorithm 2.3 shows the stack-based implementation of depth-first search. You

Copyrighted material

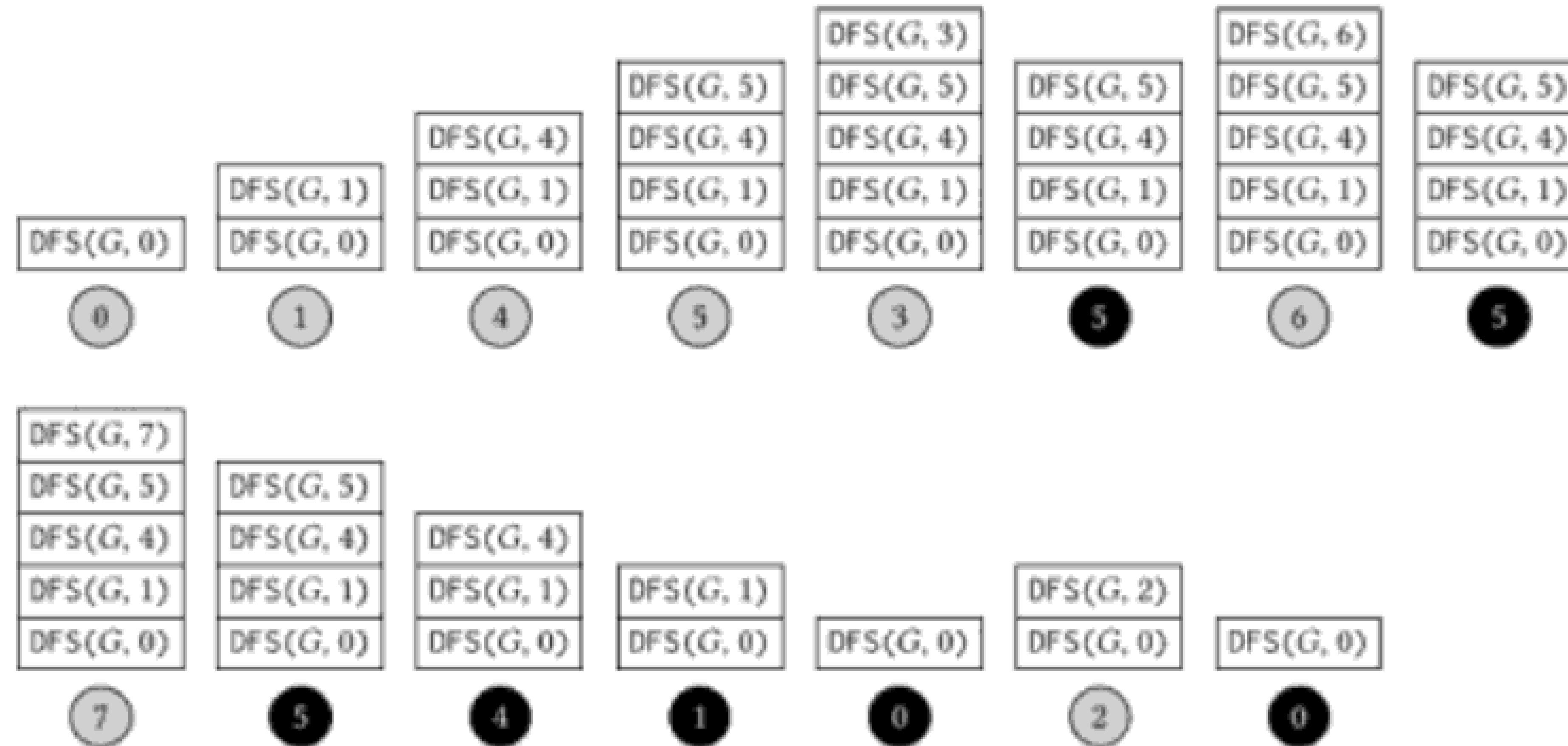


Figure 2.24
Stack evolution in depth-first search for figure 2.20.

can see the contents of the stack in figure 2.25b; we only show snapshots of the stack upon visiting a node, not when backtracking on it, in the interests of space and ink.

The algorithm works in the same way as algorithm 2.1 but uses an explicit stack instead of relying on recursion. We create the stack in line 1. This time we do not rely on an externally provided array to record our progress, creating the array *visited* ourselves in line 2; we then initialize it to all FALSE values in lines 3–4.

To emulate recursion, we push on the stack the nodes that we have yet to visit, and when we are looking for a node to visit, we go to the one at the top of the stack. To get things in motion, we push on the stack the starting node, in line 5. Then as long as there is something on the stack (line 6), we pop it (line 7), mark it as visited (line 8), and push on the stack every node in its adjacency list that we have not already visited (lines 9–11). When we are done, we return the array *visited* so that we report which nodes we were able to reach.

Because of the order nodes are put on the stack, graph traversal is depth-first but goes from higher-numbered nodes to lower-numbered ones. Whereas the recursive algorithm goes counterclockwise in our maze, this one goes clockwise.

You may notice from figure 2.25b, in the third column from the right, that node 1 is added to the stack twice. This does not make the algorithm incorrect, but we can fix it anyway. We need a *no-duplicates stack*, in which an item is

Copyrighted material

Algorithm 2.3: Graph depth-first search with a stack.

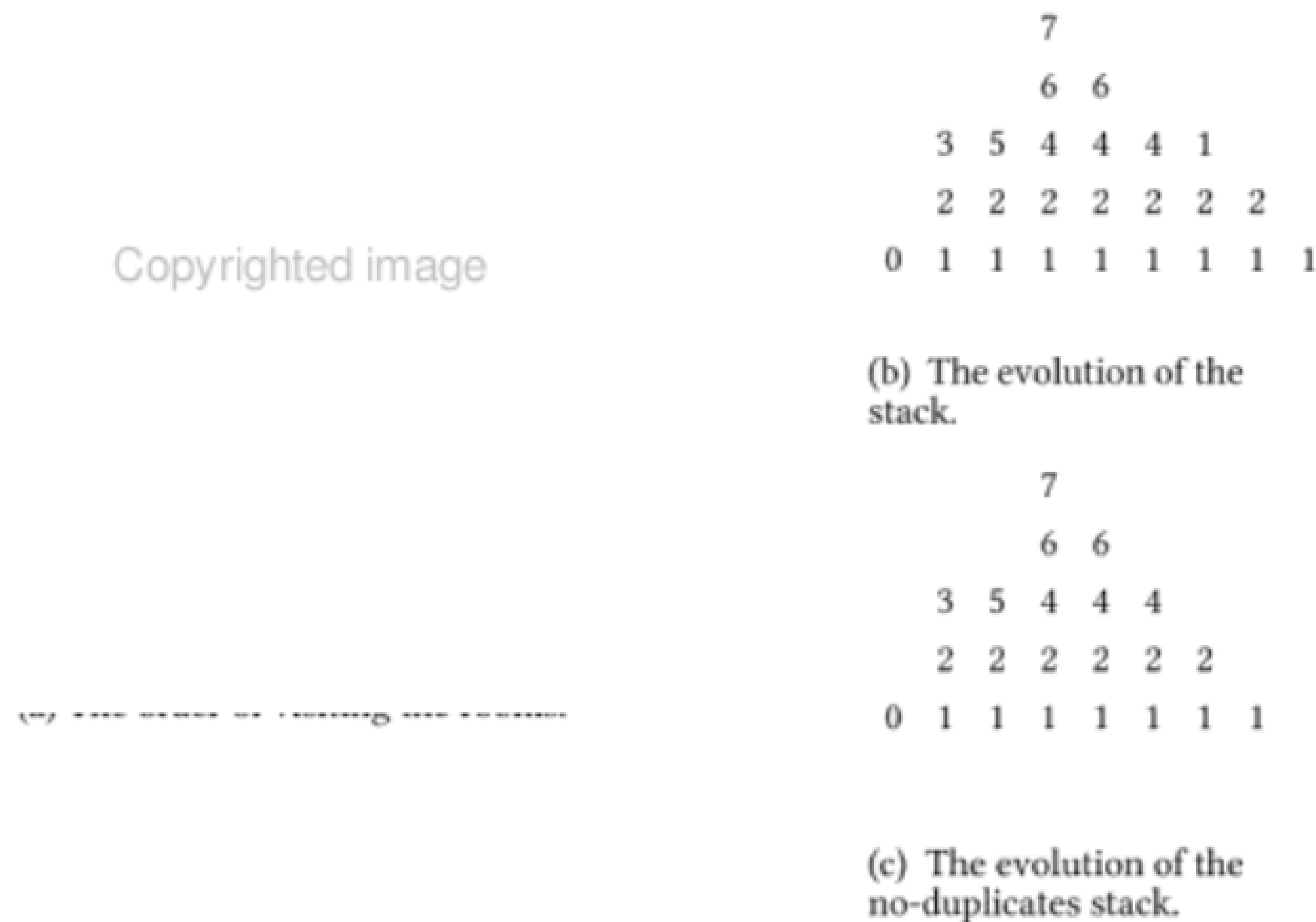
```
StackDFS(G, node) → visited
    Input: G = (V, E), a graph
           node, the starting vertex in G
    Output: visited, an array of size |V| such that visited[i] is TRUE if we
            have visited node i, FALSE otherwise

1  S ← CreateStack()
2  visited ← CreateArray(|V|)
3  for i ← 0 to |V| do
4      visited[i] ← FALSE

5  Push(S, node)
6  while not IsStackEmpty(S) do
7      c ← Pop(s)
8      visited[c] ← TRUE
9      foreach v in AdjacencyList(G, c) do
10         if not visited[v] then
11             Push(S, v)
12 return visited
```

added only if it is not already in the stack. To do this we use an additional array. An element of that array will be true if that element is currently in the stack and false otherwise. Algorithm 2.4 is the result. The algorithm is pretty much the same as algorithm 2.4 but uses the additional array *instack*, where we record those nodes that are in the stack. You can see what is happening in the stack in figure 2.25c.

You may wonder why, given that we already had algorithm 2.1, we went on to develop algorithm 2.4. Apart from it being instructive, to show how recursion really works, implicit recursion requires that the computer puts all the necessary memory state of the function on the stack at each recursive call. So it will transfer more things on the stack in figure 2.24 (where we only show the function calls) instead of simple numbers as in figure 2.25. An algorithm implemented with an explicit stack may be more economical than an equivalent recursive one.

To finish depth-first exploration, let's go back to algorithm 2.1 and examine its complexity. The complexity of algorithms 2.3 and 2.4 will be the same

Copyrighted material



(b) The evolution of the stack.

(c) The evolution of the no-duplicates stack.

Figure 2.25
Walkthrough and stack contents of algorithms 2.3 and 2.4.

as that of algorithm 2.1 because only the implementation of the recursive mechanism changes, not the overall exploration strategy. Line 2 is executed $|V|$ times, once per each vertex. Then `DFS`$(G, node)$ is called exactly once per edge, in line 4, that is, $|E|$ times. All in all, the complexity of depth-first search is $\Theta(|V| + |E|)$. We can explore the graph in time proportional to its size, which makes sense.

2.4 Breadth-first Search

As we saw, the graph exploration in depth-first search goes deep rather than wide. Suppose we would like to explore the maze in a different way, so that when we start at node 0 we visit nodes 1, 2, and 3 before we go on to visit node 4. That means that we would be casting our net wide, instead of deep. Such a search strategy is called, not surprisingly, *breadth-first search*, or BFS for short.

In breadth-first search, we can no longer rely on a thread (implicit or real) to carry us through. There is no physical means by which we can get directly from node 3 to node 4 because they are not directly connected, so the analogy

Copyrighted material

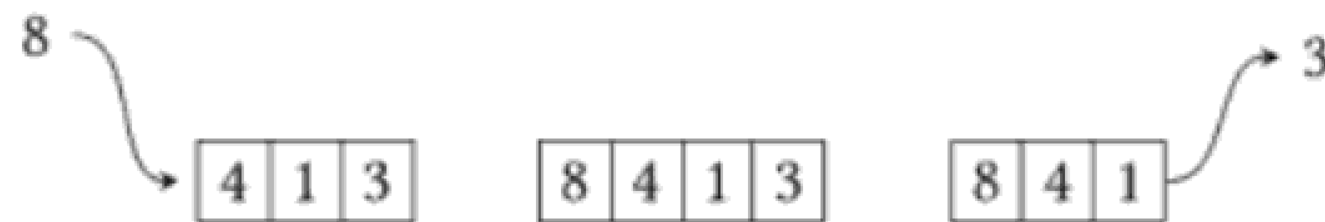


Figure 2.27
Addition and removal in a queue.

node we record its three neighbors, nodes 1, 2, and 3. We visit them in that order. When we visit node 1, we record its unvisited neighbor, node 4; that means we know we have to visit nodes 2, 3, and 4. We visit node 2, which has no unvisited neighbors, and then we go to node 3, where we record that we will have to visit node 5. The known unvisited nodes now are 4 and 5. We visit 4, then 5. When we are at node 5, we record that we need to visit nodes 6 and 7. We then visit them in that order and we are done.

To implement breadth-first search we need to use a new data structure called a *queue*, which gives us the functionality we need to keep track of the unvisited nodes below each snapshot of the graph exploration in 2.26. A queue is a sequence of things. We add things on the back of the sequence, and we remove things from its front; it works like a queue in real life, where the first one in is the first one out (unless somebody is jumping the queue). To make this clear we talk about First In First Out (FIFO) queues. We call the back of the queue its *tail* and the front of the queue its *head* (so both lists and queues have heads). You can see how addition and removal works in a queue in figure 2.27. It follows that the basic operations on a queue are:

- `CreateQueue()`, creates an empty queue.
- `Enqueue(`Q, i`)` adds item i to the tail of the queue Q.
- `Dequeue(`Q`)` removes an item from the front of the queue. In essence, it removes the head and makes the element following it the new head. If the queue is empty, then the operation is not allowed (we get an error).
- `IsQueueEmpty(`Q`)` returns TRUE if the queue Q is empty, FALSE otherwise.

With these operations at hand, we can write algorithm 2.5, which implements breadth-first search in a graph. Because the queue is filled at the tail and emptied from the head, we visit the nodes it contains by reading its contents from the right to the left, as in figure 2.26.

The algorithm is similar to algorithm 2.4. It returns an array *visited*, which indicates the nodes that we were able to reach. It uses an array *inqueue* that

Copyrighted material

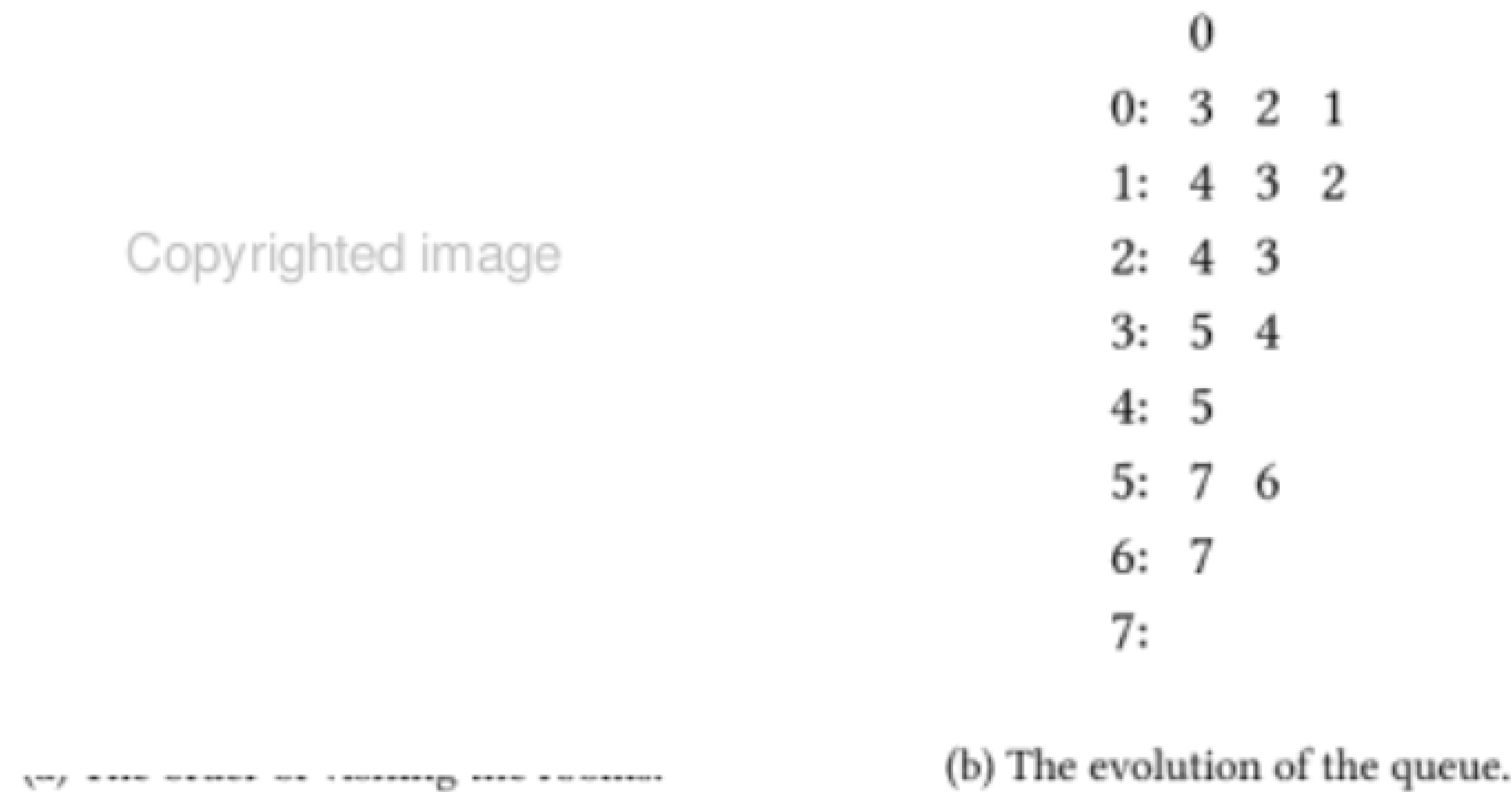
Algorithm 2.5: Graph breadth-first search.

```
BFS(G, node) → visited
    Input: G = (V, E), a graph
           node, the starting vertex in G
    Output: visited, an array of size |V| such that visited[i] is TRUE if we
            have visited node i, FALSE otherwise

1  Q ← CreateQueue()
2  visited ← CreateArray(|V|)
3  inqueue ← CreateArray(|V|)
4  for i ← 0 to |V| do
5      visited[i] ← FALSE
6      inqueue[i] ← FALSE

7  Enqueue(Q, node)
8  inqueue[node] ← TRUE
9  while not IsQueueEmpty(Q) do
10     c ← Dequeue(Q)
11     inqueue[c] ← FALSE
12     visited[c] ← TRUE
13     foreach v in AdjacencyList(G, c) do
14         if not visited[v] and not inqueue[v] then
15             Enqueue(Q, v)
16             inqueue[v] ← TRUE
17 return visited
```

keeps track of which nodes are currently in the queue. In the beginning of the algorithm, we initialize the queue we will be using (line 1), then we create and initialize *visited* and *inqueue* (lines 2–6).

The queue will always contain the nodes we know exist but we have not visited yet. In the beginning we only node the starting node, so we add it to the queue (line 7) and we keep track of that in *inqueue*. Then as long as the queue is not empty (lines 9–16), we take off the element at the head of the queue (line 10), record the fact (line 11), and mark it as visited (line 12). Then for every node in its adjacency list (lines 13–16) that is neither visited nor in the queue (line 14), we put it in the queue (line 15) and record that the node entered the queue (line 16). In this way it will come off the queue in some

Copyrighted material



Copyrighted image

(b) The evolution of the queue.

Figure 2.28
Walkthrough and queue contents of algorithm 2.5.

future iteration of the main loop of the algorithm. Figure 2.28 is a condensed version of figure 2.26.

Examining the algorithm's complexity, line 9 will be executed $|V|$ times. Then the loop starting at line 13 will be executed once for every edge of the graph, or $|E|$ times. So the complexity of breadth-first search is $\Theta(|V| + |E|)$, the same with depth-first search. That is quite pleasant; it means that we have two algorithms for graph search at our disposal, with the same complexity, but each one of them will explore the graph with a different, yet correct strategy. We can choose and pick which one fits better a particular problem we have to solve.

Notes

The foundations of graph theory were laid down by Leonhard Euler in 1736, when he presented a paper examining whether it would be possible to cross the seven bridges of Königsberg once and only once (back then Königsberg was in Prussia, today it is called Kaliningrad and it is in Russia, and only five bridges remain). The answer was negative [56]; because the original paper is in Latin, which may not be your forte, you can check the book by Biggs,

Copyrighted material

Lloyd, and Wilson [19], which contains a translation, along with lots of other interesting historical material on graph theory.

For an easy approach to graph theory, you can read the introductory book by Benjamin, Chartrand, and Zhang [15]. If you want to go deeper, you can check the book by Bondy and Murty [25]. In recent years, offshoots of graph theory treat many different aspects of all kinds of networks; see for example the books by Barabási [10], Newman [150], David and Kleinberg [48], and Watts [214]. The study of graphs in networks (of different kinds), the web, and the Internet can be seen as three different disciplines [203], where graphs are applied to explain different phenomena arising from large interconnected structures.

Depth-first search has a long provenance; a 19th-century French mathematician, Charles Pierre Trémaux, published a version of it; for a comprehensive account of this and other aspects of graph theory, see the book by Even [57]. Hopcroft and Tarjan presented depth-first search in computers and argued for representing graphs using adjacency lists [197, 96]; see also the short, classic text by Tarjan on data structures and graphs [199].

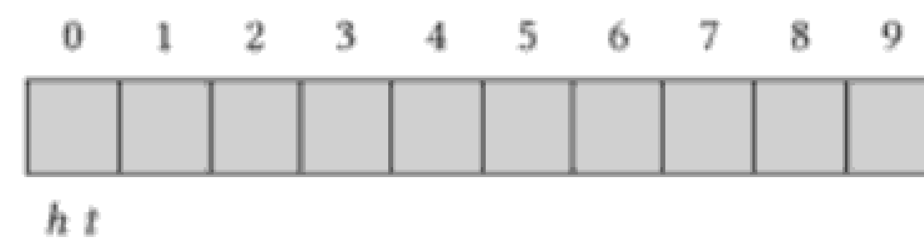
Maze exploration lay behind breadth-first search, published in the 1950s by E. F. Moore [145]. It was also discovered independently as an algorithm for routing wires on circuit boards by C. Y. Lee [126].

Exercises

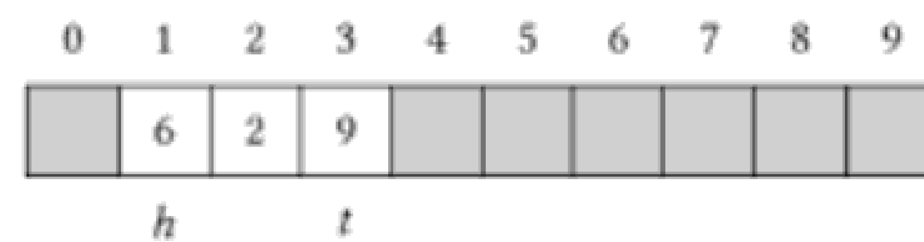
1. There are good quality implementations of lists in all popular programming languages, but it is instructive, and not difficult, to implement your own. The basic ideas are in figures 2.13 and 2.15. An empty list is a pointer to NULL. When you insert an element in the list's head, you need to adjust the list to point to the new head and make the newly inserted head point to where the list was pointing before its insertion—that is, the previous head or NULL. To insert an element after an existing element, you need to adjust the previous element's link to point to the newly inserted element and make the newly inserted element point to where the previous element was pointing before the insertion. To search for an item in the list, you need to start from the head and check each item, following the links, until you find the one you are looking for or NULL. To remove an item, you have to look for it first; once you find it, you need to make the pointer pointing to it point to the next item, or NULL if you are removing the last item.

Copyrighted material

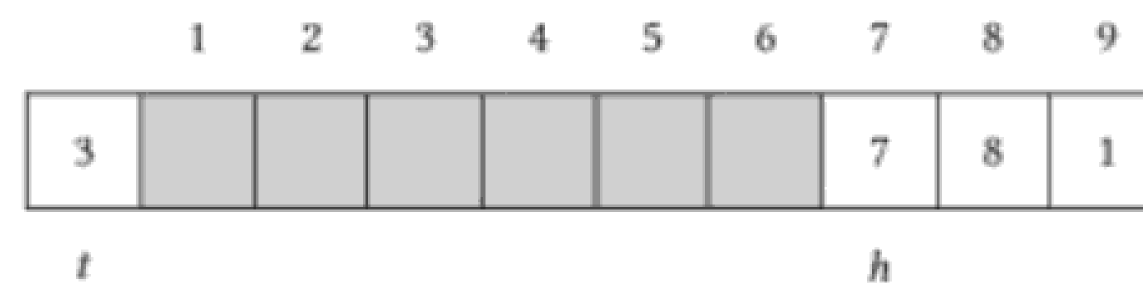
2. A queue can be implemented using an array, in which you keep track of the index of the head, h, and the tail, t, of the queue. Initially both the head and the tail are equal to 0:



When you insert an item in the queue, you increase the index of the head; similarly, when you remove an item from the queue, you increase the index of the tail. After inserting 5, 6, 2, and 9 and removing 5, the array will be:



If the array can hold n items, when the head or the tail reach the $(n - 1)$th item they wrap around to position 0. So, after several more insertions and removals the queue may look like this:



Implement a queue with this idea. The queue will be empty when the head reaches the tail and full when the tail is about to trample the head.

3. Implement depth-first search (either recursively or not) and breadth-first search using the adjacency matrix representation instead of the adjacency list one that we have used.

4. Depth-first search can be used to create mazes, not just to explore them. We start with a graph with $n \times n$ nodes that are arranged in a grid; for example, if $n = 10$ and we name the nodes by their (x, y) position we have:

Copyrighted material

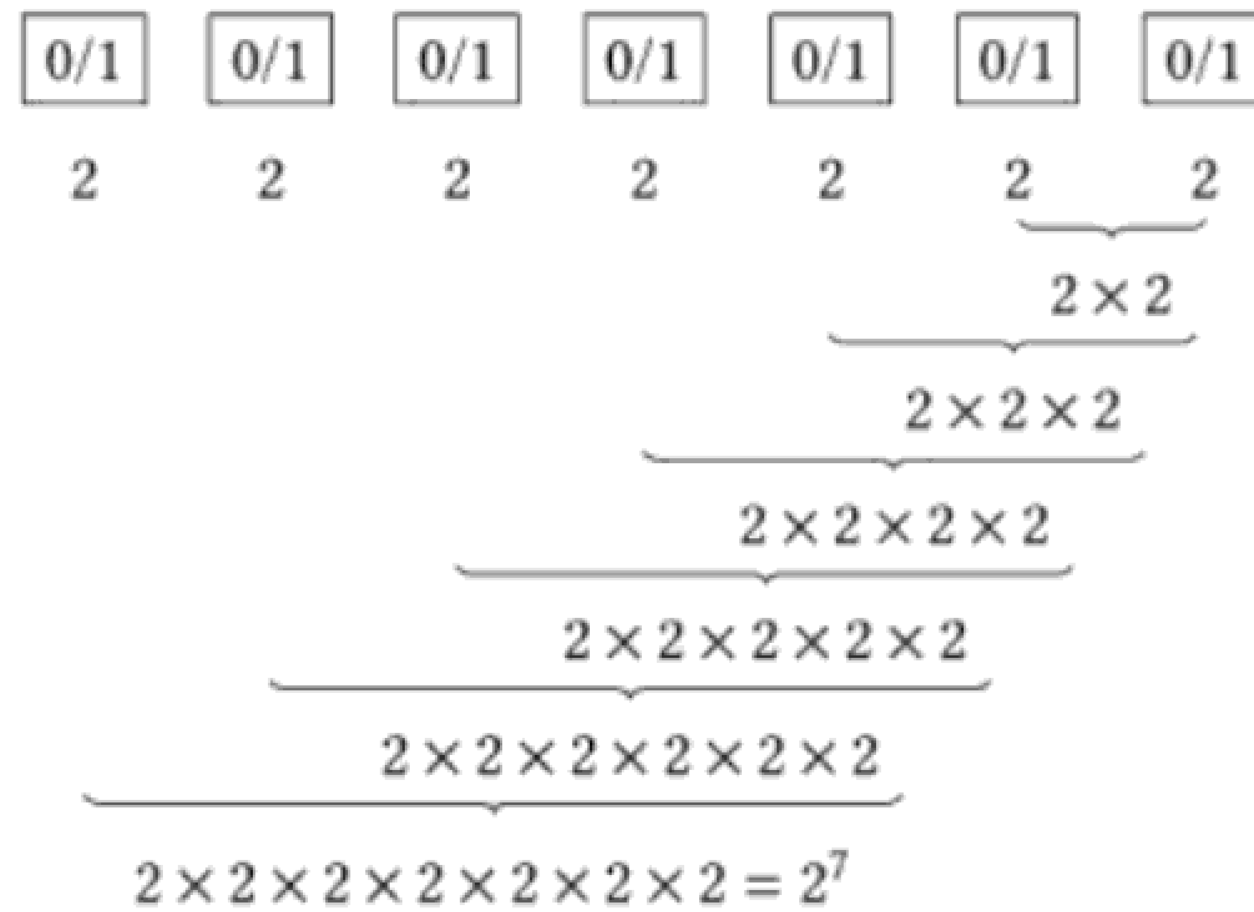


Figure 3.1
Number of possible characters represented by 7 bits.

You can see the ASCII encoding in table 3.1. Each character corresponds to a unique bit pattern, which in turn corresponds to a unique number, the value of the pattern in the binary system. The table runs in 8 rows of 16 elements each; for convenience we use the hexadecimal number system for the columns. There is nothing special about hexadecimal numbers. Instead of using the nine digits 0, 1, ..., 9, we use the sixteen digits 0, 1, ..., 9, A, B, C, D, E, F. The number A in hexadecimal is 10, the number B is 11, until the number F, which is 15. Remember that in decimal numbers, the value of a number such as 53 is $5 \times 10 + 3$, and in general a number made up of digits $D_n D_{n-1} \ldots D_1 D_0$ has value $D_n \times 10^n + D_{n-1} \times 10^{n-1} + \cdots + D_1 \times 10^1 + D_0 \times 10^0$. In hexadecimal the logic is exactly the same, but instead of 10 we use 16 as the base of our calculations. The number $H_n H_{n-1} \ldots H_1 H_0$ in hexadecimal has value $H_n \times 16^n + H_{n-1} \times 16^{n-1} + \cdots + H_1 \times 16^1 + H_0 \times 16^0$. For example, the number 20 in hexadecimal has value $2 \times 16 + 0 = 32$ and the number 1B in hexadecimal has value $1 \times 16 + 11 = 27$. Usually we prefix hexadecimal numbers by 0x to make clear we are using the hexadecimal number system and avoid any confusions. So we write 0x20 to make clear that this is written in hexadecimal and not in decimal. We also write 0x1B, although it is clear that this cannot be a number in decimal, just to keep things consistent.

By the way, the logic we described above works with other bases apart from 10 and 16; the *binary number system* directly follows when we adopt the number 2 as the *base* of our calculations. The value of a binary number made up of bits $B_n B_{n-1} \ldots B_1 B_0$ is $B_n \times 2^n + B_{n-1} \times 2^{n-1} + \cdots + B_1 \times 2^1 + B_0 \times 2^0$.

Copyrighted material

All these are examples of *positional number systems*, that is, number systems where the value of a number is derived by the position of the digits in it and the base of the number system. The general rule for finding the value is in a base b system is:

$$X_n X_{n-1} \ldots X_1 X_0 = X_n \times b^n + X_{n-1} \times b^{n-1} + \cdots + X_1 \times b^1 + X_0 \times b^0$$

If you substitute 2, 10, or 16 for b, then you get the formulas we used above. A generic notation to work with number in different number systems is $(X)_b$. For example, $(27)_{10} = (1\text{B})_{16}$.

You may wonder at this point why we go into all this trouble with hexadecimal numbers. Computers store data in memory in multiples of bytes, where a byte contains 8 bits. If you go back to figure 3.1, you will see that four bits make up $2 \times 2 \times 2 \times 2 = 2^4 = 16$ patterns. With a single hexadecimal digit, we can represent all patterns made from four bits. Then by splitting a byte in two components of four bits each, we can represent all possible bytes by splitting them down in half and using just two hexadecimal characters, from 0x0 to 0xFF. Take, for instance, the byte 11100110. If we split it in half, we get 1110 and 0110. We treat each one of them as a binary number. The binary number 1010 has value 14, as it is $2^3 + 2^2 + 2^1$. The binary number 0110 has value 6, as it is $2^2 + 2^1$. The hexadecimal number with value 14 is 0xE and the hexadecimal number with value 6 is 0x6. Therefore, we can write byte 11100110 as 0xE6. That is more elegant than 230, which is the decimal representation of the number; also, there is no easy way to derive 230 apart from doing the full calculations $2^7 + 2^6 + 2^5 + 2^2 + 2^1$, while from the hexadecimal equivalent we have immediately $\text{E} \times 16 + 6 = 14 \times 16 + 6$.

If you are still not convinced of the usefulness of hexadecimal, note that you can write down cool numbers such as 0xCAFEBABE. As it happens, 0xCAFEBABE is used to identify compiled files of programs written in the Java programming language. Spelling English words using hexadecimal characters is called *Hexspeak*, and if you search around you will find some resourceful examples.

Back to ASCII, the first 33 characters and the 128th character are control characters. These were originally intended to control devices, for instance, printers, that use ASCII. Most of them are not used anymore, apart from some exceptions that are still relevant. So the character 32 (0x20, the 33rd character since we start from 0) is the space character; character 127 (0x7F) stands for delete; character 27 (0x1B) is the escape character, and characters 10 (0xA) and 13 (0xD) stand for carriage return and line feed, respectively: these are

Copyrighted material

Table 3.1
The ASCII encoding.

	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

Table 3.2
ASCII encoding example.

I		a	m		s	e	a
0x49	0x20	0x61	0x6D	0x20	0x73	0x65	0x61
1001001	100000	1100001	1101101	100000	1110011	1100101	1100001
t	e	d		i	n		a
0x74	0x65	0x64	0x20	0x69	0x6E	0x20	0x61
1110100	1100101	1100100	100000	1101001	1101110	100000	1100001
n		o	f	f	i	c	e
0x6E	0x20	0x6F	0x66	0x66	0x69	0x63	0x65
1101110	100000	1101111	1100110	1100110	1101001	1100011	1100101

used to start a new line (depending on the operating system of the computer, only carriage return or both of them are required). Other characters are more exotic. For instance, character 7 was intended to ring a bell on teletypes.

By using table 3.1 you can find that the sentence "I am seated in an office" corresponds to the ASCII sequences in hexadecimal and binary of table 3.2. Because each character corresponds to a binary number with seven bits and the sentence contains 24 characters, we need $24 \times 7 = 168$ bits.

3.1 Compression

Can we do better than that? If we can somehow find a way to represent text in a more compact way, we could save many storage bits; and taking into account the amount of textual digital information that we store every day, the savings could turn out to be huge. Indeed, a lot of the information we store is compressed in one way or another and decompressed when we want to read it.

Copyrighted material