LEARNING MADE EASY

# Algorithms

for dummies®
A Wiley Brand

Relate algorithms to real-world uses

Develop skills for using algorithms

Learn to use Python® to test how algorithms work

John Paul Mueller
Luca Massaron

# Algorithms

by John Paul Mueller and Luca Massaron

for dummies®
A Wiley Brand

# Algorithms For Dummies®

Published by: **John Wiley & Sons, Inc.,** 111 River Street, Hoboken, NJ 07030-5774, www.wiley.com

Published simultaneously in Canada

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002. For technical support, please visit https://hub.wiley.com/community/support/dummies.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at http://booksupport.wiley.com. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2017936606

ISBN 978-1-119-33049-3 (pbk); ISBN 978-1-119-33053-0 (ebk); ISBN 978-1-119-33052-3 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

# Contents at a Glance

# Table of Contents

# Introduction

You need to learn about algorithms for school or work. Yet, all the books you've tried on the subject end up being more along the lines of really good sleep-inducing aids rather than texts to teach you something. Assuming that you can get past the arcane symbols obviously written by a demented two-year-old with a penchant for squiggles, you end up having no idea of why you'd even want to know anything about them. Most math texts are boring! However, *Algorithms For Dummies* is different. The first thing you'll note is that this book has a definite lack of odd symbols (especially of the squiggly sort) floating about. Yes, you see a few (it is a math book, after all), but what you find instead are clear instructions for using algorithms that actually have names and a history behind them to perform useful tasks. You'll encounter simple coding techniques that perform amazing things that will intrigue your friends and certainly make them jealous as you perform amazing feats of math that they can't begin to understand. You get all this without having to strain your brain, even a little, and you won't even fall asleep (well, unless you really want to do so).

## About This Book

*Algorithms For Dummies* is the math book that you wanted in college but didn't get. You discover, for example, that algorithms aren't new. After all, the Babylonians used algorithms to perform simple tasks as early as 1,600 BC. If the Babylonians could figure this stuff out, certainly you can, too! This book actually has three things that you won't find in most math books:

- Algorithms that have actual names and a historical basis so that you can remember the algorithm and know why someone took time to create it
- Simple explanations of how the algorithm performs amazing feats of data manipulation, data analysis, or probability prediction
- Code that shows how to use the algorithm without actually dealing with arcane symbols that no one without a math degree can understand

Part of the emphasis of this book is on using the right tools. This book uses Python to perform various tasks. Python has special features that make working with algorithms significantly easier. For example, Python provides access to a huge array of packages that let you do just about anything you can imagine, and more than a few that you can't. However, unlike many texts that use Python, this one doesn't bury you in packages. We use a select group of packages that provide great flexibility with a lot of functionality, but don't require you to pay anything. You can go through this entire book without forking over a cent of your hard-earned money.

You also discover some interesting techniques in this book. The most important is that you don't just see the algorithms used to perform tasks; you also get an explanation of how the algorithms work. Unlike many other books, *Algorithms For Dummies* enables you to fully understand what you're doing, but without requiring you to have a PhD in math. Every one of the examples shows the expected output and tells you why that output is important. You aren't left with the feeling that something is missing.

Of course, you might still be worried about the whole programming environment issue, and this book doesn't leave you in the dark there, either. At the beginning, you find complete installation instructions for Anaconda, which is the Python language Integrated Development Environment (IDE) used for this book. In addition, quick primers (with references) help you understand the basic Python programming that you need to perform. The emphasis is on getting you up and running as quickly as possible, and to make examples straightforward and simple so that the code doesn't become a stumbling block to learning.

To help you absorb the concepts, this book uses the following conventions:

- Text that you're meant to type just as it appears in the book is in **bold**. The exception is when you're working through a step list: Because each step is bold, the text to type is not bold.
- Words that we want you to type in that are also in *italics* are used as placeholders, which means that you need to replace them with something that works for you. For example, if you see "Type ***Your Name*** and press Enter," you need to replace *Your Name* with your actual name.
- We also use *italics* for terms we define. This means that you don't have to rely on other sources to provide the definitions you need.
- Web addresses and programming code appear in `monofont`. If you're reading a digital version of this book on a device connected to the Internet, you can click the live link to visit that website, like this: `http://www.dummies.com`.
- When you need to click command sequences, you see them separated by a special arrow, like this: File ➪ New File, which tells you to click File and then New File.

# Foolish Assumptions

You might find it difficult to believe that we've assumed anything about you — after all, we haven't even met you yet! Although most assumptions are indeed foolish, we made certain assumptions to provide a starting point for the book.

The first assumption is that you're familiar with the platform you want to use, because the book doesn't provide any guidance in this regard. (Chapter 3 does, however, tell you how to install Anaconda; Chapter 4 provides a basic Python language overview; and Chapter 5 helps you understand how to perform the essential data manipulations using Python.) To give you the maximum information about Python with regard to algorithms, this book doesn't discuss any platform-specific issues. You really do need to know how to install applications, use applications, and generally work with your chosen platform before you begin working with this book.

This book isn't a math primer. Yes, you see lots of examples of complex math, but the emphasis is on helping you use Python to perform common tasks using algorithms rather than learning math theory. However, you do get explanations of many of the algorithms used in the book so that you can understand how the algorithms work. Chapters 1 and 2 guide you through a better understanding of precisely what you need to know in order to use this book successfully.

This book also assumes that you can access items on the Internet. Sprinkled throughout are numerous references to online material that will enhance your learning experience. However, these added sources are useful only if you actually find and use them.

# Icons Used in This Book

As you read this book, you encounter icons in the margins that indicate material of interest (or not, as the case may be). Here's what the icons mean:

TIP

Tips are nice because they help you save time or perform some task without a lot of extra work. The tips in this book are time-saving techniques or pointers to resources that you should try so that you can get the maximum benefit from Python, or in performing algorithm-related or data analysis–related tasks.

WARNING

We don't want to sound like angry parents or some kind of maniacs, but you should avoid doing anything that's marked with a Warning icon. Otherwise, you might find that your application fails to work as expected, you get incorrect answers from seemingly bulletproof algorithms, or (in the worst-case scenario) you lose data.

TECHNICAL STUFF

Whenever you see this icon, think advanced tip or technique. You might find these tidbits of useful information just too boring for words, or they could contain the solution you need to get a program running. Skip these bits of information whenever you like.

REMEMBER

If you don't get anything else out of a particular chapter or section, remember the material marked by this icon. This text usually contains an essential process or a bit of information that you must know to work with Python, or to perform algorithm-related or data analysis–related tasks successfully.

# Beyond the Book

This book isn't the end of your Python or algorithm learning experience — it's really just the beginning. We provide online content to make this book more flexible and better able to meet your needs. That way, as we receive email from you, we can address questions and tell you how updates to Python, or its associated add-ons affect book content. In fact, you gain access to all these cool additions:

- **Cheat sheet:** You remember using crib notes in school to make a better mark on a test, don't you? You do? Well, a cheat sheet is sort of like that. It provides you with some special notes about tasks that you can do with Python, Anaconda, and algorithms that not every other person knows. To find the cheat sheet for this book, go to `www.dummies.com` and search for *Algorithms For Dummies Cheat Sheet.* It contains really neat information such as finding the algorithms that you commonly need to perform specific tasks.
- **Updates:** Sometimes changes happen. For example, we might not have seen an upcoming change when we looked into our crystal ball during the writing of this book. In the past, this possibility simply meant that the book became outdated and less useful, but you can now find updates to the book at `www.dummies.com/go/algorithmsfd`.

  In addition to these updates, check out the blog posts with answers to reader questions and demonstrations of useful book-related techniques at `http://blog.johnmuellerbooks.com/`.
- **Companion files:** Hey! Who really wants to type all the code in the book and reconstruct all those plots manually? Most readers prefer to spend their time actually working with Python, performing tasks using algorithms, and seeing the interesting things they can do, rather than typing. Fortunately for you, the examples used in the book are available for download, so all you need to do is read the book to learn algorithm usage techniques. You can find these files at `www.dummies.com/go/algorithmsfd`.

# Where to Go from Here

It’s time to start your algorithm learning adventure! If you’re completely new to algorithms, you should start with Chapter 1 and progress through the book at a pace that allows you to absorb as much of the material as possible. Make sure to read about Python because the book uses this language as needed for the examples.

If you’re a novice who’s in an absolute rush to get going with algorithms as quickly as possible, you can skip to Chapter 3 with the understanding that you may find some topics a bit confusing later. If you already have Anaconda installed, you can skim Chapter 3. To use this book, you must install Python version 3.4. The examples won’t work with the 2.*x* version of Python because this version doesn’t support some of the packages we use.

Readers who have some exposure to Python, and have the appropriate language versions installed, can save reading time by moving directly to Chapter 6. You can always go back to earlier chapters as necessary when you have questions. However, you do need to understand how each technique works before moving to the next one. Every technique, coding example, and procedure has important lessons for you, and you could miss vital content if you start skipping too much information.

# 1

# Getting Started

## IN THIS PART . . .

Discover how you can use algorithms to perform practical tasks.

Understand how algorithms are put together.

Install and configure Python to work with algorithms.

Use Python to work with algorithms.

Perform basic algorithm manipulations using Python.

IN THIS CHAPTER

- Defining what is meant by *algorithm*
- Relying on computers to use algorithms to provide solutions
- Determining how issues differ from solutions
- Performing data manipulation so that you can find a solution

# Chapter 1

# Introducing Algorithms

If you're in the majority of people, you're likely confused as you open this book and begin your adventure with algorithms because most texts never tell you what an algorithm is, much less why you'd want to use one. Most texts assume that you already know something about algorithms and that you are reading about them to refine and elevate your knowledge. Interestingly enough, some books actually provide a confusing definition for *algorithm* that doesn't really define it after all, and sometimes even equates it to some other form of abstract, numeric, or symbolic expression.

The first section of this chapter is dedicated to helping you understand precisely what the term *algorithm* means and why you benefit from knowing how to use algorithms. Far from being arcane, algorithms are actually used all over the place, and you have probably used or been helped by them for years without really knowing it. In truth, algorithms are becoming the spine that supports and regulates what is important in an increasingly complex and technological society like ours.

This chapter also discusses how you use computers to create solutions to problems using algorithms, how to distinguish between issues and solutions, and what you need to do to manipulate data to discover a solution. The goal of this chapter is to help you differentiate between algorithms and other tasks that people perform that they confuse with algorithms. In short, you discover why you really want to know about algorithms and how to apply them to data.

## Describing Algorithms

Even though people have solved algorithms manually for literally thousands of years, doing so can consume huge amounts of time and require many numeric computations, depending on the complexity of the problem you want to solve. Algorithms are all about finding solutions, and the speedier and easier, the better. A huge gap exists between mathematical algorithms historically created by geniuses of their time, such as Euclid, Newton, or Gauss, and modern algorithms created in universities as well as private research and development laboratories. The main reason for this gap is the use of computers. Using computers to solve problems by employing the appropriate algorithm speeds up the task significantly, which is the reason that the development of new algorithms has progressed so fast since the appearance of powerful computer systems. In fact, you may have noticed that more and more solutions to problems appear quickly today, in part, because computer power is both cheap and constantly increasing. Given their ability to solve problems using algorithms, computers (sometimes in the form of special hardware) are becoming ubiquitous.

When working with algorithms, you consider the inputs, desired outputs, and process (a sequence of actions) used to obtain a desired output from a given input. However, you can get the terminology wrong and view algorithms in the wrong way because you haven't really considered how they work in a real-world setting. The third section of the chapter discusses algorithms in a real-world manner, that is, by viewing the terminologies used to understand algorithms and to present algorithms in a way that shows that the real-world is often less than perfect. Understanding how to describe an algorithm in a realistic manner also makes it possible to temper expectations to reflect the realities of what an algorithm can actually do.

This book views algorithms in many ways. However, because it provides an overview telling how algorithms are changing and enriching people's lives, the focus is on algorithms used to manipulate data with a computer providing the required processing. With this in mind, the algorithms you work with in this book require data input in a specific form, which sometimes means changing the data to match the algorithm's requirements. Data manipulation doesn't change the content of the data. What it does do is change the presentation and form of the data so that an algorithm can help you see new patterns that weren't apparent before (but were actually present in the data all along).

Sources of information about algorithms often present them in a way that proves confusing because they're too sophisticated or downright incorrect. Although you may find other definitions, this book uses the following definitions for terms that people often confuse with algorithms (but aren't):

- **Equation:** Numbers and symbols that, when taken as a whole, equate to a specific value. An equation always contains an equals sign so that you know that the numbers and symbols represent the specific value on the other side of the equals sign. Equations generally contain variable information presented as a symbol, but they're not required to use variables.

- **Formula:** A combination of numbers and symbols used to express information or ideas. Formulas normally present mathematical or logical concepts, such as defining the Greatest Common Divisor (GCD) of two integers (the video at `https://www.khanacademy.org/math/in-sixth-grade-math/playing-numbers/highest-common-factor/v/greatest-common-divisor` tells how this works). Generally, they show the relationship between two or more variables. Most people see a formula as a special kind of equation.

REMEMBER

- **Algorithm:** A sequence of steps used to solve a problem. The sequence presents a unique method of addressing an issue by providing a particular solution. An algorithm need not represent mathematical or logical concepts, even though the presentations in this book often do fall into that category because people most commonly use algorithms in this manner. Some special formulas are also algorithms, such as the quadratic formula. In order for a process to represent an algorithm, it must be
  - **Finite:** The algorithm must eventually solve the problem. This book discusses problems with a known solution so that you can evaluate whether an algorithm solves the problem correctly.
  - **Well-defined:** The series of steps must be precise and present steps that are understandable. Especially because computers are involved in algorithm use, the computer must be able to understand the steps to create a usable algorithm.
  - **Effective:** An algorithm must solve all cases of the problem for which someone defined it. An algorithm should always solve the problem it has to solve. Even though you should anticipate some failures, the incidence of failure is rare and occurs only in situations that are acceptable for the intended algorithm use.

With these definitions in mind, the following sections help to clarify the precise nature of algorithms. The goal isn't to provide a precise definition for algorithms, but rather to help you understand how algorithms fit into the grand scheme of things so that you can develop your own understanding of what algorithms are and why they're so important.

## Defining algorithm uses

An algorithm always presents a series of steps and doesn't necessarily perform these steps to solve a math formula. The scope of algorithms is incredibly large. You can find algorithms that solve problems in science, medicine, finance, industrial production and supply, and communication. Algorithms provide support for all parts of a person's daily life. Any time a sequence of actions achieving something in our life is finite, well-defined, and effective, you can view it as an algorithm. For example, you can turn even something as trivial and simple as making toast into an algorithm. In fact, the making toast procedure often appears in computer science classes, as discussed at http://brianaspinall.com/now-thats-how-you-make-toast-using-computer-algorithms/.

TIP

Unfortunately, the algorithm on the site is flawed. The instructor never removes the toast from the wrapper and never plugs the toaster in, so the result is damaged plain bread still in its wrapper stuffed into a nonfunctional toaster (see the discussion at http://blog.johnmuellerbooks.com/2013/03/04/procedures-in-technical-writing/ for details). Even so, the idea is the correct one, yet it requires some slight, but essential, adjustments to make the algorithm finite and effective.

One of the most common uses of algorithms is as a means of solving formulas. For example, when working with the GCD of two integer values, you can perform the task manually by listing each of the factors for the two integers and then selecting the greatest factor that is common to both. For example, GCD(20, 25) is 5 because 5 is the largest number that divides into both 20 and 25. However, processing every GCD manually (which is actually a kind of algorithm) is time consuming and error prone, so the Greek mathematician Euclid (https://en.wikipedia.org/wiki/Euclid) created an algorithm to perform the task. You can see the Euclidean method demonstrated at https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm.

However, a single formula, which is a presentation of symbols and numbers used to express information or ideas, can have multiple solutions, each of which is an algorithm. In the case of GCD, another common algorithm is one created by Lehmer (see https://www.imsc.res.in/~kapil/crypto/notes/node11.html and https://en.wikipedia.org/wiki/Lehmer%27s_GCD_algorithm for details). Because you can solve any formula multiple ways, people often spend a great deal of time comparing algorithms to determine which one works best in a given situation. (See a comparison of Euclid to Lehmer at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.31.693&rep=rep1&type=pdf.)

REMEMBER

Because our society and its accompanying technology are gaining momentum, running faster and faster, we need algorithms that can keep the pace. Scientific achievements such as sequencing the human genome were possible in our age because scientists found algorithms that run fast enough to complete the task. Measuring which algorithm is better in a given situation, or in an average usage situation, is really serious stuff and a topic of discussion among computer scientists.

When it comes to computer science, the same algorithm can see multiple presentations. For example, you can present the Euclidean algorithm in both a recursive and an iterative manner, as explained at http://cs.stackexchange.com/questions/1447/what-is-most-efficient-for-gcd. In short, algorithms present a method of solving formulas, but it would be a mistake to say that just one acceptable algorithm exists for any given formula or that there is only one acceptable presentation of an algorithm. Using algorithms to solve problems of various sorts has a long history — it isn't something that has just happened.

Even if you limit your gaze to computer science, data science, artificial intelligence, and other technical areas, you find many kinds of algorithms — too many for a single book. For example, *The Art of Computer Programming*, by Donald E. Knuth (Addison-Wesley), spans 3,168 pages in four volumes (see `http://www.amazon.com/exec/obidos/ASIN/0321751043/datacservip0f-20/`) and still doesn't manage to cover the topic (the author intended to write more volumes). However, here are some interesting uses for you to consider:

- **Searching:** Locating information or verifying that the information you see is the information you want is an essential task. Without this ability, it wouldn't be possible to perform many tasks online, such as finding the website on the Internet selling the perfect coffee pot for your office.
- **Sorting:** Determining which order to use to present information is important because most people today suffer from information overload, and putting information in order is one way to reduce the onrush of data. You likely learned as a child that when you place your toys in order, it's easier to find and play with a toy that interests you, compared to having toys scattered randomly everywhere. Imagine going to Amazon, finding that over a thousand coffee pots are for sale there, and yet not being able to sort them in order of price or most positive review. Moreover, many complex algorithms require data in the proper order to work dependably, therefore ordering is an important requisite for solving more problems.
- **Transforming:** Converting one sort of data to another sort of data is critical to understanding and using the data effectively. For example, you might understand imperial weights just fine, but all your sources use the metric system. Converting between the two systems helps you understand the data. Likewise, the Fast Fourier Transform (FFT) converts signals between the time domain and the frequency domain so that it becomes possible to make things like your Wi-Fi router work.
- **Scheduling:** Making the use of resources fair to all concerned is another way in which algorithms make their presence known in a big way. For example, timing lights at intersections are no longer simple devices that count down the seconds between light changes. Modern devices consider all sorts of issues, such as the time of day, weather conditions, and flow of traffic. Scheduling comes in many forms, however. For example, consider how your computer runs multiple tasks at the same time. Without a scheduling algorithm, the operating system might grab all the available resources and keep your application from doing any useful work.

- **Graph analysis:** Deciding on the shortest line between two points finds all sorts of uses. For example, in a routing problem, your GPS couldn't function without this particular algorithm because it could never direct you along city streets using the shortest route from point A to point B.
- **Cryptography:** Keeping data safe is an ongoing battle with hackers constantly attacking data sources. Algorithms make it possible to analyze data, put it into some other form, and then return it to its original form later.
- **Pseudorandom number generation:** Imagine playing games that never varied. You start at the same place; perform the same steps, in the same manner, every time you play. Without the capability to generate seemingly random numbers, many computer tasks become impossible.

REMEMBER

This list presents an incredibly short overview. People use algorithms for many different tasks and in many different ways, and constantly create new algorithms to solve both existing problems and new problems. The most important issue to consider when working with algorithms is that given a particular input, you should expect a specific output. Secondary issues include how many resources the algorithm requires to perform its task and how long it takes to complete the task. Depending on the kind of issue and the sort of algorithm used, you may also need to consider issues of accuracy and consistency.

## Finding algorithms everywhere

The previous section mentions the toast algorithm for a specific reason. For some reason, making toast is probably the most popular algorithm ever created. Many grade-school children write their equivalent of the toast algorithm long before they can even solve the most basic math. It's not hard to imagine how many variations of the toast algorithm exist and what the precise output is of each of them. The results likely vary by individual and the level of creativity employed. In short, algorithms appear in great variety and often in unexpected places.

Every task you perform on a computer involves algorithms. Some algorithms appear as part of the computer hardware. (They are embedded, thus you hear of embedded microprocessors.) The very act of booting a computer involves the use of an algorithm. You also find algorithms in operating systems, applications, and every other piece of software. Even users rely on algorithms. Scripts help direct users to perform tasks in a specific way, but those same steps could appear as written instructions or as part of an organizational policy statement.

Daily routines often devolve into algorithms. Think about how you spend your day. If you're like most people, you perform essentially the same tasks every day in the same order, making your day an algorithm that solves the problem of how to live successfully while expending the least amount of energy possible. After all, that's what a routine does; it makes us efficient.

Emergency procedures often rely on algorithms. You take the emergency card out of the packet in front of you in the plane. On it are a series of pictographs showing how to open the emergency door and extend the slide. In some cases, you might not even see words, but the pictures convey the procedure required to perform the task and solve the problem of getting out of the plane in a hurry. Throughout this book, you see the same three elements for every algorithm:

1. Describe the problem.
2. Create a series of steps to solve the problem (well defined).
3. Perform the steps to obtain a desired result (finite and effective).

## Using Computers to Solve Problems

The term *computer* sounds quite technical and possibly a bit overwhelming to some people, but people today are neck deep (possibly even deeper) in computers. You wear at least one computer, your smartphone, most of the time. If you have any sort of special device, such as a pacemaker, it also includes a computer. Your smart TV contains at least one computer, as does your smart appliance. A car can contain as many as 30 computers in the form of embedded microprocessors that regulate fuel consumption, engine combustion, transmission, steering, and stability (according to a *New York Times* article at `http://www.nytimes.com/2010/02/05/technology/05electronics.html`) and more lines of code than a jet fighter. The automated cars appearing in the car market will require even more embedded microprocessors and algorithms of greater complexity. A computer exists to solve problems quickly and with less effort than solving them manually. Consequently, it shouldn’t surprise you that this book uses still more computers to help you understand algorithms better.

Computers vary in a number of ways. The computer in your watch is quite small; the one on your desktop quite large. Supercomputers are immense and contain many smaller computers all tasked to work together to solve complex issues, such as tomorrow’s weather. The most complex algorithms rely on special computer functionality to obtain solutions to the issues people design them to solve. Yes, you could use lesser resources to perform the task, but the trade-off is waiting a lot longer for an answer or getting an answer that lacks sufficient accuracy to provide a useful solution. In some cases, you wait so long that the answer is no longer important. With the need for both speed and accuracy in mind, the following sections discuss some special computer features that can affect algorithms.

## Leveraging modern CPUs and GPUs

General-purpose processors, CPUs, started out as a means to solve problems using algorithms. However, their general-purpose nature also means that a CPU can perform a great many other tasks, such as moving data around or interacting with external devices. A general-purpose processor does many things well, which means that it can perform the steps required to complete an algorithm, but not necessarily fast. In fact, owners of early general-purpose processors could add math coprocessors (special math-specific chips) to their systems to gain a speed advantage (see `http://www.computerhope.com/jargon/m/mathcopr.htm` for details). Today, general-purpose processors have the math coprocessor embedded into them, so when you get an Intel i7 processor, you actually get multiple processors in a single package.

TIP

Interestingly enough, Intel still markets specialty processor add-ons, such as the Xeon Phi processor used with the Xeon chips (see `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html` and `https://en.wiki2.org/wiki/Intel_Xeon_Phi` for details). You use the Xeon Phi chip alongside a Xeon chip when performing compute-intensive tasks such as machine learning (see *Machine Learning For Dummies,* by John Mueller and Luca Massaron, for details on how machine learning uses algorithms to determine how to perform various tasks that help you use data to predict the unknown and to organize information meaningfully).

You may wonder why this section mentions Graphics Processing Units (GPUs). After all, GPUs are supposed to take data, manipulate it in a special way, and then display a pretty picture onscreen. Any computer hardware can serve more than one purpose. It turns out that GPUs are particularly adept at performing data transformations, which is a key task for solving algorithms in many cases. A GPU is a special-purpose processor, but one with capabilities that lend themselves to faster algorithm execution. It shouldn't surprise you to discover that people who create algorithms spend a lot of time thinking outside the box, which means that they often see methods of solving issues in nontraditional approaches.

The point is that CPUs and GPUs form the most commonly used chips for performing algorithm-related tasks. The first performs general-purpose tasks quite well, and the second specializes in providing support for math-intensive tasks, especially those that involve data transformations. Using multiple cores makes parallel processing (performing more than one algorithmic step at a time) possible. Adding multiple chips increases the number of cores available. Having more cores adds speed, but a number of factors keeps the speed gain to a minimum. Using two i7 chips won't produce double the speed of just one i7 chip.

## Working with special-purpose chips

A math coprocessor and a GPU are two examples of common special-purpose chips in that you don't see them used to perform tasks such as booting the system. However, algorithms often require the use of uncommon special-purpose chips to solve problems. This isn't a hardware book, but spending a little time looking around can show you all sorts of interesting chips, such as the new artificial neurons that IBM is working on (see the story at `http://www.computerworld.com/article/3103294/computer-processors/ibm-creates-artificial-neurons-from-phase-change-memory-for-cognitive-computing.html`). Imagine performing algorithmic processing using memory that simulates the human brain. It would create an interesting environment for performing tasks that might not otherwise be possible today.

Neural networks, a technology that is used to simulate human thought and make deep learning techniques possible for machine learning scenarios, are now benefitting from the use of specialized chips, such as the Tesla P100 from NVidia (see the story at `https://www.technologyreview.com/s/601195/a-2-billion-chip-to-accelerate-artificial-intelligence/` for details). These kinds of chips not only perform algorithmic processing extremely fast, but learn as they perform the tasks, making them faster still with each iteration. Learning computers will eventually power robots that can move (after a fashion) on their own, akin to the robots seen in the movie *I Robot* (see one such robot described at `http://www.cbsnews.com/news/this-creepy-robot-is-powered-by-a-neural-network/`). There are also special chips that perform tasks such as visual recognition (see `https://www.technologyreview.com/s/537211/a-better-way-to-build-brain-inspired-chips/` for details).

REMEMBER

No matter how they work, specialized processors will eventually power all sorts of algorithms that will have real-world consequences. You can already find many of these real-world applications in a relatively simple form. For example, imagine the tasks that a pizza-making robot would have to solve — the variables it would have to consider on a real-time basis. This sort of robot already exists (this is just one example of the many industrial robots used to produce material goods by employing algorithms), and you can bet that it relies on algorithms to describe what to do, as well as on special chips to ensure that the tasks are done quickly (see the story at `http://www.bloomberg.com/news/articles/2016-06-24/inside-silicon-valley-s-robot-pizzeria`).

TECHNICAL STUFF

Eventually, it might even be possible to use the human mind as a processor and output the information through a special interface. Some companies are now experimenting with putting processors directly into the human brain to enhance its ability to process information (see the story at `https://www.washingtonpost.com/news/the-switch/wp/2016/08/15/putting-a-computer-in-your-brain-is-no-longer-science-fiction/` for details). Imagine a system in which humans can solve algorithms at the speed of computers, but with the creative "what if" potential of humans.

## Leveraging networks

Unless you have unlimited funds, using some algorithms effectively may not be possible, even with specialized chips. In that case, you can network computers together. Using special software, one computer, a master, can use the processors of all slave computers running an *agent* (a kind of in-memory background application that makes the processor available). Using this approach, you can solve incredibly complex problems by offloading pieces of the problem to a number of slave computers. As each computer in the network solves its part of the problem, it sends the results back to the master, which puts the pieces together to create a consolidated answer, a technique called *cluster computing.*

Lest you think this is the stuff of science fiction, people are already using cluster computing techniques in all sorts of interesting ways. For example, the article at `http://www.zdnet.com/article/build-your-own-supercomputer-out-of-raspberry-pi-boards/` details how you can build your own supercomputer by combining multiple Raspberry Pi (`https://www.raspberrypi.org/`) boards into a single cluster.

*Distributed computing,* another version of cluster computing (but with a looser organization) is also popular. In fact, you can find a list of distributed computing projects at `http://www.distributedcomputing.info/projects.html`. The list of projects includes some major endeavors, such as Search for Extraterrestrial Intelligence (SETI). You can also donate your computer's extra processing power to work on a cure for cancer. The list of potential projects is amazing.

Networks also let you access other people's processing power in an unattached form. For example, Amazon Web Services (AWS) and other vendors provide the means to use their computers to perform your work. A network connection can make the remote computers feel as if they're part of your own network. The point is that you can use networking in all sorts of ways to create connections between computers to solve a variety of algorithms that would be too complicated to solve using just your system.

## Leveraging available data

Part of solving an algorithm has nothing to do with processing power, creative thinking outside the box, or anything of a physical nature. To create a solution to most problems, you also need data on which to base a conclusion. For example, in the toast-making algorithm, you need to know about the availability of bread, a toaster, electricity to power the toaster, and so on before you can solve the problem of actually making toast. The data becomes important because you can't finish the algorithm when missing even one element of the required solution. Of course, you may need additional input data as well. For example, the person wanting the toast may not like rye. If this is the case and all you have is rye bread to use, the presence of bread still won't result in a successful result.

Data comes from all sorts of sources and in all kinds of forms. You can stream data from a source such as a real-time monitor, access a public data source, rely on private data in a database, scrape the data from websites, or get it in myriad other ways too numerous to mention here. The data may be static (unchanging) or dynamic (constantly changing). You may find that the data is complete or missing elements. The data may not appear in the right form (such as when you get imperial units and require metric units when solving a weight problem). The data may appear in a tabular format when you need it in some other form. It may reside in an unstructured way (for instance in a NoSQL database or just in a bunch of different data files) when you need the formal formatting of a relational database. In short, you need to know all sorts of things about the data used with your algorithm in order to solve problems with it.

REMEMBER

Because data comes in so many forms and you need to work with it in so many ways, this book pays a lot of attention to data. Starting in Chapter 6, you discover just how data structure comes into play. Moving on to Chapter 7, you begin looking at how to search through data to find what you need. Chapters 12 through 14 help you work with big data. However, you can find some sort of data-specific information in just about every chapter of the book because without data, an algorithm can't solve any problems.

# Distinguishing between Issues and Solutions

This book discusses two parts of the algorithmic view of the real world. On the one hand, you have *issues*, which are problems that you need to solve. An issue can describe the desired output of an algorithm or it can describe a hurdle you must overcome to obtain the desired output. *Solutions* are the methods, or steps, used to address the issues. A solution can relate to just one step or many steps within the algorithm. In fact, the output of an algorithm, the response to the last step, is a solution. The following sections help you understand some of the important aspects of issues and solutions.

## Being correct and efficient

Using algorithms is all about getting an acceptable answer. The reason you look for an acceptable answer is that some algorithms generate more than one answer in response to fuzzy input data. Life often makes precise answers impossible to get. Of course, getting a precise answer is always the goal, but often you end up with an acceptable answer instead.

Getting the most precise answer possible may take too much time. When you get a precise answer but that answer comes too late to use, the information becomes useless and you've wasted your time. Choosing between two algorithms that address the same issue may come down to a choice between speed and precision. A fast algorithm may not generate a precise answer, but the answer may still work well enough to provide useful output.

Wrong answers can be a problem. Creating a lot of wrong answers fast is just as bad as creating a lot of precisely correct answers slowly. Part of the focus of this book is helping you find the middle ground between too fast and too slow, and between inaccurate and too accurate. Even though your math teacher stressed the need for providing the correct answer in the way expressed by the book you used at the time, real-world math often involves weighing choices and making middle-ground decisions that affect you in ways you might not think possible.

## Discovering there is no free lunch

You may have heard the common myth that you can have everything in the way of computer output without putting much effort into deriving the solution. Unfortunately, no absolute solution exists to any problem, and better answers are often quite costly. When working with algorithms, you quickly discover the need to provide additional resources when you require precise answers quickly. The size and complexity of the data sources you use greatly affect the solution resolution as well. As size and complexity increase, you find that the need to add resources increases as well.

## Adapting the strategy to the problem

Part 5 of this book looks at strategies you can use to decrease the cost of working with algorithms. The best mathematicians use tricks to get more output from less computing. For example, you can create an ultimate algorithm to solve an issue, or you can use a host of simpler algorithms to solve the same issue, but using multiple processors. The host of simple algorithms will usually work faster and better than the single, complex algorithm, even though this approach seems counterintuitive.

## Describing algorithms in a lingua franca

Algorithms do provide a basis for communication between people, even when those individuals have different perspectives and speak different languages. For example, Bayes' Theorem (the probability of an event occurring given certain premises; see https://betterexplained.com/articles/an-intuitive-and-short-explanation-of-bayes-theorem/ for a quick explanation of this amazing theorem)

```
P(B|E) = P(E|B)*P(B)/P(E)
```

appears the same whether you speak English, Spanish, Chinese, German, French, or any other language. Regardless what language you speak, the algorithm looks the same and acts the same given the same data. Algorithms help cross all sorts of divides that serve to separate humans from each other by expressing ideas in a form that anyone can prove. As you go through this book, you discover the beauty and magic that algorithms can provide in communicating even subtle thoughts to others.

TIP

Apart from universal mathematical notations, algorithms take advantage of programming languages as a means for explaining and communicating the formulas they solve. You can find all the sorts of algorithms in C, C++, Java, Fortran, Python (as in this book), and other languages. Some writers rely on pseudocode to overcome the fact that an algorithm may be proposed in a programming language that you don't know. *Pseudocode* is a way to describe computer operations by using common English words.

## Facing hard problems

An important consideration when working with algorithms is that you can use them to solve issues of any complexity. The algorithm doesn't think, have emotion, or care how you use it (or even abuse it). You can use algorithms in any way required to solve an issue. For example, the same group of algorithms used to perform facial recognition to act as an alternative to computer passwords (for security purposes) can find terrorists lurking in an airport or recognize a lost child wandering the streets. The same algorithm has different uses; how to use it depends on the interests of the user. Part of the reason you want to read this book carefully is to help you solve those hard problems that may require only a simple algorithm to address.

# Structuring Data to Obtain a Solution

Humans think about data in nonspecific ways and apply various rules to the same data to understand it in ways that computers never can. A computer's view of data is structured, simple, uncompromising, and most definitely not creative. When humans prepare data for a computer to use, the data often interacts with the algorithms in unexpected ways and produces undesirable output. The problem is one in which the human fails to appreciate the limited view of data that a computer has. The following sections describe two aspects of data that you see illustrated in many of the chapters to follow.

## Understanding a computer's point of view

A computer has a simple view of data, but it's also a view that humans typically don't understand. For one thing, everything is a number to a computer because computers aren't designed to work with any other kind of data. Humans see characters on the computer display and assume that the computer interacts with the data in that manner, but the computer doesn't understand the data or its implications. The letter *A* is simply the number 65 to the computer. In fact, it's not truly even the number 65. The computer sees a series of electrical impulses that equate to a binary value of 0100 0001.

Computers also don't understand the whole concept of uppercase and lowercase. To a human, the lowercase *a* is simply another form of the uppercase *A*, but to a computer they're two different letters. A lowercase *a* appears as the number 97 to the computer (a binary value of 0110 0001).

If these simple sorts of single letter comparisons could cause such problems between humans and computers, it isn't hard to imagine what happens when humans start assuming too much about other kinds of data. For example, a computer can't hear or appreciate music. Yet, music comes out of the computer speakers. The same holds true for graphics. A computer sees a series of 0s and 1s, not a graphic containing a pretty scene of the countryside.

REMEMBER

It's important to consider data from the computer's perspective when using algorithms. The computer sees only 0s and 1s, nothing else. Consequently, when you start working through the needs of the algorithm, you must view the data in that manner. You may actually find it beneficial to know that the computer's view of data makes some solutions easier to find, not harder. You discover more about this oddity in viewing data as the book progresses.

## Arranging data makes the difference

Computers also have a strict idea about the form and structure of data. When you begin working with algorithms, you find that a large part of the job involves making the data appear in a form that the computer can use when using the algorithm to find a solution to an issue. Although a human can mentally see patterns in data that isn't arranged precisely right, computers really do need the precision to find the same pattern. The benefit of this precision is that computers can often make new patterns visible. In fact, that's one of the main reasons to use algorithms with computers — to help locate new patterns and then use those patterns to perform other tasks. For example, a computer may recognize a customer's spending pattern so that you can use the information to generate more sales automatically.

IN THIS CHAPTER

» Considering how to solve a problem

» Using a divide-and-conquer approach to solving problems

» Understanding the greedy approach to solving problems

» Determining the costs of problem solutions

» Performing algorithm measurements

# Chapter 2
# Considering Algorithm Design

As stated in Chapter 1, an algorithm consists of a series of steps used to solve a problem. In most cases, input data provides the basis of solving the problem and sometimes offers constraints that any solution must consider before anyone will see the algorithm as being effective. The first section of this chapter helps you consider the *problem solution* (the solution to the problem you're trying to solve). It helps you understand the need to create algorithms that are both flexible (in that they can handle a wide range of data inputs) and effective (in that they yield the desired output).

Some problems are quite complex. In fact, you look at them at first and may decide that they're too complicated to solve. Feeling overwhelmed by a problem is common. The most common way to solve the issue is to divide the problem into smaller pieces, each of which is manageable on its own. The divide-and-conquer approach to problem solving, discussed in this chapter's second section, originally referred to warfare (see `http://classroom.synonym.com/civilization-invented-divide-conquer-strategy-12746.html` for a history of this approach). However, people use the same ideas to cut problems of all sorts down to size.

REMEMBER

The third section of the chapter refers to the greedy approach to problem solving. Greed normally has a negative connotation, but not in this case. A *greedy algorithm* is one that makes an optimal choice at each problem-solving stage. By doing so, it hopes to obtain an overall optimal solution to solve the problem. Unfortunately, this strategy doesn't always work, but it's always worth a try. It often yields a *good enough* solution, making it a good baseline.

No matter what problem-solving approach you choose, every algorithm comes with costs. Being good shoppers, people who rely heavily on algorithms want the best possible deal, which means performing a cost/benefit analysis. Of course, getting the best deal also assumes that a person using the algorithm has some idea of what sort of solution is good enough. Getting a solution that is too precise or one that offers too much in the way of output is often wasteful, so part of keeping costs under control is getting what you need as output and nothing more.

To know what you have with an algorithm, you need to know how to measure it in various ways. Measurements create a picture of usability, size, resource usage, and cost in your mind. More important, measurements offer the means of making comparisons. You can't compare algorithms without measurements. Until you can compare the algorithms, you can't choose the best one for a task.

## Starting to Solve a Problem

Before you can solve any problem, you must understand it. It isn't just a matter of sizing up the problem, either. Knowing that you have certain inputs and require certain outputs is a start, but that's not really enough to create a solution. Part of the solution process is to

- Discover how other people have created new problem solutions
- Know what resources you have on hand
- Determine the sorts of solutions that worked for similar problems in the past
- Consider what sorts of solutions haven't produced a desirable result

The following sections help you understand these phases of solving a problem. Realize that you won't necessarily perform these phases in order and that sometimes you revisit a phase after getting more information. The process of starting a problem solution is iterative; you keep at it until you have a good understanding of the problem at hand.

## Modeling real-world problems

Real-world problems differ from those found in textbooks. When creating a textbook, the author often creates a simple example to help the reader understand the basic principles at work. The example models just one aspect of a more complex problem. A real-world problem may require that you combine several techniques to create a complete solution. For example, to locate the best answer to a problem, you may:

1. Need to sort the answer set by a specific criterion.
2. Perform some sort of filtering and transformation.
3. Search the result.

Without this sequence of steps, comparing each of the answers adequately may prove impossible, and you end up with a less-than-optimal result. A series of algorithms used together to create a desired result is an *ensemble*. You can read about their use in machine learning in *Machine Learning For Dummies,* by John Paul Mueller and Luca Massaron (Wiley). The article at `https://www.toptal.com/machine-learning/ensemble-methods-machine-learning` gives you a quick overview of how ensembles work.

However, real-world problems are even more complex than simply looking at static data or iterating that data only once. For example, anything that moves, such as a car, airplane, or robot, receives constant input. Each updated input includes error information that a real-world solution will need to incorporate into the result in order to keep these machines working properly. In addition to other algorithms, the constant calculations require the proportional integral derivative (PID) algorithm (see `http://www.ni.com/white-paper/3782/en/` for a detailed explanation of this algorithm) to control the machine using a feedback loop. Every calculation brings the solution used to control the machine into better focus, which is why machines often go through a settling stage when you first turn them on. (If you work with computers regularly, you might be used to the idea of iterations. PIDs are for continuous systems; therefore, there are no iterations.) Finding the right solution is called *settling time* — the time during which the algorithm controlling the machine hasn't yet found the right answer.

When modeling a real-world problem, you must also consider non-obvious issues that crop up. An obvious solution, even one based on significant mathematical input and solid theory, may not work. For example, during WWII, the allies had a serious problem with bomber losses. Therefore, the engineers analyzed every bullet hole in every plane that came back. After the analysis, the engineers used their solution to armor the allied planes more heavily to ensure that more of them would come back. It didn't work. Enter Abraham Wald. This mathematician suggested a non-obvious solution: Put armor plating in all the places that lacked bullet holes (because the areas with bullet holes are already strong enough;

otherwise the plane wouldn't have returned). The resulting solution did work and is now used as the basis for *survivor bias* (the fact that the survivors of an incident often don't show what actually caused a loss) in working with algorithms. You can read more about this fascinating bit of history at `http://www.macgetit.com/solving-problems-of-wwii-bombers/`. The point is that biases and other problems in modeling problems can create solutions that don't work.

Real-world modeling may also include the addition of what scientists normally consider undesirable traits. For example, scientists often consider noise undesirable because it hides the underlying data. Consider a hearing aid, which removes noise to enable someone to hear better (see the discussion at `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4111515/` for details). Many methods exist for removing noise, some of which you can find in this book starting with Chapter 9 as part of other topic discussions. However, as counterintuitive as it might seem, adding noise also requires an algorithm that provides useful output. For example, Ken Perlin wanted to get rid of the machine-like look of computer-generated graphics in 1983 and created an algorithm to do so. The result is Perlin noise (see `http://paulbourke.net/texture_colour/perlin/` for details). The effect is so useful that Ken won an Academy Award for his work (see `http://mrl.nyu.edu/~perlin/doc/oscar.html` for details). Other people, such as Steven Worley, have created other sorts of noise that affect graphics in other ways (see the discussion at `http://procworld.blogspot.com/2011/05/hello-worley.html`, which compares Perlin noise to Worley noise). The point is that whether you need to remove or add noise depends on the problem domain you want to solve. A real-world scenario often requires choices that may not be obvious when working in the lab or during the learning process.

REMEMBER

The main gist of this section is that solutions often require several iterations to create, you may have to spend a lot of time refining them, and obvious solutions may not work at all. When modeling a real-world problem, you do begin with the solutions found in textbooks, but then you must move beyond theory to find the actual solution to your problem. As this book progresses, you're exposed to a wide variety of algorithms — all of which help you find solutions. The important thing to remember is that you may need to combine these examples in various ways and discover methods for interacting with data so that it lends itself to finding patterns that match the output you require.

## Finding solutions and counterexamples

The previous section introduces you to the vagaries of discovering real-world solutions that consider issues that solutions found in the lab can't consider. However, just finding a solution — even a good one — isn't sufficient because even good solutions fail on occasion. Playing the devil's advocate by locating counterexamples is an important part of starting to solve a problem. The purpose of counterexamples is to

- Potentially disprove the solution
- Provide boundaries that define the solution better
- Consider situations in which the hypothesis used as a basis for the solution remains untested
- Help you understand the limits of the solution

A common scenario that illustrates a solution and counterexample is the statement that all prime numbers are odd. (*Prime numbers* are integers that can be divided only by themselves and 1 to produce an integer result.) Of course, the number 2 is prime, but it's also even, which makes the original statement false. Someone making the statement could then qualify it by saying that all prime numbers are odd except 2. The partial solution to the problem of finding all the prime numbers is that you need to find odd numbers, except in the case of 2, which is even. In this second case, disproving the solution is no longer possible, but adding to the original statement provides a boundary.

By casting doubt on the original assertion, you can also consider situations in which the hypothesis, all prime numbers except 2 are odd, may not hold true. For example, 1 is an odd number but isn't considered prime (see the discussion at `https://primes.utm.edu/notes/faq/one.html` for details). So now the original statement has two boundaries, and you must restate it as follows: Prime numbers are greater than 1 and usually odd, except for 2, which is even. The boundaries for prime numbers are better defined by locating and considering counterexamples. Just in case you're wondering, 0 is also not considered a prime number, for the reasons discussed at `http://math.stackexchange.com/questions/539174/is-zero-a-prime-number`.

TIP

As the complexity of a problem grows, the potential for finding counterexamples grows as well. An essential rule to consider is that, as with reliability, having more failure points means greater potential for a failure to occur. Thinking of algorithms in this way is important. Ensembles of simple algorithms can produce better results with fewer potential counterexamples than a single complex algorithm.

## Standing on the shoulders of giants

A myth that defies explanation is that the techniques currently used to process huge quantities of data are somehow new. Yes, new algorithms do appear all the time, but the basis for these algorithms is all of the algorithms that have gone before. In fact, when you think about Sir Isaac Newton, you might think of someone who invented something new, yet even he stated (using correct spelling for his time), "If I have seen further it is by standing on the sholders of Giants" (see `https://en.wikiquote.org/wiki/Isaac_Newton` for additional quotes and insights).

The fact is that the algorithms you use today weren't even new in the days of Aristotle (see http://plato.stanford.edu/entries/aristotle-mathematics/ for a discussion of how Aristotle used math) and Plato (see http://www.storyofmathematics.com/greek_plato.html for a discussion of how Plato used math). The origins of algorithms in use today are so hidden in history that the best that anyone can say is that math relies on adaptations of knowledge from ancient times. The use of algorithms since antiquity should give you a certain feeling of comfort because the algorithms in use today are based on knowledge tested for thousands of years.

This isn't to say that some mathematicians haven't overturned the apple cart over the years. For example, John Nash's theory, Nash Equilibrium, significantly changed how economics are considered today (see https://www.khanacademy.org/economics-finance-domain/microeconomics/nash-equilibrium-tutorial for a basic tutorial on this theory). Of course, recognition for such work comes slowly (and sometimes not at all). Nash had to wait for a long time before he received much in the way of professional recognition (see the story at https://www.princeton.edu/main/news/archive/S42/72/29C63/index.xml) despite having won a Nobel Prize in economics for his contributions. Just in case you're interested, John Nash's story is depicted in the movie *A Beautiful Mind*, which contains some much-debated scenes, including one containing a claim that the Nash Equilibrium somehow overturns some of the work of Adam Smith, another contributor to economic theories. (See one such discussion at https://www.quora.com/Was-Adam-Smith-wrong-as-claimed-by-John-Nash-in-the-movie-A-Beautiful-Mind.)

## Dividing and Conquering

If solving problems were easy, everyone would do it. However, the world is still filled with unsolved problems and the condition isn't likely to change anytime soon, for one simple reason: Problems often appear so large that no solution is imaginable. Ancient warriors faced a similar problem. An opposing army would seem so large and their forces so small as to make the problem of winning a war unimaginably hard, perhaps impossible. Yet, by dividing the opposing army into small pieces and attacking it a little at a time, a small army could potentially defeat a much larger opponent. (The ancient Greeks, Romans, and Napoleon Bonaparte were all great users of the divide-and-conquer strategy; see *Napoleon For Dummies*, by J. David Markham [Wiley], for details.)

You face the same problem as those ancient warriors. Often, the resources at your disposal seem quite small and inadequate. However, by dividing a huge problem into small pieces so that you can understand each piece, you can eventually create a solution that works for the problem as a whole. Algorithms have this premise at their core: to use steps to solve problems one small piece at a time. The following sections help you understand the divide-and-conquer approach to problem solving in more detail.

## Avoiding brute-force solutions

A *brute-force solution* is one in which you try each possible answer, one at a time, to locate the best possible answer. It's thorough, this much is certain, but it also wastes time and resources in most cases. Testing every answer, even when it's easy to prove that a particular answer has no chance of success, wastes time that an algorithm can use on answers that have a better chance of success. In addition, testing the various answers using this approach generally wastes resources, such as memory. Think of it this way: You want to break the combination for a lock, so you begin at 0, 0, 0, even though you know that this particular combination has no chance of success given the physical characteristics of combination locks. A brute-force solution would proceed with testing 0, 0, 0 anyway and then move on to the equally ridiculous 0, 0, 1.

REMEMBER

It's important to understand that every solution type does come with advantages, sometimes quite small. A brute-force solution has one such advantage. Because you test every answer anyway, you don't need to perform any sort of preprocessing when working with a brute-force solution. The time saved in skipping the preprocessing, though, is unlikely to ever pay back the time lost in trying every answer. However, you may find occasion to use a brute-force solution when

- Finding a solution, if one exists, is essential.
- The problem size is limited.
- You can use heuristics to reduce the size of the solution set.
- Simplicity of implementation is more important than speed.

## Starting by making it simpler

The brute-force solution has a serious drawback. It looks at the entire problem at one time. It's sort of like going into a package and hunting book by book through the shelves without ever considering any method of making your search simpler. The divide-and-conquer approach to package searches is different. In this case, you begin by dividing the package into children's and adults' sections. After that, you divide the adults' section into categories. Finally, you search just the part of the category that contains the book of interest. This is the purpose of classification systems such as the Dewey Decimal System (see `https://en.wikipedia.org/wiki/List_of_Dewey_Decimal_classes` for a list of classes, hierarchical divisions, and sections). The point is that divide and conquer simplifies the problem. You make things faster and easier by reducing the number of book candidates.

The divide part of divide and conquer is an essential way to understand a problem better as well. Trying to understand the layout of an entire package could prove difficult. However, knowing that the book on comparative psychology you want to find appears as part of Class 100 in Division 150 of Section 156 makes your job easier. You can understand this smaller problem because you know that every Section 156 book will contain something about the topic you wish to know about. Algorithms work the same way. By making the problem simpler, you can create a set of simpler steps to finding a problem solution, which reduces the time to find the solution, reduces the number of resources used, and improves your chances of finding precisely the solution you need.

## Breaking down a problem is usually better

After you have divided a problem into manageable pieces, you need to conquer the piece in question. This means creating a precise problem definition. You don't want just any book on comparative psychology; you want one written by George Romanes. Knowing that the book you want appears in Section 156 of the Dewey Decimal System is a good start, but it doesn't solve the problem. Now you need a process for reviewing every book in Section 156 for the specific book you need. The process might go further still and look for books with specific content. To make this process viable, you must break the problem down completely, define precisely what you need, and then, after you understand the problem thoroughly, use the correct set of steps (algorithm) to find what you need.

### ALGORITHMS HAVE NO ABSOLUTES

You may think that you can create a scenario in which you can say that you always use a particular kind of algorithm to solve a particular kind of problem. However, this isn't the case. For example, you can find discussions of the relative merits of using brute-force techniques against certain problems as compared to divide and conquer. It shouldn't surprise you to discover that divide and conquer doesn't win in every situation. For example, when looking for the maximum value in an array, a brute-force approach can win the day when the array isn't sorted. You can read a discussion of this particular topic at `http://stackoverflow.com/questions/11043226/why-do-divide-and-conquer-algorithms-often-run-faster-than-brute-force`. The interesting thing is that the brute-force approach also uses fewer resources in this particular case. Always remember that rules have exceptions and knowing the exceptions can save you time and effort later.

# Learning that Greed Can Be Good

In some cases, you can't see the end of a solution process or even know whether you're winning the war. The only thing you can really do is to ensure that you win the individual battles to create a problem solution in hopes of also winning the war. A greedy method to problem solving uses this approach. It looks for an overall solution such that it chooses the best possible outcome at each problem solution stage.

REMEMBER

It seems that winning each battle would necessarily mean winning the war as well, but sometimes the real world doesn't work that way. A *Pyrrhic victory* is one in which someone wins every battle but ends up losing the war because the cost of the victory exceeds the gains of winning by such a large margin. You can read about five Pyrrhic victories at `http://www.history.com/news/history-lists/5-famous-Pyrrhic-victories`. The important lesson from these histories is that a greedy algorithm often does work, but not always, so you need to consider the best overall solution to a problem rather than become blinded by interim wins. The following sections describe how to avoid the Pyrrhic victory when working with algorithms.

## Applying greedy reasoning

Greedy reasoning is often used as part of an optimization process. The algorithm views the problem one step at a time and focuses just on the step at hand. Every greedy algorithm makes two assumptions:

- You can make a single optimal choice at a given step.
- By choosing the optimal selection at each step, you can find an optimal solution for the overall problem.

You can find many greedy algorithms, each optimized to perform particular tasks. Here are some common examples of greedy algorithms used for graph analysis (see Chapter 9 for more about graphs) and data compression (see Chapter 14 for more about data compression) and the reason you might want to use them:

- **Kruskal's Minimum Spanning Tree (MST):** This algorithm actually demonstrates one of the principles of greedy algorithms that people might not think about immediately. In this case, the algorithm chooses the edge between two nodes with the smallest value, not the greatest value as the word *greedy* might initially convey. This sort of algorithm might help you find the shortest path between two locations on a map or perform other graph-related tasks.
- **Prim's MST:** This algorithm splits an undirected graph (one in which direction isn't considered) in half. It then selects the edge that connects the two halves such that the total weight of the two halves is the smallest that it can be. You

might find this algorithm used in a maze game to locate the shortest distance between the start and the finish of the maze.

- **Huffman Encoding:** This algorithm is quite famous in computers because it forms the basis for many data-compression techniques. The algorithm assigns a code to every unique data entry in a stream of entries, such that the most commonly used data entry receives the shortest code. For example, the letter *E* would normally receive the shortest code when compressing English text, because you use it more often than any other letter in the alphabet. By changing the encoding technique, you can compress the text and make it considerably smaller, reducing transmission time.

## Reaching a good solution

Scientists and mathematicians use greedy algorithms so often that Chapter 15 covers them in depth. However, it's important to realize that what you really want is a good solution, not just a particular solution. In most cases, a good solution provides optimal results of the sort you can measure, but the word *good* can include many meanings, depending on the problem domain. You must ask what problem you want to solve and which solution solves the problem in a manner that best meets your needs. For example, when working in engineering, you might need to weigh solutions that consider weight, size, cost, or other considerations, or perhaps some combination of all these outputs that meet a specific requirement.

To put this issue into context, say that you build a coin machine that creates change for particular monetary amounts using the fewest coins possible (perhaps as part of an automatic checkout at a store). The reason to use the fewest coins possible is to reduce equipment wear, the weight of coins needed, and the time required to make change (your customers are always in a hurry, after all). A greedy solution solves the problem by using the largest coins possible. For example, to output $0.16 in change, you use a dime ($0.10), a nickel ($0.05), and a penny ($0.01).

REMEMBER

A problem occurs when you're unable to use every coin type in creating a solution. The change machine might be out of nickels, for example. To provide $0.40 in change, a greedy solution would start with a quarter ($0.25) and a dime ($0.10). Unfortunately, there are no nickels, so the coin machine then outputs five pennies (5 × $0.01) for a total of seven coins. The optimal solution in this case is to use four dimes instead (4 × $0.10). As a result, the greedy algorithm provides a particular solution, but not a good (optimal) solution in this case. The change-making problem receives considerable attention because it's so hard to solve. You can find additional information in discussions such as "Combinatorics of the Change-Making Problem," by Anna Adamaszeka and Michal Adamaszek (see `http://www.sciencedirect.com/science/article/pii/S0195669809001292` for details).

# Computing Costs and Following Heuristics

Even when you find a good solution, one that is both efficient and effective, you still need to know precisely what the solution costs. You may find that the cost of using a particular solution is still too high, even when everything else is considered. Perhaps the answer comes almost, but not quite, on time or it uses too many computing resources. The search for a good solution involves creating an environment in which you can fully test the algorithm, the states it creates, the operators it uses to change those states, and the time required to derive a solution.

Often, you find that a *heuristic approach,* one that relies on self-discovery and produces sufficiently useful results (not necessarily optimal, but good enough) is the method you actually need to solve a problem. Getting the algorithm to perform some of the required work for you saves time and effort because you can create algorithms that see patterns better than humans do. Consequently, self-discovery is the process of allowing the algorithm to show you a potentially useful path to a solution (but you must still count on human intuition and understanding to know whether the solution is the right one). The following sections describe techniques you can use to compute the cost of an algorithm using heuristics as a method of discovering the actual usefulness of any given solution.

## Representing the problem as a space

A *problem space* is an environment in which a search for a solution takes place. A set of states and the operators used to change those states represent the problem space. For example, consider a tile game that has eight tiles in a 3-x-3 frame. Each tile shows one part of a picture, and the tiles start in some random order so that the picture is scrambled. The goal is to move one tile at a time to place all the tiles in the right order and reveal the picture. You can see an example of this sort of puzzle at `http://mypuzzle.org/sliding`.

The combination of the start state, the randomized tiles, and the goal state — the tiles in a particular order — is the *problem instance.* You could represent the puzzle graphically using a *problem space graph.* Each node of the problem space graph presents a state (the eight tiles in a particular position). The edges represent operations, such as to move tile number eight up. When you move tile eight up, the picture changes — it moves to another state.

Winning the game by moving from the start state to the goal state isn't the only consideration. To solve the game efficiently, you need to perform the task in the least number of moves possible, which means using the smallest number of operators. The minimum number of moves used to solve the puzzle is the *problem depth.*

You must consider several factors when representing a problem as a space. For example, you must consider the maximum number of nodes that will fit in memory, which represents the *space complexity*. When you can't fit all the nodes in memory at one time, the computer must store some nodes in other locations, such as the hard drive, which can slow the algorithm considerably. To determine whether the nodes will fit in memory, you must consider the *time complexity*, which is the maximum number of nodes created to solve the problem. In addition, it's important to consider the *branching factor*, which is the average number of nodes created in the problem space graph to solve a problem.

## Going random and being blessed by luck

Solving a search problem using brute-force techniques (described in "Avoiding brute-force techniques," earlier in this chapter) is possible. The advantage of this approach is that you don't need any domain-specific knowledge to use one of these algorithms. A brute-force algorithm tends to use the simplest possible approach to solving the problem. The disadvantage is that a brute-force approach works well only for a small number of nodes. Here are some of the common brute-force search algorithms:

- **Breadth-first search:** This technique begins at the root node, explores each of the child nodes first, and only then moves down to the next level. It progresses level by level until it finds a solution. The disadvantage of this algorithm is that it must store every node in memory, which means that it uses a considerable amount of memory for a large number of nodes. This technique can check for duplicate nodes, which saves time, and it always comes up with a solution.
- **Depth-first search:** This technique begins at the root node and explores a set of connected child nodes until it reaches a leaf node. It progresses branch by branch until it finds a solution. The disadvantage of this algorithm is that it can't check for duplicate nodes, which means that it might traverse the same node paths more than once. In fact, this algorithm may not find a solution at all, which means that you must define a cutoff point to keep the algorithm from searching infinitely. An advantage of this approach is that it's memory efficient.
- **Bidirectional search:** This technique searches simultaneously from the root node and the goal node until the two search paths meet in the middle. An advantage of this approach is that it's time efficient because it finds the solution faster than many other brute-force solutions. In addition, it uses memory more efficiently than other approaches and always finds a solution. The main disadvantage is complexity of implementation, translating into a longer development cycle.

## Using a heuristic and a cost function

For some people, the word *heuristic* just sounds complicated. It would be just as easy to say that the algorithm makes an educated guess and then tries again when it fails. Unlike brute-force methods, heuristic algorithms learn. They also use cost functions to make better choices. Consequently, heuristic algorithms are more complex, but they have a distinct advantage in solving complex problems. As with brute-force algorithms, there are many heuristic algorithms and each comes with its own set of advantages, disadvantages, and special requirements. The following list describes a few of the most common heuristic algorithms:

- **Pure heuristic search:** The algorithm expands nodes in order of their cost. It maintains two lists. The closed list contains the nodes it has already explored; the open list contains the nodes it must yet explore. In each iteration, the algorithm expands the node with the lowest possible cost. All its child nodes are placed in the closed list and the individual child node costs are calculated. The algorithm sends the child nodes with a low cost back to the open list and deletes the child nodes with a high cost. Consequently, the algorithm performs an intelligent, cost-based search for the solution.
- **A * search:** The algorithm tracks the cost of nodes as it explores them using the equation: $f(n) = g(n) + h(n)$, where
  - n is the node identifier.
  - $g(n)$ is the cost of reaching the node so far.
  - $h(n)$ is the estimated cost to reach the goal from the node.
  - $f(n)$ is the estimated cost of the path from n to the goal.

  The idea is to search the most promising paths first and avoid expensive paths.
- **Greedy best-first search:** The algorithm always chooses the path that is closest to the goal using the equation: $f(n) = h(n)$. This particular algorithm can find solutions quite quickly, but it can also get stuck in loops, so many people don't consider it an optimal approach to finding a solution.

# Evaluating Algorithms

Gaining insights into precisely how algorithms work is important because otherwise you can't determine whether an algorithm actually performs in the way you need it to. In addition, without good measurements, you can't perform accurate comparisons to know whether you really do need to discover a new method of solving a problem when an older solution works too slowly or uses too many resources. The reality is that you'll use algorithms made by others most of the time, potentially devising a few of your own. Knowing the basis to use to compare different solutions and deciding between them is an essential skill when dealing with algorithms.

The issue of efficiency has been part of discovering and designing new algorithms since the concept of algorithms first came into being, which is why you see so many different algorithms competing to solve the same problem (sometimes a real embarrassment of riches). The concept of measuring the size of the functions within an algorithm and analyzing how the algorithm works isn't new; both Ada Lovelace and Charles Babbage considered the problems of algorithm efficiency in reference to computers as early as 1843 (see a short history of the Babbage engine at `http://www.computerhistory.org/babbage/adalovelace/`).

Donald Knuth (`http://www-cs-faculty.stanford.edu/~uno/`), computer scientist, mathematician, professor emeritus at Stanford University, and author of the milestone, multivolume book *The Art of Computer Programming* (Addison-Wesley), devoted much of his research and studies to comparing algorithms. He strived to formalize how to estimate the resource needs of algorithms in a mathematical way and to allow a correct comparison between alternative solutions. He coined the term *analysis of algorithms,* which is the branch of computer science devoted to understanding how algorithms work in a formal way. The analysis measures resources required in terms of the number of operations an algorithm requires to reach a solution or by its occupied space (such as the storage an algorithm requires in computer memory).

Analysis of algorithms requires some mathematical understanding and some computations, but it's extremely beneficial in your journey to discover, appreciate, and effectively use algorithms. This topic is considerably more abstract than other topics in this book. To make the discussion less theoretical, later chapters present more practicalities of such measurement by examining algorithms together in detail. The following sections provide you with the basics.

## Simulating using abstract machines

The more operations an algorithm requires, the more complex it is. Complexity is a measure of algorithm efficiency in terms of time usage because each operation takes some time. Given the same problem, complex algorithms are generally less favorable than simple algorithms because complex algorithms require more time. Think about those times when speed of execution makes the difference, such as in the medical or financial sector, or when flying on automatic pilot on an airplane or space rocket. Measuring algorithm complexity is a challenging task, though a necessary one if you want to employ the right solution. The first measurement technique uses abstract machines like the Random Access Machine (RAM).

REMEMBER

RAM also stands for Random-Access Memory, which is the internal memory that your computer uses when running programs. Even though it uses the same acronym, a Random-Access Machine is something completely different.

*Abstract machines* aren't real computers, but theoretical ones, computers that are imagined in their functioning. You use abstract machines to consider how well an algorithm would work on a computer without testing it on the real thing, yet bound by the type of hardware you'd use. A RAM computer performs basic arithmetic operations and interacts with information in memory, that's all. Every time a RAM computer does anything, it takes a time step (a time unit). When you evaluate an algorithm in a RAM simulation, you count time steps using the following procedure:

1. Count each simple operation (arithmetic ones) as a time step.
2. Break complex operations into simple arithmetic operations and count time steps as defined in Step 1.
3. Count every data access from memory as one time step.

To perform this accounting, you write a pseudocode version of your algorithm (as mentioned in Chapter 1) and perform these steps using paper and pencil. In the end, it's a simple approach based on a basic idea of how computers work, a useful approximation that you can use to compare solutions regardless of the power and speed of your hardware or the programming language you use.

REMEMBER

Using a simulation is different from running the algorithm on a computer because you use a standard and predefined input. Real computer measurements require that you run the code and verify the time required to run it. Running code on a computer is actually a *benchmark*, another form of efficiency measurement, in which you also account for the application environment (such as the type of hardware used and the software implementation). A benchmark is useful but lacks generalization. Consider, for instance, how newer hardware can quickly execute an algorithm that took ages on your previous computer.

## Getting even more abstract

Measuring a series of steps devised to achieve a solution to a problem poses quite a few challenges. The previous section discusses counting time steps (number of operations), but sometimes you also need to compute space (such as the memory an algorithm consumes). You consider space when your problem is greedy for resources. Depending on the problem, you may consider an algorithm better when it works efficiently with regard to one of these resource consumption aspects:

- Running time
- Computer memory requirements
- Hard-disk usage
- Power consumption
- Data-transmission speed in a network

Some of these aspects relate to others in an inverse manner, so if, for instance, you want speedier execution time, you can increase memory or power consumption to get it. Not only can you have different efficiency configurations when running an algorithm, you can also change the hardware characteristics and software implementation to accomplish your goals. In terms of hardware, using a supercomputer or a general-purpose computer does matter, and the software, or language used to write the algorithm, is definitely a game changer. In addition, the quantity and kind of data you feed the algorithm could result in better or worse performance measurements.

RAM simulations count time because when you can employ a solution in so many environments and its resource usage depends on so many factors, you have to find a way to simplify comparisons so that they become standard. Otherwise, you can't compare possible alternatives. The solution is, as so often happens with many other problems, to use a single measure and say that one size fits all. In this case, the measure is time, which you make equal to the number of operations, that is, the complexity of the algorithm.

A RAM simulation places the algorithm in a situation that's both language and machine agnostic (it's independent of programming language and computer type). However, explaining how a RAM simulation works to others requires quite an effort. The analysis of algorithms proposes to use the number of operations you get from a RAM simulation and turn them into a mathematical function expressing how your algorithm behaves in terms of time, which is a quantification of the steps or operations required when the number of data inputs grows. For instance, if your algorithm sorts objects, you can express complexity using a function that reports how many operations it needs depending on the number of objects it receives.

## Working with functions

A function in mathematics is simply a way to map some inputs to a response. Expressed in a different way, a *function* is a transformation (based on math operations) that transforms (maps) your input to an answer. For certain values of input (usually denoted by the letters *x* or *n*), you have a corresponding answer using the math that defines the function. For instance, a function like `f(n) = 2n` tells you that when your input is a number *n*, your answer is the number *n* multiplied by 2.

Using the size of the input does make sense given that this is a time-critical age and people's lives are crammed with a growing quantity of data. Making everything a mathematical function is a little less intuitive, but a function describing how an algorithm relates its solution to the quantity of data it receives is something you can analyze without specific hardware or software support. It's also easy to compare with other solutions, given the size of your problem. Analysis of algorithms is really a mind-blowing concept because it reduces a complex series of steps into a mathematical formula.

Moreover, most of the time, an analysis of algorithms isn't even interested in defining the function exactly. What you really want to do is compare a target function with another function. These comparison functions appear within a set of proposed functions that perform poorly when contrasted to the target algorithm. In this way, you don't have to plug numbers into functions of greater or lesser complexity; instead, you deal with simple, premade, and well-known functions. It may sound rough, but it's more effective and is similar to classifying the performance of algorithms into categories, rather than obtaining an exact performance measurement.

The set of generalized functions is called *Big O* notation, and in this book, you often encounter this small set of functions (put into parentheses and preceded by a capital O) used to represent the performance of algorithms. Figure 2-1 shows the analysis of an algorithm. A Cartesian coordinate system can represent its function as measured by RAM simulation, where the *abscissa* (the x coordinate) is the size of the input and the *ordinate* (the y coordinate) is its resulting number of operations. You can see three curves represented. Input size matters. However, quality also matters (for instance, when ordering problems, it's faster to order an input which is already almost ordered). Consequently, the analysis shows a worst case, $f_1(n)$, an average case, $f_2(n)$, and a best case, $f_3(n)$. Even though the average case might give you a general idea, what you really care about is the worst case, because problems may arise when your algorithm struggles to reach a solution. The Big O function is the one that, after a certain $n_0$ value (the threshold for considering an input big), always results in a larger number of operations given the same input than the worst-case function $f_1$. Thus, the Big O function is even more pessimistic than the one representing your algorithm, so that no matter the quality of input, you can be sure that things cannot get worse than that.

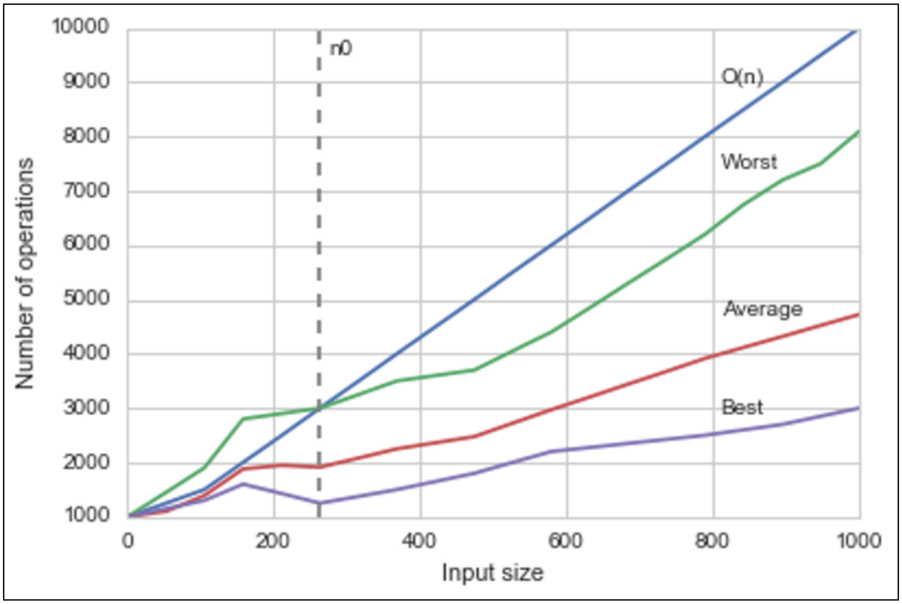


**FIGURE 2-1:** Complexity of an algorithm in case of best, average, and worst input case.

Many possible functions can result in worse results, but the choice of functions offered by the Big O notation that you can use is restricted because its purpose is to simplify complexity measurement by proposing a standard. Consequently, this section contains just the few functions that are part of the Big O notation. The following list describes them in growing order of complexity:

- **Constant complexity O(1):** The same time, no matter how much input you provide. In the end, it is a constant number of operations, no matter how long the input data is. This level of complexity is quite rare in practice.
- **Logarithmic complexity O(log n):** The number of operations grows at a slower rate than the input, making the algorithm less efficient with small inputs and more efficient with larger ones. A typical algorithm of this class is the binary search, as described in Chapter 7 on arranging and searching data.
- **Linear complexity O(n):** Operations grow with the input in a 1:1 ratio. A typical algorithm is iteration, which is when you scan input once and apply an operation to each element of it. Chapter 5 discusses iterations.
- **Linearithmic complexity O(n log n):** Complexity is a mix between logarithmic and linear complexity. It is typical of some smart algorithms used to order data, such as Mergesort, Heapsort, and Quicksort. Chapter 7 tells you about most of them.

- **Quadratic complexity O($n^2$):** Operations grow as a square of the number of inputs. When you have one iteration inside another iteration (nested iterations, in computer science), you have quadratic complexity. For instance, you have a list of names and, in order to find the most similar ones, you compare each name against all the other names. Some less efficient ordering algorithms present such complexity: bubble sort, selection sort, and insertion sort. This level of complexity means that your algorithms may run for hours or even days before reaching a solution.
- **Cubic complexity O($n^3$):** Operations grow even faster than quadratic complexity because now you have multiple nested iterations. When an algorithm has this order of complexity and you need to process a modest amount of data (100,000 elements), your algorithm may run for years. When you have a number of operations that is a power of the input, it is common to refer to the algorithm as *running in polynomial time.*
- **Exponential complexity O($2^n$):** The algorithm takes twice the number of previous operations for every new element added. When an algorithm has this complexity, even small problems may take forever. Many algorithms doing exhaustive searches have exponential complexity. However, the classic example for this level of complexity is the calculation of Fibonacci numbers (which, being a recursive algorithm, is dealt with in Chapter 5).
- **Factorial complexity O(n!):** A real nightmare of complexity because of the large number of possible combinations between the elements. Just imagine: If your input is 100 objects and an operation on your computer takes $10^{-6}$ seconds (a reasonable speed for every computer, nowadays), you will need about $10^{140}$ years to complete the task successfully (an impossible amount of time since the age of the universe is estimated as being $10^{14}$ years). A famous factorial complexity problem is the traveling salesman problem, in which a salesman has to find the shortest route for visiting many cities and coming back to the starting city (presented in Chapter 18).

IN THIS CHAPTER

- **Using Python to discover how algorithms work**
- **Considering the various Python distributions**
- **Performing a Python installation on Linux**
- **Performing a Python installation on OS X**
- **Performing a Python installation on Windows**
- **Obtaining and installing the datasets used in this book**

# Chapter 3 Using Python to Work with Algorithms

You have many good choices when it comes to using computer assistance to discover the wonders of algorithms. For example, apart from Python, many people rely on MATLAB and many others use R. In fact, some people use all three and then compare the sorts of outputs they get (see one such comparison at `https://www.r-bloggers.com/evaluating-optimization-algorithms-in-matlab-python-and-r/`). If you just had the three choices, you'd still need to think about them for a while and might choose to learn more than one language, but you actually have more than three choices, and this book can't begin to cover them all. If you get deep into the world of algorithms, you discover that you can use all programming languages to write algorithms and that some are appreciated because they boil everything down to simple operations, such as the RAM simulation described in Chapter 2. For instance, Donald Knuth, winner of the Turing Award, wrote examples in Assembly language in his book *The Art of Computer Programming* (Addison-Wesley). Assembly language is a programming language that resembles machine code, the language used natively by computers (but not understandable by most humans).

This book uses Python for a number of good reasons, including the community support it enjoys and the fact that it's full featured, yet easy to learn. Python is also a verbose language, resembling how a human creates instructions rather than how a computer interprets them. The first section of this chapter fills in the details of why this book uses Python for the examples, but also tells you why other options are useful and why you may need to consider them as your journey continues.

When you speak a human language, you add nuances of meaning by employing specific word combinations that others in your community understand. The use of nuanced meaning comes naturally and represents a dialect. In some cases, dialects also form because one group wants to demonstrate a difference with another group. For example, Noah Webster wrote and published *A Grammatical Institute of the English Language,* in part to remove the influence of the British aristocracy from the American public (see `http://connecticuthistory.org/noah-webster-and-the-dream-of-a-common-language/` for details). Likewise, computer languages often come with flavors, and vendors purposely add extensions that make their product unique to provide a reason to buy their product over another offering.

The second section of the chapter introduces you to various Python distributions, each of which provides a Python dialect. This book uses Analytics Anaconda, which is the product you should use to obtain the best results from your learning experience. Using another product, essentially another dialect, can cause problems in making the examples work — the same sort of thing that happens sometimes when someone who speaks British English talks to someone who speaks American English. However, knowing about other distributions can be helpful when you need to obtain access to features that Anaconda may not provide.

The next three sections of this chapter help you install Anaconda on your platform. The examples in this book are tested on the Linux, Mac OS X, and Windows platforms. They may also work with other platforms, but the examples aren't tested on these platforms, so you have no guarantee that they'll work. By installing Anaconda using the procedures found in this chapter, you reduce the chance of getting an installation that won't work with the example code. To use the examples in this book, you must install Anaconda 4.2.0 with support for Python 3.5. Other versions of Anaconda and Python may not work with the example code because, as with human language dialects, they could misunderstand the instructions that the code provides.

Algorithms work with data in specific ways. To see particular output from an algorithm, you need consistent data. Fortunately, the Python community is busy creating datasets that anyone can use for testing purposes. This allows the community to repeat results that others get without having to download custom datasets from an unknown source. The final section of this chapter helps you get and install the datasets needed for the examples.

## Considering the Benefits of Python

To work with algorithms on a computer, you need some means of communicating with the computer. If this were *Star Trek*, you could probably just tell the computer what you want and it would dutifully perform the task for you. In fact, Scotty seems quite confused about the lack of a voice computer interface in *Star Trek IV* (see `http://www.davidalison.com/2008/07/keyboard-vs-mouse.html` for details). The point is that you still need to use the mouse and keyboard, along with a special language, to communicate your ideas to the computer because the computer isn't going to make an effort to communicate with you. Python is one of a number of languages that is especially adept at making it easy for nondevelopers to communicate ideas to the computer, but it isn't the only choice. The following paragraphs help you understand why this book uses Python and what your other choices are.

### Understanding why this book uses Python

Every computer language available today translates algorithms into a form that the computer can process. In fact, languages like ALGOL (ALGOrithmic Language) and FORTRAN (FORmula TRANslation) make this focus clear. Remember the definition of an algorithm from Chapter 1 as being a sequence of steps used to solve a problem. The method used to perform this translation differs by language, and the techniques used by some languages are quite arcane, requiring specialized knowledge even to make an attempt.

REMEMBER

Computers speak only one language, *machine code* (the 0s and 1s that a computer interprets to perform tasks), which is so incredibly hard for humans to speak that early developers created a huge array of alternatives. Computer languages exist to make human communication with computers easier. Consequently, if you find yourself struggling to make anything work, perhaps you have the wrong language. It's always best to have more than one language at your fingertips so that you can perform computer communications with ease. Python happens to be one of the languages that works exceptionally well for people who work in disciplines outside application development.

Python is the vision of a single person, Guido van Rossum (see his home page at `https://gvanrossum.github.io/`). You might be surprised to learn that Python has been around for a long time — Guido started the language in December 1989 as a replacement for the ABC language. Not much information is available as to the precise goals for Python, but it does retain ABC's capability to create applications using less code. However, it far exceeds the capability of ABC to create applications of all types, and in contrast to ABC, boasts four programming styles. In short, Guido took ABC as a starting point, found it limited, and created a new language without those limitations. It's an example of creating a new language that really is better than its predecessor.

Python has gone through a number of iterations and currently has two development paths. The 2.*x* path is backward compatible with previous versions of Python; the 3.*x* path isn't. The compatibility issue is one that figures into how you use Python to perform algorithm-related tasks because at least some of the packages won't work with 3.*x*. In addition, some versions use different licensing because Guido was working at various companies during Python's development. You can see a listing of the versions and their respective licenses at `https://docs.python.org/3/license.html`. The Python Software Foundation (PSF) owns all current versions of Python, so unless you use an older version, you really don't need to worry about the licensing issue.

TECHNICAL STUFF

Guido actually started Python as a *skunkworks project* (a project developed by a small and loosely structured group of people). The core concept was to create Python as quickly as possible, yet create a language that is flexible, runs on any platform, and provides significant potential for extension. Python provides all these features and many more. Of course, there are always bumps in the road, such as figuring out just how much of the underlying system to expose. You can read more about the Python design philosophy at `http://python-history.blogspot.com/2009/01/pythons-design-philosophy.html`. The history of Python at `http://python-history.blogspot.com/2009/01/introduction-and-overview.html` also provides some useful information.

The original development (or design) goals for Python don't quite match what has happened to the language since that time. Guido originally intended Python as a second language for developers who needed to create one-off code but who couldn't quite achieve their goals using a scripting language. The original target audience for Python was the C developer. You can read about these original goals in the interview at `http://www.artima.com/intv/pyscale.html`.

You can find a number of applications written in Python today, so the idea of using it solely for scripting didn't come to fruition. In fact, you can find listings of Python applications at `https://www.python.org/about/apps/` and `https://www.python.org/about/success/`.

Naturally, with all these success stories to go on, people are enthusiastic about adding to Python. You can find lists of Python Enhancement Proposals (PEPs) at `http://legacy.python.org/dev/peps/`. These PEPs may or may not see the light of day, but they prove that Python is a living, growing language that will continue to provide features that developers truly need to create great applications of all types.

## Working with MATLAB

Python has advantages over many other languages by offering multiple coding styles, fantastic flexibility, and great extensibility, but it's still a programming language. If you honestly don't want to use a programming language, you do have other options, such as MATLAB (`https://www.mathworks.com/products/matlab/`), which focuses more on algorithms. MATLAB is still a scripting language of a sort, and to perform any significant tasks with it, you still need to know a little about coding, but not as much as with Python.

One of the major issues with using MATLAB is the price you pay. Unlike Python, MATLAB requires a monetary investment on your part (see `https://www.mathworks.com/pricing-licensing/` for licensing costs). The environment is indeed easier to use, but as with most things, there is no free lunch, and you must consider the cost differential as part of determining which product to use.

Many people are curious about MATLAB, that is, its strengths and weaknesses when compared to Python. This book doesn't have room to provide a full comparison, but you can find a great overview at `http://www.pyzo.org/python_vs_matlab.html`. In addition, you can call Python packages from MATLAB using the techniques found at `https://www.mathworks.com/help/matlab/call-python-libraries.html`. In fact, MATLAB also works with the following:

- MEX (`https://www.mathworks.com/help/matlab/call-mex-file-functions.html`)
- C (`https://www.mathworks.com/help/matlab/using-c-shared-library-functions-in-matlab-.html`)
- Java (`https://www.mathworks.com/help/matlab/using-java-libraries-in-matlab.html`)
- .NET (`https://www.mathworks.com/help/matlab/using-net-libraries-in-matlab.html`)
- COM (`https://www.mathworks.com/help/matlab/using-com-objects-in-matlab.html`)

Therefore, you don't necessarily have to choose between MATLAB and Python (or other language), but the more Python features you use, the easier it becomes to simply work with Python and skip MATLAB. You can discover more about MATLAB in *MATLAB For Dummies*, by Jim Sizemore and John Paul Mueller (Wiley).

## Considering other algorithm testing environments

A third major contender for algorithm-related work is R. The R programming language, like Python, is free of charge. It also supports a large number of packages and offers great flexibility. Some of the programming constructs are different, however, and some people find R harder to use than Python. Most people view R as the winner when it comes to performing statistics, but they see the general-purpose nature of Python as having major benefits (see the articles at `https://www.datacamp.com/community/tutorials/r-or-python-for-data-analysis` and `http://www.kdnuggets.com/2015/05/r-vs-python-data-science.html`). The stronger community support for Python is also a major advantage.

As previously mentioned, you can use any computer programming language to perform algorithm-related work, but most languages have a specific purpose in mind. For example, you can perform algorithm-related tasks using a language such as Structured Query Language (SQL), but this language focuses on data management, so some algorithm-related tasks might become convoluted and difficult to perform. A significant lack in SQL is the ability to plot data with ease and to perform some of the translations and transformations that algorithm-specific work requires. In short, you need to consider what you plan to do when choosing a language. This book uses Python because it truly is the best overall language to perform the tasks at hand, but it's important to realize that you may need another language at some point.

# Looking at the Python Distributions

You can quite possibly obtain a generic copy of Python and add all the packages required to work with algorithms to it. The process can be difficult because you need to ensure that you have all the required packages in the correct versions to guarantee success. In addition, you need to perform the configuration required to make sure that the packages are accessible when you need them. Fortunately, going through the required work is not necessary because numerous Python products that work well with algorithms are available for you to use. These products provide everything needed to get started with algorithm-related projects.

REMEMBER

You can use any of the packages mentioned in the following sections to work with the examples in this book. However, the book's source code and downloadable source code rely on Continuum Analytics Anaconda 4.2.0 because this particular package works on every platform this book supports: Linux, Mac OS X, and Windows. The book doesn't mention a specific package in the chapters that follow, but any screenshots reflect how things look when using Anaconda on Windows. You may need to tweak the code to use another package, and the screens will look different if you use Anaconda on some other platform.

WARNING

Windows 10 presents some serious installation issues when working with Python. You can read about these issues on my (John's) blog at `http://blog.johnmuellerbooks.com/2015/10/30/python-and-windows-10/`. Given that so many readers of my other Python books have sent feedback saying that Windows 10 doesn't provide a good environment, I can't recommend Windows 10 as a Python platform for this book. If you're working with Windows 10, simply be aware that your road to a Python installation will be a rocky one.

## Obtaining Analytics Anaconda

The basic Anaconda package is a free download that you obtain at `https://store.continuum.io/cshop/anaconda/`. Simply click Download Anaconda to obtain access to the free product. You do need to provide an email address to get a copy of Anaconda. After you provide your email address, you go to another page, where you can choose your platform and the installer for that platform. Anaconda supports the following platforms:

- Windows 32-bit and 64-bit (the installer may offer you only the 64-bit or 32-bit version, depending on which version of Windows it detects)
- Linux 32-bit and 64-bit
- Mac OS X 64-bit

Because package support for Python 3.5 has gotten better than previous 3.*x* versions, you see both Python 3.*x* and 2.*x* equally supported on the Analytics site. This book uses Python 3.5 because the package support is now substantial enough and stable enough to support all the programming examples, and because Python 3.*x* represents the future direction of Python.

TIP

You can obtain Anaconda with older versions of Python. If you want to use an older version of Python, click the installer archive link near the bottom of the page. You should use an older version of Python only when you have a pressing need to do so.

The Miniconda installer can potentially save time by limiting the number of features you install. However, trying to figure out precisely which packages you do need is an error-prone and time-consuming process. In general, you want to perform a full installation to ensure that you have everything needed for your projects. Even a full install doesn't require much time or effort to download and install on most systems.

The free product is all you need for this book. However, when you look on the site, you see that many other add-on products are available. These products can help you create robust applications. For example, when you add Accelerate to the mix, you obtain the capability to perform multicore and GPU-enabled operations. The use of these add-on products is outside the scope of this book, but the Anaconda site provides details on using them.

## Considering Enthought Canopy Express

Enthought Canopy Express is a free product for producing both technical and scientific applications using Python. You can obtain it at `https://www.enthought.com/canopy-express/`. Click Download Free on the main page to see a listing of the versions that you can download. Only Canopy Express is free; the full Canopy product comes at a cost. However, you can use Canopy Express to work with the examples in this book. Canopy Express supports the following platforms:

- Windows 32-bit and 64-bit
- Linux 32-bit and 64-bit
- Mac OS X 32-bit and 64-bit

Choose the platform and version you want to download. When you click Download Canopy Express, you see an optional form for providing information about yourself. The download starts automatically, even if you don't provide personal information to the company.

One of the advantages of Canopy Express is that Enthought is heavily involved in providing support for both students and teachers. People also can take classes, including online classes, that teach the use of Canopy Express in various ways (see `https://training.enthought.com/courses`).

## Considering pythonxy

The pythonxy Integrated Development Environment (IDE) is a community project hosted on Google at `http://python-xy.github.io/`. It's a Windows-only product, so you can't easily use it for cross-platform needs. (In fact, it supports only

Windows Vista, Windows 7, and Windows 8.) However, it does come with a full set of packages, and you can easily use it for this book if you want.

Because pythonxy uses the GNU General Public License (GPL) v3 (see `http://www.gnu.org/licenses/gpl.html`), you have no add-ons, training, or other paid features to worry about. No one will come calling at your door hoping to sell you something. In addition, you have access to all the source code for pythonxy, so you can make modifications if you want.

## Considering WinPython

The name tells you that WinPython is a Windows-only product that you can find at `http://winpython.sourceforge.net/`. This product is actually a spin-off of pythonxy and isn't meant to replace it. Quite the contrary: WinPython is simply a more flexible way to work with pythonxy. You can read about the motivation for creating WinPython at `http://sourceforge.net/p/winpython/wiki/Roadmap/`.

The bottom line for this product is that you gain flexibility at the cost of friendliness and a little platform integration. However, for developers who need to maintain multiple versions of an IDE, WinPython may make a significant difference. When using WinPython with this book, make sure to pay particular attention to configuration issues or you'll find that even the downloadable code has little chance of working.

# Installing Python on Linux

You use the command line to install Anaconda on Linux — you're given no graphical installation option. Before you can perform the install, you must download a copy of the Linux software from the Continuum Analytics site. You can find the required download information in the "Obtaining Analytics Anaconda" section, earlier in this chapter. The following procedure should work fine on any Linux system, whether you use the 32-bit or 64-bit version of Anaconda:

1. **Open a copy of Terminal.**

   The Terminal window appears.

2. **Change directories to the downloaded copy of Anaconda on your system.**

   The name of this file varies, but normally it appears as `Anaconda3-4.2.0-Linux-x86.sh` for 32-bit systems and `Anaconda3-4.2.0-Linux-x86_64.sh` for 64-bit systems. The version number is embedded as part of the filename. In this case, the filename refers to version 4.2.0, which is the version used for this book. If you use some other version, you may experience problems with the source code and need to make adjustments when working with it.

3. **Type** bash Anaconda3-4.2.0-Linux-x86.sh **(for the 32-bit version) or** bash Anaconda3-4.2.0-Linux-x86_64.sh **(for the 64-bit version) and press Enter.**

   An installation wizard starts that asks you to accept the licensing terms for using Anaconda.

4. **Read the licensing agreement and accept the terms using the method required for your version of Linux.**

   The wizard asks you to provide an installation location for Anaconda. The book assumes that you use the default location of ~/anaconda. If you choose some other location, you may have to modify some procedures later in the book to work with your setup.

5. **Provide an installation location (if necessary) and press Enter (or click Next).**

   The application extraction process begins. After the extraction is complete, you see a completion message.

6. **Add the installation path to your `PATH` statement using the method required for your version of Linux.**

   You're ready to begin using Anaconda.

# Installing Python on MacOS

The Mac OS X installation comes in only one form: 64-bit. Before you can perform the install, you must download a copy of the Mac software from the Continuum Analytics site. You can find the required download information in the "Obtaining Analytics Anaconda" section, earlier in this chapter.

The installation files come in two forms. The first depends on a graphical installer; the second relies on the command line. The command-line version works much like the Linux version described in the "Installing Python on Linux" section of this chapter. The following steps help you install Anaconda 64-bit on a Mac system using the graphical installer:

1. **Locate the downloaded copy of Anaconda on your system.**

   The name of this file varies, but normally it appears as `Anaconda3-4.2.0-MacOSX-x86_64.pkg`. The version number is embedded as part of the filename. In this case, the filename refers to version 4.2.0, which is the version used for this book. If you use some other version, you may experience problems with the source code and need to make adjustments when working with it.

2. **Double-click the installation file.**

   An introduction dialog box appears.

3. **Click Continue.**

   The wizard asks whether you want to review the Read Me materials. You can read these materials later. For now, you can safely skip the information.

4. **Click Continue.**

   The wizard displays a licensing agreement. Be sure to read through the licensing agreement so that you know the terms of usage.

5. **Click I Agree if you agree to the licensing agreement.**

   The wizard asks you to provide a destination for the installation. The destination controls whether the installation is for an individual user or a group.

   WARNING

   You may see an error message stating that you can't install Anaconda on the system. The error message occurs because of a bug in the installer and has nothing to do with your system. To get rid of the error message, choose the Install Only for Me option. You can't install Anaconda for a group of users on a Mac system.

6. **Click Continue.**

   The installer displays a dialog box containing options for changing the installation type. Click Change Install Location if you want to modify where Anaconda is installed on your system. (The book assumes that you use the default path of ~/anaconda.) Click Customize if you want to modify how the installer works. For example, you can choose not to add Anaconda to your `PATH` statement. However, the book assumes that you have chosen the default install options, and no good reason exists to change them unless you have another copy of Python 3.5 installed somewhere else.

7. **Click Install.**

   The installation begins. A progress bar tells you how the installation process is progressing. When the installation is complete, you see a completion dialog box.

8. **Click Continue.**

   You're ready to begin using Anaconda.

# Installing Python on Windows

Anaconda comes with a graphical installation application for Windows, so getting a good install means using a wizard, as you would for any other installation. Of course, you need a copy of the installation file before you begin, and you can find the required download information in the "Obtaining Analytics Anaconda" section, earlier in this chapter. The following procedure should work fine on any Windows system, whether you use the 32-bit or the 64-bit version of Anaconda:

1. **Locate the downloaded copy of Anaconda on your system.**

   The name of this file varies, but normally it appears as `Anaconda3-4.2.0-Windows-x86.exe` for 32-bit systems and `Anaconda3-4.2.0-Windows-x86_64.exe` for 64-bit systems. The version number is embedded as part of the filename. In this case, the filename refers to version 4.2.0, which is the version used for this book. If you use some other version, you may experience problems with the source code and need to make adjustments when working with it.

2. **Double-click the installation file.**

   (You may see an Open File – Security Warning dialog box that asks whether you want to run this file. Click Run if you see this dialog box pop up.) You see an Anaconda 4.2.0 Setup dialog box similar to the one shown in Figure 3-1. The exact dialog box that you see depends on which version of the Anaconda installation program you download. If you have a 64-bit operating system, using the 64-bit version of Anaconda is always best so that you obtain the best possible performance. This first dialog box tells you when you have the 64-bit version of the product.

3. **Click Next.**

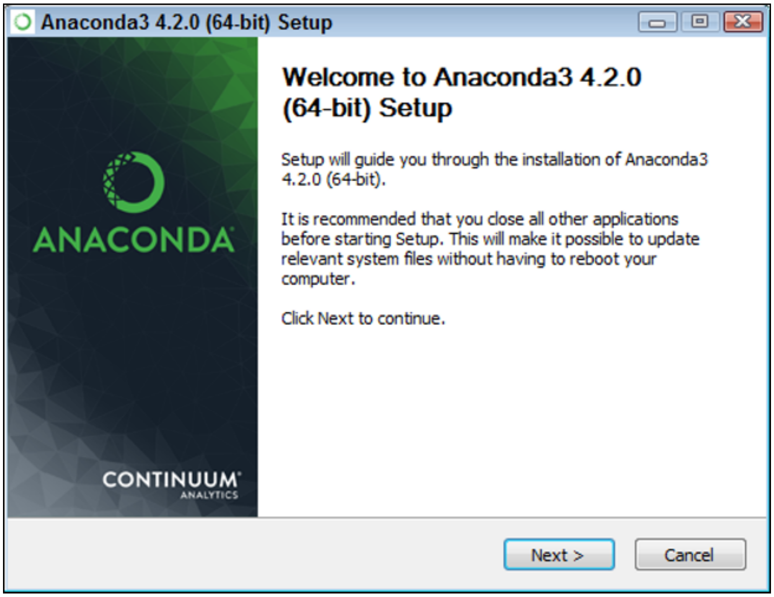
   The wizard displays a licensing agreement. Be sure to read through the licensing agreement so that you know the terms of usage.

4. **Click I Agree if you agree to the licensing agreement.**

   You're asked what sort of installation type to perform, as shown in Figure 3-2. In most cases, you want to install the product just for yourself. The exception is if you have multiple people using your system and they all need access to Anaconda.

5. **Choose one of the installation types and then click Next.**

   The wizard asks where to install Anaconda on disk, as shown in Figure 3-3. The book assumes that you use the default location. If you choose some other location, you may have to modify some procedures later in the book to work with your setup.



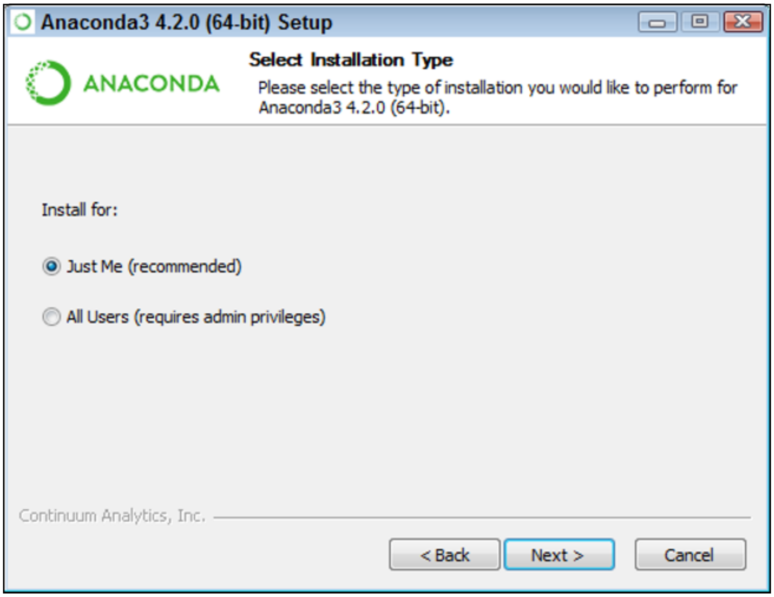
Welcome to Anaconda3 4.2.0 (64-bit) Setup

Setup will guide you through the installation of Anaconda3 4.2.0 (64-bit).

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

**FIGURE 3-1:** The setup process begins by telling you whether you have the 64-bit version.



Select Installation Type

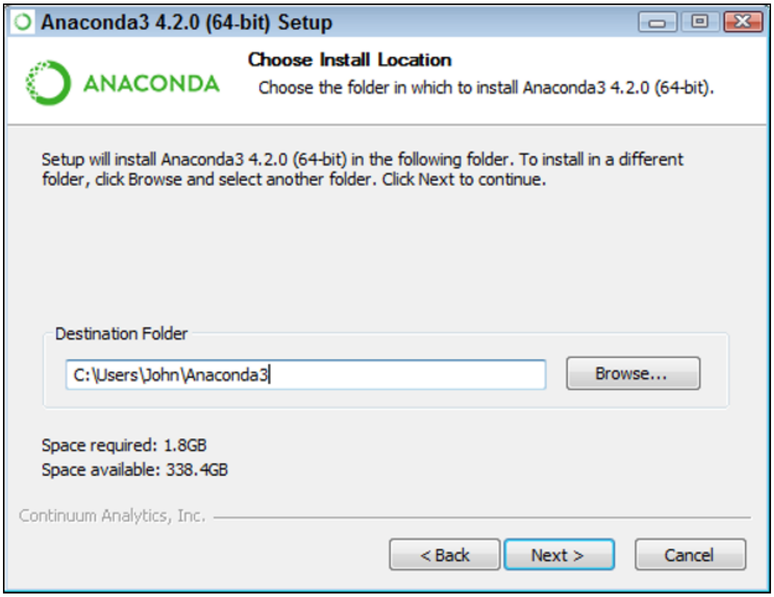
Please select the type of installation you would like to perform for Anaconda3 4.2.0 (64-bit).

Install for:

● Just Me (recommended)

○ All Users (requires admin privileges)

**FIGURE 3-2:** Tell the wizard how to install Anaconda on your system.


Anaconda3 4.2.0 (64-bit) Setup

Choose Install Location

Choose the folder in which to install Anaconda3 4.2.0 (64-bit).

Setup will install Anaconda3 4.2.0 (64-bit) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

Destination Folder

C:\Users\John\Anaconda3

Space required: 1.8GB
Space available: 338.4GB

**FIGURE 3-3:** Specify an installation location.

6. **Choose an installation location (if necessary) and then click Next.**

   You see the Advanced Installation Options, shown in Figure 3-4. These options are selected by default, and no good reason exists to change them in most cases. You might need to change them if Anaconda won't provide your default Python 3.5 (or Python 2.7) setup. However, the book assumes that you've set up Anaconda using the default options.

7. **Change the advanced installation options (if necessary) and then click Install.**

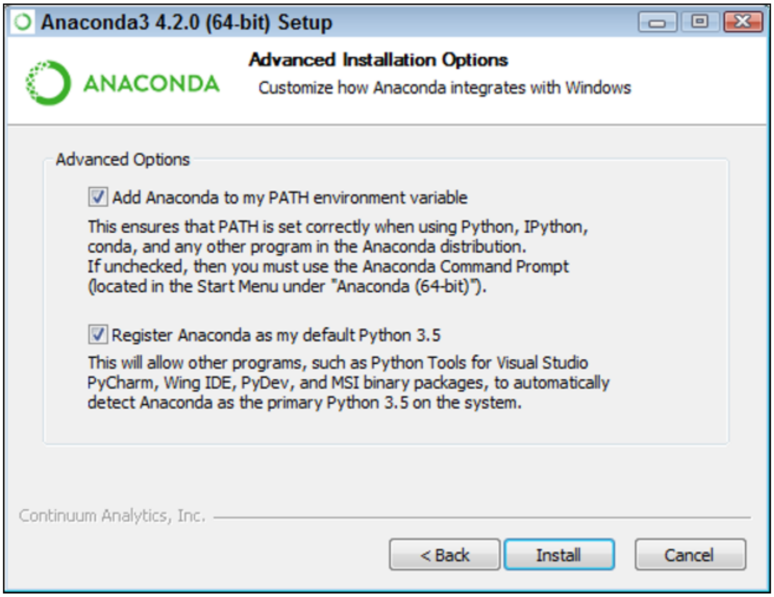
   You see an Installing dialog box with a progress bar. The installation process can take a few minutes, so get yourself a cup of coffee and read the comics for a while. When the installation process is over, you see a Next button enabled.

8. **Click Next.**

   The wizard tells you that the installation is complete.

9. **Click Finish.**

   You're ready to begin using Anaconda.


Anaconda3 4.2.0 (64-bit) Setup
ANACONDA
Advanced Installation Options
Customize how Anaconda integrates with Windows
Advanced Options

☑ Add Anaconda to my PATH environment variable

This ensures that PATH is set correctly when using Python, IPython, conda, and any other program in the Anaconda distribution. If unchecked, then you must use the Anaconda Command Prompt (located in the Start Menu under "Anaconda (64-bit)").

☑ Register Anaconda as my default Python 3.5

This will allow other programs, such as Python Tools for Visual Studio PyCharm, Wing IDE, PyDev, and MSI binary packages, to automatically detect Anaconda as the primary Python 3.5 on the system.

**FIGURE 3-4:** Configure the advanced installation options.

## A WORD ABOUT THE SCREENSHOTS

As you work your way through the book, you use an IDE of your choice to open the Python and Jupyter Notebook files containing the book's source code. Every screenshot that contains IDE-specific information relies on Anaconda because Anaconda runs on all three platforms supported by the book. The use of Anaconda doesn't imply that it's the best IDE or that the authors are making any sort of recommendation for it; Anaconda simply works well as a demonstration product.

When you work with Anaconda, the name of the graphical (GUI) environment, Jupyter Notebook, is precisely the same across all three platforms, and you won't even see any significant difference in the presentation. (Jupyter Notebook is an evolution of IPython, so you may see online resources refer to IPython Notebook.) The differences that you do see are minor, and you should ignore them as you work through the book. With this in mind, the book does rely heavily on Windows 7 screenshots. When working on a Linux, Mac OS X, or other Windows version platform, you should expect to see some differences in presentation, but these differences shouldn't reduce your ability to work with the examples.

# Downloading the Datasets and Example Code

This book is about using Python to perform machine learning tasks. Of course, you can spend all your time creating the example code from scratch, debugging it, and only then discovering how it relates to machine learning, or you can take the easy way and download the prewritten code from the Dummies site (see the Introduction of this book for details) so that you can get right to work. Likewise, creating datasets large enough for algorithm learning purposes would take quite a while. Fortunately, you can access standardized, precreated data sets quite easily by using features provided in some of the data science packages (which also work just fine for all sorts of purposes, including learning to work with algorithms). The following sections help you download and use the example code and datasets so that you can save time and get right to work with algorithm-specific tasks.

## Using Jupyter Notebook

To make working with the relatively complex code in this book easier, you use Jupyter Notebook. This interface lets you easily create Python notebook files that can contain any number of examples, each of which can run individually. The program runs in your browser, so which platform you use for development doesn't matter; as long as it has a browser, you should be okay.

### Starting Jupyter Notebook

Most platforms provide an icon to access Jupyter Notebook. Just click this icon to access Jupyter Notebook. For example, on a Windows system, you choose Start ➪ All Programs ➪ Anaconda 3 ➪ Jupyter Notebook. Figure 3-5 shows how the interface looks when viewed in a Firefox browser. The precise appearance on your system depends on the browser you use and the kind of platform you have installed.

If you have a platform that doesn't offer easy access through an icon, you can use these steps to access Jupyter Notebook:

1. **Open a Command Prompt or Terminal Window on your system.**

   The window opens so that you can type commands.

2. **Change directories to the `\Anaconda3\Scripts` directory on your machine.**

   Most systems let you use the `CD` command for this task.

3. **Type** python jupyter-notebook-script.py **and press Enter.**

   The Jupyter Notebook page opens in your browser.

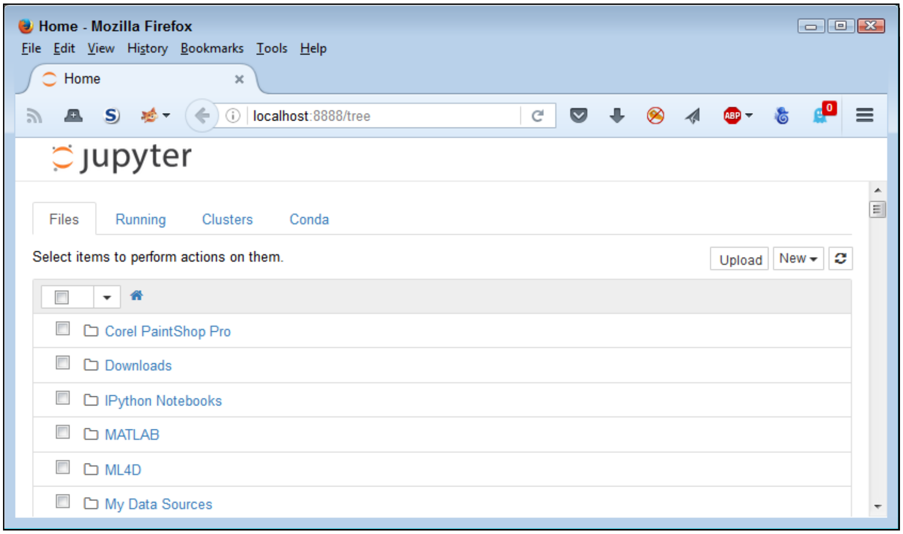


**FIGURE 3-5:** Jupyter Notebook provides an easy method to create machine learning examples.

## Stopping the Jupyter Notebook server

No matter how you start Jupyter Notebook (or just Notebook, as it appears in the remainder of the book), the system generally opens a command prompt or terminal window to host Jupyter Notebook. This window contains a server that makes the application work. After you close the browser window when a session is complete, select the server window and press Ctrl+C or Ctrl+Break to stop the server.

# Defining the code repository

The code you create and use in this book will reside in a repository on your hard drive. Think of a *repository* as a kind of filing cabinet where you put your code. Notebook opens a drawer, takes out the folder, and shows the code to you. You can modify it, run individual examples within the folder, add new examples, and simply interact with your code in a natural manner. The following sections get you started with Notebook so that you can see how this whole repository concept works.

## Defining the book's folder

It pays to organize your files so that you can access them easier later. This book keeps its files in the A4D (*Algorithms For Dummies*) folder. Use these steps within Notebook to create a new folder.

1. **Choose New ➪ Folder.**

   Notebook creates a new folder named Untitled Folder, as shown in Figure 3-6. The file appears in alphanumeric order, so you may not initially see it. You must scroll down to the correct location.

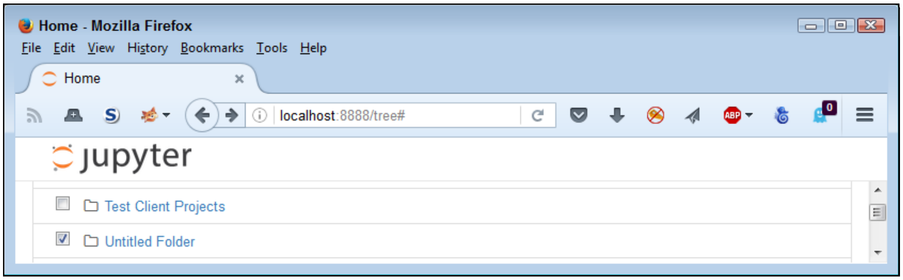
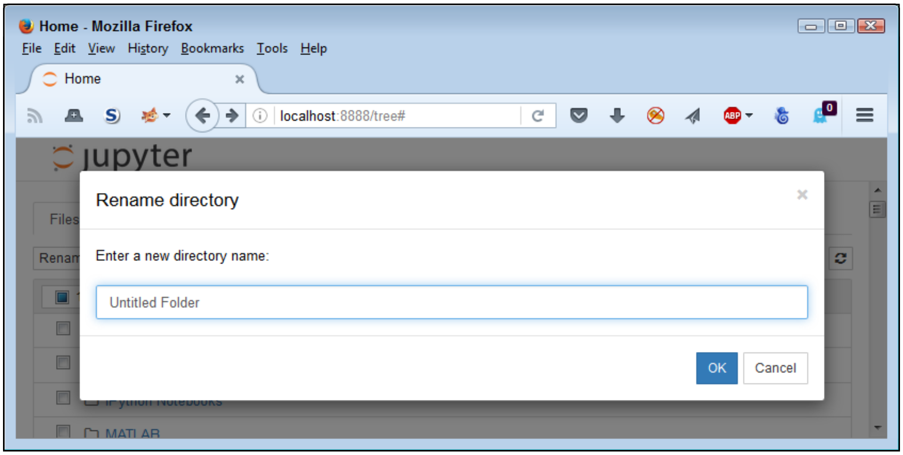


**FIGURE 3-6:** New folders appear with a name of Untitled Folder.

2. **Select the box next to the Untitled Folder entry.**
3. **Click Rename at the top of the page.**

   You see a Rename Directory dialog box like the one shown in Figure 3-7.



Rename directory

Enter a new directory name:

Untitled Folder

OK Cancel

**FIGURE 3-7:** Rename the folder so that you remember the kinds of entries it contains.

4. **Type** A4D **and click OK.**

   Notebook changes the name of the folder for you.

5. **Click the new A4D entry in the list.**

   Notebook changes the location to the A4D folder in which you perform tasks related to the exercises in this book.

## Creating a new notebook

Every new notebook is like a file folder. You can place individual examples within the file folder, just as you would sheets of paper into a physical file folder. Each example appears in a cell. You can put other sorts of things in the file folder, too, but you see how these things work as the book progresses. Use these steps to create a new notebook:

1. **Click New ➪ Python (default).**

   A new tab opens in the browser with the new notebook, as shown in Figure 3-8. Notice that the notebook contains a cell and that Notebook has highlighted the cell so that you can begin typing code in it. The title of the notebook is Untitled right now. That's not a particularly helpful title, so you need to change it.

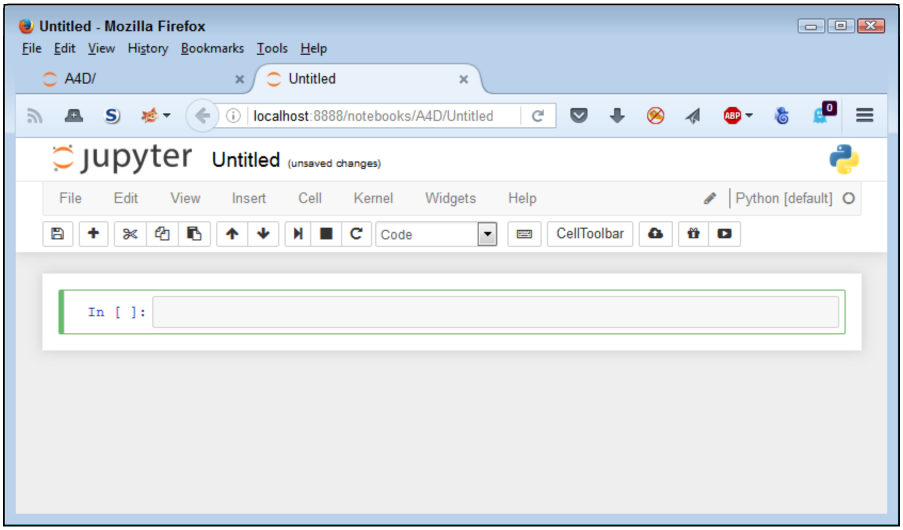

Untitled - Mozilla Firefox
File Edit View History Bookmarks Tools Help
A4D/
Untitled
localhost:8888/notebooks/A4D/Untitled
jupyter Untitled (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help
Python [default]
Code
CellToolbar
In [ ]:

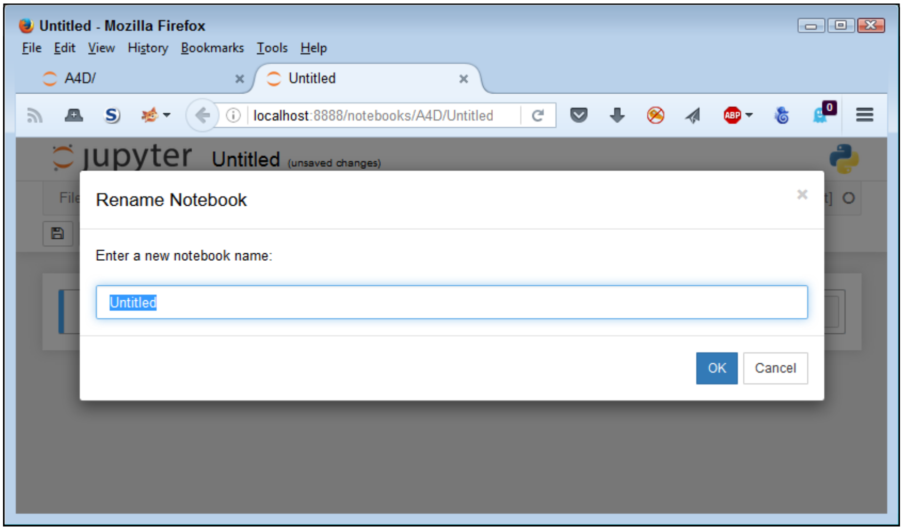
**FIGURE 3-8:** A notebook contains cells that you use to hold code.

2. **Click Untitled on the page.**

   Notebook asks what you want to use as a new name, as shown in Figure 3-9.

3. **Type** A4D; 03; Sample **and press Enter.**

   The new name tells you that this is a file for *Algorithms For Dummies,* Chapter 3, Sample.ipynb. Using this naming convention lets you easily differentiate these files from other files in your repository.


Untitled - Mozilla Firefox
File Edit View History Bookmarks Tools Help
A4D/
Untitled
localhost:8888/notebooks/A4D/Untitled
jupyter Untitled (unsaved changes)
OK
Cancel

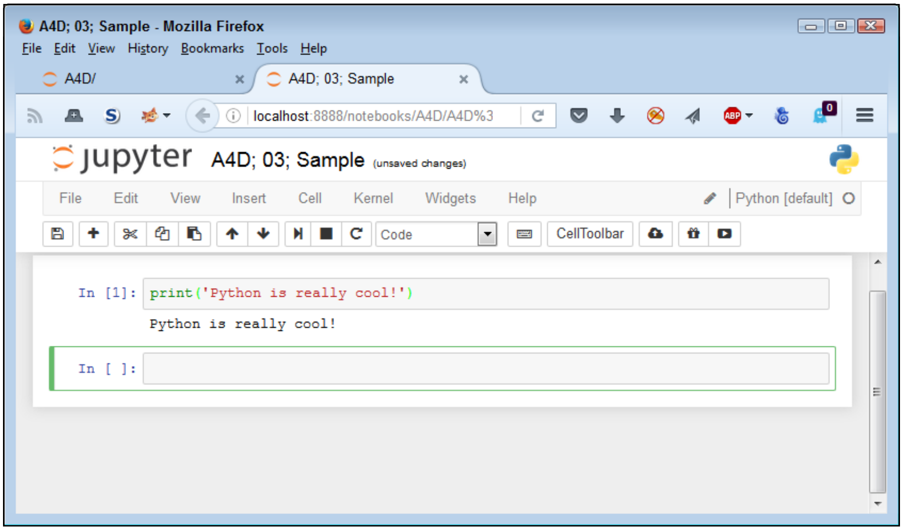
Rename Notebook

Enter a new notebook name:

Untitled

FIGURE 3-9: Provide a new name for your notebook.

Of course, the Sample notebook doesn't contain anything just yet. Place the cursor in the cell, type **print('Python is really cool!')**, and then click the Run button (the button with the right-pointing arrow on the toolbar). You see the output shown in Figure 3-10. The output is part of the same cell as the code. (The code resides in a square box and the output resides outside that square box, but both are within the cell.) However, Notebook visually separates the output from the code so that you can tell them apart. Notebook automatically creates a new cell for you.


A4D; 03; Sample - Mozilla Firefox
File Edit View History Bookmarks Tools Help
A4D/
A4D; 03; Sample
localhost:8888/notebooks/A4D/A4D%3
jupyter A4D; 03; Sample (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help
Python [default]
Code
CellToolbar

```
In [1]: print('Python is really cool!')

        Python is really cool!

In [ ]:
```

FIGURE 3-10: Notebook uses cells to store your code.

When you finish working with a notebook, shutting it down is important. To close a notebook, choose File➪Close and Halt. You return to the Home page, where you can see that the notebook you just created is added to the list, as shown in Figure 3-11.

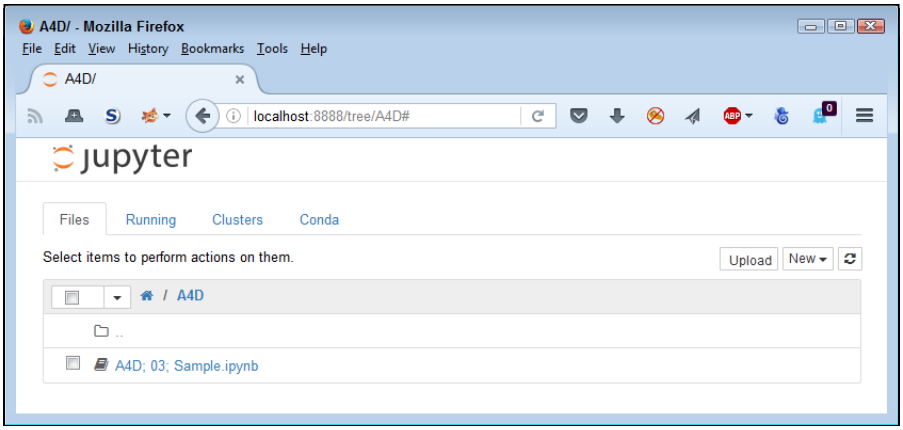


**FIGURE 3-11:** Any notebooks you create appear in the repository list.

## Exporting a notebook

Creating notebooks and keeping them all to yourself isn't much fun. At some point, you want to share them with other people. To perform this task, you must export your notebook from the repository to a file. You can then send the file to someone else, who will import it into his or her repository.

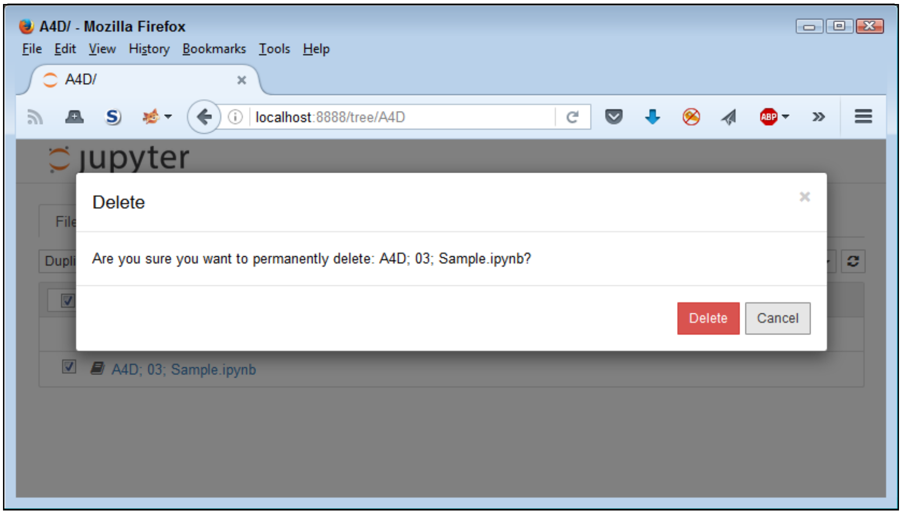
The previous section shows how to create a notebook named A4D; 03; Sample. You can open this notebook by clicking its entry in the repository list. The file reopens so that you can see your code again. To export this code, choose File➪Download As➪Notebook (.ipynb). What you see next depends on your browser, but you generally see some sort of dialog box for saving the notebook as a file. Use the same method for saving the IPython Notebook file as you use for any other file you save using your browser.

## Removing a notebook

Sometimes notebooks get outdated or you simply don't need to work with them any longer. Rather than allow your repository to get clogged with files you don't need, you can remove these unwanted notebooks from the list. Use these steps to remove the file:

1. **Select the box next to the A4D; 03; Sample.ipynb entry.**
2. **Click the trash can icon (Delete) at the top of the page.**

   You see a Delete notebook warning message like the one shown in Figure 3-12.



Delete

Are you sure you want to permanently delete: A4D; 03; Sample.ipynb?

**FIGURE 3-12:** Notebook warns you before removing any files from the repository.

3. **Click Delete.**

   The file gets removed from the list.

## Importing a notebook

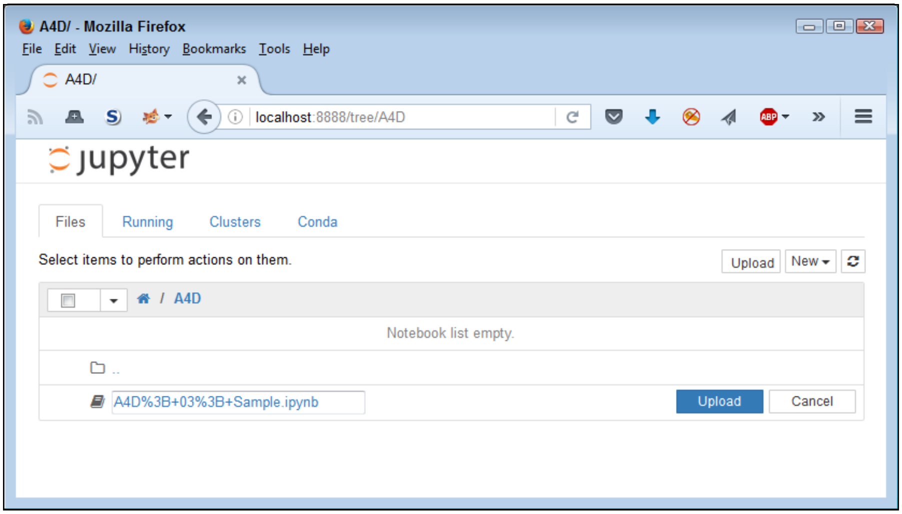
To use the source code from this book, you must import the downloaded files into your repository. The source code comes in an archive file that you extract to a location on your hard drive. The archive contains a list of `.ipynb` (IPython Notebook) files containing the source code for this book (see the Introduction for details on downloading the source code). The following steps tell how to import these files into your repository:

1. **Click Upload at the top of the page.**

   What you see depends on your browser. In most cases, you see some type of File Upload dialog box that provides access to the files on your hard drive.

2. **Navigate to the directory containing the files that you want to import into Notebook.**
3. **Highlight one or more files to import and click the Open (or other, similar) button to begin the upload process.**

   You see the file added to an upload list, as shown in Figure 3-13. The file isn't part of the repository yet — you've simply selected it for upload.



Select items to perform actions on them.

Notebook list empty.

A4D%3B+03%3B+Sample.ipynb

**FIGURE 3-13:** The files that you want to add to the repository appear as part of an upload list consisting of one or more filenames.

TIP

When you export a file, Notebook converts any special characters to a form that your system will accept with greater ease. Figure 3-13 shows this conversion in action. The semicolons appear as %3B, and spaces appear as a + (plus sign). You must change these characters to their Notebook form to see the title as you expect it.

4. **Click Upload.**

   Notebook places the file in the repository so that you can begin using it.

## Understanding the datasets used in this book

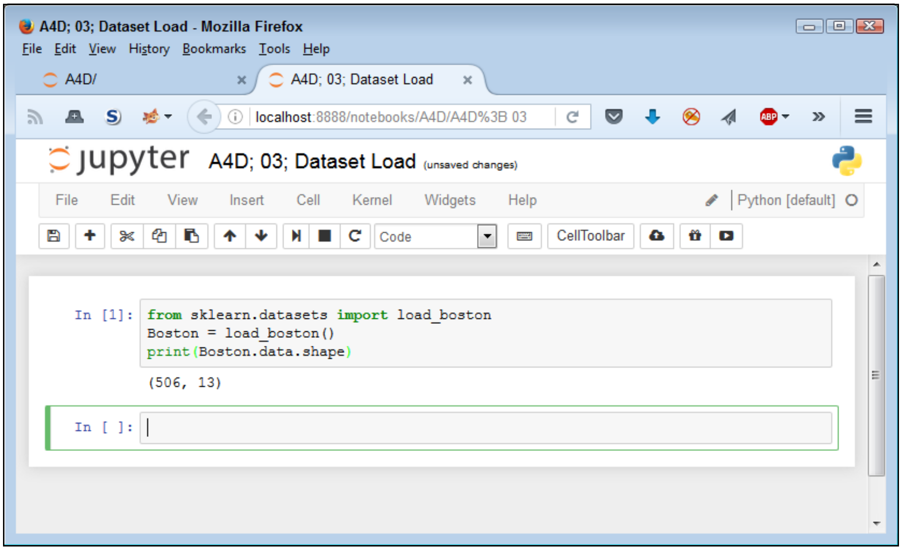
This book uses a number of datasets, all of which appear in the scikit-learn package. These datasets demonstrate various ways in which you can interact with data, and you use them in the examples to perform a variety of tasks. The following list provides a quick overview of the function used to import each of the datasets into your Python code:

- `load_boston()`: Regression analysis with the Boston house-prices dataset
- `load_iris()`: Classification with the iris dataset
- `load_diabetes()`: Regression with the diabetes dataset
- `load_digits([n_class])`: Classification with the digits dataset
- `fetch_20newsgroups(`*subset='train'*`)`: Data from 20 newsgroups
- `fetch_olivetti_faces()`:Olivetti faces dataset from AT&T

The technique for loading each of these datasets is the same across examples. The following example shows how to load the Boston house-prices dataset. You can find the code in the `A4D; 03; Dataset Load.ipynb` notebook.

```
from sklearn.datasets import load_boston
Boston = load_boston()
print(Boston.data.shape)

(506, 13)
```

To see how the code works, click Run Cell. The output from the `print()` call is `(506, 13)`. You can see the output shown in Figure 3-14.


A4D; 03; Dataset Load - Mozilla Firefox
File Edit View History Bookmarks Tools Help
A4D/
A4D; 03; Dataset Load
localhost:8888/notebooks/A4D/A4D%3B 03
jupyter A4D; 03; Dataset Load (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help
Python [default]
Code
CellToolbar
In [1]:
In [ ]:

```
from sklearn.datasets import load_boston
Boston = load_boston()
print(Boston.data.shape)
```

```
(506, 13)
```

**FIGURE 3-14:** The Boston object contains the loaded dataset.

IN THIS CHAPTER

- » Performing numeric and logic-based tasks
- » Working with strings
- » Performing tasks with dates
- » Packaging code by using functions
- » Making decisions and repeating steps
- » Managing data in memory
- » Reading data in storage objects
- » Finding data faster by using dictionaries

# Chapter 4

# Introducing Python for Algorithm Programming

A recipe is a kind of algorithm because it helps you cook tasty food by using a series of steps (and thereby get rid of your hunger). You can devise many ways to create a sequence of steps that solve a problem. Procedures of every variety and description abound, all of which describe a sequence of steps used to solve a problem. Not every sequence of steps is concrete. Mathematical notations present a series of steps to solve a numeric problem, but many people view them as so many oddly shaped symbols in an arcane language that few can understand. A computer language can turn the arcane language into a concrete form of English-like statements that solve the problem in a manner that works for most humans.

The previous chapter in this book, Chapter 3, helps you install a copy of Python to work with the examples in this book. You use Python throughout the book to solve numeric problems using algorithms that you can also express in mathematical notation. The reason that this book uses a programming language is to turn those oddly shaped abstract symbols into something that most people can understand and use to solve real-world problems.

Before you can use Python to perform tasks with algorithms, you need at least a passing knowledge of how Python works. This chapter isn't designed to make you a Python expert. However, it does provide you with enough information to make sense of the example code with the commentary provided. The various sections help you understand how Python performs tasks in a concrete manner. For example, you need to know how Python works with various kinds of data in order to determine what the example code is doing with that data. You find the essentials of working with numeric, logical, string, and date data in the first three sections.

Imagine a cookbook, or any book for that matter, that provided steps for performing every task that the book tells you how to perform as one long narrative without any breaks. Trying to find a specific recipe (or other procedure) would become impossible and the book would be useless. In fact, no one would write such a book. The fourth section of the chapter discusses functions, which are akin to the individual recipes in a cookbook. You can combine functions to create an entire program, much as you would combine recipes to create an entire dinner.

The next four sections discuss various ways to manage data, which means reading, writing, modifying, and erasing it as needed. You also need to know how to make decisions and what to do when you need to perform the same set of steps more than one time. Data is a resource, just as flour, sugar, and other ingredients are resources you use when working with a recipe. The different kinds of data require different techniques to make them into an application that solves the problem proposed by an algorithm. These sections tell you about the various ways to manipulate data and work with it to solve problems.

# Working with Numbers and Logic

Interacting with algorithms involves working with data of various sorts, but much of the work involves numbers. In addition, you use logical values to make decisions about the data you use. For example, you might need to know whether two values are equal or whether one value is greater than another value. Python supports these number and logic value types:

- Any whole number is an *integer.* For example, the value 1 is a whole number, so it's an integer. On the other hand, 1.0 isn't a whole number; it has a decimal part to it, so it's not an integer. Integers are represented by the `int` data type. On most platforms, you can store numbers between –9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 within an `int` (which is the maximum value that fits in a 64-bit variable).

- Any number that includes a decimal portion is a *floating-point* value. For example, 1.0 has a decimal part, so it's a floating-point value. Many people get confused about whole numbers and floating-point numbers, but the difference is easy to remember. If you see a decimal in the number, it's a floating-point value. Python stores floating-point values in the `float` data type. The maximum value that a floating point variable can contain is $\pm 1.7976931348623157 \times 10^{308}$ and the minimum value that a floating point variable can contain is $\pm 2.2250738585072014 \times 10^{-308}$ on most platforms.
- A *complex number* consists of a real number and an imaginary number that are paired together. In case you've completely forgotten about complex numbers, you can read about them at `http://www.mathsisfun.com/numbers/complex-numbers.html`. The imaginary part of a complex number always appears with a *j* after it. So if you want to create a complex number with 3 as the real part and 4 as the imaginary part, you make an assignment like this: `myComplex = 3 + 4j`.
- Logical arguments require Boolean values, which are named after George Bool. When using a Boolean value in Python, you rely on the `bool` type. A variable of this type can contain only two values: `True` or `False`. You can assign a value by using the `True` or `False` keywords, or you can create an expression that defines a logical idea that equates to true or false. For example, you could say `myBool = 1 > 2`, which would equate to False because 1 is most definitely not greater than 2.

Now that you have the basics down, it's time to see the data types in action. The following paragraphs provide a quick overview of how you can work with both numeric and logical data in Python.

## Performing variable assignments

When working with applications, you store information in *variables.* A variable is a kind of storage box. Whenever you want to work with the information, you access it using the variable. If you have new information that you want to store, you put it in a variable. Changing information means accessing the variable first and then storing the new value in the variable. Just as you store things in boxes in the real world, so you store things in variables (a kind of storage box) when working with applications. To store data in a variable, you *assign* the data to it using any of a number of *assignment operators* (special symbols that tell how to store the data). Table 4-1 shows the assignment operators that Python supports.

**TABLE 4-1** Python Assignment Operators

| Operator | Description | Example |
|---|---|---|
| = | Assigns the value found in the right operand to the left operand | MyVar = 5 results in MyVar containing 5 |
| += | Adds the value found in the right operand to the value found in the left operand and places the result in the left operand | MyVar += 2 results in MyVar containing 7 |
| -= | Subtracts the value found in the right operand from the value found in the left operand and places the result in the left operand | MyVar -= 2 results in MyVar containing 3 |
| *= | Multiplies the value found in the right operand by the value found in the left operand and places the result in the left operand | MyVar *= 2 results in MyVar containing 10 |
| /= | Divides the value found in the left operand by the value found in the right operand and places the result in the left operand | MyVar /= 2 results in MyVar containing 2.5 |
| %= | Divides the value found in the left operand by the value found in the right operand and places the remainder in the left operand | MyVar %= 2 results in MyVar containing 1 |
| **= | Determines the exponential value found in the left operand when raised to the power of the value found in the right operand and places the result in the left operand | MyVar ** 2 results in MyVar containing 25 |
| //= | Divides the value found in the left operand by the value found in the right operand and places the integer (whole number) result in the left operand | MyVar //= 2 results in MyVar containing 2 |

## Doing arithmetic

Storing information in variables makes it easily accessible. However, to perform any useful work with the variable, you usually perform some type of arithmetic operation on it. Python supports the common arithmetic operators you use to perform tasks by hand. They appear in Table 4-2.

**TABLE 4-2** Python Arithmetic Operators

| Operator | Description | Example |
|---|---|---|
| + | Adds two values together | 5 + 2 = 7 |
| - | Subtracts the right operand from the left operand | 5 – 2 = 3 |
| * | Multiplies the right operand by the left operand | 5 * 2 = 10 |
| / | Divides the left operand by the right operand | 5 / 2 = 2.5 |
| % | Divides the left operand by the right operand and returns the remainder | 5 % 2 = 1 |
| ** | Calculates the exponential value of the right operand by the left operand | 5 ** 2 = 25 |
| // | Performs integer division, in which the left operand is divided by the right operand and only the whole number is returned (also called floor division) | 5 // 2 = 2 |

Sometimes you need to interact with just one variable. Python supports a number of *unary operators*, those that work with just one variable, as shown in Table 4-3.

**TABLE 4-3** Python Unary Operators

| Operator | Description | Example |
|---|---|---|
| ~ | Inverts the bits in a number so that all the 0 bits become 1 bits and vice versa | ~4 results in a value of –5 |
| - | Negates the original value so that positive becomes negative and vice versa | –(–4) results in 4 and –4 results in –4 |
| + | Is provided purely for the sake of completeness; returns the same value that you provide as input | +4 results in a value of 4 |

Computers can perform other sorts of math tasks because of the way in which the processor works. It's important to remember that computers store data as a series of individual bits. Python lets you access these individual bits by using *bitwise operators*, as shown in Table 4-4.

**TABLE 4-4** Python Bitwise Operators

| Operator | Description | Example |
|---|---|---|
| & (And) | Determines whether both individual bits within two operators are true and sets the resulting bit to true when they are. | 0b1100 & 0b0110 = 0b0100 |
| \| (Or) | Determines whether either of the individual bits within two operators are true and sets the resulting bit to true when they are. | 0b1100 \| 0b0110 = 0b1110 |
| ^ (Exclusive or) | Determines whether just one of the individual bits within two operators is true and sets the resulting bit to true when one is. When both bits are true or both bits are false, the result is false. | 0b1100 ^ 0b0110 = 0b1010 |
| ~ (One's complement) | Calculates the one's complement value of a number. | ~0b1100 = –0b1101<br>~0b0110 = –0b0111 |
| << (Left shift) | Shifts the bits in the left operand left by the value of the right operand. All new bits are set to 0 and all bits that flow off the end are lost. | 0b00110011 << 2 = 0b11001100 |
| >> (Right shift) | Shifts the bits in the left operand right by the value of the right operand. All new bits are set to 0 and all bits that flow off the end are lost. | 0b00110011 >> 2 = 0b00001100 |

## Comparing data by using Boolean expressions

Using arithmetic to modify the content of variables is a kind of data manipulation. To determine the effect of data manipulation, a computer must compare the current state of the variable against its original state or the state of a known value. In some cases, detecting the status of one input against another is also necessary. All these operations check the relationship between two variables, so the resulting operators are relational operators, as shown in Table 4-5.

**TABLE 4-5** Python Relational Operators

| Operator | Description | Example |
|---|---|---|
| == | Determines whether two values are equal. Notice that the relational operator uses two equals signs. A mistake many developers make is using just one equals sign, which results in one value being assigned to another. | 1 == 2 is False |
| != | Determines whether two values are not equal. Some older versions of Python allowed you to use the <> operator in place of the != operator. Using the <> operator results in an error in current versions of Python. | 1 != 2 is True |
| > | Verifies that the left operand value is greater than the right operand value. | 1 > 2 is False |
| < | Verifies that the left operand value is less than the right operand value. | 1 < 2 is True |
| >= | Verifies that the left operand value is greater than or equal to the right operand value. | 1 >= 2 is False |
| <= | Verifies that the left operand value is less than or equal to the right operand value. | 1 <= 2 is True |

Sometimes a relational operator can't tell the whole story of the comparison of two values. For example, you might need to check a condition in which two separate comparisons are needed, such as `MyAge > 40` and `MyHeight < 74`. The need to add conditions to the comparison requires a logical operator of the sort shown in Table 4-6.

**TABLE 4-6** Python Logical Operators

| Operator | Description | Example |
|---|---|---|
| and | Determines whether both operands are true. | True and True is True<br>True and False is False<br>False and True is False<br>False and False is False |
| or | Determines when one of two operands is true. | True or True is True<br>True or False is True<br>False or True is True<br>False or False is False |
| not | Negates the truth value of a single operand. A true value becomes false and a false value becomes true. | not True is False<br>not False is True |

Computers provide order to comparisons by making some operators more significant than others. The ordering of operators is *operator precedence.* Table 4-7 shows the operator precedence of all the common Python operators, including a few you haven't seen as part of a discussion yet. When making comparisons, always consider operator precedence because otherwise, the assumptions you make about a comparison outcome will likely be wrong.

**TABLE 4-7** Python Operator Precedence

| Operator | Description |
|---|---|
| () | You use parentheses to group expressions and to override the default precedence so that you can force an operation of lower precedence (such as addition) to take precedence over an operation of higher precedence (such as multiplication). |
| ** | Exponentiation raises the value of the left operand to the power of the right operand. |
| ~ + - | Unary operators interact with a single variable or expression. |
| * / % // | Multiply, divide, modulo, and floor division. |
| + - | Addition and subtraction. |
| >> << | Right and left bitwise shift. |
| & | Bitwise AND. |
| ^ \| | Bitwise exclusive OR and standard OR. |
| <= < > >= | Comparison operators. |

*(continued)*

**TABLE 4-7** *(continued)*

| Operator | Description |
|---|---|
| == != | Equality operators. |
| = %= /= //= -= += *= **= | Assignment operators. |
| is<br>is not | Identity operators. |
| in<br>not in | Membership operators. |
| not or and | Logical operators. |

# Creating and Using Strings

Of all the data types, strings are the most easily understood by humans and not understood at all by computers. A *string* is simply any grouping of characters you place within double quotation marks. For example, `myString = "Python is a great language."` assigns a string of characters to `myString`.

REMEMBER

The main reason to use strings when working with algorithms is to provide user interaction — either as requests for input or as a means of making output easier to understand. You can also perform analysis of string data as part of working with algorithms, but the computer doesn't actually require strings as part of its sequence of steps to obtain a solution to a problem. In fact, the computer doesn't see letters at all. Every letter you use is represented by a number in memory. For example, the letter *A* is actually the number 65. To see this for yourself, type **ord("A")** at the Python prompt and press Enter. You see 65 as output. You can convert any single letter to its numeric equivalent using the `ord()` command.

## STARTING IPython

Most of the book relies on Jupyter Notebook (see Chapter 3) because it provides methods for creating, managing, and interacting with complex coding examples. However, sometimes you need a simple interactive environment to use for quick tests, which is the route this chapter uses. Anaconda comes with two such environments, IPython and Jupyter QT Console. Of the two, IPython is the simplest to use, but both environments provide similar functionality. To start IPython, simply click its entry in the Anaconda3 folder on your system. For example, when working with Windows, you choose Start ➪ All Programs ➪ Anaconda3 ➪ IPython. You can also start IPython in a console or terminal window by typing **IPython** and pressing Enter.

Because the computer doesn't really understand strings, but strings are so useful in writing applications, you sometimes need to convert a string to a number. You can use the `int()` and `float()` commands to perform this conversion. For example, if you type **myInt = int("123")** and press Enter at the Python prompt, you create an `int` named `myInt` that contains the value `123`.

REMEMBER

You can convert numbers to a string as well by using the `str()` command. For example, if you type **myStr = str(1234.56)** and press Enter, you create a string containing the value `"1234.56"` and assign it to `myStr`. The point is that you can go back and forth between strings and numbers with great ease. Later chapters demonstrate how these conversions make many seemingly impossible tasks quite doable.

As with numbers, you can use some special operators with strings (and many objects). The *member operators* enable you to determine when a string contains specific content. Table 4-8 shows these operators.

**TABLE 4-8** Python Membership Operators

| Operator | Description | Example |
|---|---|---|
| in | Determines whether the value in the left operand appears in the sequence found in the right operand | "Hello" in "Hello Goodbye" is True |
| not in | Determines whether the value in the left operand is missing from the sequence found in the right operand | "Hello" not in "Hello Goodbye" is False |

The discussion in this section also makes it obvious that you need to know the kind of data that variables contain. You use the *identity operators* to perform this task, as shown in Table 4-9.

**TABLE 4-9** Python Identity Operators

| Operator | Description | Example |
|---|---|---|
| is | Evaluates to true when the type of the value or expression in the right operand points to the same type in the left operand | type(2) is int is True |
| is not | Evaluates to true when the type of the value or expression in the right operand points to a different type than the value or expression in the left operand | type(2) is not int is False |

# Interacting with Dates

Dates and times are items that most people work with quite a bit. Society bases almost everything on the date and time that a task needs to be or was completed. We make appointments and plan events for specific dates and times. Most of our day revolves around the clock. When working with algorithms, the date or time at which a particular step in a sequence occurs can be just as important as how the step occurs and what happens as a result of performing the step. Algorithms rely on date and time to organize data so that humans can better understand the data and the resulting output of the algorithm.

Because of the time-oriented nature of humans, it's a good idea to look at how Python deals with interacting with date and time (especially storing these values for later use). As with everything else, computers understand only numbers — date and time don't really exist. The algorithm, not the computer, relies on date and time to help organize the series of steps performed to solve a problem.

REMEMBER

To work with dates and times, you must issue a special `import datetime` command. Technically, this act is called *importing a module.* Don't worry about how the command works right now — just use it whenever you want to do something with date and time.

Computers do have clocks inside them, but the clocks are for the humans using the computer. Yes, some software also depends on the clock, but again, the emphasis is on human needs rather than anything the computer might require. To get the current time, you can simply type **datetime.datetime.now()** and press Enter. You see the full date and time information as found on your computer's clock, such as `datetime.datetime(2016, 12, 20, 10, 37, 24, 460099)`.

You may have noticed that the date and time are a little hard to read in the existing format. Say that you want to get just the current date, and in a readable format. To accomplish this task, you access just the date portion of the output and convert it into a string. Type **str(datetime.datetime.now().date())** and press Enter. You now have something a little more usable, such as `'2016-12-20'`.

Interestingly enough, Python also has a `time()` command, which you can use to obtain the current time. You can obtain separate values for each of the components that make up date and time using the `day`, `month`, `year`, `hour`, `minute`, `second`, and `microsecond` values. Later chapters help you understand how to use these various date and time features to make working with algorithms easier.

# Creating and Using Functions

Every step in an algorithm normally requires a single line of Python code — an English-like instruction that tells the computer how to move the problem solution one step closer to completion. You combine these lines of code to achieve a desired result. Sometimes you need to repeat the instructions with different data, and in some cases your code becomes so long that it's hard to keep track of what each part does. Functions serve as organization tools that keep your code neat and tidy. In addition, functions make it easy to reuse the instructions you've created as needed with different data. This section of the chapter tells you all about functions. More important, in this section you start creating your first serious applications in the same way that professional developers do.

## Creating reusable functions

You go to your closet, take out pants and shirt, remove the labels, and put them on. At the end of the day, you take everything off and throw it in the trash. Hmmm . . . that really isn't what most people do. Most people take the clothes off, wash them, and then put them back into the closet for reuse. Functions are reusable, too. No one wants to keep repeating the same task; it becomes monotonous and boring. When you create a function, you define a package of code that you can use over and over to perform the same task. All you need to do is tell the computer to perform a specific task by telling it which function to use. The computer faithfully executes each instruction in the function absolutely every time you ask it to do so.

REMEMBER

When you work with functions, the code that needs services from the function is named the *caller,* and it calls upon the function to perform tasks for it. Much of the information you see about functions refers to the caller. The caller must supply information to the function, and the function returns information to the caller.

At one time, computer programs didn't include the concept of code reusability. As a result, developers had to keep reinventing the same code. It didn't take long for someone to come up with the idea of functions, though, and the concept has evolved over the years until functions have become quite flexible. You can make functions do anything you want. Code reusability is a necessary part of applications to

- Reduce development time
- Reduce programmer error
- Increase application reliability

- Allow entire groups to benefit from the work of one programmer
- Make code easier to understand
- Improve application efficiency

In fact, functions do a whole list of things for applications in the form of reusability. As you work through the examples in this book, you see how reusability makes your life significantly easier. If not for reusability, you'd still be programming by plugging 0s and 1s into the computer by hand.

Creating a function doesn't require much work. To see how functions work, open a copy of IPython and type in the following code (pressing Enter at the end of each line):

```
def SayHello():
    print('Hello There!')
```

To end the function, you press Enter a second time after the last line. A function begins with the keyword `def` (for define). You provide a function name, parentheses that can contain function *arguments* (data used in the function), and a colon. The editor automatically indents the next line for you. Python relies on whitespace to define *code blocks* (statements that are associated with each other in a function).

You can now use the function. Simply type **SayHello()** and press Enter. The parentheses after the function name are important because they tell Python to execute the function rather than tell you that you are accessing a function as an object (to determine what it is). You see `Hello There!` as the output.

## Calling functions

Functions can accept arguments (additional bits of data) and return values. The capability to exchange data makes functions far more useful than they otherwise might be. The following sections describe how to call functions in a variety of ways to both send and receive data.

### Sending requirement arguments

A function can require the caller to provide arguments to it. A required argument is a variable that must contain data for the function to work. Open a copy of IPython and type the following code:

```
def DoSum(Value1, Value2):
    return Value1 + Value2
```

You have a new function, `DoSum()`. This function requires that you provide two arguments to use it. At least, that's what you've heard so far. Type **DoSum()** and press Enter. You see an error message like this one:

```
TypeError
    Traceback (most recent call last)
<ipython-input-2-a37c1b30cd89> in <module>()
----> 1 DoSum()

TypeError: DoSum() missing 2 required positional
    arguments: 'Value1' and 'Value2'
```

Trying `DoSum()` with just one argument would result in another error message. To use `DoSum()` ,you must provide two arguments. To see how this works, type **DoSum(1, 2)** and press Enter. You see the expected result of `3`.

TIP

Notice that `DoSum()` provides an output value of `3` when you supply `1` and `2` as inputs. The `return` statement provides the output value. Whenever you see `return` in a function, you know that the function provides an output value.

## Sending arguments by keyword

As your functions become more complex and the methods to use them do as well, you may want to provide a little more control over precisely how you call the function and provide arguments to it. Until now, you have *positional arguments*, which means that you have supplied values in the order in which they appear in the argument list for the function definition. However, Python also has a method for sending arguments by keyword. In this case, you supply the name of the argument followed by an equals sign (=) and the argument value. To see how this works, open a copy of IPython and type the following code:

```
def DisplaySum(Value1, Value2):
    print(str(Value1) + ' + ' + str(Value2) + ' = ' +
    str((Value1 + Value2)))
```

REMEMBER

Notice that the `print()` function argument includes a list of items to print and that those items are separated by plus signs (+). In addition, the arguments are of different types, so you must convert them using the `str()` function. Python makes it easy to mix and match arguments in this manner. This function also introduces the concept of automatic line continuation. The `print()` function actually appears on two lines, and Python automatically continues the function from the first line to the second.

Next, it's time to test DisplaySum(). Of course, you want to try the function using positional arguments first, so type **DisplaySum(2, 3)** and press Enter. You see the expected output of 2 + 3 = 5. Now type **DisplaySum(Value2 = 3, Value1 = 2)** and press Enter. Again, you receive the output 2 + 3 = 5 even though the position of the arguments has been reversed.

## Giving function arguments a default value

Whether you make the call using positional arguments or keyword arguments, the functions to this point have required that you supply a value. Sometimes a function can use default values when a common value is available. Default values make the function easier to use and less likely to cause errors when a developer doesn't provide an input. To create a default value, you simply follow the argument name with an equals sign and the default value. To see how this works, open a copy of IPython and type the following code:

```
def SayHello(Greeting = "No Value Supplied"):
    print(Greeting)
```

The SayHello() function provides an automatic value for Greeting when a caller doesn't provide one. When someone tries to call SayHello() without an argument, it doesn't raise an error. Type **SayHello()** and press Enter to see for yourself — you see the default message. Type **SayHello("Howdy!")** to see a normal response.

## Creating functions with a variable number of arguments

In most cases, you know precisely how many arguments to provide with your function. It pays to work toward this goal whenever you can because functions with a fixed number of arguments are easier to troubleshoot later. However, sometimes you simply can't determine how many arguments the function will receive at the outset. For example, when you create a Python application that works at the command line, the user might provide no arguments, the maximum number of arguments (assuming there is one), or any number of arguments in between.

Fortunately, Python provides a technique for sending a variable number of arguments to a function. You simply create an argument that has an asterisk in front of it, such as *VarArgs. The usual technique is to provide a second argument that contains the number of arguments passed as an input. To see how this works, open a copy of IPython and type the following code:

```
def DisplayMulti(ArgCount = 0, *VarArgs):
    print('You passed ' + str(ArgCount) + ' arguments.',
    VarArgs)
```

Notice that the `print()` function displays a string and then the list of arguments. Because of the way this function is designed, you can type **DisplayMulti()** and press Enter to see that you can pass zero arguments. To see multiple arguments at work, type **DisplayMulti(3, 'Hello', 1, True)** and press Enter. The output of `('You passed 3 arguments.', ('Hello', 1, True))` shows that you need not pass values of any particular type.

# Using Conditional and Loop Statements

Algorithms often require steps that make decisions or perform some steps more than one time. For example, you might need to throw out a value that doesn't fit with the rest of the data, which requires making a decision, or you might need to process the data more than once to obtain a desired result, such as when you filter the data. Python accommodates this need by providing special statements that make decisions or let you perform steps more than once, as described in the sections that follow.

## Making decisions using the if statement

You use if statements regularly in everyday life. For example, you may say to yourself, "If it's Wednesday, I'll eat tuna salad for lunch." The Python `if` statement is a little less verbose, but it follows precisely the same pattern. To see how this works, open a copy of IPython and type the following code:

```
def TestValue(Value):
    if Value == 5:
        print('Value equals 5!')
    elif Value == 6:
        print('Value equals 6!')
    else:
        print('Value is something else.')
        print('It equals ' + str(Value))
```

Every Python `if` statement begins, oddly enough, with the word *if.* When Python sees `if`, it knows that you want it to make a decision. After the word *if* comes a condition. A *condition* simply states what sort of comparison you want Python to make. In this case, you want Python to determine whether `Value` contains the value `5`.

WARNING

Notice that the condition uses the relational equality operator, ==, and not the assignment operator, =. A common mistake that developers make is to use the assignment operator rather than the equality operator. Using the assignment operator in place of the equality operator will cause your code to malfunction.

The condition always ends with a colon (:). If you don't provide a colon, Python doesn't know that the condition has ended and will continue to look for additional conditions on which to base its decision. After the colon comes any tasks you want Python to perform.

You may need to perform multiple tasks using a single if statement. The `elif` clause makes it possible to add an additional condition and associated tasks. A *clause* is an addendum to a previous condition, which is an `if` statement in this case. The `elif` clause always provides a condition, just as the `if` statement does, and it has its own associated set of tasks to perform.

Sometimes you need to do something no matter what the condition might be. In this case, you add the `else` clause. The `else` clause tells Python to do something in particular when the conditions of the `if` statement aren't met.

REMEMBER

Notice how indenting is becoming more important as the functions become more complex. The function contains an `if` statement. The `if` statement contains just one `print()` statement. The `else` clause contains two `print()` statements.

To see this function in action, type **TestValue(1)** and press Enter. You see the output from the `else` clause. Type **TestValue(5)** and press Enter. The output now reflects the `if` statement output. Type **TestValue(6)** and press Enter. The output now shows the results of the `elif` clause. The result is that this function is more flexible than previous functions in the chapter because it can make decisions.

## Choosing between multiple options using nested decisions

*Nesting* is the process of placing a subordinate statement within another statement. You can nest any statement within any other statement, in most cases. To see how this works, open a copy of IPython and type the following code:

```
def SecretNumber():
    One = int(input("Type a number between 1 and 10: "))
    Two = int(input("Type a number between 1 and 10: "))

    if (One >= 1) and (One <= 10):
        if (Two >= 1) and (Two <= 10):
            print('Your secret number is: ' + str(One * Two))
        else:
            print("Incorrect second value!")
    else:
        print("Incorrect first value!")
```

In this case, `SecretNumber()` asks you to provide two inputs. Yes, you can get inputs from a user when needed by using the `input()` function. The `int()` function converts the inputs to a number.

There are two levels of `if` statement this time. The first level checks for the validity of the number in `One`. The second level checks for the validity of the number in `Two`. When both `One` and `Two` have values between 1 and 10, .`SecretNumber()` outputs a secret number for the user.

To see `SecretNumber()` in action, type **SecretNumber()** and press Enter. Type **20** and press Enter when asked for the first input value, and type `10` and press Enter when asked for the second. You see an error message telling you that the first value is incorrect. Type **SecretNumber()** and press Enter again. This time, use values of 10 and 20. The function will tell you that the second input is incorrect. Try the same sequence again using input values of 10 and 10.

## Performing repetitive tasks using the for loop

Sometimes you need to perform a task more than one time. You use the `for` loop statement when you need to perform a task a specific number of times. The `for` loop has a definite beginning and a definite end. The number of times that this loop executes depends on the number of elements in the variable you provide. To see how this works, open a copy of IPython and type the following code:

```
def DisplayMulti(*VarArgs):
    for Arg in VarArgs:
        if Arg.upper() == 'CONT':
            continue
            print('Continue Argument: ' + Arg)
        elif Arg.upper() == 'BREAK':
            break
            print('Break Argument: ' + Arg)
        print('Good Argument: ' + Arg)
```

In this case, the `for` loop attempts to process each element in `VarArgs`. Notice that there is a nested `if` statement in the loop and it tests for two ending conditions.

In most cases, the code skips the if statement and simply prints the argument. However, when the if statement finds the words CONT or BREAK in the input values, it performs one of these two tasks:

- continue: Forces the loop to continue from the current point of execution with the next entry in VarArgs.
- break: Stops the loop from executing.

TIP

The keywords can appear using any combination of uppercase and lowercase letters, such as ConT, because the upper() function converts them to uppercase. The DisplayMulti() function can process any number of input strings. To see it in action, type **DisplayMulti('Hello', 'Goodbye', 'First', 'Last')** and press Enter. You see each of the input strings presented on a separate line in the output. Now type **DisplayMulti('Hello', 'Cont', 'Goodbye', 'Break', 'Last')** and press Enter. Notice that the words Cont and Break don't appear in the output because they're keywords. In addition, the word Last doesn't appear in the output because the for loop ends before this word is processed.

## Using the while statement

The while loop statement continues to perform tasks until such time that a condition is no longer true. As with the for statement, the while statement supports both the continue and break keywords for ending the loop prematurely. To see how this works, open a copy of IPython and type the following code:

```
def SecretNumber():
   GotIt = False
   while GotIt == False:
      One = int(input("Type a number between 1 and 10: "))
      Two = int(input("Type a number between 1 and 10: "))

      if (One >= 1) and (One <= 10):
         if (Two >= 1) and (Two <= 10):
            print('Secret number is: ' + str(One * Two))
            GotIt = True
            continue
         else:
            print("Incorrect second value!")
      else:
         print("Incorrect first value!")
      print("Try again!")
```

This is an expansion of the SecretNumber() function first described in the "Choosing between multiple options using nested decisions" section, earlier in this chapter. However, in this case, the addition of a while loop statement means that the function continues to ask for input until it receives a valid response.

To see how the while statement works, type **SecretNumber()** and press Enter. Type **20** and press Enter for the first prompt. Type **10** and press Enter for the second prompt. The example tells you that the first number is wrong and then tells you to try again. Try a second time using values of 10 and 20. This time, the second number is wrong and you still need to try again. On the third try, use values of 10 and 10. This time, you get a secret number. Notice that the use of a continue clause means that the application doesn't tell you to try again.

# Storing Data Using Sets, Lists, and Tuples

When working with algorithms, it's all about the data. Python provides a host of methods for storing data in memory. Each method has advantages and disadvantages. Choosing the most appropriate method for your particular need is important. The following sections discuss three common techniques used for storing data for data science needs.

## Creating sets

Most people have used sets at one time or another in school to create lists of items that belong together. These lists then became the topic of manipulation using math operations such as intersection, union, difference, and symmetric difference. Sets are the best option to choose when you need to perform membership testing and remove duplicates from a list. You can't perform sequence-related tasks using sets, such as indexing or slicing. To see how you can work with sets, start a copy of IPython and type the following code:

```
SetA = set(['Red', 'Blue', 'Green', 'Black'])
SetB = set(['Black', 'Green', 'Yellow', 'Orange'])
SetX = SetA.union(SetB)
SetY = SetA.intersection(SetB)
SetZ = SetA.difference(SetB)
```

You now have five different sets to play with, each of which has some common elements. To see the results of each math operation, type **print('{0}\n{1}\n{2}'.**

**format(SetX, SetY, SetZ))** and press Enter. You see one set printed on each line, like this:

```
{'Blue', 'Orange', 'Red', 'Green', 'Black', 'Yellow'}
{'Green', 'Black'}
{'Blue', 'Red'}
```

TIP

The outputs show the results of the math operations: `union()`, `intersection()`, and `difference()`. Python's fancier print formatting can be useful in working with collections such as sets. The `format()` function tells Python which objects to place within each of the placeholders in the string. A *placeholder* is a set of curly brackets (`{}`) with an optional number in it. The *escape character* (essentially a kind of control or special character), `/n`, provides a newline character between entries. You can read more about fancy formatting at `https://docs.python.org/3/tutorial/inputoutput.html`.

You can also test relationships between the various sets. For example, type **SetA.issuperset(SetY)** and press Enter. The output value of `True` tells you that `SetA` is a superset of `SetY`. Likewise, if you type **SetA.issubset(SetX)** and press Enter, you find that `SetA` is a subset of `SetX`.

It's important to understand that sets are either mutable or immutable. All the sets in this example are mutable, which means that you can add or remove elements from them. For example, if you type **SetA.add('Purple')** and press Enter, `SetA` receives a new element. If you type **SetA.issubset(SetX)** and press Enter now, you find that `SetA` is no longer a subset of `SetX` because `SetA` has the `'Purple'` element in it.

## Creating lists

The Python specification defines a list as a kind of sequence. *Sequences* simply provide some means of allowing multiple data items to exist together in a single storage unit, but as separate entities. Think about one of those large mail holders you see in apartment buildings. A single mail holder contains a number of small mailboxes, each of which can contain mail. Python supports other kinds of sequences as well:

- **Tuples:** A tuple is a collection that's used to create complex, list-like sequences. An advantage of tuples is that you can nest the content of a tuple. This feature lets you create structures that can hold employee records or x-y coordinate pairs.
- **Dictionaries:** As with the real dictionaries, you create key/value pairs when using the dictionary collection (think of a word and its associated definition).

A dictionary provides incredibly fast search times and makes ordering data significantly easier.

- **Stacks:** Most programming languages support stacks directly. However, Python doesn't support the stack, although a workaround exists for that. A stack is a last in/first out (LIFO) sequence. Think of a pile of pancakes: You can add new pancakes to the top and also take them off the top. A stack is an important collection that you can simulate in Python by using a list.
- **Queues:** A queue is a first in/first out (FIFO) collection. You use it to track items that need to be processed in some way. Think of a queue as a line at the bank. You go into the line, wait your turn, and are eventually called to talk with a teller.
- **Deques:** A double-ended queue (deque) is a queue-like structure that lets you add or remove items from either end, but not from the middle. You can use a deque as a queue or a stack or any other kind of collection to which you're adding and from which you're removing items in an orderly manner (in contrast to lists, tuples, and dictionaries, which allow randomized access and management).

Of all the sequences, lists are the easiest to understand and are the most directly related to a real-world object. Working with lists helps you become better able to work with other kinds of sequences that provide greater functionality and improved flexibility. The point is that the data is stored in a list much as you would write it on a piece of paper: One item comes after another. The list has a beginning, a middle, and an end. Python numbers the items in the list. (Even if you might not normally number the list items in real life, using a numbered list makes the items easy to access.) To see how you can work with lists, start a copy of IPython and type the following code:

```
ListA = [0, 1, 2, 3]
ListB = [4, 5, 6, 7]
ListA.extend(ListB)
ListA
```

When you type the last line of code, you see the output of `[0, 1, 2, 3, 4, 5, 6, 7]`. The `extend()` function adds the members of `ListB` to `ListA`. Besides extending lists, you can also add to them by using the `append()` function. Type **ListA.append(-5)** and press Enter. When you type **ListA** and press Enter again, you see that Python has added −5 to the end of the list. You may find that you need to remove items again, and you do that by using the `remove()` function. For example, type **ListA.remove(-5)** and press Enter. When you list `ListA` again by typing **ListA** and pressing Enter, you see that the added entry is gone.

REMEMBER

Lists also support *concatenation* by using the plus (+) sign to add one list to another. For example, if you type **ListX = ListA + ListB** and press Enter, you find that the newly created ListX contains both ListA and ListB in it, with the elements of ListA coming first.

## Creating and using tuples

A tuple is a collection used to create complex lists, in which you can embed one tuple within another. This embedding lets you create hierarchies with tuples. A hierarchy can be something as simple as the directory listing of your hard drive or an organizational chart for your company. The idea is that you can create complex data structures using a tuple.

REMEMBER

Tuples are *immutable,* which means that you can't change them. You can create a new tuple with the same name and modify it in some way, but you can't modify an existing tuple. Lists are mutable, which means that you can change them. So a tuple can seem at first to be at a disadvantage, but immutability has all sorts of advantages, such as being more secure as well as faster. In addition, immutable objects are easier to use with multiple processors. To see how you can work with tuples, start a copy of IPython and type the following code:

```
MyTuple = (1, 2, 3, (4, 5, 6, (7, 8, 9)))
```

MyTuple is nested three levels deep. The first level consists of the values 1, 2, 3, and a tuple. The second level consists of the values 4, 5, 6, and yet another tuple. The third level consists of the values 7, 8, and 9. To see how this works, type the following code into IPython:

```
for Value1 in MyTuple:
    if type(Value1) == int:
        print(Value1)
    else:
        for Value2 in Value1:
            if type(Value2) == int:
                print("\t", Value2)
            else:
                for Value3 in Value2:
                   print("\t\t", Value3)
```

When you run this code, you find that the values really are at three different levels. You can see the indentations showing the level:

```
1
2
3
        4
        5
        6
                7
                8
                9
```

TIP

It is possible to perform tasks such as adding new values, but you must do it by adding the original entries and the new values to a new tuple. In addition, you can add tuples to an existing tuple only. To see how this works, type **MyNewTuple = MyTuple.___add___((10, 11, 12, (13, 14, 15)))** and press Enter. `MyNewTuple` contains new entries at both the first and second levels, like this: `(1, 2, 3, (4, 5, 6, (7, 8, 9)), 10, 11, 12, (13, 14, 15))`.

# Defining Useful Iterators

The chapters that follow use all kinds of techniques to access individual values in various types of data structures. For this section, you use two simple lists, defined as the following:

```
ListA = ['Orange', 'Yellow', 'Green', 'Brown']
ListB = [1, 2, 3, 4]
```

The simplest method of accessing a particular value is to use an index. For example, if you type **ListA[1]** and press Enter, you see `'Yellow'` as the output. All indexes in Python are zero based, which means that the first entry is 0, not 1.

Ranges present another simple method of accessing values. For example, if you type **ListB[1:3]** and press Enter, the output is `[2, 3]`. You could use the range as input to a `for` loop, such as

```
for Value in ListB[1:3]:
    print(Value)
```

Instead of the entire list, you see just `2` and `3` as outputs, printed on separate lines. The range has two values separated by a colon. However, the values are optional. For example, `ListB[:3]` would output `[1, 2, 3]`. When you leave out a value, the range starts at the beginning or the end of the list, as appropriate.

Sometimes you need to process two lists in parallel. The simplest method of doing this is to use the zip() function. Here's an example of the zip() function in action:

```
for Value1, Value2 in zip(ListA, ListB):
    print(Value1, '\t', Value2)
```

This code processes both ListA and ListB at the same time. The processing ends when the for loop reaches the shortest of the two lists. In this case, you see the following:

```
Orange  1
Yellow  2
Green   3
Brown   4
```

REMEMBER

This is the tip of the iceberg. You see a host of iterator types used throughout the book. The idea is to enable you to list just the items you want, rather than all the items in a list or other data structure. Some of the iterators used in upcoming chapters are a little more complicated than what you see here, but this is an important start.

# Indexing Data Using Dictionaries

A dictionary is a special kind of sequence that uses a name and value pair. The use of a name makes it easy to access particular values with something other than a numeric index. To create a dictionary, you enclose name and value pairs in curly brackets. Create a test dictionary by typing **MyDict = {'Orange':1, 'Blue':2, 'Pink':3}** and pressing Enter.

To access a particular value, you use the name as an index. For example, type **MyDict[‘Pink’]** and press Enter to see the output value of 3. The use of dictionaries as data structures makes it easy to access incredibly complex data sets using terms that everyone can understand. In many other respects, working with a dictionary is the same as working with any other sequence.

Dictionaries do have some special features. For example, type **MyDict.keys()** and press Enter to see a list of the keys. You can use the values() function to see the list of values in the dictionary.

IN THIS CHAPTER

- **Using matrixes and vectors to perform calculations**
- **Obtaining the correct combinations**
- **Employing recursive techniques to obtain specific results**
- **Considering ways to speed calculations**

# Chapter 5

# Performing Essential Data Manipulations Using Python

Chapter 4 discusses the use of Python as a means for expressing in concrete terms those arcane symbols often used in mathematical representations of algorithms. In that chapter, you discover the various language constructs used to perform tasks in Python. However, simply knowing how to control a language by using its constructs to perform tasks isn't enough. The goal of mathematical algorithms is to turn one kind of data into another kind of data. *Manipulating data* means taking raw input and doing something with it to achieve a desired result. (As with data science, this is a topic covered in *Python for Data Science For Dummies,* by John Paul Mueller and Luca Massaron [Wiley].) For example, until you do something with traffic data, you can't see the patterns that emerge that tell you where to spend additional money in improvements. The traffic data in its raw form does nothing to inform you — you must manipulate it to see the pattern in a useful manner. Therefore, those arcane symbols are useful after all. You use them as a sort of machine to turn raw data into something helpful, which is what you discover in this chapter.

In times past, people actually had to perform the various manipulations to make data useful by hand, which required advanced knowledge of math. Fortunately, you can find Python packages to perform most of these manipulations using a little code. You don't have to memorize arcane manipulations anymore — just know which Python features to use. That's what this chapter helps you achieve. You discover the means to perform various kinds of data manipulations using easily accessed Python packages designed especially for the purpose. The chapter begins with vector and matrix manipulations. Later sections discuss techniques such as recursion that can make the tasks even simpler and perform some tasks that are nearly impossible using other means. You also discover how to speed up the calculations so that you spend less time manipulating the data and more time doing something really interesting with it, such as discovering just how to keep quite so many traffic jams from occurring.

## Performing Calculations Using Vectors and Matrixes

To perform useful work with Python, you often need to work with larger amounts of data that comes in specific forms. These forms have odd-sounding names, but the names are quite important. The three terms you need to know for this chapter are as follows:

- **Scalar:** A single base data item. For example, the number 2 shown by itself is a scalar.
- **Vector:** A one-dimensional array (essentially a list) of data items. For example, an array containing the numbers 2, 3, 4, and 5 would be a vector. You access items in a vector using a zero-based *index,* a pointer to the item you want. The item at index 0 is the first item in the vector, which is 2 in this case.
- **Matrix:** A two-or-more-dimensional array (essentially a table) of data items. For example, an array containing the numbers 2, 3, 4, and 5 in the first row and 6, 7, 8, and 9 in the second row is a matrix. You access items in a matrix using a zero-based row-and-column index. The item at row 0, column 0 is the first item in the matrix, which is 2 in this case.

Python provides an interesting assortment of features on its own, as described in Chapter 4, but you'd still need to do a lot of work to perform some tasks. To reduce the amount of work you do, you can rely on code written by other people and found in packages. The following sections describe how to use the NumPy package to perform various tasks on scalars, vectors, and matrixes.

## Understanding scalar and vector operations

The NumPy package provides essential functionality for scientific computing in Python. To use `numpy`, you import it using a command such as `import numpy as np`. Now you can access `numpy` using the common two-letter abbreviation `np`.

REMEMBER

Python provides access to just one data type in any particular category. For example, if you need to create a variable that represents a number without a decimal portion, you use the integer data type. Using a generic designation like this is useful because it simplifies code and gives the developer a lot less to worry about. However, in scientific calculations, you often need better control over how data appears in memory, which means having more data types, something that `numpy` provides for you. For example, you might need to define a particular scalar as a `short` (a value that is 16 bits long). Using `numpy`, you could define it as `myShort = np.short(15)`. You could define a variable of precisely the same size using the `np.int16` function. The NumPy package provides access to a side assortment of data types described at `https://docs.scipy.org/doc/numpy/reference/arrays.scalars.html`.

Use the `numpy array` function to create a vector. For example, `myVect = np.array([1, 2, 3, 4])` creates a vector with four elements. In this case, the vector contains standard Python integers. You can also use the `arange` function to produce vectors, such as `myVect = np.arange(1, 10, 2)`, which fills `myVect` with `array([1, 3, 5, 7, 9])`. The first input tells the starting point, the second the stopping point, and the third the step between each number. A fourth argument lets you define the data type for the vector. You can also create a vector with a specific data type. All you need to do is specify the data type like this: `myVect = np.array(np.int16([1, 2, 3, 4]))` to fill `myVect` with a vector like this: `array([1, 2, 3, 4], dtype=int16)`.

In some cases, you need special `numpy` functions to create a vector (or a matrix) of a specific type. For example, some math tasks require that you fill the vector with ones. In this case, you use the `ones` function like this: `myVect = np.ones(4, dtype=np.int16)` to fill `myVect` with ones of specific data types like this: `array([1, 1, 1, 1], dtype=int16)`. You can also use a `zeros` function to fill a vector with zeros.

TIP

You can perform basic math functions on vectors as a whole, which makes this incredibly useful and less prone to errors that can occur when using programming constructs such as loops to perform the same task. For example, `myVect + 1` produces an output of `array([2, 3, 4, 5])` when working with standard Python integers. If you choose to work with the `numpy int16` data type, `myVect + 1` produces `array([2, 3, 4, 5], dtype=int16)`. Note that the output tells you specifically which data type is in use. As you might expect, `myVect - 1` produces an output of `array([0, 1, 2, 3])`. You can even use vectors in more complex math scenarios, such as `2 ** myVect`, where the output is `array([ 2,  4,  8, 16], dtype=int32)`. When used in this manner, however, `numpy` often assigns a specific type to the output, even when you define a vector using standard Python integers.

As a final thought on scalar and vector operations, you can also perform both logical and comparison tasks. For example, the following code performs comparison operations on two arrays:

```
a = np.array([1, 2, 3, 4])
b = np.array([2, 2, 4, 4])

a == b
array([False,  True, False,  True], dtype=bool)

a < b
array([ True, False,  True, False], dtype=bool)
```

Starting with two vectors, `a` and `b`, the code checks whether the individual elements in a equal those in b. In this case, `a[0]` doesn't equal `b[0]`. However, `a[1]` does equal `b[1]`. The output is a vector of type `bool` that contains true or false values based on the individual comparisons. Likewise, you can check for instances when `a < b` and produce another vector containing the truth-values in this instance.

Logical operations rely on special functions. You check the logical output of the Boolean operators AND, OR, XOR, and NOT. Here is an example of the logical functions:

```
a = np.array([True, False, True, False])
b = np.array([True, True, False, False])

np.logical_or(a, b)
array([ True,  True,  True, False], dtype=bool)

np.logical_and(a, b)
array([ True, False, False, False], dtype=bool)

np.logical_not(a)
array([False,  True, False,  True], dtype=bool)

np.logical_xor(a, b)
array([False,  True,  True, False], dtype=bool)
```

You can also use numeric input to these functions. When using numeric input, a 0 is false and a 1 is true. As with comparisons, the functions work on an element-by-element basis even though you make just one call. You can read more about the logic functions at https://docs.scipy.org/doc/numpy-1.10.0/reference/routines.logic.html.

## Performing vector multiplication

Adding, subtracting, or dividing vectors occurs on an element-by-element basis, as described in the previous section. However, when it comes to multiplication, things get a little odd. In fact, depending on what you really want to do, things can become quite odd indeed. Consider the sort of multiplication discussed in the previous section. Both myVect * myVect and np.multiply(myVect, myVect) produce an element-by-element output of array([ 1,  4,  9, 16]).

WARNING

Unfortunately, an element-by-element multiplication can produce incorrect results when working with algorithms. In many cases, what you really need is a *dot product,* which is the sum of the products of two number sequences. When working with vectors, the dot product is always the sum of the individual element-by-element multiplications and results in a single number. For example, myVect.dot(myVect) results in an output of 30. If you sum the values from the element-by-element multiplication, you find that they do indeed add up to 30. The discussion at https://www.mathsisfun.com/algebra/vectors-dot-product.html tells you about dot products and helps you understand where they might fit in with algorithms. You can learn more about the linear algebra manipulation functions for numpy at https://docs.scipy.org/doc/numpy/reference/routines.linalg.html.

## Creating a matrix is the right way to start

Many of the same techniques you use with vectors also work with matrixes. To create a basic matrix, you simply use the array function as you would with a vector, but you define additional dimensions. A *dimension* is a direction in the matrix. For example, a two-dimensional matrix contains rows (one direction) and columns (a second direction). The array call myMatrix = np.array([[1,2,3], [4,5,6], [7,8,9]]) produces a matrix containing three rows and three columns, like this:

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Note how you embed three lists within a container list to create the two dimensions. To access a particular array element, you provide a row and column index value, such as myMatrix[0, 0] to access the first value of 1. You can produce matrixes with any number of dimensions using a similar technique. For example, myMatrix = np.array([[[1,2], [3,4]], [[5,6], [7,8]]]) produces a three-dimensional matrix with an x, y, and z axis that looks like this:

```
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])
```

In this case, you embed two lists, within two container lists, within a single container list that holds everything together. In this case, you must provide an x, y, and z index value to access a particular value. For example, myMatrix[0, 1, 1] accesses the value 4.

TIP

In some cases, you need to create a matrix that has certain start values. For example, if you need a matrix filled with ones at the outset, you can use the ones function. The call to myMatrix = np.ones([4,4], dtype=np.int32) produces a matrix containing four rows and four columns filled with int32 values, like this:

```
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])
```

Likewise, a call to myMatrix = np.ones([4,4,4], dtype=np.bool) will create a three-dimensional array. This time, the matrix will contain Boolean values of True. There are also functions for creating a matrix filled with zeros, the identity matrix, and for meeting other needs. You can find a full listing of vector and matrix array-creation functions at https://docs.scipy.org/doc/numpy/reference/routines.array-creation.html.

REMEMBER

The NumPy package supports an actual matrix class. The matrix class supports special features that make it easier to perform matrix-specific tasks. You discover these features later in the chapter. For now, all you really need to know is how to create a matrix of the matrix data type. The easiest method is to make a call similar to the one you use for the array function, but using the mat function instead, such as myMatrix = np.mat([[1,2,3], [4,5,6], [7,8,9]]), which produces the following matrix:

```
matrix([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
```

You can also convert an existing array to a matrix using the `asmatrix` function. Use the `asarray` function to convert a `matrix` object back to an `array` form.

WARNING

The only problem with the `matrix` class is that it works on only two-dimensional matrixes. If you attempt to convert a three-dimensional matrix to the `matrix` class, you see an error message telling you that the shape is too large to be a matrix.

## Multiplying matrixes

Multiplying two matrixes involves the same concerns as multiplying two vectors (as discussed in the "Performing vector multiplication" section, earlier in this chapter). The following code produces an element-by-element multiplication of two matrixes.

```
a = np.array([[1,2,3],[4,5,6]])
b = np.array([[1,2,3],[4,5,6]])

a*b
array([[ 1,  4,  9],
       [16, 25, 36]])
```

WARNING

Note that `a` and `b` are the same shape, two rows and three columns. To perform an element-by-element multiplication, the two matrixes must be the same shape. Otherwise, you see an error message telling you that the shapes are wrong. As with vectors, the `multiply` function also produces an element-by-element result.

Dot products work completely differently with matrixes. In this case, the number of columns in matrix `a` must match the number of rows in matrix `b`. However, the number of rows in matrix `a` can be any number, and the number of columns in matrix `b` can be any number as long as you multiply `a` by `b`. For example, the following code produces a correct dot product:

```
a = np.array([[1,2,3],[4,5,6]])
b = np.array([[1,2,3],[3,4,5],[5,6,7]])

a.dot(b)
array([[22, 28, 34],
       [49, 64, 79]])
```

Note that the output contains the number of rows found in matrix a and the number of columns found in matrix b. So how does this all work? To obtain the value found in the output array at index [0,0] of 22, you sum the values of a[0,0]*b[0,0] (which is 1), a[0,1]*b[1,0] (which is 6), and a[0,2]*b[2,0] (which is 15) to obtain the value of 22. The other entries work precisely the same way.

TIP

An advantage of using the NumPy matrix class is that some tasks become more straightforward. For example, multiplication works precisely as you expect it should. The following code produces a dot product using the matrix class:

```
a = np.mat([[1,2,3],[4,5,6]])
b = np.mat([[1,2,3],[3,4,5],[5,6,7]])

a*b
matrix([[22, 28, 34],
        [49, 64, 79]])
```

The output with the * operator is the same as using the dot function with an array. This example also points out that you must know whether you're using an array or a matrix object when performing tasks such as multiplying two matrixes.

TIP

To perform an element-by-element multiplication using two matrix objects, you must use the numpy multiply function.

## Defining advanced matrix operations

This book takes you through all sorts of interesting matrix operations, but you use some of them commonly, which is why they appear in this chapter. When working with arrays, you sometimes get data in a shape that doesn't work with the algorithm. Fortunately, numpy comes with a special reshape function that lets you put the data into any shape needed. In fact, you can use it to reshape a vector into a matrix, as shown in the following code:

```
changeIt = np.array([1,2,3,4,5,6,7,8])

changeIt
array([1, 2, 3, 4, 5, 6, 7, 8])

changeIt.reshape(2,4)
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

```
changeIt.reshape(2,2,2)
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])
```

REMEMBER

The starting shape of `changeIt` is a vector, but using the `reshape` function turns it into a matrix. In addition, you can shape the matrix into any number of dimensions that work with the data. However, you must provide a shape that fits with the required number of elements. For example, calling `changeIt.reshape(2,3,2)` will fail because there aren't enough elements to provide a matrix of that size.

You may encounter two important matrix operations in some algorithm formulations. They are the transpose and inverse of a matrix. *Transposition* occurs when a matrix of shape n x m is transformed into a matrix m x n by exchanging the rows with the columns. Most texts indicate this operation by using the superscript *T*, as in $A^T$. You see this operation used most often for multiplication in order to obtain the right dimensions. When working with `numpy`, you use the `transpose` function to perform the required work. For example, when starting with a matrix that has two rows and four columns, you can transpose it to contain four rows with two columns each, as shown in this example:

```
changeIt
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])

np.transpose(changeIt)
array([[1, 5],
       [2, 6],
       [3, 7],
       [4, 8]])
```

You apply *matrix inversion* to matrixes of shape m x m, which are square matrixes that have the same number of rows and columns. This operation is quite important because it allows the immediate resolution of equations involving matrix multiplication, such as y=bX, where you have to discover the values in the vector b. Because most scalar numbers (exceptions include zero) have a number whose multiplication results in a value of 1, the idea is to find a matrix inverse whose multiplication will result in a special matrix called the *identity matrix.* To see an identity matrix in `numpy`, use the `identity` function like this:

```
np.identity(4)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

Note that an identity matrix contains all ones on the diagonal. Finding the inverse of a scalar is quite easy (the scalar number n has an inverse of $n^{-1}$ that is 1/n). It's a different story for a matrix. Matrix inversion involves quite a large number of computations. The inverse of a matrix A is indicated as $A^{-1}$. When working with `numpy`, you use the `linalg.inv` function to create an inverse. The following example shows how to create an inverse, use it to obtain a dot product, and then compare that dot product to the identity matrix by using the `allclose` function.

```
a = np.array([[1,2], [3,4]])
b = np.linalg.inv(a)

np.allclose(np.dot(a,b), np.identity(2))
True
```

REMEMBER

Sometimes, finding the inverse of a matrix is impossible. When a matrix cannot be inverted, it is referred to as a *singular matrix* or a *degenerate matrix.* Singular matrixes aren't the norm; they're quite rare.

# Creating Combinations the Right Way

Shaping data often involves viewing the data in multiple ways. Data isn't simply a sequence of numbers — it presents a meaningful sequence that, when ordered the proper way, conveys information to the viewer. Creating the right data combinations by manipulating data sequences is an essential part of making algorithms do what you want them to do. The following sections look at three data-shaping techniques: permutations, combinations, and repetitions.

## Distinguishing permutations

When you receive raw data, it appears in a specific order. The order can represent just about anything, such as the log of a data input device that monitors something like a production line. Perhaps the data is a series of numbers representing the number of products made at any particular moment in time. The reason that you receive the data in a particular order is important, but perhaps that order doesn't lend itself to obtaining the output you need from an algorithm. Perhaps creating a data *permutation,* a reordering of the data so that it presents a different view, will help achieve a desired result.

You can view permutations in a number of ways. One method of viewing a permutation is as a random presentation of the sequence order. In this case, you can use the `numpy` `random.permutation` function, as shown here:

```
a = np.array([1,2,3])
np.random.permutation(a)
array([2, 3, 1])
```

The output on your system will likely vary from the output shown. Each time you run this code, you receive a different random ordering of the data sequence, which comes in handy with algorithms that require you to randomize the dataset to obtain the desired results. For example, sampling is an essential part of data analytics, and the technique shown is an efficient way to perform this task.

Another way to view the issue is the need to obtain all the permutations for a dataset so that you can try each one in turn. To perform this task, you need to import the `itertools` package. The following code shows a technique you can use to obtain a list of all the permutations of a particular vector:

```
from itertools import permutations

a = np.array([1,2,3])

for p in permutations(a):
    print(p)

(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```

TIP

To save the list of sets, you could always create a blank list and rely on the `append` function to add each set to the list instead of printing the items one at a time, as shown in the code. The resulting list could serve as input to an algorithm designed to work with multiple sets. You can read more about `itertools` at `https://docs.python.org/3/library/itertools.html`.

## Shuffling combinations

In some cases, you don't need an entire dataset; all you really need are a few of the members in combinations of a specific length. For example, you might have a dataset containing four numbers and want only two number combinations from it.

(The ability to obtain parts of a dataset is a key function for generating a fully connected graph, which is described in Part 3 of the book.) The following code shows how to obtain such combinations:

```
from itertools import combinations

a = np.array([1,2,3,4])

for comb in combinations(a, 2):
    print(comb)

(1, 2)
(1, 3)
(1, 4)
(2, 3)
(2, 4)
(3, 4)
```

The output contains all the possible two-number combinations of `a`. Note that this example uses the `itertools` `combinations` function (the `permutations` function appears in the previous section). Of course, you might not need all those combinations; perhaps a random subset of them would work better. In this case, you can rely on the `random.sample` function to come to your aid, as shown here:

```
pool = []

for comb in combinations(a, 2):
    pool.append(comb)


random.sample(pool, 3)
[(1, 4), (3, 4), (1, 2)]
```

The precise combinations you see as output will vary. However, the idea is that you've limited your dataset in two ways. First, you're not using all the data elements all the time, and second, you're not using all the possible combinations of those data elements. The effect is to create a relatively random-looking set of data elements that you can use as input to an algorithm.

TIP

Another variation of this theme is to create a complete list but randomize the order of the elements. The act of randomizing the list order is shuffling, and you use the `random.shuffle` function to do it. In fact, Python provides a whole host of randomizing methods that you can see at `https://docs.python.org/3/library/random.html`. Many of the later examples in this book also rely on randomization to help obtain the correct output from algorithms.

## Facing repetitions

Repeated data can unfairly weight the output of an algorithm so that you get inaccurate results. Sometimes you need unique values to determine the outcome of a data manipulation. Fortunately, Python makes it easy to remove certain types of repeated data. Consider this example:

```
a = np.array([1,2,3,4,5,6,6,7,7,1,2,3])
b = np.array(list(set(a)))

b
array([1, 2, 3, 4, 5, 6, 7])
```

REMEMBER

In this case, `a` begins with an assortment of numbers in no particular order and with plenty of repetitions. In Python, a set never contains repeated data. Consequently, by converting the list in `a` to a `set` and then back to a `list`, and then placing that list in an `array`, you obtain a vector that has no repeats.

# Getting the Desired Results Using Recursion

Recursion is an elegant method of solving many computer problems that relies on the capability of a function to continue calling itself until it satisfies a particular condition. The term *recursion* actually comes from the Latin verb *recurrere*, which means to run back.

When you use recursion, you solve a problem by calling the same function multiple times but modifying the terms under which you call it. The main reason for using recursion is that it provides an easier way to solve problems when working with some algorithms because it mimics the way a human would solve it. Unfortunately, recursion is not an easy tool because it requires some effort to understand how to build a recursive routine and it can cause out-of-memory problems on your computer if you don't set some memory settings. The following sections detail how recursion works and give you an example of how recursion works in Python.

## Explaining recursion

Many people have a problem using recursion because they can't easily visualize how it works. In most cases, you call a Python function, it does something, and then it stops. However, in recursion, you call a Python function, it does something, and then it calls itself repeatedly until the task reaches a specific condition — but all those previous calls are still active. The calls unwind themselves one at a time until the first call finally ends with the correct answer, and this unwinding process is where most people encounter a problem. Figure 5-1 shows how recursion looks when using a flow chart.

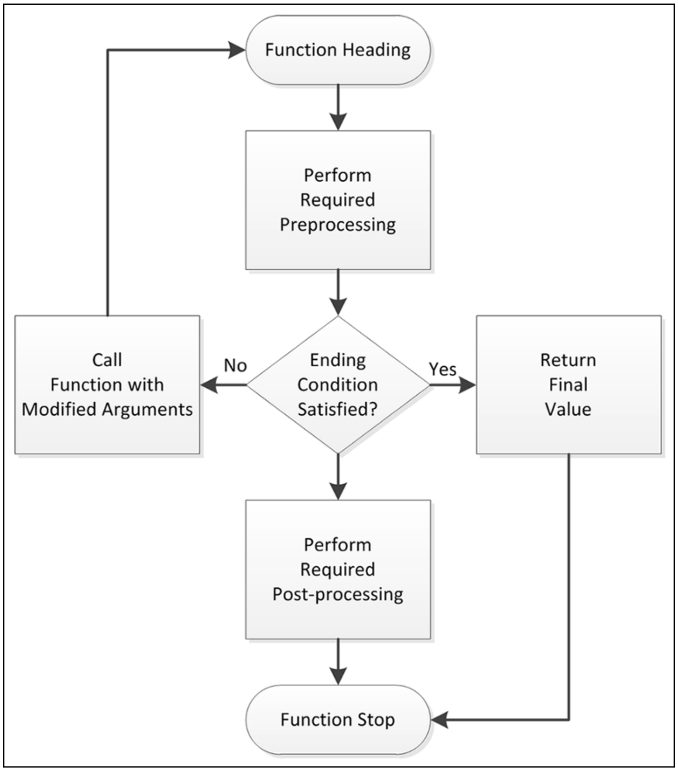

Function Heading
Perform Required Preprocessing
Call Function with Modified Arguments
No
Ending Condition Satisfied?
Yes
Return Final Value
Perform Required Post-processing
Function Stop

**FIGURE 5-1:** In the recursion process, a function continuously calls itself until it meets a condition.

Notice the conditional in the center. To make recursion work, the function must have such a conditional or it could become an endless loop. The conditional determines one of two things:

- The conditions for ending recursion haven't been met, so the function must call itself again.
- The conditions for ending recursion have been met, so the function returns a final value that is used to calculate the ending result.

REMEMBER

When a function calls itself, it doesn't use the same arguments that were passed to it. If it continuously used the same arguments, the condition would never change and the recursion would never end. Consequently, recursion requires that subsequent calls to the function must change the call arguments in order to bring the function closer to an ending solution.

One of the most common examples of recursion for all programming languages is the calculation of a factorial. A *factorial* is the multiplication of a series of numbers between a starting point and an ending point in which each number in the series is one less than the number before it. For example, to calculate 5! (read as five factorial) you multiple 5 * 4 * 3 * 2 * 1. The calculation represents a perfect and simple example of recursion. Here's the Python code you can use to perform the calculation. (You can find this code in the `A4D; 05; Recursion.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
def factorial(n):
    print("factorial called with n = ", str(n))
    if n == 1 or n == 0:
        print("Ending condition met.")
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))

factorial called with n =  5
factorial called with n =  4
factorial called with n =  3
factorial called with n =  2
factorial called with n =  1
Ending condition met.
120
```

The code meets the ending condition when `n == 1`. Each successive call to `factorial` uses `factorial(n-1)`, which reduces the starting argument by 1. The output shows each successive call to factorial and the meeting of the final condition. The result, 120, equals 5! (five factorial).

It's important to realize that there isn't just one method for using recursion to solve a problem. As with any other programming technique, you can find all sorts of ways to accomplish the same thing. For example, here's another version of the factorial recursion that uses fewer lines of code but effectively performs the same task:

```
def factorial(n):
    print("factorial called with n = ", str(n))
    if n > 1:
        return n * factorial(n-1)
    print("Ending condition met.")
    return 1

print(factorial(5))

factorial called with n =  5
factorial called with n =  4
factorial called with n =  3
factorial called with n =  2
factorial called with n =  1
Ending condition met.
120
```

REMEMBER

Note the difference. Instead of checking the ending condition, this version checks the continuation condition. As long as `n` is greater than `1`, the code will continue to make recursive calls. Even though this code is shorter than the previous version, it's also less clear because now you must think about what condition will end the recursion.

## Eliminating tail call recursion

Many forms of recursion rely on a tail call. In fact, the example in the previous section does. A *tail call* occurs any time the recursion makes a call to the function as the last thing before it returns. In the previous section, the line `return n * factorial(n-1)` is the tail call.

Tail calls aren't necessarily bad, and they represent the manner in which most people write recursive routines. However, using a tail call forces Python to keep track of the individual call values until the recursion rewinds. Each call consumes memory. At some point, the system will run out of memory and the call will fail, causing your algorithm to fail as well. Given the complexity and huge datasets used by some algorithms today, tail calls can cause considerable woe to anyone using them.

With a little fancy programming, you can potentially eliminate tail calls from your recursive routines. You can find a host of truly amazing techniques online, such as the use of a trampoline, as explained at `http://blog.moertel.com/posts/2013-06-12-recursion-to-iteration-4-trampolines.html`. However, the simplest approach to take when you want to eliminate recursion is to create an iterative alternative that performs the same task. For example, here is a `factorial` function that uses iteration instead of recursion to eliminate the potential for memory issues:

```
def factorial(n):
    print("factorial called with n = ", str(n))
    result = 1
    while n > 1:
        result = result * n
        n = n - 1
        print("Current value of n is ", str(n))
    print("Ending condition met.")
    return result

print(factorial(5))

factorial called with n =  5
Current value of n is  4
Current value of n is  3
Current value of n is  2
Current value of n is  1
Ending condition met.
120
```

The basic flow of this function is the same as the recursive function. A `while` loop replaces the recursive call, but you still need to check for the same condition and continue looping until the data meets the condition. The result is the same. However, replacing recursion with iteration is nontrivial in some cases, as explored in the example at `http://blog.moertel.com/posts/2013-06-03-recursion-to-iteration-3.html`.

# Performing Tasks More Quickly

Obviously, getting tasks done as quickly as possible is always ideal. However, you always need to carefully weigh the techniques you use to achieve this. Trading a little memory to perform a task faster is great as long as you have the memory to spare. Later chapters in the book explore all sorts of ways to perform tasks faster, but you can try some essential techniques no matter what sort of algorithm you're working with at any given time. The following sections explore some of these techniques.

## Considering divide and conquer

Some problems look overwhelming when you start them. Take, for example, writing a book. If you consider the entire book, writing it is an overwhelming task. However, if you break the book into chapters and consider just one chapter, the problem seems a little more doable. Of course, an entire chapter can seem a bit daunting, too, so you break the task down into first-level headings, which seems even more doable, but still not quite doable enough. The first-level headings could contain second-level headings and so on until you have broken down the problem of writing about a topic into short articles as much as you can. Even a short article can seem too hard, so you break it down into paragraphs, then into sentences, and finally into individual words. Writing a single word isn't too hard. So, writing a book comes down to writing individuals words —lots of them. This is how divide and conquer works. You break a problem down into smaller problems until you find a problem that you can solve without too much trouble.

Computers can use the divide-and-conquer approach as well. Trying to solve a huge problem with an enormous dataset could take days — assuming that the task is even doable. However, by breaking the big problem down into smaller pieces, you can solve the problem much faster and with fewer resources. For example, when searching for an entry in a database, searching the entire database isn't necessary if you use a sorted database. Say that you're looking for the word *Hello* in the database. You can begin by splitting the database in half (letters *A* through *M* and letters *N* through *Z*). The numeric value of the *H* in *Hello* (a value of 72 when using a standard ASCII table) is less than *M* (a value of 77) in the alphabet, so you look at the first half of the database rather than the second. Splitting the remaining half again (letters *A* through *G* and letters *H* through *M*), you now find that you need the second half of the remainder, which is now only a quarter of the database. Further splits eventually help you find precisely what you want by searching only a small fraction of the entire database. You call this search approach a *binary search.* The problem becomes one of following these steps:

1. Split the content in question in half.
2. Compare the keys for the content with the search term.
3. Choose the half that contains the key.
4. Repeat Steps 1 through 3 until you find the key.

REMEMBER

Most divide-and-conquer problems follow a similar approach, even though some of these approaches become quite convoluted. For example, instead of just splitting the database in half, you might split it into thirds in some cases. However, the goal is the same in all cases: Divide the problem into a smaller piece and determine whether you can solve the problem using just that piece as a generalized case. After you find the generalized case that you know how to solve, you can use that piece to solve any other piece as well. The following code shows an extremely simple version of a binary search that assumes that you have the list sorted. (You can find this code in the `A4D; 05; Binary Search.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
def search(searchList, key):
    mid = int(len(searchList) / 2)
    print("Searching midpoint at ", str(searchList[mid]))

    if mid == 0:
        print("Key Not Found!")
        return key

    elif key == searchList[mid]:
        print("Key Found!")
        return searchList[mid]

    elif key > searchList[mid]:
        print("searchList now contains ",
              searchList[mid:len(searchList)])
        search(searchList[mid:len(searchList)], key)

    else:
        print("searchList now contains ",
              searchList[0:mid])
        search(searchList[0:mid], key)

aList = list(range(1, 21))
search(aList, 5)

Searching midpoint at  11
searchList now contains  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Searching midpoint at  6
searchList now contains  [1, 2, 3, 4, 5]
Searching midpoint at  3
searchList now contains  [3, 4, 5]
Searching midpoint at  4
searchList now contains  [4, 5]
Searching midpoint at  5
Key Found!
```

This recursive approach to the binary search begins with `aList` containing the numbers 1 through 20. It searches for a value of `5` in `aList`. Each iteration of the recursion begins by looking for the list's midpoint, `mid`, and then using that midpoint to determine the next step. When the `key` matches the midpoint, the value is found in the list and the recursion ends.

REMEMBER

Note that this example makes one of two recursive calls. When `key` is greater than the midpoint value of the existing list, `searchList[mid]`, the code calls `search` again with just the right side of the remaining list. In other words, every call to `search` uses just half the list found in the previous call. When `key` is less than or equal to `searchList[mid]`, search receives the left half of the existing list.

WARNING

The list may not contain a search value, so you must always provide an escape method for the recursion or the stack will fill, resulting in an error message. In this case, the escape occurs when `mid == 0`, which means that there is no more `searchList` to search. For example, if you change `search(aList, 5)` to `search(aList, 22)`, you obtain the following output instead:

```
Searching midpoint at  11
searchList now contains  [11, 12, 13, 14, 15, 16, 17, 18,
 19, 20]
Searching midpoint at  16
searchList now contains  [16, 17, 18, 19, 20]
Searching midpoint at  18
searchList now contains  [18, 19, 20]
Searching midpoint at  19
searchList now contains  [19, 20]
Searching midpoint at  20
searchList now contains  [20]
Searching midpoint at  20
Key Not Found!
```

Note also that the code looks for the escape condition before performing any other work to ensure that the code doesn't inadvertently cause an error because of the lack of `searchList` content. When working with recursion, you must remain proactive or endure the consequences later.

## Distinguishing between different possible solutions

Recursion is part of many different algorithmic programming solutions, as you see in the upcoming chapters. In fact, it's hard to get away from recursion in many cases because an iterative approach proves nonintuitive, cumbersome, and time consuming. However, you can create a number of different versions of the same solution, each of which has its own characteristics, flaws, and virtues.

The solution that this chapter doesn't consider is sequential search, because a sequential search generally takes longer than any other solution you can employ. In a best-case scenario, a sequential search requires just one comparison to complete the search, but in a worst-case scenario, you find the item you want as the last check. As an average, sequential search requires `(n+1)/2` checks or O(n) time to complete.

The binary search in the previous section does a much better job than a sequential search does. It works on logarithmic time or O(log n). In a best-case scenario, it takes only one check, as with a sequential search, but the output from the example shows that even a worst-case scenario, where the value doesn't even appear in the list, takes only six checks rather than the 21 checks that a sequential search would require.

TIP

This book covers a wide variety of search and sort algorithms because searching and sorting represent two major categories of computer processing. Think about how much time you spend Googling data each day. In theory, you might spend entire days doing nothing but searching for data. Search routines work best with sorted data, so you see the need for efficient search and sort routines. Fortunately, you don't have to spend hours trying to figure out which search and sort routines work best. Sites such as Big-O Cheat Sheet, `http://bigocheatsheet.com/`, provide you with the data needed to determine which solution performs best.

WARNING

If you look at performance times alone, however, the data you receive can mislead you into thinking that a particular solution will work incredibly well for your application when in fact it won't. You must also consider the kind of data you work with, the complexity of creating the solution, and a host of other factors. That's why later examples in this book also consider the pros and cons of each approach — the hidden dangers of choosing a solution that seems to have potential and then fails to produce the desired result.

# 2 Understanding the Need to Sort and Search

## IN THIS PART . . .

Use various Python data structures.

Work with trees and graphs.

Sort data to make algorithms work faster.

Search data to locate precisely the right information quickly.

Employ hashing techniques to create smaller data indexes.

**IN THIS CHAPTER**

- Defining why data requires structure
- Working with stacks, queues, lists, and dictionaries
- Using trees to organize data
- Using graphs to represent data with relations

# Chapter 6

# Structuring Data

Raw data is just that: raw. It's not structured or cleaned in any way. You might find some parts of it missing or damaged in some way, or simply that it won't work for your problem. In fact, you're not entirely sure just what you're getting because it's raw.

Before you can do anything with most data, you must structure it in some manner so that you can begin to see what the data contains (and, sometimes, what it doesn't). Structuring data entails organizing it in some way so that all the data has the same attributes, appearance, and components. For example, you might get data from one source that contains dates in string form and another source that uses date objects. To use the information, you must make the kinds of data match. Data sources might also structure the data differently. One source might have the last and first name in a single field; another source might use individual fields for the same information. An important part of structuring data is organization. You aren't changing the data in any way — simply making the data more useful. (Structuring data contrasts with remediating or shaping the data where you sometimes do change values to convert one data type to another or experience a loss of accuracy, such as with dates, when moving between data sources.)

Python provides access to a number of organizational structures for data. The book uses these structures, especially stacks, queues, and dictionaries, for many of the examples. Each data structure provides a different means of working with the data and a different set of tools for performing tasks such as sorting the data into a particular order. This chapter presents you with the most common organizational methods, including both trees and graphs (both of which are so important that they appear in their own sections).

# Determining the Need for Structure

Structure is an essential element in making algorithms work. As shown in the binary search example in Chapter 5, implementing an algorithm using structured data is much easier than trying to figure out how to interpret the data in code. For example, the binary search example relies on having the data in sorted order. Trying to perform the required comparisons with unsorted data would require a lot more effort and potentially prove impossible to implement. With all this in mind, you need to consider the structural requirements for the data you use with your algorithms, as discussed in the following sections.

## Making it easier to see the content

An essential need to meet as part of working with data is to understand the data content. A search algorithm works only when you understand the dataset so that you know what to search for using the algorithm. Looking for words when the dataset contains numbers is an impossible task that always results in errors. Yet, search errors due to a lack of understanding of dataset content are a common occurrence even with the best search engines. Humans make assumptions about dataset content that cause algorithms to fail. Consequently, the better you can see and understand the content through structured formatting, the easier it becomes to perform algorithm-based tasks successfully.

However, even looking at the content is often error prone when dealing with humans and computers. For example, if you attempt to search for a number formatted as a string when the dataset contains the numbers formatted as integers, the search will fail. Computers don't automatically translate between strings and integers as humans do. In fact, computers see everything as numbers, and strings are only an interpretation imposed on the numbers by a programmer. Therefore, when searching for "1" (the string), the computer sees it as a request for the number 49 when using ASCII characters. To find the numeric value 1, you must search for a 1 as an integer value.

REMEMBER

Structure also enables you to discover nuanced data details. For example, a telephone number can appear in the form (555)555-1212. If you perform a search or other algorithm task using the form 1(555)555-1212, the search might fail because of the addition of a 1 at the beginning of the search term. These sorts of issues cause significant problems because most people see the two forms as equal, but the computer doesn't. The computer sees two completely different forms and even sees them as being two different lengths. Trying to impose form on humans rarely works and generally results in frustration that makes using the algorithm even harder, so structure imposed through data manipulation becomes even more important.

## Matching data from various sources

Interacting with data from a single source is one problem; interacting with data from several sources is quite another. However, datasets today generally come from more than one source, so you need to understand the complications that using multiple data sources can cause. When working with multiple data sources, you must do the following:

- Determine whether both datasets contain all the required data. Two designers are unlikely to create datasets that contain precisely the same data, in the same format, of the same type, and in the same order. Consequently, you need to consider whether the datasets provide the data you need or whether you need to remediate the data in some way to obtain the desired result, as discussed in the next section.
- Check both datasets for data type issues. One dataset could have dates input as strings, and another could have the dates input as actual date objects. Inconsistencies between data types will cause problems for an algorithm that expects data in one form and receives it in another.
- Ensure that all datasets place the same meaning on data elements. Data created by one source might have a different meaning than data created by another source. For example, the size of an integer can vary across sources, so you might see a 16-bit integer from one source and a 32-bit integer from another. Lower values have the same meaning, but the 32-bit integer can contain larger values, which can cause problems with the algorithm. Dates can also cause problems because they often rely on storing so many milliseconds since a given date (such as JavaScript, which stores the number of milliseconds since 01 January, 1970 UTC). The computer sees only numbers; humans add meaning to these numbers so that applications interpret them in specific ways.
- Verify the data attributes. Data items have specific attributes, which is why Chapter 4 tells you all about how Python interprets various data types. Chapter 5 points out that this interpretation can change when using `numpy`. In fact, you find that data attributes change between environments, and developers can change them even more by creating custom data types. To combine data from various sources, you must understand these attributes to ensure that you interpret the data correctly.

REMEMBER

The more time you spend verifying the compatibility of data from each of the sources you want to use for a dataset, the less likely you are to encounter problems when working with an algorithm. Data incompatibility issues don't always appear as outright errors. In some cases, an incompatibility can cause other issues, such as errant results that look correct but provide misleading information.

Combining data from multiple sources may not always mean creating a new dataset that looks precisely like the source datasets, either. In some cases, you create data aggregates or perform other forms of manipulation to create new data from the existing data. Analysis takes all sorts of forms, and some of the more exotic forms can produce terrible errors when used incorrectly. For example, one data source could provide general customer information and a second data source could provide customer-buying habits. Mismatches between the two sources might match customers with incorrect buying habit information and cause problems when you try to market new products to these customers. As an extreme example, consider what would happen when combining patient information from several sources and creating combined patient entries in a new data source with all sorts of mismatches. A patient without a history of a particular disease could end up with records showing diagnosis and care of the disease.

## Considering the need for remediation

After you find problems with your dataset, you need to remediate it so that the dataset works properly with the algorithms you use. For example, when working with conflicting data types, you must change the data types of each data source so that they match and then create the single data source used with the algorithm. Most of this remediation, although time consuming, is straightforward. You simply need to ensure that you understand the data before making changes, which means being able to see the content in the context of what you plan to do with it. However, you need to consider what to do in two special cases: data duplication and missing data. The following sections show how to deal with these issues.

### Dealing with data duplication

Duplicated data occurs for a number of reasons. Some of them are obvious. A user could enter the same data more than once. Distractions cause people to lose their place in a list or sometimes two users enter the same record. Some of the sources are less obvious. Combining two or more datasets could create multiple records when the data appears in more than one location. You could also create data duplications when using various data-shaping techniques to create new data from existing data sources. Fortunately, packages such as Pandas let you remove duplicate data, as shown in the following example. (You can find this code in the `A4D; 06; Remediation.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
import pandas as pd

df = pd.DataFrame({'A': [0,0,0,0,0,1,0],
                   'B': [0,2,3,5,0,2,0],
                   'C': [0,3,4,1,0,2,0]})
```

```
print(df, "\n")

df = df.drop_duplicates()
print(df)

   A  B  C
0  0  0  0
1  0  2  3
2  0  3  4
3  0  5  1
4  0  0  0
5  1  2  2
6  0  0  0

   A  B  C
0  0  0  0
1  0  2  3
2  0  3  4
3  0  5  1
5  1  2  2
```

The drop_duplicates function removes the duplicate records found in rows 4 and 6 in this example. By reading your data from a source into a pandas DataFrame, you can quickly remove the extra entries so that the duplicates don't unfairly weight the output of any algorithms you use.

## Dealing with missing values

Missing values can also skew the results of an algorithm's output. In fact, they can cause some algorithms to react oddly or even raise an error. The point is that missing values cause problems with your data, so you need to remove them. You do have many options when working with missing values. For example, you could simply set them to a standard value, such as 0 for integers. Of course, using a standard setting could also skew the results. Another approach is to use the mean of all the values, which tends to make the missing values not count. Using a mean is the approach taken in the following example

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'A': [0,0,1,None],
                   'B': [1,2,3,4],
                   'C': [np.NAN,3,4,1]},
                 dtype=int)
```

```
print(df, "\n")

values = pd.Series(df.mean(), dtype=int)
print(values, "\n")

df = df.fillna(values)
print(df)

      A  B    C
0     0  1  NaN
1     0  2    3
2     1  3    4
3  None  4    1

A    0
B    2
C    2
dtype: int32

   A  B  C
0  0  1  2
1  0  2  3
2  1  3  4
3  0  4  1
```

The `fillna` function enables you to get rid of the missing values whether they're not a number (NAN) or simply missing (None). You can supply the missing data values in a number of forms. This example relies on a series that contains the mean for each separate column of data (much as you would do when working with a database).

REMEMBER

Note that the code is careful not to introduce errors into the output by ensuring that `values` is of the right data type. Normally, the `mean` function outputs floating-point values, but you can force the series it fills into the right type. Consequently, the output not only lacks missing values but also does contain values of the correct type.

## Understanding other remediation issues

Remediation can take a number of other forms. Sometimes a user provides inconsistent or incorrect input. Applications don't always enforce data input rules, so users can enter incorrect state or region names. Misspellings also occur. Sometimes values are out of range or are simply impossible in a given situation. You may not always be able to clean your data completely on the first try. Often, you become aware of a problem by running the algorithm and noting that the results are skewed in some way or that the algorithm doesn't work at all (even if it did work on a subset of the data). When in doubt, check your data for potential remediation needs.

# Stacking and Piling Data in Order

Python provides a number of storage methodologies, as discussed in Chapter 4. As you've already seen in Chapter 5, and this chapter, packages often offer additional storage methods. Both NumPy and Pandas provide storage alternatives that you might consider when working through various data structuring problems.

REMEMBER

A common problem of data storage isn't just the fact that you need to store the data, but that you must store it in a particular order so that you can access it when necessary. For example, you may want to ensure that the first item you place on a stack of items to process is also the first item you actually do process. With this data-ordering issue in mind, the following sections describe the standard Python methods for ensuring orderly data storage that let you have a specific processing arrangement.

## Ordering in stacks

A stack provides last in/first out (LIFO) data storage. The NumPy package provides an actual stack implementation. In addition, Pandas associates stacks with objects such as the `DataFrame`. However, both packages hide the stack implementation details, and seeing how a stack works really does help. Consequently, the following example implements a stack using a standard Python `list`. (You can find this code in the `A4D; 06; Stacks, Queues, and Dictionaries.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
MyStack = []
StackSize = 3

def DisplayStack():
   print("Stack currently contains:")
   for Item in MyStack:
      print(Item)

def Push(Value):
   if len(MyStack) < StackSize:
      MyStack.append(Value)
   else:
      print("Stack is full!")

def Pop():
   if len(MyStack) > 0:
      print("Popping: ", MyStack.pop())
   else:
      print("Stack is empty.")

Push(1)
Push(2)
Push(3)
DisplayStack()

Push(4)

Pop()
DisplayStack()

Pop()
Pop()
Pop()

Stack currently contains:
1
2
3
Stack is full!
Popping:  3
Stack currently contains:
1
2
Popping:  2
Popping:  1
Stack is empty.
```

The example ensures that the stack maintains the integrity of the data and works with it in the order you expect. The code relies on simple `list` manipulation, but it's effective in providing a stack representation that you can use for any need.

TIP

Python lists are ordered lists of data values that are easy and intuitive to use. From an algorithm perspective, they often don't perform well because they store the list elements in computer memory and access them using an index and *memory pointers* (a number that provides the memory address of the data). They work exactly the way a book index or a package does. Lists don't have knowledge of their content.

When your application makes a data request, the list scans through all its items, which is even slower. When data is scattered across your computer's memory, lists must gather the data from each location individually and slowing access more.

## Using queues

Unlike stacks, queues are first in/first out (FIFO) data structures. As with stacks, you can find predefined implementations in many packages, including both NumPy and Pandas. Fortunately, you can also find a specific `queue` implementation in Python, which you find demonstrated in the following code:

```
import queue

MyQueue = queue.Queue(3)

print("Queue empty: ", MyQueue.empty())

MyQueue.put(1)
MyQueue.put(2)
MyQueue.put(3)
print("Queue full: ", MyQueue.full())

print("Popping: ", MyQueue.get())
print("Queue full: ", MyQueue.full())

print("Popping: ", MyQueue.get())
print("Popping: ", MyQueue.get())
print("Queue empty: ", MyQueue.empty())

Queue empty:  True
Queue full:  True
Popping:  1
Queue full:  False
Popping:  2
Popping:  3
Queue empty:  True
```

Using the built-in `queue` requires a lot less code than building a stack from scratch using a `list`, but notice how the two differ in output. The stack example pushes 1, 2, and 3 onto the stack, so the first value popped from the stack is 3. However, in this example, pushing 1, 2, and 3 onto the `queue` results in a first popped value of 1.

## Finding data using dictionaries

Creating and using a `dictionary` is much like working with a `list` except that you must now define a key and value pair. The great advantage of this data structure is that dictionaries can quickly provide access to specific data items using the key. There are limits to the kinds of keys you can use. Here are the special rules for creating a key:

- **The key must be unique.** When you enter a duplicate key, the information found in the second entry wins; the first entry replaces the second.
- **The key must be immutable.** This rule means that you can use strings, numbers, or tuples for the key. You can't, however, use a `list` for a key.

  REMEMBER

  The difference between mutable and immutable values is that immutable values can't change. To change the value of a string, for example, Python actually creates a new string that contains the new value and gives the new string the same name as the old one. It then destroys the old string.

TIP

Python dictionaries are the software implementation of a data structure called a *hash table*, an array that maps keys to values. Chapter 7 explains hashes in detail and how using hashes can help dictionaries perform faster. You have no restrictions on the values you provide. A value can be any Python object, so you can use a dictionary to access an employee record or other complex data. The following example helps you understand how to use dictionaries better:

```
Colors = {"Sam": "Blue", "Amy": "Red", "Sarah": "Yellow"}

print(Colors["Sarah"])
print(Colors.keys())

for Item in Colors.keys():
   print("{0} likes the color {1}."
      .format(Item, Colors[Item]))

Colors["Sarah"] = "Purple"
Colors.update({"Harry": "Orange"})
del Colors["Sam"]

print(Colors)

Yellow
dict_keys(['Sarah', 'Amy', 'Sam'])
Sarah likes the color Yellow.
Amy likes the color Red.
```

```
Sam likes the color Blue.
{'Harry': 'Orange', 'Sarah': 'Purple', 'Amy': 'Red'}
```

As you can see, a dictionary always has a key and value pair separated from each other by a colon (:). Instead of using an index to access individual values, you use the key. The special `keys` function lets you obtain a list of keys that you can manipulate in various ways. For example, you can use the keys to perform iterative processing of the data values that the dictionary contains.

TIP

Dictionaries are a bit like individual tables within a database. You can update, add, and delete records to a dictionary as shown. The `update` function can overwrite or add new entries to the dictionary.

# Working with Trees

A tree structure looks much like the physical object in the natural world. Using trees helps you organize data quickly and find it in a shorter time than using other data-storage techniques. You commonly find trees used for search and sort routines, but they have many other purposes as well. The following sections help you understand trees at a basic level. You find trees used in many of the examples in upcoming chapters.

## Understanding the basics of trees

Building a tree works much like building a tree in the physical world. Each item you add to the tree is a *node*. Nodes connect to each other using *links.* The combination of nodes and links forms a structure that looks much like a tree, as shown in Figure 6-1.

REMEMBER

Note that the tree has just one root node— just as with a physical tree. The *root node* provides the starting point for the various kinds of processing you perform. Connected to the root node are either branches or leaves. A *leaf node* is always an ending point for the tree. *Branch nodes* support either other branches or leaves. The type of tree shown in Figure 6-1 is a binary tree because each node has, at most, two connections.

In looking at the tree, Branch B is the child of the Root node. That's because the Root node appears first in the list. Leaf E and Leaf F are both children of Branch B, making Branch B the parent of Leaf E and Leaf F. The relationship between nodes is important because discussions about trees often consider the child/parent relationship between nodes. Without these terms, discussions of trees could become quite confusing.

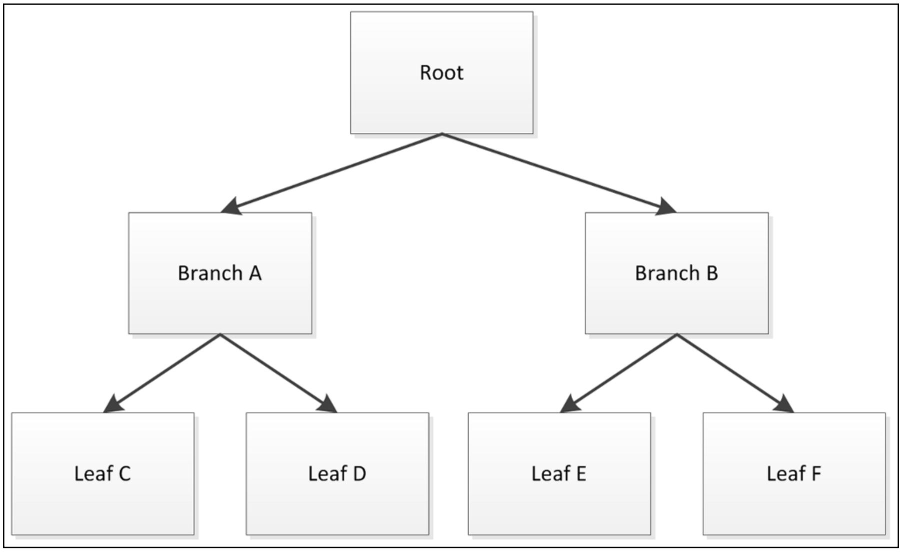


**FIGURE 6-1:** A tree in Python looks much like the physical alternative.

## Building a tree

Python doesn't come with a built-in tree object. You must either create your own implementation or use a tree supplied with a package. A basic tree implementation requires that you create a class to hold the tree data object. The following code shows how you can create a basic tree class. (You can find this code in the `A4D; 06; Trees.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
class binaryTree:
    def __init__(self, nodeData, left=None, right=None):
        self.nodeData = nodeData
        self.left  = left
        self.right = right

    def __str__(self):
        return str(self.nodeData)
```

All this code does is create a basic tree object that defines the three elements that a node must include: data storage, left connection, and right connection. Because leaf nodes have no connection, the default value for `left` and `right` is `None`. The class also includes a method for printing the content of `nodeData` so that you can see what data the node stores.

Using this simple tree requires that you not try to store anything in `left` or `right` other than a reference to another node. Otherwise, the code will fail because there isn't any error trapping. The nodeData entry can contain any value. The following code shows how to use the binaryTree class to build the tree shown in Figure 6-1:

```
tree = binaryTree("Root")
BranchA = binaryTree("Branch A")
BranchB = binaryTree("Branch B")
tree.left = BranchA
tree.right = BranchB

LeafC = binaryTree("Leaf C")
LeafD = binaryTree("Leaf D")
LeafE = binaryTree("Leaf E")
LeafF = binaryTree("Leaf F")
BranchA.left = LeafC
BranchA.right = LeafD
BranchB.left = LeafE
BranchB.right = LeafF
```

You have many options when building a tree, but building it from the top down (as shown in this code) or the bottom up (in which you build the leaves first) are two common methods. Of course, you don't really know whether the tree actually works at this point. *Traversing the tree* means checking the links and verifying that they actually do connect as you think they should. The following code shows how to use recursion (as described in Chapter 5) to traverse the tree you just built.

```
def traverse(tree):
    if tree.left != None:
        traverse(tree.left)
    if tree.right != None:
        traverse(tree.right)
    print(tree.nodeData)

traverse(tree)

Leaf C
Leaf D
Branch A
Leaf E
Leaf F
Branch B
Root
```

As the output shows, the `traverse` function doesn't print anything until it gets to the first leaf. It then prints both leaves and the parent of those leaves. The traversal follows the left branch first, and then the right branch. The root node comes last.

TECHNICAL STUFF

There are different kinds of data storage structures. Here is a quick list of the kinds of structures you commonly find:

- **Balanced trees:** A kind of tree that maintains a balanced structure through reorganization so that it can provide reduced access times. The number of elements on the left size differs from the number on the right side by at most one.
- **Unbalanced trees:** A tree that places new data items wherever necessary in the tree without regard to balance. This method of adding items makes building the tree faster but reduces access speed when searching or sorting.
- **Heaps:** A sophisticated tree that allows data insertions into the tree structure. The use of data insertion makes sorting faster. You can further classify these trees as max heaps and min heaps, depending on the tree's capability to immediately provide the maximum or minimum value present in the tree.

Later in the book, you find algorithms that use balanced trees, unbalanced trees, and heaps. For instance, Chapter 9 discusses the Dijkstra algorithm and Chapter 14 discusses Huffman encoding. As part of these discussions, the book provides pictures and code to explain how each data structure functions and its role in making the algorithm work.

# Representing Relations in a Graph

Graphs are another form of common data structure used in algorithms. You see graphs used in places like maps for GPS and all sorts of other places where the top down approach of a tree won't work. The following sections describe graphs in more detail.

## Going beyond trees

A graph is a sort of a tree extension. As with trees, you have nodes that connect to each other to create relationships. However, unlike binary trees, a graph can have more than one or two connections. In fact, graph nodes often have a multitude of connections. To keep things simple, though, consider the graph shown in Figure 6-2.

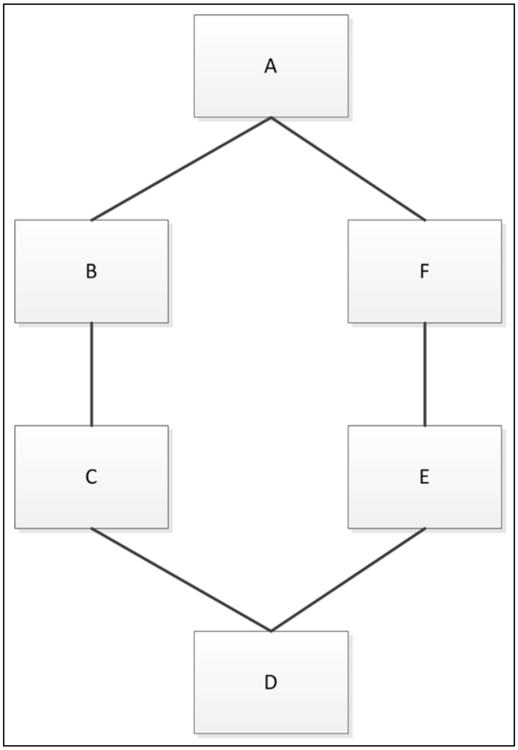


**FIGURE 6-2:** Graph nodes can connect to each other in myriad ways.

In this case, the graph creates a ring where A connects to both B and F. However, it need not be that way. A could be a disconnected node or could also connect to C. A graph shows connectivity between nodes in a way that is useful for defining complex relationships.

Graphs also add a few new twists that you might not have thought about before. For example, a graph can include the concept of directionality. Unlike a tree, which has parent/child relationships, a graph node can connect to any other node with a specific direction in mind. Think about streets in a city. Most streets are bidirectional, but some are one-way streets that allow movement in only one direction.

The presentation of a graph connection might not actually reflect the realities of the graph. A graph can designate a weight to a particular connection. The weight could define the distance between two points, define the time required to traverse the route, or provide other sorts of information.

## Building graphs

Most developers use dictionaries (or sometimes lists) to build graphs. Using a dictionary makes building the graph easy because the key is the node name and the values are the connections for that node. For example, here is a dictionary that creates the graph shown in Figure 6-2. (You can find this code in the `A4D; 06; Graphs.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
graph = {'A': ['B', 'F'],
         'B': ['A', 'C'],
         'C': ['B', 'D'],
         'D': ['C', 'E'],
         'E': ['D', 'F'],
         'F': ['E', 'A']}
```

This dictionary reflects the bidirectional nature of the graph in Figure 6-2. It could just as easily define unidirectional connections or provide nodes without any connections at all. However, the dictionary works quite well for this purpose, and you see it used in other areas of the book. Now it’s time to traverse the graph using the following code:

```
def find_path(graph, start, end, path=[]):
        path = path + [start]

        if start == end:
            print("Ending")
            return path

        for node in graph[start]:
            print("Checking Node ", node)

            if node not in path:
                print("Path so far ", path)

                newp = find_path(graph, node, end, path)
                if newp:
                    return newp

find_path(graph, 'B', 'E')

Checking Node  A
Path so far  ['B']
Checking Node  B
Checking Node  F
Path so far  ['B', 'A']
Checking Node  E
Path so far  ['B', 'A', 'F']
Ending

['B', 'A', 'F', 'E']
```

Later chapters discuss how to find the shortest path. For now, the code finds only a path. It begins by building the path node by node. As with all recursive routines, this one requires an exit strategy, which is that when the `start` value matches the `end` value, the path ends.

Because each node in the graph can connect to multiple nodes, you need a `for` loop to check each of the potential connections. When the node in question already appears in the path, the code skips it. Otherwise, the code tracks the current path and recursively calls `find_path` to locate the next node in the path.

IN THIS CHAPTER

- » Performing sorts using Mergesort and Quicksort
- » Conducting searches using trees and the heap
- » Considering the uses for hashing and dictionaries

# Chapter 7

# Arranging and Searching Data

Data surrounds you all the time. In fact, you really can't get away from it. Everything from the data needed to make business work to the nutritional guide on the side of your box of cereal relies on data. The four data operations are create, read, update, and delete (CRUD), which focus on the need to access the data you need to perform just about every task in life quickly and easily. That's why having the means to arrange and search data in a number of ways is essential. Unless you can access the data when you want in the manner you want, the CRUD required to make your business work will become quite cruddy indeed. Consequently, this is an especially important chapter for everyone who wants to make an application shine.

The first section of this chapter focuses on sorting data. Placing data in an order that makes it easy to perform CRUD operations is important because the less code you need to make data access work, the better. In addition, even though sorting data might not seem particularly important, sorted data makes searches considerably faster, as long as the sort matches the search. Sorting and searching go together: You sort the data in a way that makes searching faster.

The second section of the chapter discusses searching. You won't be surprised to learn that many different ways are available to search for data. Some of these techniques are slower than others; some have attributes that make them attractive to developers. The fact is that no perfect search strategy exists, but the exploration for such a method continues.

The final section of the chapter looks at hashing and dictionaries. The use of indexing makes sorting and searching significantly faster but also comes with trade-offs that you need to consider (such as the use of additional resources). An *index* is a kind of pointer or an address. It's not the data, but it points to the data, much as your address points to your home. A block-by-block manual search for your home in the city would be time consuming because the person looking for you would need to ask each person at each address whether you're there, but finding your address in the phone book and then using that address to locate your home is much faster.

# Sorting Data Using Mergesort and Quicksort

Sorting is one of the essentials of working with data. Consequently, a lot of people have come up with a lot of different ways in which to sort data over the years. All these techniques result in ordered data, but some work better than others do, and some work exceptionally well for specific tasks. The following sections help you understand the need for searching as well as consider the various search options.

## Defining why sorting data is important

A case can be made for not sorting data. After all, the data is still accessible, even if you don't sort it — and sorting takes time. Of course, the problem with unsorted data is the same problem as that junk drawer in your kitchen (or wherever you have your junk drawer — assuming that you can find it at all). Looking for anything in the junk drawer is time consuming because you can't even begin to guess where to find something. Rather than just reach in and take what you want, you must take out myriad other items that you don't want in an effort to find the one item you need. Unfortunately, the item you need might not be in the junk drawer in the first place—you might have thrown it out or put it in a different drawer.

The junk drawer in your home is just like unsorted data on your system. When the data is unsorted, you need to search one item at a time, and you don't even know whether you'll find what you need without searching every item in the dataset first. It's a frustrating way to work with data. The binary search example in the "Considering divide and conquer" section of Chapter 5 points out the need for sorting quite well. Imagine trying to find an item in a list without sorting it first. Every search becomes a time-consuming sequential search.

REMEMBER

Of course, simply sorting the data isn't enough. If you have an employee database sorted by last name, yet need to look up an employee by birth date, the sorting isn't useful. (Say you want to find all of the employees who have a birthday on a certain day.) To find the birth date you need, you must still search the entire dataset one item at a time. Consequently, sorting must focus on a particular need. Yes, you needed the employee database sorted by department at one point and by last name at another time, but now you need it sorted by birth date in order to use the dataset effectively.

The need to maintain several sorted orders for the same data is the reason that developers created indexes. Sorting a small index is faster than sorting the entire dataset. The index maintains a specific data order and points to the full dataset so that you can find what you need extremely fast. By maintaining an index for each sort requirement, you can effectively cut data access time and allow several people to access the data at the same time in the order in which they need to access it. The "Relying on Hashing" section, later in this chapter, gives you an idea of how indexing works and why you really need it in some cases, despite the additional time and resources needed to maintain the indexes.

TECHNICAL STUFF

Many ways are available to categorize sorting algorithms. One of these ways is the speed of the sort. When considering how effective a particular sort algorithm is at arranging the data, timing benchmarks typically look at two factors:

- **Comparisons:** To move data from one location in a dataset to another, you need to know where to move it, which means comparing the target data to other data in the dataset. Having fewer comparisons means better performance.
- **Exchanges:** Depending on how you write an algorithm, the data may not get to its final location in the dataset on the first try. The data might actually move several times. The number of exchanges affects speed considerably because now you're actually moving data from one location to another in memory. Fewer and smaller exchanges (such as when using indexes) means better performance.

## Ordering data naïvely

Ordering data naively means to order it using brute-force methods — without any regard whatsoever to making any kind of guess as to where the data should appear in the list. In addition, these techniques tend to work with the entire dataset instead of applying approaches that would likely reduce sorting time (such as the divide and conquer technique described in Chapter 5). However, these searches are also relatively easy to understand, and they use resources efficiently. Consequently, you shouldn't rule them out completely. Even though many searches fall into this category, the following sections look at the two most popular approaches.

## Using a selection sort

The selection sort replaced a predecessor, the bubble sort, because it tends to provide better performance than the bubble sort. Even though both sorts have a worst-case sort speed of $O(n^2)$, the selection sort performs fewer exchanges. A selection sort works in one of two ways: It either looks for the smallest item in the list and places it in the front of the list (ensuring that the item is in its correct location) or looks for the largest item and places it in the back of the list. Either way, the sort is exceptionally easy to implement and guarantees that items immediately appear in the final location once moved (which is why some people call it an in-place comparison sort). Here's an example of a selection sort. (You can find this code in the `A4D; 07; Sorting Techniques.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
data = [9, 5, 7, 4, 2, 8, 1, 10, 6, 3]

for scanIndex in range(0, len(data)):
    minIndex = scanIndex

    for compIndex in range(scanIndex + 1, len(data)):
        if data[compIndex] < data[minIndex]:
            minIndex = compIndex

    if minIndex != scanIndex:
        data[scanIndex], data[minIndex] = \
            data[minIndex], data[scanIndex]
        print(data)

[1, 5, 7, 4, 2, 8, 9, 10, 6, 3]
[1, 2, 7, 4, 5, 8, 9, 10, 6, 3]
[1, 2, 3, 4, 5, 8, 9, 10, 6, 7]
[1, 2, 3, 4, 5, 6, 9, 10, 8, 7]
[1, 2, 3, 4, 5, 6, 7, 10, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 10, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

## Switching to an insertion sort

An insertion sort works by using a single item as a starting point and adding items to the left or right of it based on whether these items are less than or greater than the selected item. As the number of sorted items builds, the algorithm checks new items against the sorted items and inserts the new item into the right position in the list. An insertion sort has a best-case sort speed of $O(n)$ and a worst case sort speed of $O(n^2)$.

TECHNICAL STUFF

An example of best-case sort speed is when the entire dataset is already sorted because the insertion sort won't have to move any values. An example of the worst-case sort speed is when the entire dataset is sorted in reverse order because every insertion will require moving every value that already appears in the output. You can read more about the math involved in this sort at `https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/analysis-of-insertion-sort`.

The insertion sort is still a brute-force method of sorting items, but it can require fewer comparisons than a selection sort. Here's an example of an insertion sort:

```
data = [9, 5, 7, 4, 2, 8, 1, 10, 6, 3]

for scanIndex in range(1, len(data)):
    temp = data[scanIndex]

    minIndex = scanIndex

    while minIndex > 0 and temp < data[minIndex - 1]:
        data[minIndex] = data[minIndex - 1]
        minIndex -= 1

    data[minIndex] = temp
    print(data)

[5, 9, 7, 4, 2, 8, 1, 10, 6, 3]
[5, 7, 9, 4, 2, 8, 1, 10, 6, 3]
[4, 5, 7, 9, 2, 8, 1, 10, 6, 3]
[2, 4, 5, 7, 9, 8, 1, 10, 6, 3]
[2, 4, 5, 7, 8, 9, 1, 10, 6, 3]
[1, 2, 4, 5, 7, 8, 9, 10, 6, 3]
[1, 2, 4, 5, 7, 8, 9, 10, 6, 3]
[1, 2, 4, 5, 6, 7, 8, 9, 10, 3]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

## Employing better sort techniques

As sort technology improves, the sort algorithms begin taking a more intelligent approach to getting data into the right order. The idea is to make the problem smaller and easier to manage. Rather than work with an entire dataset, smart sorting algorithms work with individual items, reducing the work required to perform the task. The following sections discuss two such smart sorting techniques.

## Rearranging data with Mergesort

A Mergesort works by applying the divide and conquer approach. The sort begins by breaking the dataset into individual pieces and sorting the pieces. It then merges the pieces in a manner that ensures that it has sorted the merged piece. The sorting and merging continues until the entire dataset is again a single piece. The worst-case sort speed of the Mergesort is O(n log n), which makes it considerably faster than the techniques used in the previous section (because log n is always smaller than n). This sort actually requires the use of two functions. The first function works recursively to split the pieces apart and put them back together again.

```
data = [9, 5, 7, 4, 2, 8, 1, 10, 6, 3]

def mergeSort(list):
    # Determine whether the list is broken into
    # individual pieces.
    if len(list) < 2:
        return list

    # Find the middle of the list.
    middle = len(list)//2

    # Break the list into two pieces.
    left = mergeSort(list[:middle])
    right = mergeSort(list[middle:])

    # Merge the two sorted pieces into a larger piece.
    print("Left side: ", left)
    print("Right side: ", right)
    merged = merge(left, right)
    print("Merged ", merged)
    return merged
```

The second function performs the actual task of merging the two sides using an iterative process. Here's the code used to merge the two pieces:

```
def merge(left, right):
    # When the left side or the right side is empty,
    # it means that this is an individual item and is
    # already sorted.
    if not len(left):
        return left
    if not len(right):
        return right
    # Define variables used to merge the two pieces.
    result = []
    leftIndex = 0
    rightIndex = 0
    totalLen = len(left) + len(right)

    # Keep working until all of the items are merged.
    while (len(result) < totalLen):

        # Perform the required comparisons and merge
        # the pieces according to value.
        if left[leftIndex] < right[rightIndex]:
            result.append(left[leftIndex])
            leftIndex+= 1
        else:
            result.append(right[rightIndex])
            rightIndex+= 1

        # When the left side or the right side is longer,
        # add the remaining elements to the result.
        if leftIndex == len(left) or \
            rightIndex == len(right):
                result.extend(left[leftIndex:]
                              or right[rightIndex:])
                break

    return result

mergeSort(data)
```

The `print` statements in the code help you see how the merging process works. Even though the process seems quite complex, it really is relatively straightforward when you work through the merging process shown here.

```
Left side:  [9]
Right side:  [5]
Merged  [5, 9]
Left side:  [4]
Right side:  [2]
Merged  [2, 4]
Left side:  [7]
Right side:  [2, 4]
Merged  [2, 4, 7]
Left side:  [5, 9]
```

```
Right side:  [2, 4, 7]
Merged  [2, 4, 5, 7, 9]
Left side:  [8]
Right side:  [1]
Merged  [1, 8]
Left side:  [6]
Right side:  [3]
Merged  [3, 6]
Left side:  [10]
Right side:  [3, 6]
Merged  [3, 6, 10]
Left side:  [1, 8]
Right side:  [3, 6, 10]
Merged  [1, 3, 6, 8, 10]
Left side:  [2, 4, 5, 7, 9]
Right side:  [1, 3, 6, 8, 10]
Merged  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

## Solving sorting issues the best way using Quicksort

The Quicksort is one of the fastest methods of sorting data. In reading about Mergesort and Quicksort online, you find that some people prefer to use one over the other in a given situation. For example, most people feel that a Quicksort works best for sorting arrays, and Mergesort works best for sorting linked lists (see the discussion at `http://www.geeksforgeeks.org/why-quick-sort-preferred-for-arrays-and-merge-sort-for-linked-lists/`). Tony Hoare wrote the first version of Quicksort in 1959, but since that time, developers have written many other versions of Quicksort. The average sort time of a Quicksort is O(n log n), but the worst-case sort time is $O(n^2)$.

The first part of the task is to partition the data. The code chooses a pivot point that determines the left and right side of the sort. Here is the partitioning code for this example:

```
data = [9, 5, 7, 4, 2, 8, 1, 10, 6, 3]

def partition(data, left, right):
    pivot = data[left]
    lIndex = left + 1
    rIndex = right
    while True:
        while lIndex <= rIndex and data[lIndex] <= pivot:
            lIndex += 1
        while rIndex >= lIndex and data[rIndex] >= pivot:
            rIndex -= 1
        if rIndex <= lIndex:
            break
        data[lIndex], data[rIndex] = \
            data[rIndex], data[lIndex]
        print(data)

    data[left], data[rIndex] = data[rIndex], data[left]
    print(data)
    return rIndex
```

TECHNICAL STUFF

## UNDERSTANDING QUICKSORT WORST-CASE PERFORMANCE

Quicksort seldom incurs the worst-case sort time. However, even modified versions of the Quicksort can have a worst-case sort time of O(n$^2$) when one of these events occurs:

- The dataset is already sorted in the desired order.
- The dataset is sorted in reverse order.
- All the elements in the dataset are the same.

All these problems occur because of the pivot point that a less intelligent sort function uses. Fortunately, using the right programming technique can mitigate these problems by defining something other than the leftmost or rightmost index as the pivot point. The techniques that modern Quicksort versions rely on include:

- Choosing a random index
- Choosing the middle index of the partition
- Choosing the median of the first, middle, and last element of the partition for the pivot (especially for longer partitions)

The inner loop of this example continuously searches for elements that are in the wrong place and swaps them. When the code can no longer swap items, it breaks out of the loop and sets a new pivot point, which it returns to the caller. This is the iterative part of the process. The recursive part of the process handles the left and right side of the dataset, as shown here:

```
def quickSort(data, left, right):
    if right <= left:
        return
    else:
        pivot = partition(data, left, right)
        quickSort(data, left, pivot-1)
        quickSort(data, pivot+1, right)

    return data

quickSort(data, 0, len(data)-1)
```

The amount of comparisons and exchanges for this example are relatively small compared to the other examples. Here is the output from this example:

```
[9, 5, 7, 4, 2, 8, 1, 3, 6, 10]
[6, 5, 7, 4, 2, 8, 1, 3, 9, 10]
[6, 5, 3, 4, 2, 8, 1, 7, 9, 10]
[6, 5, 3, 4, 2, 1, 8, 7, 9, 10]
[1, 5, 3, 4, 2, 6, 8, 7, 9, 10]
[1, 5, 3, 4, 2, 6, 8, 7, 9, 10]
[1, 2, 3, 4, 5, 6, 8, 7, 9, 10]
[1, 2, 3, 4, 5, 6, 8, 7, 9, 10]
[1, 2, 3, 4, 5, 6, 8, 7, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

# Using Search Trees and the Heap

Search trees enable you to look for data quickly. Chapter 5 introduces you to the idea of a binary search, and the "Working with Trees" section of Chapter 6 helps you understand trees to some extent. Obtaining data items, placing them in sorted order in a tree, and then searching that tree is one of the faster ways to find information.

A special kind of tree structure is the *binary heap*, which places each of the node elements in a special order. The root node always contains the smallest value. When viewing the branches, you see that upper-level branches are always a smaller value than lower-level branches and leaves. The effect is to keep the tree balanced and in a predictable order so that searching becomes extremely efficient. The cost is in keeping the tree balanced. The following sections describe how search trees and the heap work in detail.

## Considering the need to search effectively

Of all the tasks that applications do, searching is the more time consuming and also the one required most. Even though adding data (and sorting it later) does require some amount of time, the benefit to creating and maintaining a dataset comes from using it to perform useful work, which means searching it for important information. Consequently, you can sometimes get by with less efficient CRUD functionality and even a less-than-optimal sort routine, but searches must proceed as efficiently as possible. The only problem is that no one search performs every task with absolute efficiency, so you must weigh your options based on what you expect to do as part of the search routines.

Two of the more efficient methods of searching involve the use of the binary search tree (BST) and binary heap. Both of the search techniques rely on a tree-like structure to hold the keys used to access data elements. However, the arrangement of the two methods is different, which is why one has advantages over the other when performing certain tasks. Figure 7-1 shows the arrangement for a BST.

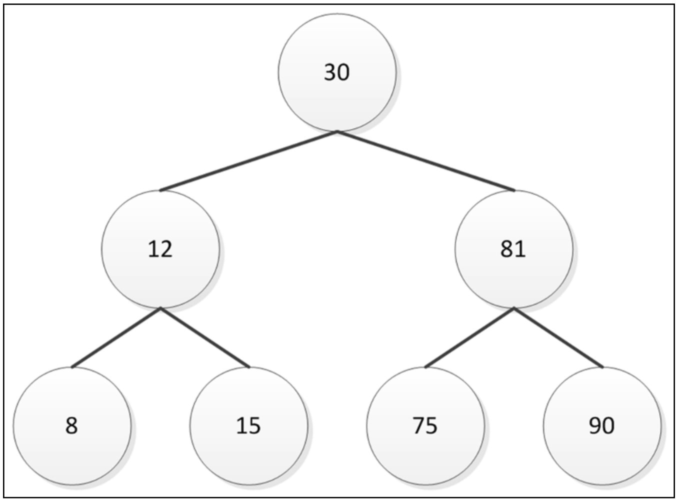


**FIGURE 7-1:** The arrangement of keys when using a BST.

Note how the keys follow an order in which lesser numbers appear to the left and greater numbers appear to the right. The root node contains a value that is in the middle of the range of keys, giving the BST an easily understood balanced approach to storing the keys. Contrast this arrangement to the binary heap shown in Figure 7-2.

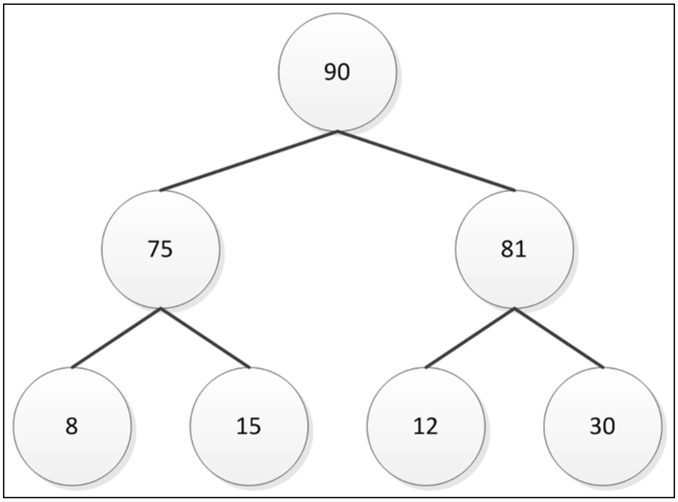


**FIGURE 7-2:** The arrangement of keys when using a binary heap.

Each level contains values that are less than the previous level, and the root contains the maximum key value for the tree. In addition, in this particular case, the lesser values appear on the left and the greater on the right (although this order isn't strictly enforced). The figure actually depicts a *binary max heap.* You can also create a *binary min heap* in which the root contains the lowest key value and each level builds to higher values, with the highest values appearing as part of the leaves.

REMEMBER

As previously noted, BST has some advantages over binary heap when used to perform a search. The following list provides some of the highlights of these advantages:

- Searching for an element requires O(log n) time, contrasted to O(n) time for a binary heap.
- Printing the elements in order requires only O(log n) time, contrasted to O(n log n) time for a binary heap.
- Finding the floor and ceiling requires O(log n) time.
- Locating Kth smallest/largest element requires O(log n) time when the tree is properly configured.

Whether these times are important depends on your application. BST tends to work best in situations in which you spend more time searching and less time building the tree. A binary heap tends to work best in dynamic situations in which keys change regularly. The binary heap also offers advantages, as described in the following list:

- Creating the required structures requires fewer resources because binary heaps rely on arrays, making them cache friendlier as well.
- Building a binary heap requires O(n) time, contrasted to BST, which requires O(n log n) time.
- Using pointers to implement the tree isn't necessary.
- Relying on binary heap variations (for example, the Fibonacci Heap) offers advantages such as increase and decrease key times of O(1) time.

## Building a binary search tree

You can build a BST using a variety of methods. Some people simply use a dictionary; others use custom code (see the article at `https://interactivepython.org/courselib/static/pythonds/Trees/SearchTreeImplementation.html` and `http://code.activestate.com/recipes/577540-python-binary-search-tree/` as examples). However, most developers don't want to reinvent the wheel when it comes to BST. With this in mind, you need a package, such as `bintrees`, which provides all the required functionality to create and interact with BST using a minimum of code. To download and install `bintrees`, open a command prompt, type **pip install bintrees**, and press Enter. You see `bintrees` installed on your system. The documentation for this package appears at `https://pypi.python.org/pypi/bintrees/2.0.6`.

You can use `bintrees` for all sorts of needs, but the example in this section looks specifically at a BST. In this case, the tree is unbalanced. The following code shows how to build and display a BST using `bintrees`. (You can find this code in the `A4D; 07; Search Techniques.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
from bintrees import BinaryTree

data = {3:'White', 2:'Red', 1:'Green', 5:'Orange',
        4:'Yellow', 7:'Purple', 0:'Magenta'}

tree = BinaryTree(data)
tree.update({6:'Teal'})

def displayKeyValue(key, value):
    print('Key: ', key, 'Value: ', value)

tree.foreach(displayKeyValue)
print('Item 3 contains: ', tree.get(3))
print('The maximum item is: ', tree.max_item())
```

```
Key:  0 Value:  Magenta
Key:  1 Value:  Green
Key:  2 Value:  Red
Key:  3 Value:  White
Key:  4 Value:  Yellow
Key:  5 Value:  Orange
Key:  6 Value:  Teal
Key:  7 Value:  Purple
Item 3 contains:  White
The maximum item is:  (7, 'Purple')
```

To create a binary tree, you must supply key and value pairs. One way to perform this task is to create a dictionary as shown. After you create the tree, you can use the `update` function to add new entries. The entries must include a key and value pair as shown.

TIP

This example uses a function to perform a task with the data in `tree`. In this case, the function merely prints the key and value pairs, but you could use the tree as input to an algorithm for analysis (among other tasks). The function, `displayKeyValue`, acts as input to the `foreach` function, which displays the key and value pairs as output. You also have access to myriad other features, such as using `get` to obtain a single item or `max_item` to obtain the maximum item stored in `tree`.

## Performing specialized searches using a binary heap

As with BST, you have many ways to implement a binary heap. Writing one by hand or using a dictionary does work well, but relying on a package makes things considerably faster and more reliable. The `heapq` package comes with Python, so you don't even need to install it. You can find the documentation for this package at `https://docs.python.org/3/library/heapq.html`. The following example shows how to build and search a binary heap using `heapq`:

```
import heapq

data = {3:'White', 2:'Red', 1:'Green', 5:'Orange',
        4:'Yellow', 7:'Purple', 0:'Magenta'}

heap = []
for key, value in data.items():
    heapq.heappush(heap, (key, value))
heapq.heappush(heap, (6, 'Teal'))
heap.sort()
```

```
for item in heap:
    print('Key: ', item[0], 'Value: ', item[1])
print('Item 3 contains: ', heap[3][1])
print('The maximum item is: ', heapq.nlargest(1, heap))

Key:  0 Value:  Magenta
Key:  1 Value:  Green
Key:  2 Value:  Red
Key:  3 Value:  White
Key:  4 Value:  Yellow
Key:  5 Value:  Orange
Key:  6 Value:  Teal
Key:  7 Value:  Purple
Item 3 contains:  White
The maximum item is:  [(7, 'Purple')]
```

The example code performs the same tasks and provides the same output as the example in the previous section, except that it relies on a binary heap in this case. The dataset is the same as before. However, note the difference in the way you add the data to the heap using `heappush`. In addition, after adding a new item, you must call `sort` to ensure that the items appear in sorted order. Manipulating the data is much like manipulating a list, as contrasted to the dictionary approach used for `bintrees`. Whichever approach you use, it pays to choose an option that works well with the application you want to create and provides the fastest possible search times for the tasks you perform.

# Relying on Hashing

A major problem with most sort routines is that they sort all the data in a dataset. When the dataset is small, you hardly notice the amount of data that the sort routine attempts to move. However, as the dataset gets larger, the data movement becomes noticeable as you sit staring at the screen for hours on end. A way around this problem is to sort just the key information. A *key* is the identifying data for a particular data record. When you interact with an employee record, the employee name or number usually serves as a key for accessing all the other information you have about the employee. It's senseless to sort all the employee information when you really need only the keys sorted, which is what using hashing is all about. When working with these data structures, you gain a major speed advantage by sorting the smaller amount of data presented by the keys, rather than the records as a whole.

## Putting everything into buckets

Until now, the search and sort routines in the book work by performing a series of comparisons until the algorithm finds the correct value. The act of performing comparisons slows the algorithms because each comparison takes some amount of time to complete.

A smarter way to perform the task involves predicting the location of a particular data item in the data structure (whatever that structure might be) before actually looking for it. That's what a *hash table* does —provides the means to create an index of keys that points to individual items in a data structure so that an algorithm can easily predict the location of the data. Placing keys into the index involves using a *hash function* that turns the key into a numeric value. The numeric value acts as an index into the hash table, and the hash table provides a pointer to the full record in the dataset. Because the hash function produces repeatable results, you can predict the location of the required data. In many cases, a hash table provides a search time of O(1). In other words, you need only one comparison to find the data.

REMEMBER

A hash table contains a specific number of *slots* that you can view as buckets for holding data. Each slot can hold one data item. The number of filled slots when compared to the number of available slots is the *load factor.* When the load factor is high, the potential for *collisions* (where two data entries have the same hash value) becomes greater as well. The next section of the chapter discusses how to avoid collisions, but all you really need to know for now is that they can occur.

One of the more typical methods for calculating the hash value for an input is to obtain the modulus of the value divided by the number of slots. For example, if you want to store the number 54 into a hash table containing 15 slots, the hash value is 9. Consequently, the value 54 goes into slot 9 of the hash table when the slots are numbers from 0 through 14 (given that the table has 15 slots). A real hash table will contain a considerably greater number of slots, but 15 works fine for the purposes of this section. After placing the item into the hash slot, you can use the hash function a second time to find its location.

TIP

Theoretically, if you have a perfect hash function and an infinite number of slots, every value you present to the hash function will produce a unique value. In some cases, the hash calculation can become quite complex to ensure unique values most of the time. However, the more complex the hash calculation, the less benefit you receive from hashing, so keeping things simple is the best way to go.

Hashing can work with all sorts of data structures. However, for the purposes of demonstration, the following example uses a simple list to hold the original data and a second list to hold the resulting hash. (You can find this code in the `A4D; 07; Hashing.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
data = [22, 40, 102, 105, 23, 31, 6, 5]
hash_table = [None] * 15
tblLen = len(hash_table)

def hash_function(value, table_size):
    return value % table_size

for value in data:
    hash_table[hash_function(value, tblLen)] = value

print(hash_table)

[105, 31, None, None, None, 5, 6, 22, 23, None, 40, None,
 102, None, None]
```

To find a particular value again, you just run it through `hash_function`. For example, `print(hash_table[hash_function(102, tblLen)])` displays `102` as output after locating its entry in `hash_table`. Because the hash values are unique in this particular case, `hash_function` can locate the needed data every time.

## Avoiding collisions

A problem occurs when two data entries have the same hash value. If you simply write the value into the hash table, the second entry will overwrite the first, resulting in a loss of data. *Collisions,* the use of the same hash value by two values, require you to have some sort of strategy in mind for handling them. Of course, the best strategy is to avoid the collision in the first place.

One of the methods for avoiding collisions is to ensure that you have a large enough hash table. Keeping the load factor low is your first line of defense against having to become creative in the use of your hash table. However, even with a large table, you can't always avoid collisions. Sometimes the potential dataset is so large, but the used dataset is so small, that avoiding the problem becomes impossible. For example, if you have a school with 400 children and rely on their social security number for identification, collisions are inevitable because no one is going to create a hash table with a billion entries for that many children. The waste of memory would be enormous. Consequently, a hash function may have to use more than just a simple modulus output to create the hash value. Here are some techniques you can use to avoid collisions:

- **Partial values:** When working with some types of information, part of that information repeats, which can create collisions. For example, the first three digits of a telephone number can repeat for a given area, so removing those numbers and using just the remaining four may help solve a collision problem.

- **Folding:** Creating a unique number might be as easy as dividing the original number into pieces, adding the pieces together, and using the result for the hash value. For example, using the telephone number 555-1234, the hash could begin by breaking it into pieces: 55 51 234, and then adding the result together to obtain 340 as the number used to generate the hash.
- **Mid-square:** The hash squares the value in question, uses some number of digits from the center of the resulting number, and discards the rest of those digits. For example, consider the value 120. When squared, you get 14,400. The hash could use 440 to generate the hash value and discard the 1 from the left and the 0 from the right.

Obviously, there are as many ways to generate the hash as someone has imagination to create them. Unfortunately, no amount of creativity is going to solve every collision problem, and collisions are still likely to occur. Therefore, you need another plan. When a collision does occur, you can use one of the following methods to address it:

- **Open addressing:** The code stores the value in the next open slot by looking through the slots sequentially until it finds an open slot to use. The problem with this approach is that it assumes an open slot for each potential value, which may not be the case. In addition, open addressing means that the search slows considerably after the load factor increases. You can no longer find the needed value on the first comparison.
- **Rehashing:** The code hashes the hash value plus a constant. For example, consider the value 1,020 when working with a hash table containing 30 slots and a constant of 100. The hash value in this case is 22. However, if slot 22 already contains a value, rehashing (`(22 + 100) % 30`) produces a new hash value of 2. In this case, you don't need to search the hash table sequentially for a value. When implemented correctly, a search might still include a low number of comparisons to find the target value.
- **Chaining:** Each slot in the hash table can hold multiple values. You can implement this approach by using a list within a list. Every time a collision occurs, the code simply appends the value to the list in the target slot. This approach offers the benefit of knowing that the hash will always produce the correct slot, but the list within that slot will still require some sort of sequential (or other) search to find the specific value.

## Creating your own hash function

You may at times need to create custom hash functions in order to meet the needs of the algorithm you use or to improve its performance. Apart from cryptographic uses (which deserve a book alone), Chapter 12 presents common algorithms that leverage different hash functions, such as the *Bloom Filter*, the *HyperLogLog*, and the *Count-Min Sketch*, that leverage the properties of custom hash functions to extract information from huge amounts of data.

You can find many examples of different hash functions in the Python `hashlib` package. The `hashlib` package contains algorithms such as these:

- **Secure Hash Algorithms (SHA):** These algorithms include SHA1, SHA224, SHA256, SHA384, and SHA512. Released by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS), SHA algorithms provide support for security applications and protocols.
- **RSA's MD5 algorithm:** Initially designed for security applications, this hash turned into a popular way to checksum files. Checksums reduce files to a single number that enables you to determine whether the file was modified since hash creation (it lets you determine whether the file you downloaded wasn't corrupted and hasn't been altered by a hacker). To ensure file integrity, just check whether the MD5 checksum of your copy corresponds to the original one communicated by the author of the file.

TIP

If `hashlib` isn't available on your Python installation, you can install the package using the `pip install hashlib` command from a command shell. The algorithms in `hashlib` work well for simple applications when used alone.

## DISCOVERING UNEXPECTED USES OF HASHES

Apart from the algorithms detailed in this book, other important algorithms are based on hashes. For instance, the *Locality-sensitive Hashing* (LSH) algorithm relies on a large number of hash functions to stitch apparently separated information together. If you wonder how marketing companies and intelligence services put different chunks of information together based on names and addresses that aren't identical (for example, guessing that "Los Angels," "Los Angles," and "Los Angleles" all refer to Los Angeles) the answer is LSH. LSH chunks the information to check into parts and digests it using many hash functions, resulting in the production of a special hash result, which is an address for a bucket used to hold similar words. LSH is quite complex in its implementation, but check out this material from the Massachusetts Institute of Technology (MIT): `http://www.mit.edu/~andoni/LSH/`.

However, you can combine the output of multiple hash functions when working with complex applications that rely on a large dataset. Simply sum the results of the various outputs after having done a multiplication on one or more of them. The sum of two hash functions treated in this way retains the qualities of the original hash functions even though the result is different and impossible to recover as the original elements of the sum. Using this approach means that you have a brand-new hash function to use as your secret hash recipe for algorithms and applications.

The following code snippet relies on the `hashlib` package and the `md5` and `sha1` hash algorithms. You just provide a number to use for the multiplication inside the hash sum. (Because numbers are infinite, you have a function that can produce infinite hashes.)

```
from hashlib import md5, sha1

def hash_f(element, i, length):
    """ Function to create many hash functions """
    h1 = int(md5(element.encode('ascii')).hexdigest(),16)
    h2 = int(sha1(element.encode('ascii')).hexdigest(),16)
    return (h1 + i*h2) % length

print (hash_f("CAT", 1, 10**5))
64018

print (hash_f("CAT", 2, 10**5))
43738
```

REMEMBER

If you wonder where to find other uses of hash tables around you, check out Python's dictionaries. Dictionaries are, in fact, hash tables, even though they have a smart way to deal with collisions and you won't lose your data because two hashed keys casually have the same result. The fact that the dictionary index uses a hash is also the reason for its speed in checking whether a key is present. In addition, the use of a hash explains why you can't use every data type as a key. The key you choose must be something that Python can turn into a hash result. Lists, for instance, are unhashable because they are mutable; you can change them by adding or removing elements. Nevertheless, if you transform your list into a string, you can use it as a key for a dictionary in Python.

# 3 Exploring the World of Graphs

## IN THIS PART . . .

Discover graph essentials that help you draw, measure, and analyze graphs.

Interact with graphs to locate nodes, sort graph elements, and find the shortest path.

Work with social media in graph form.

Explore graphs to find patterns and make decisions based on those patterns.

Use the PageRank algorithm to rate web pages.

IN THIS CHAPTER

- » Defining why networks are important
- » Demonstrating graph drawing techniques
- » Considering graph functionality
- » Using numeric formats to represent graphs

# Chapter 8
# Understanding Graph Basics

G*raphs* are structures that present a number of nodes (or vertexes) connected by a number of edges or arcs (depending on the representation). When you think about a graph, think about a structure like a map, where each location on the map is a node and the streets are the edges. This presentation differs from a tree where each path ends up in a leaf node. Remember from Chapter 7 that a tree could look like an organizational chart or a family hierarchy. Most important, tree structures actually do look like trees and have a definite start and a definite end. This chapter begins by helping you understand the importance of networks, which are a kind of graph commonly used for all sorts of purposes.

REMEMBER

You can represent graphs in all sorts of ways, most of them abstract. Unless you're really good at visualizing things in your mind (most people aren't), you need to know how to draw a graph so you can actually see it. People rely on their vision to understand how things work. The act of turning the numbers that represent a graph into a graphic visualization is *plotting.* Languages like Python excel at plotting because it's such an incredibly important feature. In fact, it's one of the reasons that this book uses Python rather than another language, such as C (which is good at performing a completely different set of tasks).

After you can visualize a graph, it's important to know what to do with the graphic representation. This chapter starts you off by measuring graph functionality.

You do things like count the edges and vertexes to determine things like graph complexity. Seeing a graph also enables you to perform tasks like computing centrality with greater ease. Of course, you build on what you discover in this chapter in Chapter 9.

The numeric presentation of a graph is important, even if it makes understanding the graph hard. The plot is for you, but the computer doesn't really understand the plot (despite having drawn it for you). Think of the computer as more of an abstract thinker. With the need to present a graph in a form that the computer can understand in mind, this chapter discusses three techniques for putting a graph into numeric format: matrixes, sparse representations, and lists. All these techniques have advantages and disadvantages, and you use them in specific ways in future chapters (beginning with Chapter 9). Other ways are also available to put a graph in numeric format, but these three methods will serve you well in communicating with the computer.

# Explaining the Importance of Networks

A *network* is a kind of graph that associates names with the vertexes (nodes or points), edges (arcs or lines), or both. Associating names with the graph features reduces the level of abstraction and makes understanding the graph easier. The data that the graph models becomes real in the mind of the person viewing it, even though the graph truly is an abstraction of the real world put into a form that both humans and computers can understand in different ways. The following sections help you understand the importance of networks better so that you can see how their use in this book simplifies the task of discovering how algorithms work and how you can benefit from their use.

## Considering the essence of a graph

Graphs appear as ordered pairs in the form G = (V,E), where G is the graph, V is a list of vertexes, and E is a list of edges that connect the vertexes. An edge is actually a numeric pair that expresses the two vertexes that it connects. Consequently, if you have two vertexes that represent cities, Houston (which equals 1) and Dallas (which equals 2), and you want to connect them with a road, then you create an edge, `Highway`, that contains a pair of vertex references, `Highway = [Houston, Dallas]`. The graph would appear as `G = [(Houston, Dallas)]`, which simply says that there is a first vertex, Houston, with a connection to Dallas, the second vertex. Using the order of presentation of the vertexes, Houston is adjacent to Dallas; in other words, a car would leave Houston and enter Dallas.

Graphs come in several forms. An *undirected graph* (as shown in Figure 8-1) is one in which the order of the edge entries doesn't matter. A road map would represent an undirected graph in most cases because traffic can travel along the road in both directions.

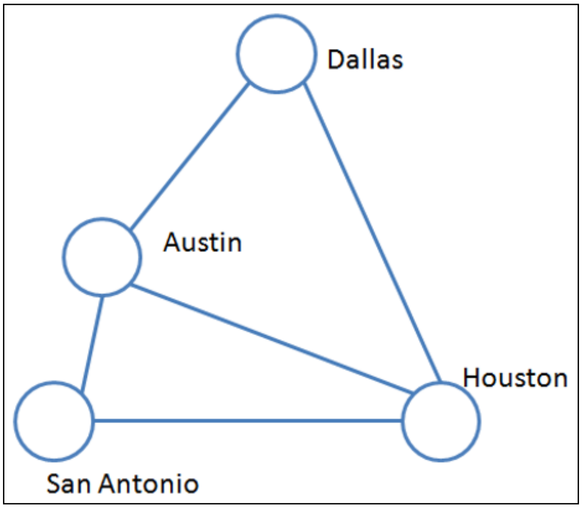


**FIGURE 8-1:** Presenting a simple undirected graph.

A *directed graph,* like the one shown in Figure 8-2, is one in which the order of the edge entries does matter because the flow is from the first entry to the second. In this case, most people call the edges *arcs* to differentiate them from undirected entries. Consider a graph representation of a traffic light sequence where Red equals 1, Yellow equals 2, and Green equals 3. The three arcs required to express the sequence are: `Go = [Red, Green]`, `Caution = [Green, Yellow]`, and `Stop = [Yellow, Red]`. The order of the entries is important because the flow from Go, to Caution, to Stop is important. Imagine the chaos that would result if the signal light chose to ignore the directed graph nature of the sequence.

A third essential kind of graph that you must consider is the mixed graph. Think about the road map again. It isn't always true that traffic flows both ways on all roads. When creating some maps, you must consider the presence of one-way streets. Consequently, you need both undirected and directed subgraphs in the same graph, which is what you get with a *mixed graph.*

Another graph type for your consideration is the *weighted graph* (shown in Figure 8-3), which is a graph that has values assigned to each of the edges or arcs. Think about the road map again. Some people want to know more than simply the direction to travel; they also want to know how far away the next destination is or how much time to allocate for getting there. A weighted graph provides this sort of information, and you use the weights in many different ways when performing calculations using graphs.

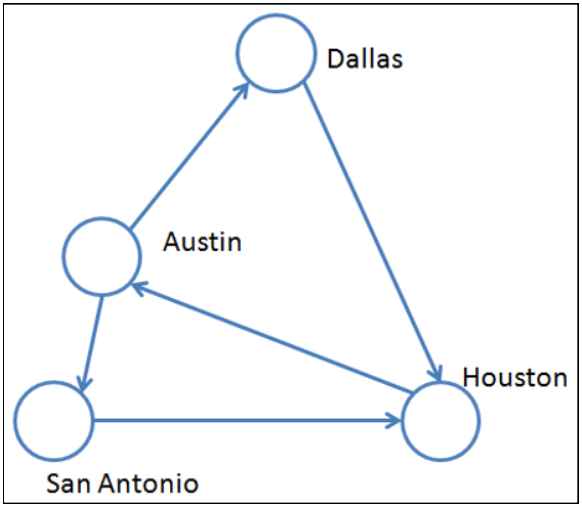


**FIGURE 8-2:** Creating the directed version of the same graph.

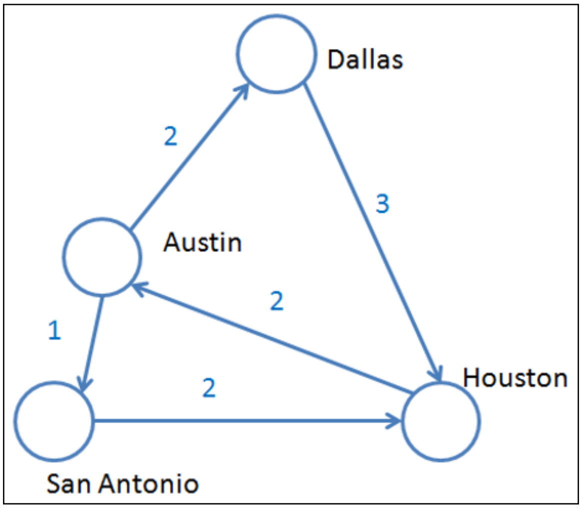


**FIGURE 8-3:** Using a weighted graph to make things more realistic.

Along with the weighted graph, you might also need a vertex-labeled graph when creating a road map. When working with a *vertex-labeled graph*, each vertex has a name associated with it. Consider looking at a road map where the mapmaker hasn't labeled the towns. Yes, you can see the towns, but you don't know which one is which without labels. You can find additional graph types described at `http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/Graphs.html`.

## Finding graphs everywhere

Graphs might seem like one of those esoteric math features that you found boring in school, but graphs are actually quite exciting because you use them all the time without really thinking about it. Of course, it helps to know that you won't normally deal with the numbers behind the graphs. Think about a map. What you see is a graph, but you see it in graphic format, with cities, roads, and all sorts of other features. The thing is, when you see a map, you think about a map, not a graph (but your GPS does see a graph, which is why it can always suggest the shortest route to your destination). If you were to start looking around, you'd find many common items that are graphs but are called something else.

REMEMBER

Some graphs aren't visual in nature, but you still don't see them as graphs. For example, telephone menu systems are a form of directional graph. In fact, for their seeming simplicity, telephone graphs are actually somewhat complex. They can include loops and all sorts of other interesting structures. Something you might try is to map out the graph for a menu system at some point. You might be surprised at just how complex some of them can be.

Another form of menu system appears as part of applications. To perform tasks, most applications take you through a series of steps in a special kind of subapplication called a wizard. The use of wizards make seemingly complex applications much easier to use, but to make the wizards work, the application developer must create a graph depicting the series of steps.

It may surprise you to find that even recipes in cookbooks are a kind of graph (and creating a pictorial representation of the relationships between ingredients can prove interesting). Each ingredient in the recipe is a node. The nodes connect using the edges created by the instructions for mixing the ingredients. Of course, a recipe is just a kind of chemistry, and chemical graphics show the relationship between elements in a molecule. (Yes, people actually are having this discussion; you can see one such thread at `http://stackoverflow.com/questions/7749073/representing-a-cooking-recipe-in-a-graph-database`.)

REMEMBER

The point is that you see these graphs all the time, but you don't see them as graphs — you see them as something else, such as a recipe or a chemical formula. Graphs can represent many kinds of relationships between objects, implying an order sequence, time dependence, or causality.

## Showing the social side of graphs

Graphs have social implications because they often reflect relationships between people in various settings. One of the most obvious uses of graphs is the organizational chart. Think about it. Each node is a different person in the organization, with edges connecting the nodes to show the various relationships between individuals. The same holds true for all sorts of graphs, such as those that show family history. However, in the first case, the graph is undirected because communication flows both ways between managers and subordinates (although the nature of the conversation differs based on direction). In the second case, the graph is directed because two parents bear children. The flow shows the direction of heredity from a founding member to the current children.

Social media benefits from the use of graphs as well. For example, a whole industry exists for analyzing the relationships between tweets on Twitter (see http://twittertoolsbook.com/10-awesome-twitter-analytics-visualization-tools/ for an example of just some of these tools). The analysis relies on the use of graphs to discover the relationships between individual tweets.

However, you don't have to look at anything more arcane than email to see graphs used for social needs. The Enron corpus includes the 200,399 email messages of 158 senior executives, dumped onto the Internet by the Federal Energy Regulatory Commission (FERC). Scientists and scholars have used this corpus to create many social graphs to disclose how the seventh largest company in the United States needed to file bankruptcy in 2001 (see https://www.technologyreview.com/s/515801/the-immortal-life-of-the-enron-e-mails/ to learn how this corpus has helped and is actually helping advance the analysis of complex graphs).

Even your computer has social graphs on it. No matter which email application you use, you can group emails in various ways, and these grouping methods normally rely on graphs to provide a structure. After all, trying to follow the flow of discussion without knowing which messages are responses to other messages is a lost cause. Yes, you could do it, but as the number of messages increases, the effort requires more and more time until it's wasted because of time constraints most people have.

## Understanding subgraphs

Relationships depicted by graphs can become quite complex. For example, when depicting city streets, most streets allow traffic in both directions, making an undirected graph perfect for representation purposes. However, some streets allow traffic in only one direction, which means that you need a directed graph in this case. The combination of two-way and one-way streets makes representation using a single graph type impossible (or at least inconvenient). Mixing undirected and directed graphs in a single graph means that you must create subgraphs to depict each graph type and then connect the subgraphs in a larger graph. Some graphs that contain subgraphs are so common that they have specific names, which is a mixed graph in this case.

Subgraphs are useful for other purposes as well. For example, you might want to analyze a loop within a graph, which means describing that loop as a subgraph. You don't need the entire graph, just the nodes and edges required to perform the analysis. All sorts of disciplines use this approach. Yes, developers use it to ensure that parts of an application work as expected, but city engineers also use it to understand the nature of traffic flow in a particularly busy section of the city.

Medical professionals also use subgraphs to understand the flow of blood or other liquids between organs in the body. The organs are the nodes and the blood vessels are the edges. In fact, many of these graphs are weighted — it's essential to know how much blood is flowing, not just that it's flowing.

Complex graphs can also hide patterns that you need to know about. For example, the same cycle can appear in multiple parts of the graph, or you might see the same cycle within different graphs. By creating a subgraph from the cycle, you can easily perform comparisons within the same graph or between graphs to see how they compare. For example, a biologist might want to compare the cycle of mutation for one animal against the cycle of mutation for another animal. To make this comparison, the biologist would need to create the representation as a subgraph of the processes for the entire animal. (You can see an interesting view of this particular use of graphs at `http://www.sciencedirect.com/science/article/pii/S1359027896000569`.). The graph appears near the beginning of the article as Figure 1.

# Defining How to Draw a Graph

A few people can visualize data directly in their minds. However, most people really do need a graphic presentation of the data in order to understand it. This point is made clear by the use of graphics in business presentations. You could tell others about last year's sales by presenting tables of numbers. After a while, most of your audience would nod off and you'd never get your point across. The reason is simple: The tables of numbers are precise and present a lot of information, but they don't do it in a way that people understand.

REMEMBER

Plotting the data and showing the sales numbers as a bar chart helps people see the relationships between the numbers with greater ease. If you want to point out that sales are increasing each year, a bar chart with bars of increasing length makes this point quickly. Interestingly enough, using the plot actually presents the data in a less accurate way. Trying to see that the company made $3,400,026.15 last year and $3,552,215.82 this year when looking at a bar chart is nearly impossible. Yes, the table would show this information, but people don't really need to know that level of detail — they simply need to see the annual increase, the contrast in earnings from year to year. However, your computer is interested in details, which is why plots are for humans and matrixes are for computers.

The following sections help you discover the wonders of plotting. You get a quick overview of how plots work with Python. Of course, these principles appear in later chapters in a more detailed form. These sections provide a start so that you can more easily understand the plots presented later.

## Distinguishing the key attributes

Before you can draw a graph, you need to know about graph attributes. As previously mentioned, graphs consist of nodes (or vertexes) and either edges (for undirected graphs) or arcs (for directed graphs). Any graph that you want to draw will contain these elements. However, how you represent these elements depends partly upon the package you choose to use. For the sake of simplicity, the book relies on a combination of two packages:

- **NetworkX (**`https://networkx.github.io/`**):** Contains code for drawing graphs.
- **matplotlib (**`http://matplotlib.org/`**):** Provides access to all sorts of drawing routines, some of which can display graphs created by NetworkX.

REMEMBER

To use packages in Python, you must import them. When you need to use external packages, you must add special code, such as the following lines of code that provide access to `matplotlib` and `networkx`. (You can find this code in the `A4D; 08; Draw Graph.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline
```

TIP

The special `%matplotlib inline` entry lets you see your plots directly in the Notebook rather than as an external graphic. Using this entry means that you can create a Notebook with graphics already included so that you don't have to run the code again to see the results you received in the past.

Now that you have access to the packages, you create a graph. In this case, a graph is a sort of container that holds the key attributes that define the graph. Creating a container lets you draw the graph so that you can see it later. The following code creates a NetworkX `Graph` object.

```
AGraph = nx.Graph()
```

Adding the key attributes to `AGraph` comes next. You must add both nodes and edges using the following code.

```
Nodes = range(1,5)
Edges = [(1,2), (2,3), (3,4), (4,5), (1,3), (1,5)]
```

As previously mentioned, `Edges` describe connections between `Nodes`. In this case, `Nodes` contains values from 1 through 5, so `Edges` contains connections between those values.

Of course, the Nodes and Edges are just sitting there now and won't appear as part of AGraph. You must put them into the container to see them. Use the following code to add the Nodes and Edges to AGraph.

```
AGraph.add_nodes_from(Nodes)
AGraph.add_edges_from(Edges)
```

The NetworkX package contains all sorts of functions you can use to interact with individual nodes and edges, but the approach shown here is the fastest way to do things. Even so, you might find that you want to add additional edges later. For example, you might want to add an edge between 2 and 4, in which case you would call the AGraph.add_edge(2, 4) function.

## Drawing the graph

You can interact in all sorts of ways with the AGraph container object that you created in the previous section, but many of those ways to interact are abstract and not very satisfying if you're a visually oriented person. Sometimes it's just nice to see what an object contains by looking at it. The following code displays the graph contained in AGraph:

```
nx.draw(AGraph, with_labels=True)
```

The draw() function provides various arguments that you can use to dress up the display, such as modifying the node color using the node_color argument and the edge color using the edge_color argument. Figure 8-4 shows the graph contained in AGraph.

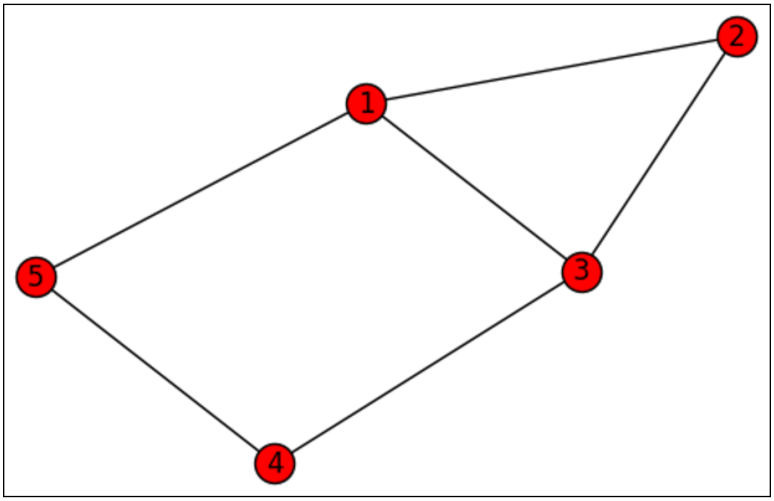


**FIGURE 8-4:** Seeing what a graph contains makes it easier to understand.

### DIFFERENCES IN FIGURE OUTPUT

Figure 8-4 shows typical output. However, your graph might appear to be slightly different from the one shown. For example, the triangle could appear at the bottom instead of the top, or the angles between the nodes could vary. The connections between the nodes matter most, so slight differences in actual appearance aren't important. Running the code several times would demonstrate that the orientation of the graph changes, along with the angles between edges. You see this same difference in other screenshots in the book. Always view the image with node connections in mind, rather than expecting a precise match between your output and the book's output.

# Measuring Graph Functionality

After you can visualize and understand a graph, you need to consider the question of which parts of the graph are important. After all, you don't want to spend your time performing analysis on data that doesn't really matter in the grand scheme of things. Think about someone who is analyzing traffic flow to improve the street system. The intersections represent vertexes and the streets represent edges along which the traffic flows. By knowing how the traffic flows, that is, which vertexes and edges see the most traffic, you can start thinking about which roads to widen and which need more repair because more traffic uses them.

However, just looking at individual streets isn't enough. A new skyscraper may bring with it a lot of traffic that affects an entire area. The skyscraper represents a central point around which traffic flow becomes more important. The most important vertexes are those central to the new skyscraper. Calculating *centrality*, the most important vertexes in a graph, can help you understand which parts of the graph require more attention. The following sections discuss the basic issues you must consider when measuring *graph functionality*, which is the capability of the graph to model a specific problem.

## Counting edges and vertexes

As graphs become more complex, they convey more information, but they also become harder to understand and manipulate. The number of edges and vertexes in a graph determines graph complexity. However, you use the combination of edges and vertexes to tell the full story. For example, you can have a node that isn't connected to the other nodes in any way. It's legal to create such a node in a graph to represent a value that lacks connections to the others. Using the following code, you can easily determine that node 6 has no connections to the others because it lacks any edge information. (You can find this code in the `A4D; 08; Graph Measurements.ipynb` file.)

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

AGraph = nx.Graph()

Nodes = range(1,5)
Edges = [(1,2), (2,3), (3,4), (4,5), (1,3), (1,5)]

AGraph.add_nodes_from(Nodes)
AGraph.add_edges_from(Edges)

AGraph.add_node(6)
sorted(nx.connected_components(AGraph))

[{1, 2, 3, 4, 5}, {6}]
```

The output from this code shows that nodes 1 through 5 are connected and that node 6 lacks a connection. Of course, you can remedy this situation by adding another edge by using the following code and then checking again:

```
AGraph.add_edge(1,6)
sorted(nx.connected_components(AGraph))

[{1, 2, 3, 4, 5, 6}]
```

The output now shows that every one of the nodes connects to at least one other node. However, you don't know which nodes have the most connections. The count of edges to a particular node is the *degree.* The higher the degree, the more complex the node becomes. By knowing the degree, you can develop an idea of which nodes are most important. The following code shows how to obtain the degree for the example graph.

```
nx.degree(AGraph).values()

dict_values([4, 2, 3, 2, 2, 1])
```

The degree values appear in node order, so node 1 has four connections and node 6 has only one connection. Consequently, node 1 is the most important, followed by node 3, which has three connections.

## USE OF WHITESPACE IN OUTPUT

The output for this example appears on two lines in the book, even though it appears on just one line in Jupyter Notebook. The addition of whitespace helps the output appear in a readable size on the page — it doesn't affect the actual information. Other examples in the book also show output on multiple lines, even when it appears on a single line in Jupyter Notebook.

When modeling real-world data, such as the tweets about a particular topic, the nodes also tend to cluster. You might think of this tendency as a kind of trending — what people feel is important now. The fancy math term for this tendency is *clustering*, and measuring this tendency helps you understand which group of nodes is most important in a graph. Here is the code you use to measure clustering for the example graph:

```
nx.clustering(AGraph)

{1: 0.16666666666666666, 2: 1.0, 3: 0.3333333333333333,
 4: 0.0, 5: 0.0, 6: 0.0}
```

The output shows that the nodes are most likely to cluster around node 2 even though node 1 has the highest degree. That's because both nodes 1 and 3 have high degrees and node 2 is between them.

REMEMBER

Clustering graphs helps aid understanding data. The technique helps show that there are nodes in the graph that are better connected and nodes that risk isolation. When you understand how elements connect in a graph, you can determine how to strengthen its structure or, on the contrary, destroy it. During the Cold war, military scientists from both the United States and the Soviet bloc studied graph clustering to better understand how to disrupt the other side's supply chain in case of a conflict.

## Computing centrality

Centrality comes in a number of different forms because importance often depends on different factors. The important elements of a graph when analyzing tweets will differ from the important elements when analyzing traffic flow. Fortunately, NetworkX provides you with a number of methods for calculating centrality. For example, you can calculate centrality based on node degrees. The following code uses the modified graph from the previous section of the chapter. (You can find this code in the `A4D; 08; Graph Centrality.ipynb` file.)

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

AGraph = nx.Graph()

Nodes = range(1,6)
Edges = [(1,2), (2,3), (3,4), (4,5), (1,3), (1,5), (1,6)]

AGraph.add_nodes_from(Nodes)
AGraph.add_edges_from(Edges)

nx.degree_centrality(AGraph)

{1: 0.8, 2: 0.4, 3: 0.6000000000000001, 4: 0.4, 5: 0.4,
 6: 0.2}
```

The values differ by the number of connections for each node. Because node 1 has four connections (it has the highest degree), it also has the highest centrality. You can see how this works by plotting the graph using a call to `nx.draw(AGraph, with_labels=True)`, as shown in Figure 8-5.

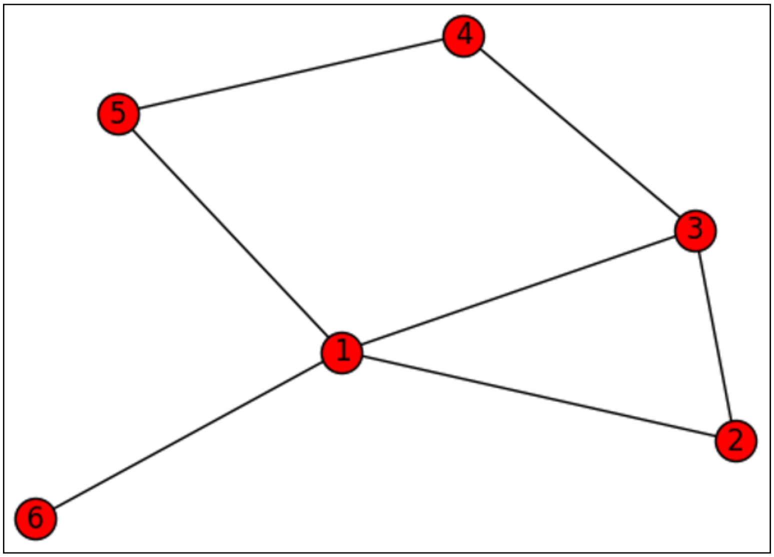


**FIGURE 8-5:** Plotting the graph can help you see degree centrality with greater ease.

Node 1 is indeed in the center of the graph with the most connections. The node 1 degree ensures that it's the most important based on the number of connections. When working with directed graphs, you can also use the `in_degree_centrality()` and `out_degree_centrality()` functions to determine degree centrality based on connection type rather than just the number of connections.

When working with traffic analysis, you might need to determine which locations are central based on their distance to other nodes. Even though a shopping center in the suburbs may have all sorts of connections to it, the fact that it is in the suburbs may reduce its impact on traffic. Yet, a supermarket in the center of the city with few connections might have a great impact on traffic because it's close to so many other nodes. To see how this works, add another node, 7, that is disconnected to the graph. The centrality of that node is infinite because no other node can reach it. The following code shows how to calculate the closeness centrality for the various nodes in the example graph:

```
AGraph.add_node(7)
nx.closeness_centrality(AGraph)

{1: 0.6944444444444445,
 2: 0.5208333333333334,
 3: 0.5952380952380952,
 4: 0.462962962962963,
 5: 0.5208333333333334,
 6: 0.4166666666666667,
 7: 0.0}
```

The output shows the centrality of each node in the graph based on its closeness to every other node. Notice that node 7 has a value of 0, which means that it's an infinite distance to every other node. On the other hand, node 1 has a high value because it's close to every node to which it has a connection. By calculating the closeness centrality, you can determine which nodes are the most important based on their location.

Another form of distance centrality is betweenness. Say that you're running a company that transfers goods throughout the city. You'd like to know which nodes have the greatest effect on these transfers. Perhaps you can route some traffic around this node to make your operation more specific. When calculating betweenness centrality, you determine the node that has the highest number of short paths coming to it. Here's the code used to perform this calculation (with the disconnected node 7 still in place):

```
nx.betweenness_centrality(AGraph)

{1: 0.36666666666666664,
 2: 0.0,
 3: 0.13333333333333333,
 4: 0.03333333333333333,
 5: 0.06666666666666667,
 6: 0.0,
 7: 0.0}
```

As you might expect, node 7 has no effect on transfer between other nodes because it has no connections to the other nodes. Likewise, because node 6 is a leaf node with only one connection to another node, it has no effect on transfers. Look again at Figure 8-5. The subgraph consisting of nodes 1, 3, 4, and 5 have the greatest effect on the transfer of items in this case. No connection exists between nodes 1 and 4, so nodes 3 and 5 act as intermediaries. In this case, node 2 acts like a leaf node.

TIP

NetworkX provides you with a number of other centrality functions. You find a complete list of these functions at `http://networkx.readthedocs.io/en/stable/reference/algorithms.centrality.html`. The important consideration is determining how you want to calculate importance. Considering centrality in light of the kind of importance you want to attach to the vertexes and edges in a graph is essential.

# Putting a Graph in Numeric Format

Precision is an important part of using algorithms. Even though too much precision hides the overall pictures from humans, computers thrive on detail. Often, the more detail you can provide, the better the results you receive. However, the form of that detail is important. To use certain algorithms, the data you provide must appear in certain forms or the result you receive won't make sense (it will contain errors or have other issues).

Fortunately, NetworkX provides a number of functions to convert your graph into forms that other packages and environments can use. These functions appear at `http://networkx.readthedocs.io/en/stable/reference/convert.html`. The following sections show how to present graph data as a NumPy (`http://www.numpy.org/`) matrix, SciPy (`https://www.scipy.org/`) sparse representation, and a standard Python list. You use these presentations as the book progresses to work with the various algorithms. (The code in the following sections appears in the `A4D; 08; Graph Conversion.ipynb` file and relies on the graph you created in the "Counting edges and vertexes" section of the chapter.)

## Adding a graph to a matrix

Using NetworkX, you can easily move your graph to a NumPy matrix and back again as needed to perform various tasks. You use NumPy to perform all sorts of data manipulation tasks. By analyzing the data in a graph, you might see patterns that wouldn't ordinarily be visible. Here's the code used to convert the graph into a matrix that NumPy can understand:

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

AGraph = nx.Graph()

Nodes = range(1,6)
Edges = [(1,2), (2,3), (3,4), (4,5), (1,3), (1,5), (1,6)]

AGraph.add_nodes_from(Nodes)
AGraph.add_edges_from(Edges)

nx.to_numpy_matrix(AGraph)

matrix([[ 0.,  1.,  1.,  0.,  1.,  1.],
        [ 1.,  0.,  1.,  0.,  0.,  0.],
        [ 1.,  1.,  0.,  1.,  0.,  0.],
        [ 0.,  0.,  1.,  0.,  1.,  0.],
        [ 1.,  0.,  0.,  1.,  0.,  0.],
        [ 1.,  0.,  0.,  0.,  0.,  0.]])
```

The resulting rows and columns show where connections exist. For example, there is no connection between node 1 and itself, so row 1, column 1, has a 0 in it. However, there is a connection between node 1 and node 2, so you see a 1 in row 1, column 2, and row 2, column 1 (which means that the connection goes both ways as an undirected connection).

The size of this matrix is affected by the number of nodes (the marix has as many rows and columns as nodes), and when it grows huge, it has many nodes to represent because the total number of cells is the square of the number of nodes. For instance, you can't represent the Internet using such a matrix because a conservative estimate calculates that at 10^10 websites, you'd need a matrix with 10^20 cells to store its structure, something impossible with the present computing capacity.

In addition, the number of nodes affects its content. If *n* is number of nodes, you find a minimum of (n–1) ones and a maximum of n(n–1) ones. The fact that the number of ones is few or large makes the graph dense or sparse, and that's relevant because if the connection between nodes are few, such as in the case of websites, more efficient solutions exist for storing graph data.

## Using sparse representations

The SciPy package also performs various math, scientific, and engineering tasks. When using this package, you can rely on a sparse matrix to hold the data. A sparse matrix is one in which only the actual connections appear in the matrix; all other entries don't exist. Using a sparse matrix saves resources because the memory requirements for a sparse matrix are small. Here is the code you use to create a SciPy sparse matrix from a NetworkX graph:

```
print(nx.to_scipy_sparse_matrix(AGraph))

  (0, 1)    1
  (0, 2)    1
  (0, 4)    1
  (0, 5)    1
  (1, 0)    1
  (1, 2)    1
  (2, 0)    1
  (2, 1)    1
  (2, 3)    1
  (3, 2)    1
  (3, 4)    1
  (4, 0)    1
  (4, 3)    1
  (5, 0)    1
```

As you can see, the entries show the various edge coordinates. Each active coordinate has a 1 associated with it. The coordinates are 0 based. This means that `(0, 1)` actually refers to a connection between nodes 1 and 2.

## Using a list to hold a graph

Depending on your needs, you might find that you also require the ability to create a dictionary of lists. Many developers use this approach to create code that performs various analysis tasks on graphs. You can see one such example at https://www.python.org/doc/essays/graphs/. The following code shows how to create a dictionary of lists for the example graph:

```
nx.to_dict_of_lists(AGraph)

{1: [2, 3, 5, 6], 2: [1, 3], 3: [1, 2, 4], 4: [3, 5],
 5: [1, 4], 6: [1]}
```

Notice that each node represents a dictionary entry, followed by a list of the nodes to which it connects. For example, node 1 connects to nodes 2, 3, 5, and 6.

IN THIS CHAPTER

- » Working with graphs
- » Performing sorting tasks
- » Reducing the tree size
- » Locating the shortest route between two points

# Chapter 9
# Reconnecting the Dots

This chapter is about working with graphs. You use graphs every day to perform a range of tasks. A *graph* is simply a set of vertexes, nodes, or points connected by edges, arcs, or lines. Putting this definition in simpler terms, every time you use a map, you use a graph. The starting point, intermediate points, and destination are all nodes. These nodes connect to each other with streets, which represent the lines. Using graphs enables you to describe relationships of various sorts. The reason that Global Positioning System (GPS) setups work is that you can use math to describe the relationships between points on the map and the streets that connect them. In fact, by the time you finish this chapter, you understand the basis used to create a GPS (but not necessarily the mechanics of making it happen). Of course, the fundamental requirement for using a graph to create a GPS is the capability to search for connections between points on the map, as discussed in the first section of the chapter.

To make sense of a graph, you need to sort the nodes, as described in the second section of the chapter, to create a specific organization. Without organization, making any sort of decision becomes impossible. An algorithm might end up going in circles or giving inconvenient output. For example, some early GPS setups didn't correctly find the shortest distance between two points, or sometimes ended up sending someone to the wrong place. Part of the reason for these problems is the need to sort the data so that you can view it in the same manner each time the algorithm traverses the nodes (providing you with a route between your home and your business).

When you view a map, you don't look at the information in the lower-right corner when you actually need to work with locations and roads in the upper-left corner.

A computer doesn't know that it needs to look in a specific place until you tell it to do so. To focus attention in a specific location, you need to reduce the graph size, as described in the third section of the chapter.

Now that the problem is simplified, an algorithm can find the shortest route between two points, as described in the fourth section of the chapter. After all, you don't want to spend any more time than is necessary in traffic fighting your way from home to the office (and back again). The concept of finding the shortest route is a bit more convoluted than you might think, so the fourth section looks at some of the specific requirements for performing routing tasks in detail.

# Traversing a Graph Efficiently

*Traversing a graph* means to search (visit) each vertex (node) in a specific order. The process of visiting a vertex can include both reading and updating it. As you traverse a graph, an unvisited vertex is *undiscovered.* After a visit, the vertex becomes *discovered* (because you just visited it) or *processed* (because the algorithm tried all the edges departing from it). The order of the search determines the kind of search performed, and many algorithms are available to perform this task. The following sections discuss two such algorithms.

## CONSIDERING REDUNDANCY

When traversing a tree, every path ends in a leaf node so that you know that you have reached the end of that path. However, when working with a graph, the nodes interconnect such that you might have to traverse some nodes more than once to explore the entire graph. As the graph becomes denser, the possibility of visiting the same node more than once increases. Dense graphs can greatly increase both computational and storage requirements.

To reduce the negative effects of visiting a node more than once, it's common to mark each visited node in some manner to show that the algorithm has visited it. When the algorithm detects that it has visited a particular node, it can simply skip that node and move onto the next node in the path. Marking visited nodes decreases the performance penalties inherent in redundancy.

Marking visited nodes also enables verification that the search is complete. Otherwise, an algorithm can end up in a loop and continue to make the rounds through the graph indefinitely.

## Creating the graph

To see how traversing a graph might work, you need a graph. The examples in this section rely on a common graph so that you can see how the two techniques work. The following code shows the adjacency list found at the end of Chapter 8. (You can find this code in the `A4D; 09; Graph Traversing.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
graph = {'A': ['B', 'C'],
         'B': ['A', 'C', 'D'],
         'C': ['A', 'B', 'D', 'E'],
         'D': ['B', 'C', 'E', 'F'],
         'E': ['C', 'D', 'F'],
         'F': ['D', 'E']}
```

The graph features a bidirectional path that goes from A, B, D, and F on one side (starting at the root) and A, C, E, and F along the second side (again, starting at the root). There are also connections (that act as possible shortcuts) going from B to C, from C to D, and from D to E. Using the NetworkX package presented in Chapter 8 lets you display the adjacency as a picture so that you can see how the vertexes and edges appear (see Figure 9-1) by using the following code:

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

Graph = nx.Graph()
for node in graph:
    Graph.add_nodes_from(node)
    for edge in graph[node]:
        Graph.add_edge(node,edge)

pos = { 'A': [0.00, 0.50], 'B': [0.25, 0.75],
        'C': [0.25, 0.25], 'D': [0.75, 0.75],
        'E': [0.75, 0.25], 'F': [1.00, 0.50]}

nx.draw(Graph, pos, with_labels=True)
nx.draw_networkx(Graph, pos)
plt.show()
```

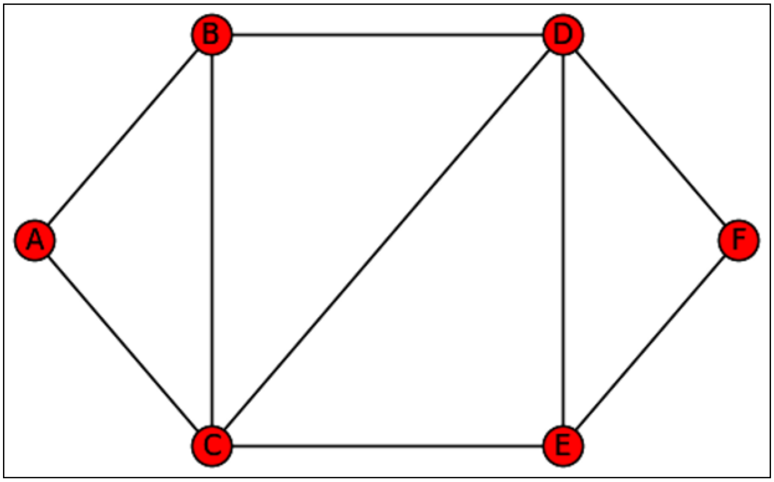


**FIGURE 9-1:** Representing the example graph by NetworkX.

## Applying breadth-first search

A breadth-first search (BFS) begins at the graph root and explores every node that attaches to the root. It then searches the next level — exploring each level in turn until it reaches the end. Consequently, in the example graph, the search explores from A to B and C before it moves on to explore D. BFS explores the graph in a systematic way, exploring vertexes all around the starting vertex in a circular fashion. It begins by visiting all the vertexes a single step from the starting vertex; it then moves two steps out, then three steps out, and so on. The following code demonstrates how to perform a breadth-first search.

```
def bfs(graph, start):
    queue = [start]
    queued = list()
    path = list()
    while queue:
        print ('Queue is: %s' % queue)
        vertex = queue.pop(0)
        print ('Processing %s' % vertex)
        for candidate in graph[vertex]:
            if candidate not in queued:
                queued.append(candidate)
                queue.append(candidate)
                path.append(vertex+'>'+candidate)
                print ('Adding %s to the queue'
                       % candidate)
    return path
```

```
steps = bfs(graph, 'A')
print ('\nBFS:', steps)

Queue is: ['A']
Processing A
Adding B to the queue
Adding C to the queue
Queue is: ['B', 'C']
Processing B
Adding A to the queue
Adding D to the queue
Queue is: ['C', 'A', 'D']
Processing C
Adding E to the queue
Queue is: ['A', 'D', 'E']
Processing A
Queue is: ['D', 'E']
Processing D
Adding F to the queue
Queue is: ['E', 'F']
Processing E
Queue is: ['F']
Processing F

BFS: ['A>B', 'A>C', 'B>A', 'B>D', 'C>E', 'D>F']
```

REMEMBER

The output shows how the algorithm searches. It's in the order that you expect — one level at a time. The biggest advantage of using BFS is that it's guaranteed to return the shortest path between two points as the first output when used to find paths.

TIP

The example code uses a simple list as a queue. As described in Chapter 4, a queue is a first in/first out (FIFO) data structure that works like a line at a bank, where the first item put into the queue is also the first item that comes out. For this purpose, Python provides an even better data structure called a deque (pronounced deck). You create it using the `deque` function from the `collections` package. It performs insertions and extractions in linear time, and you can use it as both a queue and a stack. You can discover more about the `deque` function at `https://pymotw.com/2/collections/deque.html`.

## Applying depth-first search

In addition to BFS, you can use a depth-first search (DFS) to discover the vertexes in a graph. When performing a DFS, the algorithm begins at the graph root and then explores every node from that root down a single path to the end. It then backtracks and begins exploring the paths not taken in the current search path until it reaches the root again. At that point, if other paths to take from the root are available, the algorithm chooses one and begins the same search again. The idea is to explore each path completely before exploring any other path. To make this search technique work, the algorithm must mark each vertex it visits. In this way, it knows which vertexes require a visit and can determine which path to take next. Using BFS or DFS can make a difference according to the way in which you need to traverse a graph. From a programming point of view, the difference between the two algorithms is how each one stores the vertexes to explore the following:

- A queue for BFS, a list that works according to the FIFO principle. Newly discovered vertexes don't wait long for processing.
- A stack for DFS, a list that works according to the last in/first out (LIFO) principle.

The following code shows how to create a DFS:

```
def dfs(graph, start):
    stack = [start]
    parents = {start: start}
    path = list()
    while stack:
        print ('Stack is: %s' % stack)
        vertex = stack.pop(-1)
        print ('Processing %s' % vertex)
        for candidate in graph[vertex]:
            if candidate not in parents:
                parents[candidate] = vertex
                stack.append(candidate)
                print ('Adding %s to the stack'
                       % candidate)
        path.append(parents[vertex]+'>'+vertex)
    return path[1:]

steps = dfs(graph, 'A')
print ('\nDFS:', steps)

Stack is: ['A']
Processing A
Adding B to the stack
Adding C to the stack
```

```
Stack is: ['B', 'C']
Processing C
Adding D to the stack
Adding E to the stack
Stack is: ['B', 'D', 'E']
Processing E
Adding F to the stack
Stack is: ['B', 'D', 'F']
Processing F
Stack is: ['B', 'D']
Processing D
Stack is: ['B']
Processing B

DFS: ['A>C', 'C>E', 'E>F', 'C>D', 'A>B']
```

The first line of output shows the actual search order. Note that the search begins at the root, as expected, but then follows down the left side of the graph around to the beginning. The final step is to search the only branch off the loop that creates the graph in this case, which is D.

Note that the output is not the same as for the BFS. In this case, the processing route begins with node A and moves to the opposite side of the graph, to node F. The code then retraces back to look for overlooked paths. As discussed, this behavior depends on the use of a stack structure in place of a queue. Reliance on a stack means that you could also implement this kind of search using recursion. The use of recursion would make the algorithm faster, so you could obtain results faster than when using a BFS. The trade-off is that you use more memory when using recursion.

TIP

When your algorithm uses a stack, it's using the last result available (as contrasted to a queue, where it would use the first result placed in the queue). Recursive functions produce a result and then apply themselves using that same result. A stack does exactly the same thing in an iteration: The algorithm produces a result, the result is put on a stack, and then the result is immediately taken from the stack and processed again.

## Determining which application to use

The choice between BFS and DFS depends on how you plan to apply the output from the search. Developers often employ BFS to locate the shortest route between two points as quickly as possible. This means that you commonly find BFS used in applications such as GPS, where finding the shortest route is paramount. For the purposes of this book, you also see BFS used for spanning tree, shortest path, and many other minimization algorithms.

A DFS focuses on finding an entire path before exploring any other path. You use it when you need to search in detail, rather than generally. For this reason, you often see DFS used in games, where finding a complete path is important. It's also an optimal approach to perform tasks such as finding a solution to a maze.

REMEMBER

Sometimes you have to decide between BFS and DFS based on the limitations of each technique. BFS needs lots of memory because it systematically stores all the paths before finding a solution. On the other hand, DFS needs less memory, but you have no guarantee that it'll find the shortest and most direct solution.

## Sorting the Graph Elements

The ability to search graphs efficiently relies on sorting. After all, imagine going to a library and finding the books placed in any order the library felt like putting them on the shelves. Locating a single book would take hours. A library works because the individual books appear in specific locations that make them easy to find.

Libraries also exhibit another property that's important when working with certain kinds of graphs. When performing a book search, you begin with a specific category, then a row of books, then a shelf in that row, and finally the book. You move from less specific to more specific when performing the search, which means that you don't revisit the previous levels. Therefore, you don't end up in odd parts of the library that have nothing to do with the topic at hand.

### GRAPHS WITH LOOPS

Sometimes you need to express a process in such a manner that a set of steps repeats. For example, when washing your car, you rinse, soap down, and then rinse again. However, you find a dirty spot, an area that the soap didn't clean the first time. To clean that spot, you soap it again and rinse it again to verify that the spot is gone. Unfortunately, it's a really stubborn spot, so you repeat the process again. In fact, you repeat the soap and rinse steps until the spot is clean. That's what a loop does; it creates a situation in which a set of steps repeats in one of two ways:

- **Meets a specific condition:** The spot on the car is gone.
- **Performs a specific number of times:** This is the number of repetitions you perform during the exercise.

The following sections review *Directed Acyclic Graphs (DAGs)*, which are finite directed graphs that don't have any loops in them. In other words, you start from a particular location and follow a specific route to an ending location without ever going back to the starting location. When using topological sorting, a DAG always directs earlier vertexes to later vertexes. This kind of graph has all sorts of practical uses, such as schedules, with each milestone representing a particular milestone.

## Working on Directed Acyclic Graphs (DAGs)

DAGs are one of the most important kinds of graphs because they see so many practical uses. The basic principles of DAGs are that they

- Follow a particular order so that you can't get from one vertex to another and back to the beginning vertex using any route.
- Provide a specific path from one vertex to another so that you can create a predictable set of routes.

You see DAGs used for many organizational needs. For example, a family tree is an example of a DAG. Even when the activity doesn't follow a chronological or other overriding order, the DAG enables you to create predictable routes, which makes DAGs easier to process than many other kinds of graphs you work with.

However, DAGs can use optional routes. Imagine that you're building a burger. The menu system starts with a bun bottom. You can optionally add condiments to the bun bottom, or you can move directly to the burger on the bun. The route always ends up with a burger, but you have multiple paths for getting to the burger. After you have the burger in place, you can choose to add cheese or bacon before adding the bun top. The point is that you take a specific path, but each path can connect to the next level in several different ways.

REMEMBER

So far, the chapter has shown you a few different kinds of graph configurations, some of which can appear in combination, such as a directed, weighted, dense graph:

- **Directed:** Edges have a single direction and can have these additional properties:
  - **Cyclic:** The edges form a cycle that take you back to the initial vertex after having visited the intermediary vertexes.
  - **A-cyclic:** This graph lacks cycles.
- **Undirected:** Edges connect vertexes in both directions.

- **Weighted:** Each edge has a cost associated with it, such as time, money, or energy, which you must pay to pass through it.
- **Unweighted:** All the edges have no cost or the same cost.
- **Dense:** A graph that has a large number of edges when compared to the number of vertexes.
- **Sparse:** A graph that has a small number of edges when compared to the number of vertexes.

## Relying on topological sorting

An important element of DAGs is that you can represent a myriad of activities using them. However, some activities require that you approach tasks in a specific order. This is where topological sorting comes into play. *Topological sorting* orders all the vertexes of a graph on a line with the direct edges pointing from left to right. Arranged in such a fashion, the code can easily traverse the graph and process the vertexes one after the other, in order.

When you use topological sorting, you organize the graph so that every graph vertex leads to a later vertex in the sequence. For example, when creating a schedule for building a skyscraper, you don't start at the top and work your way down. You begin with the foundation and work your way up. Each floor can represent a milestone. When you complete the second floor, you don't go to the third and then redo the second floor. Instead, you move on from the third floor to the fourth floor, and so on. Any sort of scheduling that requires you to move from a specific starting point to a specific ending point can rely on a DAG with topological sorting.

Topological sorting can help you determine that your graph has no cycles (because otherwise, you can't order the edges connecting the vertexes from left to right; at least one node will refer to a previous node). In addition, topological sorting also proves helpful in algorithms that process complex graphs because it shows the best order for processing them.

You can obtain topological sorting using the DFS traversal algorithm. Simply note the processing order of the vertexes by the algorithm. In the previous example, the output appears in this order: A, C, E, F, D, and B. Follow the sequence in Figure 9-1 and you notice that the topological sorting follows the edges on the external perimeter of graph. It then makes a complete tour: After reaching the last node of the topological sort, you're just a step away from A, the start of the sequence.

# Reducing to a Minimum Spanning Tree

Many problems that algorithms solve rely on defining a minimum of resources to use, such as defining an economical way to reach all the points on a map. This problem was paramount in the late nineteenth and early twentieth centuries when railway and electricity networks started appearing in many countries, revolutionizing transportation and ways of living. Using private companies to build such networks was expensive (it took a lot of time and workers). Using less material and a smaller workforce offered savings by reducing redundant connections.

REMEMBER

Some redundancy is desirable in critical transportation or energy networks even when striving for economical solutions. Otherwise, if only one method connects the network, it's easily disrupted accidentally or by a voluntary act (such as an act of war), interrupting services to many customers.

In Moravia, the eastern part of Czech Republic, the Czech mathematician Otakar Borůvka found a solution in 1926 that allows constructing an electrical network using the least amount of wire possible. His solution is quite efficient because it not only allows finding a way to connect all the towns in Moravia in the most economical way possible, but it had a time complexity of O(m*log n), where m is the number of edges (the electrical cable) and n the number of vertexes (the towns). Others have improved Borůvka's solution since then. (In fact, algorithm experts partially forgot and then rediscovered it.) Even though the algorithms you find in books are better designed and easier to grasp (those from Prim and Kruskal), they don't achieve better results in terms of time complexity.

A minimal spanning tree defines the problem of finding the most economical way to accomplish a task. A *spanning tree* is the list of edges required to connect all the vertexes in an undirected graph. A single graph could contain multiple spanning trees, depending on the graph arrangement, and determining how many trees it contains is a complex issue. Each path you can take from start to completion in a graph is another spanning tree. The spanning tree visits each vertex only once; it doesn't loop or do anything to repeat path elements.

When you work on an unweighted graph, the spanning trees are the same length. In unweighted graphs, all edges have the same length, and the order you visit them in doesn't matter because the run path is always the same. All possible spanning trees have the same number of edges, n-1 edges (n is the number of vertexes), of the same exact length. Moreover, any graph traversal algorithm, such as BFS or DFS, suffices to find one of the possible spanning trees.

Things become tricky when working with a weighted graph with edges of different lengths. In this case, of the many possible spanning trees, a few, or just one, have the minimum length possible. A *minimum spanning tree* is the one spanning tree that guarantees a path with the least possible edge weight. An undirected graph generally contains just one minimum spanning tree, but, again, it depends on the configuration. Think about minimum spanning trees this way: When looking at a map, you see a number of paths to get from point A to point B. Each path has places where you must turn or change roads, and each of these junctions is a vertex. The distance between vertexes represents the edge weight. Generally, one path between point A and point B provides the shortest route.

However, minimum spanning trees need not always consider the obvious. For example, when considering maps, you might not be interested in distance; you might instead want to consider time, fuel consumption, or myriad other needs. Each of these needs could have a completely different minimum spanning tree. With this in mind, the following sections help you understand minimum spanning trees better and demonstrate how to solve the problem of figuring out the smallest edge weight for any given problem. To demonstrate a minimum spanning tree solution using Python, the following code updates the previous graph by adding edge weights. (You can find this code in the `A4D; 09; Minimum Spanning Tree.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

graph = {'A': {'B':2, 'C':3},
         'B': {'A':2, 'C':2, 'D':2},
         'C': {'A':3, 'B':2, 'D':3, 'E':2},
         'D': {'B':2, 'C':3, 'E':1, 'F':3},
         'E': {'C':2, 'D':1, 'F':1},
         'F': {'D':3, 'E':1}}

Graph = nx.Graph()
for node in graph:
    Graph.add_nodes_from(node)
    for edge, weight in graph[node].items():
        Graph.add_edge(node,edge, weight=weight)

pos = { 'A': [0.00, 0.50], 'B': [0.25, 0.75],
        'C': [0.25, 0.25], 'D': [0.75, 0.75],
        'E': [0.75, 0.25], 'F': [1.00, 0.50]}

labels = nx.get_edge_attributes(Graph,'weight')
nx.draw(Graph, pos, with_labels=True)
nx.draw_networkx_edge_labels(Graph, pos,
                             edge_labels=labels)
nx.draw_networkx(Graph,pos)
plt.show()
```

Figure 9-2 shows that all edges have a value now. This value can represent something like time, fuel, or money. Weighted graphs can represent many possible optimization problems that occur in geographical space (such as movement between cities) because they represent situations in which you can come and go from a vertex.

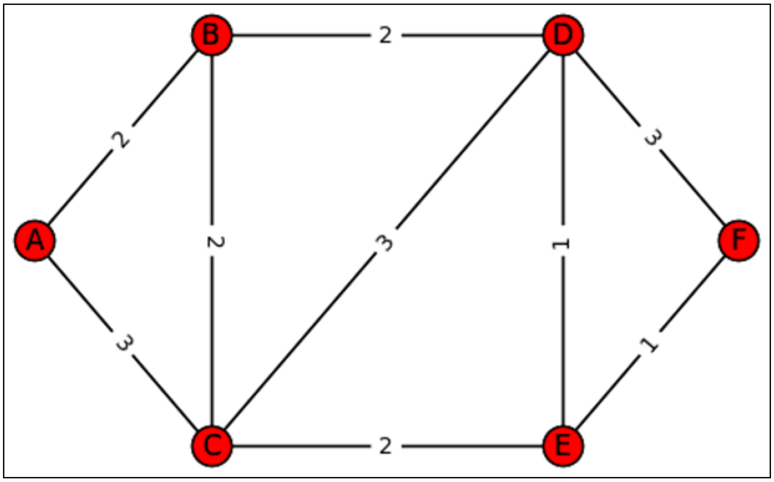


**FIGURE 9-2:** The example graph becomes weighted.

Interestingly, all edges have positive weights in this example. However, weighted graphs can have negative weights on some edges. Many situations take advantage of negative edges. For instance, they're useful when you can both gain and lose from moving between vertexes, such as gaining or losing money when transporting or trading goods, or releasing energy in a chemical process.

REMEMBER

Not all algorithms are well suited for handling negative edges. It's important to note those that can work with only positive weights.

## Discovering the correct algorithms to use

You can find many different algorithms to use to create a minimum spanning tree. The most common are greedy algorithms, which run in polynomial time. *Polynomial time* is a power of the number of edges, such as $O(n^2)$ or $O(n^3)$ (see Part 5 for additional information about polynomial time). The major factors that affect the running speed of such algorithms involve the decision-making process — that is, whether a particular edge belongs in the minimum spanning tree or whether the minimum total weight of the resulting tree exceeds a certain value. With this in mind, here are some of the algorithms available for solving a minimum spanning tree:

- **Borůvka's:** Invented by Otakar Borůvka in 1926 to solve the problem of finding the optimal way to supply electricity in Moravia. The algorithm relies on a series of stages in which it identifies the edges with the smallest weight in each stage. The calculations begin by looking at individual vertexes, finding the smallest weight for that vertex, and then combining paths to form forests of individual trees until it creates a path that combines all the forests with the smallest weight.
- **Prim's:** Originally invented by Jarnik in 1930, Prim rediscovered it in 1957. This algorithm starts with an arbitrary vertex and grows the minimum spanning tree one edge at a time by always choosing the edge with the least weight.
- **Kruskal's:** Developed by Joseph Kruskal in 1956, it uses an approach that combines Borůvka's algorithm (creating forests of individual trees) and Prim's algorithm (looking for the minimum edge for each vertex and building the forests one edge at a time).
- **Reverse-delete:** This is actually a reversal of Kruskal's algorithm. It isn't commonly used.

TIP

These algorithms use a greedy approach. Greedy algorithms appear in Chapter 2 among the families of algorithms, and you see them in detail in Chapter 15. In a *greedy approach,* the algorithm gradually arrives at a solution by taking, in an irreversible way, the best decision available at each step. For instance, if you need the shortest path between many vertexes, a greedy algorithm takes the shortest edges among those available between all vertexes.

## Introducing priority queues

Later in this chapter, you see how to implement Prim's and Kruskal's algorithm for a minimum spanning tree, and Dijkstra's algorithm for the shortest path in a graph using Python. However, before you can do that, you need a method to find the edges with the minimum weight among a set of edges. Such an operation implies ordering, and ordering elements costs time. It's a complex operation, as described in Chapter 7. Because the examples repeatedly reorder edges, a data structure called the *priority queue* comes in handy.

*Priority queues* rely on heap tree-based data structures that allow fast element ordering when you insert them inside the heap. Like the magician's magic hat, priority heaps store edges with their weights and are immediately ready to provide you with the inserted edge whose weight is the minimum among those stores.

This example uses a class that allows it to perform priority-queue comparisons that determine whether the queue contains elements and when those elements contain a certain edge (avoiding double insertions). The priority queue has another useful characteristic (whose usefulness is explained when working on Dijkstra's algorithm): If you insert an edge with a different weight than previously stored, the code updates the edge weight and rearranges the edge position in the heap.

```
from heapq import heapify, heappop, heappush

class priority_queue():
    def __init__(self):
        self.queue = list()
        heapify(self.queue)
        self.index = dict()
    def push(self, priority, label):
        if label in self.index:
            self.queue = [(w,l)
                for w,l in self.queue if l!=label]
            heapify(self.queue)
        heappush(self.queue, (priority, label))
        self.index[label] = priority
    def pop(self):
        if self.queue:
            return heappop(self.queue)
    def __contains__(self, label):
        return label in self.index
    def __len__(self):
        return len(self.queue)
```

## Leveraging Prim's algorithm

Prim's algorithm generates the minimum spanning tree for a graph by traversing the graph vertex by vertex. Starting from any chosen vertex, the algorithm adds edges using a constraint in which, if one vertex is currently part of the spanning tree and the second vertex isn't part of it, the edge weight between the two must be the least possible among those available. By proceeding in this fashion, creating cycles in the spanning tree is impossible (it could happen only if you add an edge whose vertexes are already both in the spanning tree) and you're guaranteed to obtain a minimal tree because you add the edges with the least weight. In terms of steps, the algorithm includes these three phases, with the last one being iterative:

1. Track both the edges of the minimum spanning tree and the used vertexes as they become part of the solution.

2. Start from any vertex in the graph and place it into the solution.
3. Determine whether there are still vertexes that aren't part of the solution:
   - Enumerate the edges that touch the vertexes in the solution.
   - Insert the edge with the minimum weight into the spanning tree. (This is the greedy principle at work in the algorithm: Always choose the minimum at each step to obtain an overall minimum result.)

By translating these steps into Python code, you can test the algorithm on the example weighted graph using the following code:

```
def prim(graph, start):
    treepath = {}
    total = 0
    queue = priority_queue()
    queue.push(0 , (start, start))
    while queue:
        weight, (node_start, node_end) = queue.pop()
        if node_end not in treepath:
            treepath[node_end] = node_start
            if weight:
                print("Added edge from %s" \
                      " to %s weighting %i"
                      % (node_start, node_end, weight))
                total += weight
            for next_node, weight \
            in graph[node_end].items():
                queue.push(weight , (node_end, next_node))
    print ("Total spanning tree length: %i" % total)
    return treepath

treepath = prim(graph, 'A')

Added edge from A to B weighting 2
Added edge from B to C weighting 2
Added edge from B to D weighting 2
Added edge from D to E weighting 1
Added edge from E to F weighting 1
Total spanning tree length: 8
```

The algorithm prints the processing steps, showing the edge it adds at each stage and the weight the edge adds to the total. The example displays the total sum of weights and the algorithm returns a Python dictionary containing the ending vertex as key and the starting vertex as value for each edge of the resulting spanning tree. Another function, represent_tree, turns the key and value pairs of the dictionary into a tuple and then sorts each of the resulting tuples for better readability of the tree path:

```
def represent_tree(treepath):
    progression = list()
    for node in treepath:
        if node != treepath[node]:
            progression.append((treepath[node], node))
    return sorted(progression, key=lambda x:x[0])

print (represent_tree(treepath))

[('A','B'), ('B','C'), ('B','D'), ('D','E'), ('E','F')]
```

REMEMBER

The represent_tree function reorders the output of Prim's algorithm for better readability. However, the algorithm works on an undirected graph, which means that you can traverse the edges in both directions. The algorithm incorporates this assumption because there is no edge directionality check to add to the priority queue for later processing.

## Testing Kruskal's algorithm

Kruskal's algorithm uses a greedy strategy, just as Prim's does, but it picks the shortest edges from a global pool containing all the edges (whereas Prim's evaluates the edges according to the vertexes in the spanning tree). To determine whether an edge is a suitable part of the solution, the algorithm relies on an aggregative process in which it gathers vertexes together. When an edge involves vertexes already in the solution, the algorithm discards it to avoid creating a cycle. The algorithm proceeds in the following fashion:

1. Put all the edges into a heap and sort them so that the shortest edges are on top.
2. Create a set of trees, each one containing only one vertex (so that the number of trees is the same number as the vertexes). You connect trees as an aggregate until the trees converge into a unique tree of minimal length that spans all the vertexes.
3. Repeat the following operations until the solution doesn't contain as many edges as the number of vertexes in the graph:

   **a.** Choose the shortest edge from the heap.

   **b.** Determine whether the two vertexes connected by the edge appear in different trees from among the set of connected trees.

c. When the trees differ, connect the trees using the edge (defining an aggregation).

d. When the vertexes appear in the same tree, discard the edge.

e. Repeat steps a through d for the remaining edges on the heap.

The following example demonstrates how to turn these steps into Python code:

```
def kruskal(graph):
    priority = priority_queue()
    print ("Pushing all edges into the priority queue")
    treepath = list()
    connected = dict()
    for node in graph:
        connected[node] = [node]
        for dest, weight in graph[node].items():
            priority.push(weight, (node,dest))
    print ("Totally %i edges" % len(priority))
    print ("Connected components: %s"
           % connected.values())

    total = 0
    while len(treepath) < (len(graph)-1):
        (weight, (start, end)) = priority.pop()
        if end not in connected[start]:
            treepath.append((start, end))
            print ("Summing %s and %s components:"
                   % (connected[start],connected[end]))
            print ("\tadded edge from %s " \
                   "to %s weighting %i"
                   % (start, end, weight))
            total += weight
            connected[start] += connected[end][:]
            for element in connected[end]:
                connected[element]= connected[start]
    print ("Total spanning tree length: %i" % total)
    return sorted(treepath, key=lambda x:x[0])

print ('\nMinimum spanning tree: %s' % kruskal(graph))

Pushing all edges into the priority queue
Totally 9 edges
Connected components: dict_values([['A'], ['E'], ['F'],
                                  ['B'], ['D'], ['C']])
```

```
Summing ['E'] and ['D'] components:
        added edge from E to D weighting 1
Summing ['E', 'D'] and ['F'] components:
        added edge from E to F weighting 1
Summing ['A'] and ['B'] components:
        added edge from A to B weighting 2
Summing ['A', 'B'] and ['C'] components:
        added edge from B to C weighting 2
Summing ['A', 'B', 'C'] and ['E', 'D', 'F'] components:
        added edge from B to D weighting 2
Total spanning tree length: 8

Minimum spanning tree:
[('A','B'), ('B','C'), ('B','D'), ('E','D'), ('E','F')]
```

TIP

Kruskal's algorithm offers a solution that's similar to the one proposed by Prim's algorithm. However, different graphs may provide different solutions for the minimum spanning tree when using Prim's and Kruskal's algorithms because each algorithm proceeds in different ways to reach its conclusions. Different approaches often imply different minimal spanning trees as output.

## Determining which algorithm works best

Both Prim's and Kruskal's algorithms output a single connected component, joining all the vertexes in the graph by using the least (or one of the least) long sequences of edges (a minimum spanning tree). By summing the edge weights, you can determine the length of the resulting spanning tree. Because both algorithms always provide you with a working solution, you must rely on running time and decide whether they can take on any kind of weighted graph to determine which is best.

As for running time, both algorithms provide similar results with Big-O complexity rating of O(E*log(V)), where E is the number of edges and V the number of vertexes. However, you must account for how they solve the problem because there are differences in the average expected running time.

Prim's algorithm incrementally builds a single solution by adding edges, whereas Kruskal's algorithm creates an ensemble of partial solutions and aggregates them. In creating its solution, Prim's algorithm relies on data structures that are more complex than Kruskal's because it continuously adds potential edges as candidates and keeps picking the shortest edge to proceed toward its solution. When operating on a dense graph, Prim's algorithm is preferred over Kruskal's because its priority queue based on heaps does all the sorting jobs quickly and efficiently.

REMEMBER

The example uses a priority queue based on a binary heap for the heavy job of picking up the shortest edges, but there are even faster data structures, such as the *Fibonacci heap,* which can produce faster results when the heap contains many edges. Using a Fibonacci heap, the running complexity of Prim's algorithm can mutate to O(E +V*log(V)), which is clearly advantageous if you have a lot of edges (the E component is now summed instead of multiplied) compared to the previous reported running time O(E*log(V)).

Kruskal's algorithm doesn't much need a priority queue (even though one of the examples uses one) because the enumeration and sorting of edges happens just once at the beginning of the process. Being based on simpler data structures that work through the sorted edges, it's the ideal candidate for regular, sparse graphs with fewer edges.

# Finding the Shortest Route

The shortest route between two points isn't necessarily a straight line, especially when a straight line doesn't exist in your graph. Say that you need to run electrical lines in a community. The shortest route would involve running the lines as needed between each location without regard to where those lines go. However, real life tends not to allow a simple solution. You may need to run the cables beside roads and not across private property, which means finding routes that reduce the distances as much as possible.

## Defining what it means to find the shortest path

Many applications exist for shortest-route algorithms. The idea is to find the path that offers the smallest distance between point A and point B. Finding the shortest path is useful for both transportation (how to arrive at a destination consuming the least fuel) and communication (how to route information to allow it to arrive earlier). Nevertheless, unexpected applications of the shortest-path problem may also arise in image processing (for separating contours of images), gaming (how to achieve certain game goals using the fewest moves), and many other fields in which you can reduce the problem to an undirected or directed weighted graph.

The Dijkstra algorithm can solve the shortest-path problem and has found the most uses. Edsger W. Dijkstra, a Dutch computer scientist, devised the algorithm as a demonstration of the processing power of a new computer called ARMAC (`http://www-set.win.tue.nl/UnsungHeroes/machines/armac.html`) in 1959. The algorithm initially solved the shortest distance between 64 cities in the Netherlands based on a simple graph map.

REMEMBER

Other algorithms can solve the shortest-path problem. The Bellman-Ford and Floyd-Warshall are more complex but can handle graphs with negative weights. (Negative weights can represent some problems better.) Both algorithms are beyond the scope of this book, but the site at https://www.hackerearth.com/ja/practice/algorithms/graphs/shortest-path-algorithms/tutorial/ provides additional information about them. Because the shortest-path problem involves graphs that are both weighted and directed, the example graph requires another update before proceeding (you can see the result in Figure 9-3). (You can find this code in the A4D; 09; Shortest Path.ipynb file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

graph = {'A': {'B':2, 'C':3},
         'B': {'C':2, 'D':2},
         'C': {'D':3, 'E':2},
         'D': {'F':3},
         'E': {'D':1,'F':1},
         'F': {}}

Graph = nx.DiGraph()
for node in graph:
    Graph.add_nodes_from(node)
    for edge, weight in graph[node].items():
        Graph.add_edge(node,edge, weight=weight)

pos = { 'A': [0.00, 0.50], 'B': [0.25, 0.75],
        'C': [0.25, 0.25], 'D': [0.75, 0.75],
        'E': [0.75, 0.25], 'F': [1.00, 0.50]}

labels = nx.get_edge_attributes(Graph,'weight')
nx.draw(Graph, pos, with_labels=True)
nx.draw_networkx_edge_labels(Graph, pos,
                             edge_labels=labels)
nx.draw_networkx(Graph,pos)
plt.show()
```

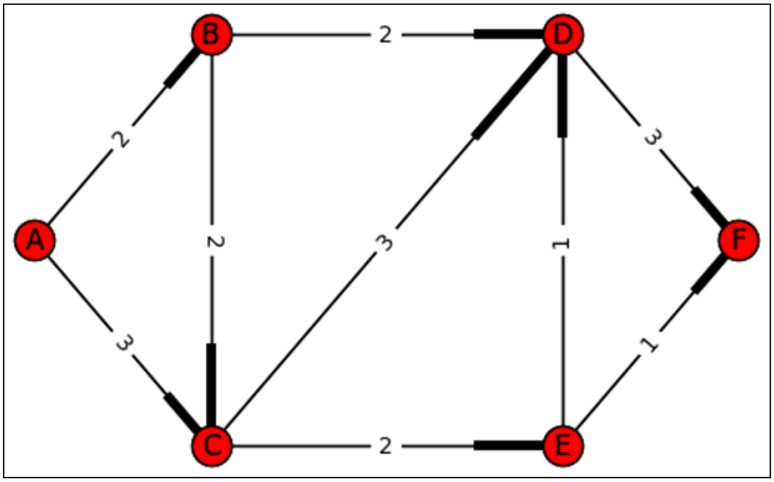


**FIGURE 9-3:** The example graph becomes weighted and directed.

## Explaining Dijkstra's algorithm

Dijkstra's algorithm requires a starting and (optionally) ending vertex as input. If you don't provide an ending vertex, the algorithm computes the shortest distance between the starting vertex and any other vertexes in the graph. When you define an ending vertex, the algorithm stops upon reading that vertex and returns the result up to that point, no matter how much of the graph remains unexplored.

The algorithm starts by estimating the distance of the other vertexes from the starting point. This is the starting belief it records in the priority queue and is set to infinity by convention. Then the algorithm proceeds to explore the neighboring nodes, similar to a BFS. This allows the algorithm to determine which nodes are near and that their distance is the weight of the connecting edges. It stores this information in the priority queue by an appropriate weight update.

REMEMBER

Naturally, the algorithm explores the neighbors because a directed edge connects them with the starting vertex. Dijkstra's algorithm accounts for the edge direction.

At this point, the algorithm moves to the nearest vertex on the graph based on the shortest edge in the priority queue. Technically, the algorithm *visits* a new vertex. It starts exploring the neighboring vertexes, excluding the vertexes that it has already visited, determines how much it costs to visit each of the unvisited vertexes, and evaluates whether the distance to visit them is less than the distance it recorded in the priority queue.

When the distance in the priority queue is infinite, this means that it's the algorithm's first visit to that vertex, and the algorithm records the shorter distance.

When the distance recorded in the priority queue isn't infinite, but it's more than the distance that the algorithm has just calculated, it means that the algorithm has found a *shortcut*, a shorter way to reach that vertex from the starting point, and it stores the information in the priority queue. Of course, if the distance recorded in the priority queue is shorter than the one just evaluated by the algorithm, the algorithm discards the information because the new route is longer. After updating all the distances to the neighboring vertexes, the algorithm determines whether it has reached the end vertex. If not, it picks the shortest edge present in the priority queue, visits it, and starts evaluating the distance to the new neighboring vertexes.

REMEMBER

As the narrative of the algorithm explained, Dijikstra's algorithm keeps a precise accounting of the cost to reach every vertex that it encounters, and it updates its information only when it finds a shorter way. The running complexity of the algorithm in Big-O notation is O(E*log(V)), where E is the number of edges and V the number of vertexes in the graph. The following code shows how to implement Dijikstra's algorithm using Python:

```
def dijkstra(graph, start, end):
    inf = float('inf')
    known = set()
    priority = priority_queue()
    path = {start: start}

    for vertex in graph:
        if vertex == start:
            priority.push(0, vertex)
        else:
            priority.push(inf, vertex)
    last = start

    while last != end:
        (weight, actual_node) = priority.pop()
        if actual_node not in known:
            for next_node in graph[actual_node]:
                upto_actual = priority.index[actual_node]
                upto_next = priority.index[next_node]
                to_next = upto_actual + \
                graph[actual_node][next_node]
                if to_next < upto_next:
                    priority.push(to_next, next_node)
                    print("Found shortcut from %s to %s"
                          % (actual_node, next_node))
                    print ("\tTotal length up so far: %i"
                           % to_next)
                    path[next_node] = actual_node

            last = actual_node
            known.add(actual_node)

    return priority.index, path

dist, path = dijkstra(graph, 'A', 'F')

Found shortcut from A to C
   Total length up so far: 3
Found shortcut from A to B
   Total length up so far: 2
Found shortcut from B to D
   Total length up so far: 4
Found shortcut from C to E
   Total length up so far: 5
Found shortcut from D to F
   Total length up so far: 7
Found shortcut from E to F
   Total length up so far: 6
```

The algorithm returns a couple of useful pieces of information: the shortest path to destination and the minimum recorded distances for the visited vertexes. To visualize the shortest path, you need a `reverse_path` function that rearranges the path to make it readable:

```
def reverse_path(path, start, end):
    progression = [end]
    while progression[-1] != start:
        progression.append(path[progression[-1]])
    return progression[::-1]

print (reverse_path(path, 'A', 'F'))

['A', 'C', 'E', 'F']
```

You can also know the shortest distance to every node encountered by querying the `dist` dictionary:

```
print (dist)

{'D': 4, 'A': 0, 'B': 2, 'F': 6, 'C': 3, 'E': 5}
```

IN THIS CHAPTER

- » Seeing social networks in graph form
- » Interacting with graph content

# Chapter 10

# Discovering Graph Secrets

Chapter 8 helps you understand the foundations of graphs as they apply to mathematics. Chapter 9 increases your knowledge by helping you see the relationship of graphs to algorithms. This chapter helps you focus on applying the theories of these previous two chapters to interact with graphs in practical ways.

The first section conveys the character of social networks by using graphs. Considering the connections created by social networks is important. For example, conversation analysis can reveal patterns that help you understand the underlying topic better than simply reading the conversations would do. A particular conversation branch might attract greater attention because it's more important than another conversation branch. Of course, you must perform this analysis while dealing with issues such as spam. Analysis of this sort can lead to all sorts of interesting conclusions, such as where to spend more advertising money in order to attract the most attention and, therefore, sales.

The second section looks at navigating graphs to achieve specific results. For example, when driving, you might need to know the best route to take between two points given that, even though one route is shorter, it also has construction that makes a second route better. Sometimes you need to randomize your search to discover a best route or a best conclusion. This section of the chapter also discusses that issue.

# Envisioning Social Networks as Graphs

Every social interaction necessarily connects with every other social interaction of the same type. For example, consider a social network such as Facebook. The links on your page connect with family members, but they also connect with outside sources that in turn connect with other outside sources. Each of your family members also has external links. Direct and indirect connections between various pages eventually link every other page together, even though the process of getting from one page to another may require the use of myriad links. Connectivity occurs in all sorts of other ways as well. The point is that studying social networks simply by viewing a Facebook page or other source of information is hard. *Social Network Analysis (SNA)* is the process of studying the interactions in social networks using graphs called *sociograms,* in which nodes (such as a Facebook page) appear as points, and ties (such as external page links) appear as lines. The following sections discuss some of the issues surrounding the study of social networks as graphs.

## Clustering networks in groups

People tend to form communities — clusters of other people who have like ideas and sentiments. By studying these clusters, attributing certain behaviors to the group as a whole becomes easier (although attributing the behavior to an individual is both dangerous and unreliable). The idea behind the study of clusters is that if a connection exists between people, they often have a common set of ideas and goals. By finding clusters, you can determine these ideas by inspecting group membership. For instance, it's common to try to find clusters of people in insurance fraud detection and tax inspection. Unexpected groups of people might raise suspicion that they're part of a group of fraudsters or tax evaders because they lack the usual reasons for people to gather in such circumstances.

*Friendship graphs* can represent how people connect with each other. The vertexes represent individuals and the edges represent their connections, such as family relationships, business contacts, or friendship ties. Typically, friendship graphs are undirected because they represent mutual relationships, and sometimes they're weighted to represent the strength of the bond between two persons.

TIP

Many studies focus on undirected graphs that concentrate solely on associations. You can also use directed graphs to show that Person A knows about Person B, but Person B doesn't even know that Person A exists. In this case, you actually have 16 different kinds of triads to consider. For the sake of simplicity, this chapter focuses solely on these four types: closed, open, connected pair, and unconnected.

When looking for clusters in a friendship graph, the connections between nodes in these clusters depend on triads — essentially, special kinds of triangles. Connections between three people can fall into these categories:

- **Closed:** All three people know each other. Think about a family setting in this case, in which everyone knows everyone else.
- **Open:** One person knows two other people, but the two other people don't know each other. Think about a person who knows an individual at work and another individual at home, but the individual at work doesn't know anything about the individual at home.
- **Connected pair:** One person knows one of the other people in a triad but doesn't know the third person. This situation involves two people who know something about each other meeting someone new — someone who potentially wants to be part of the group.
- **Unconnected:** The triad forms a group, but no one in the group knows each other. This last one might seem a bit odd, but think about a convention or seminar. The people at these events form a group, but they may not know anything about each other. However, because they have similar interests, you can use clustering to understand the behavior of the group.

Triads occur naturally in relationships, and many Internet social networks have leveraged this idea to accelerate the connections between participants. The density of connections is important for any kind of social network because a connected network can spread information and share content more easily. For instance, when LinkedIn, the professional social network (`https://www.linkedin.com/`), decided to increase the connection density of its network, it started by looking for open triads and trying to close them by inviting people to connect. Closing triads is at the foundation of LinkedIn's *Connection Suggestion algorithm.* You can discover more about how it works by reading the Quora's answer at: `https://www.quora.com/How-does-LinkedIns-People-You-May-Know-work`.

The example in this section relies on the Zachary's Karate Club sample graph described at `https://networkdata.ics.uci.edu/data.php?id=105`. It's a small graph that lets you see how networks work without spending a lot of time loading a large dataset. Fortunately, this dataset appears as part of the `networkx` package introduced in Chapter 8. The Zachary's Karate Club network represents the friendship relationships between 34 members of a karate club from 1970 to 1972. Sociologist Wayne W. Zachary used it as a topic of study. He wrote a paper on it entitled "An Information Flow Model for Conflict and Fission in Small Groups." The interesting fact about this graph and its paper is that in those years, a conflict arose in the club between one of the karate instructors (node number 0) and the president of the club (node number 33). By clustering the graph, you can almost perfectly predict the split of the club into two groups shortly after the occurrence.

Because this example also draws a graph showing the groups (so that you can visualize them easier), you also need to use the `matplotlib` package. The following code shows how to graph the nodes and edges of the dataset. (You can find this code in the `A4D; 10; Social Networks.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

graph = nx.karate_club_graph()

pos=nx.spring_layout(graph)
nx.draw(graph, pos, with_labels=True)
plt.show()
```

To display the graphic onscreen, you also need to provide a layout that determines how to position the nodes onscreen. This example uses the Fruchterman-Reingold force-directed algorithm (the call to `nx.spring_layout`). However, you can choose one of the other layouts described in the Graph Layout section at `https://networkx.github.io/documentation/networkx-1.9/reference/drawing.html`. Figure 10-1 shows the output from the example. (Your output may look slightly different.)

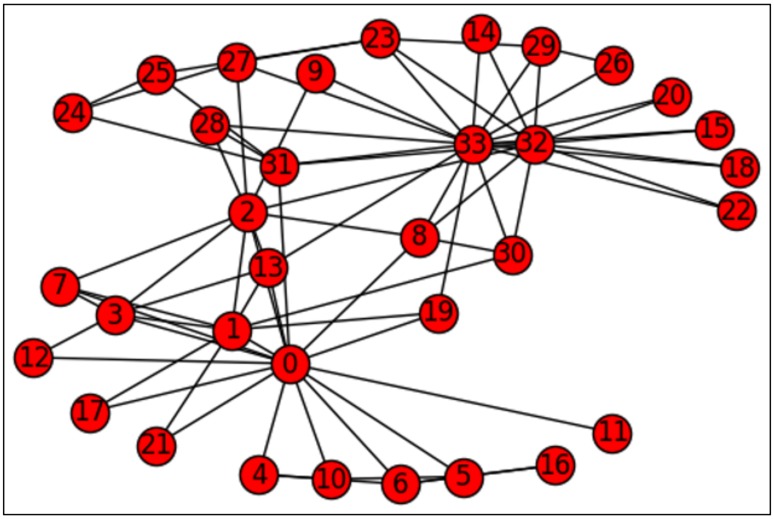


**FIGURE 10-1:** A graph showing the network clusters of relationships between friends.

REMEMBER

The Fruchterman-Reingold force-directed algorithm for generating automatic layouts of graphs creates understandable layouts with separated nodes and edges that tend not to cross by mimicking what happens in physics between electrically charged particles or magnets bearing the same sign. In looking at the graph output, you can see that some nodes have just one connection, some two, and some more than two. The edges form triads, as previously mentioned. However, the most important consideration is that Figure 10-1 clearly shows the clustering that occurs in a social network.

## Discovering communities

A group of tightly associated people often defines a community. In fact, the term *clique* applies to a group whose membership to the group is exclusive and everyone knows everyone else quite well. Most people have childhood memories of a tight group of friends at school or in the neighborhood who always spent their time together. That’s a clique.

REMEMBER

You can find cliques in undirected graphs. Directed graphs distinguish strongly between connected components when a direct connection exists between all the node pairs in the component itself. A city is an example of a strongly connected component because you can reach any destination from any starting point by following one-way and two-way streets.

Mathematically, a clique is even more rigorous because it implies a *subgraph* (a part of a network graph that you can separate from other parts as a complete element in its own right) that has maximum connectivity. In looking at various kinds of social networks, picking out the clusters is easy, but what can prove difficult is finding the cliques — the groups with maximum connectivity — within the clusters. By knowing where cliques exist, you can begin to understand the cohesive nature of a community better. In addition, the exclusive nature of cliques tends to create a group that has its own rules outside of those that might exist in the social network as a whole. The following example shows how to extract cliques and communities from the karate club graph used in the previous section:

```
graph = nx.karate_club_graph()
# Finding and printing all cliques of four
cliques = nx.find_cliques(graph)
print ('All cliques of four: %s'
       % [c for c in cliques if len(c)>=4])

# Joining cliques of four into communities
communities = nx.k_clique_communities(graph, k=4)
communities_list = [list(c) for c in communities]
nodes_list = [node for community in communities_list for
              node in community]
print ('Found these communities: %s' % communities_list)
```

```
# Printing the subgraph of communities
subgraph = graph.subgraph(nodes_list)
nx.draw(subgraph, with_labels=True)
plt.show()

All cliques of four: [[0, 1, 2, 3, 13], [0, 1, 2, 3, 7],
                      [33, 32, 8, 30], [33, 32, 23, 29]]
Found these communities: [[0, 1, 2, 3, 7, 13],
                        [32, 33, 29, 23], [32, 33, 8, 30]]
```

The example begins by extracting just the nodes in the karate club dataset that have four or more connections, and then prints the cliques with a minimum size of four. Of course, you can set any level of connections needed to obtain the desired resource. Perhaps you consider a clique a community in which each node has twenty connections, but other people might see a clique as a community where each node has just three connections.

The list of cliques doesn't really help you much, though, if you want to see the communities. To see them, you need to rely on specialized and complex algorithms to merge overlapping cliques and find clusters, such as the clique percolation method described at `https://gaplogs.net/2012/04/01/simple-community-detection-algorithms/`. The NetworkX package offers `k_clique_communities`, an implementation of the clique percolation algorithm, which results in the union of all the cliques of a certain size (the k parameter). These cliques of a certain size share k-1 elements (that is, they differ by just one component, a truly strict rule).

Clique percolation provides you with a list of all the communities found. In this example, one clique revolves around the karate instructor and another revolves around the president of the club. In addition, you can extract all the nodes that are part of a community into a single set, which helps you create a subgraph made of just communities.

Finally, you can draw the subgraph and display it. Figure 10-2 shows the output of this example, which displays the ensemble of cliques with four or more connections.

Finding cliques in graphs is a complex problem requiring many computations (it's a difficult problem) that an algorithm solves using a brute-force search, which means trying all possible subsets of vertexes to determine whether they're cliques. With some luck, because some randomization is needed for the algorithm to succeed, you can find a large clique using a simple approach whose complexity is O(n+m), where *n* is the number of vertexes and *m* the edges. The following steps describe this process.

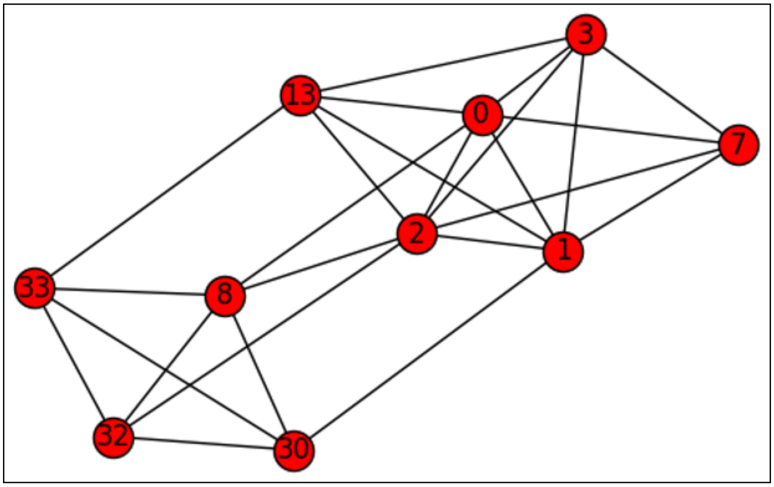


**FIGURE 10-2:** Communities often contain cliques that can prove useful for SNA.

1. Sort the vertexes by degree (which is the number of vertex connections), from the highest to the lowest.
2. Place the vertex with the highest degree into the clique (or as an alternative, randomly choose from one of the highest-degree vertexes).
3. Repeat Steps 1 and 2 until you have no more vertexes to test.
4. Verify the next vertex as being part of the clique:
   - If it's part of the clique, add it to the clique.
   - If it isn't part of the clique, repeat the test on the remaining vertexes.

At the end, after a few algorithm trials, you should have a list of vertexes that constitutes the largest clique present in the graph.

# Navigating a Graph

*Navigating* or *traversing* a graph means visiting each of the graph nodes. The purpose of navigating a graph can include determining node content or updating it as needed. When navigating a graph, it's entirely possible that you visit particular nodes more than once because of the connectivity that graphs provide. Consequently, you also need to consider marking nodes as visited after you see their content. The act of navigating a graph is important in determining how the nodes connect so that you can perform various tasks. Previous chapters discuss basic graph navigation techniques. The following sections help you understand a few of the more advanced graph navigation techniques.

## Counting the degrees of separation

The term *degrees of separation* defines the distance between nodes in a graph. When working with an undirected graph without weighted edges, each edge counts for a value of one degree of separation. However, when working with other sorts of graphs, such as maps, where each edge can represent a distance or time value, the degrees of separation can become quite different. The point is that degrees of separation indicate some sort of distance. The example in this section (and the one that follows) relies on the following graph data. (You can find this code in the `A4D; 10; Graph Navigation.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

data  = {'A': ['B', 'F', 'H'],
         'B': ['A', 'C'],
         'C': ['B', 'D'],
         'D': ['C', 'E'],
         'E': ['D', 'F', 'G'],
         'F': ['E', 'A'],
         'G': ['E', 'H'],
         'H': ['G', 'A']}

graph = nx.DiGraph(data)
pos=nx.spring_layout(graph)
nx.draw_networkx_labels(graph, pos)
nx.draw_networkx_nodes(graph, pos)
nx.draw_networkx_edges(graph, pos)
plt.show()
```

This is an expansion of the graph used in Chapter 6. Figure 10-3 shows how this graph appears so that you can visualize what the function call is doing. Note that this is a directed graph (`networkx DiGraph`) because using a directed graph has certain advantages when determining degrees of separation (and performing a wealth of other calculations).

To discover the degrees of separation between two items, you must have a starting point. For the purpose of this example, you can use node ’A’. The following code shows the required `networkx` package function call and output:

```
nx.shortest_path_length(graph, 'A')

{'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 2, 'F': 1, 'G': 2,
 'H': 1}
```

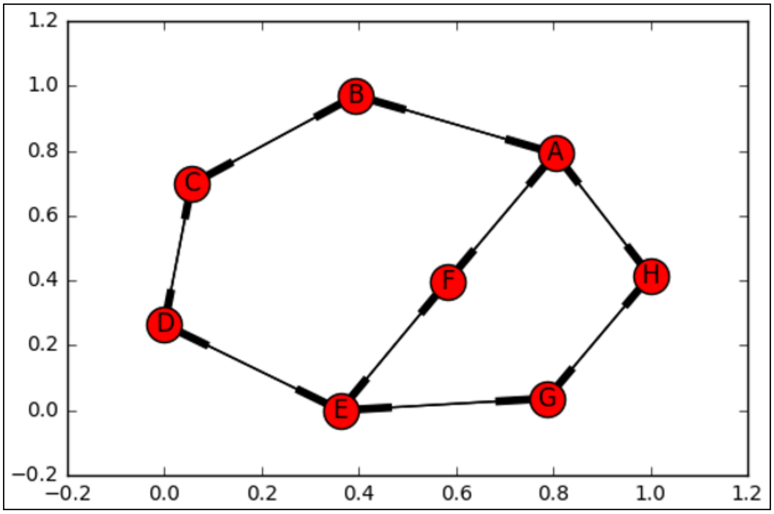


**FIGURE 10-3:** A sample graph used for navigation purposes.

The distance between node A and node A is 0, of course. The greatest degree of separation comes from node A to node D, which is 3. You can use this kind of information to determine which route to take or to perform an analysis of the cost in gas versus the cost in time of various paths. The point is that knowing the shortest distance between two points can be quite important. The `networkx` package used for this example comes in a wide array of distance-measuring algorithms, as described at `https://networkx.github.io/documentation/development/reference/algorithms.shortest_paths.html`.

TECHNICAL STUFF

To see how using a directed graph can make a big difference when performing degrees-of-separation calculations, try removing the connection between nodes A and F. Change the data so that it looks like this:

```
data  = {'A': ['B', 'H'],
         'B': ['A', 'C'],
         'C': ['B', 'D'],
         'D': ['C', 'E'],
         'E': ['D', 'F', 'G'],
         'F': ['E', 'A'],
         'G': ['E', 'H'],
         'H': ['G', 'A']}
```

When you perform the call to `nx.shortest_path_length` this time, the output becomes quite different because you can no longer go from A to F directly. Here's the new output from the call:

```
{'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 3, 'F': 4, 'G': 2,
 'H': 1}
```

Notice that the loss of the path has changed some of the degrees of separation. The distance to node F is now the longest at 4.

## Walking a graph randomly

You might find a need to walk a graph randomly. The act of walking the graph randomly, rather than look for a specific path, can simulate natural activities, such as an animal foraging for food. It also plays in to all sorts of other interesting activities, such as playing games. However, random graph walking can have practical aspects. For example, a car is held up in traffic because of an accident, so the shortest path is no longer available. In some cases, choosing a random alternative might work best because traffic along the second shortest route could be heavy as a result of the traffic jam along the shortest route.

REMEMBER

The networkx package doesn't provide the means for obtaining a random path directly. However, it does provide the means for finding all available paths, after which you can select a path from the list randomly. The following code shows how this process might work using the graph from the previous section.

```
import random
random.seed(0)

paths = nx.all_simple_paths(graph, 'A', 'H')

path_list = []
for path in paths:
    path_list.append(path)
    print("Path Candidate: ", path)

sel_path = random.randint(0, len(path_list) - 1)

print("The selected path is: ", path_list[sel_path])

Path Candidate:  ['A', 'B', 'C', 'D', 'E', 'G', 'H']
Path Candidate:  ['A', 'H']
Path Candidate:  ['A', 'F', 'E', 'G', 'H']
The selected path is:  ['A', 'H']
```

The code sets the seed to a specific value to ensure that you get the same result every time. However, by changing the seed value, you can see different results from the example code. The point is that even the simple graph shown in Figure 10-3 offers three ways to get from node A to node H (two of which are definitely longer than the selected path in this case). Choosing just one of them ensures that you get from one node to the other, albeit by a potentially roundabout way.

IN THIS CHAPTER

- **Understanding why finding what you want on the web is hard**
- **Reviewing problems that PageRank solves**
- **Implementing the PageRank algorithm with teleporting**
- **Learning how PageRank usage is evolving**

# Chapter 11

# Getting the Right Web page

The last few chapters review graphs at length. The web is one of the most interesting examples because of its extent and complexity. After providing an understanding of the basic algorithms that allow graph traversal and extraction of useful structures (such as the presence of clusters or communities), this chapter concludes the discussion of graphs by presenting the PageRank algorithm that has revolutionized people's lives as much as the web and Internet did because it makes the web usable. PageRank isn't only the engine behind Google and many other search engines, but it's also a smart way to derive latent information, such as relevance, importance, and reputation, from a graph structure.

Libraries rely on catalogues and librarians to offer an easy way to find particular texts or explore certain subjects. Books aren't all the same: Some are good at presenting certain kinds of information; some are better. Scholar recommendations make a book an authoritative source because these recommendations often appear in other books as quotes and citations. This sort of cross-reference didn't exist on the web initially. The presence of certain words in the title or in the text of the body recommended a particular web page. This approach is practically like judging a book by its title and the number of words it contains.

The PageRank algorithm changes all that by transforming the presence of the links on pages and turning them into recommendations, akin to the input of expert scholars. The growing scale of the web also plays a role in the success of the algorithm. Good signals are easy to find and distinguished from noise because they appear regularly. Noise, though confounding, is naturally casual. The larger the web, the more likely you are to get good signals for a smart algorithm like PageRank.

# Finding the World in a Search Engine

For many people, their personal and professional lives are unthinkable without the Internet and the web. The Internet network is composed of interconnected pages (among other things). The web is composed of sites that are reachable by domains, each one composed of pages and hyperlinks that connect sites internally and with other sites externally. Service and knowledge resources are available through the web if you know exactly where to look. Accessing the web is unthinkable without search engines, those sites that allow you to find anything on the web using a simple query.

## Searching the Internet for data

With an estimated size of almost 50 billion pages (`http://www.worldwidewebsize.com/`), the web isn't easy to represent. Studies describe the web as a bowtie shaped graph (see `http://www.immorlica.com/socNet/broder.pdf` and `http://vigna.di.unimi.it/ftp/papers/GraphStructureRevisited.pdf`). The web mainly consists of an interconnected core and other parts that link to that core. However, some parts are completely unreachable. By taking any road in the real world, you can go anywhere (you may have to cross the oceans to do it). On the web, you can't touch all the sites just by following its structure; some parts aren't easily accessible (they are disconnected or you're on the wrong side to reach them). If you want to find something on the web, even when time isn't a problem, you still need an index.

## Considering how to find the right data

Finding the right data has been a problem since the early years of the web, but the first search engines didn't appear until the 1990s. Search engines weren't thought of earlier because other solutions, such as simple domain listings or specialized site catalogues, worked fine. Only when these solutions stopped scaling well because of the rapidly growing size of the web did search engines such as Lycos, Magellan, Yahoo, Excite, Inktomi, and Altavista appear.

## URLs WITH .PDF EXTENSIONS

Many of the resource URLs found in this book have a .pdf extension. When you attempt to open the link, you may see a warning from your browser indicating that the .pdf file could contain a virus. It's entirely possible for a .pdf file to contain a virus (see `http://security.stackexchange.com/questions/64052/can-a-pdf-file-contain-a-virus` for a discussion of the topic). However, the research .pdf file links provided in this book are unlikely to contain viruses, so you can download them safely and then use a scanner to verify the content. As with any online content, you're always better off to be safe than sorry when it comes to files. Please do let us know if any of the .pdfs referenced in the book actually do contain viruses by writing to `John@JohnMuellerBooks.com`. In addition, please contact the webmaster for the site hosting the file.

All these search engines worked by having specialized software autonomously visit the web, using domain lists and testing hyperlinks found on the visited pages. These *spiders* explored each new link in a process called *crawling.* Spiders are pieces of software that read the pages as plain text (they can't understand images or other nontextual content).

Early search engines worked by crawling the web, collecting the information from spiders, and processing it in order to create inverted indexes. The indexes allowed retracing pages based on the words they contained. When you made a query, such inverted indexes reported all the pages containing the terms and helped score the pages, thus creating a ranking that turned into a search result (a list of ordered pages, ranging from the anticipated most useful page to the least useful page).

The scoring was quite naive because it often counted how frequently the keywords appeared on pages or whether they appeared in the titles or in the header of the page itself. Sometimes keywords were even scored more if they mixed or clustered together. Clearly, such simple indexing and scoring techniques allowed some web users to take advantage by using various tricks:

- **Web spammers:** Used their ability to fill the search results with pages containing poor content and a lot of advertising.
- **Black Hat search engine optimization (Black Hat SEO):** Used by people who employ their knowledge of search engines to make the search engine ranking higher for pages they manipulated despite their poor quality. Unfortunately, these issues still persist because every search engine, even the most evolved ones, aren't immune to people who want to game the system to obtain a higher search engine ranking. The PageRank algorithm may eliminate many of the older spammers and Black Hat SEO people, but it's not a panacea.

REMEMBER

It's essential to distinguish Black Hat SEO from White Hat SEO (usually simply SEO). People who use *White Hat SEO* are professionals who employ their knowledge of search engines to better promote valid and useful pages in a legal and ethical way.

The emergence of such actors and the possibility of manipulating search engines' results created the need for better ranking algorithms in search engines. One such result is the PageRank algorithm.

# Explaining the PageRank Algorithm

The PageRank algorithm is named after Google cofounder Larry Page. It made its first public appearance in a 1998 paper entitled "The Anatomy of a LargeScale Hypertextual Web Search Engine," by Sergey Brin and Larry Page, published by the journal *Computer Networks and ISDN Systems* (`http://ilpubs.stanford.edu:8090/361/1/1998-8.pdf`). At that time, both Brin and Page were PhD candidates, and the algorithm, the very foundation of Google's search technology, was initially a research project at Stanford University.

Simply stated, *PageRank* scores the importance of each node in a graph in such a way that the higher the score the more important the node in a graph. Determining the node importance in a graph like the web means computing whether a page is relevant as part of a query's results, thus better servicing users looking for good web content.

A page is a good response to a query when it matches the query's criteria and has prominence in the system of hyperlinks that ties pages together. The logic behind prominence is that because users build the web, a page has importance in the network for good reason (the quality and authority of the page's content is assessed by its importance in the web's network of hyperlinks).

## Understanding the reasoning behind the PageRank algorithm

In 1998, when both Brin and Page were still students at Stanford, the quality of search results was an issue for anyone using the web. Mainstream search engines of the time struggled both with an ever-growing web structure (the next part of the book discusses algorithm scaling issues and how to make them work with big data) and with a myriad of spammers.

REMEMBER

The use of spammers in this case doesn't refer to email spammers (those spammers who send unrequested emails to your Inbox) but rather to web spammers (those who know the economic importance of having pages at the top of search results). This group devised sophisticated and malicious tricks in order to fool search results. Popular hacks by web spammers of the day include:

- **Keyword stuffing:** Implies overusing particular keywords in a page to trick the search engine into thinking the page seriously discusses the keyword topic.
- **Invisible text:** Requires copying the content of a page result on top of a query into a different page using the same color for both characters and background. The copied content is invisible to users but not to the search engine's spiders (which were, and still are, just scanning textual data) and to its algorithms. The trick ranks the page with invisible text as high as the originating page in a search.
- **Cloaking:** Defines a more sophisticated variant of invisible text where, instead of text, scripts or images provide different content to search engine spiders than users actually see.

Web spammers use such tricks to trick search engines to rank pages highly, even though the page content is poor and, at best, misleading. These tricks have consequences. For instance, a user might look for information relating to university research and instead be exposed to commercial advertising or inappropriate content. Users became disappointed because they often ended up at pages unrelated to their needs, requiring them to restate their queries and to spend time digging for useful information among pages of results, wasting energy in distinguishing good references from bad ones. Scholars and experts, noting the need to cope with spam results and fearing that the development of the web could halt because users had difficulties finding what they really wanted, started working on possible solutions.

As Brin and Page worked on their solving algorithm, other ideas were drafted and publicized, or developed in parallel. One such idea is Hyper Search, by Massimo Marchiori, who first pointed out the importance of web links in determining the prominence of a web page as a factor to consider during a search: `https://www.w3.org/People/Massimo/papers/WWW6/`). Another interesting solution is a web search engine project called HITS (Hypertext-Induced Topic Search), also based on the web links structure and developed by Jon Kleinberg, a young scientist working at IBM Almaden in Silicon Valley. The interesting fact about HITS is that it classifies pages into *hubs* (a page with many links to authoritative pages) and *authorities* (pages considered authoritative by many links from hubs), something that PageRank doesn't do explicitly (but implicitly does in computations) (`http://www.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture4/lecture4.html`).

REMEMBER

When the time is ripe, the same idea or something similar often sprouts in different places. Sometimes sharing of basic ideas occurs between researchers and scientists; sometimes ideas are developed in a completely independent way (see the history of Japanese mathematician Takakazu Seki `http://www-history.mcs.st-andrews.ac.uk/history/Biographies/Seki.html`, who independently discovered many of the same things as European mathematicians such as Newton, Leibniz, and Bernoulli did around the same period). In 1998, only Brin and Page took steps to create a search engine company based on their algorithm by taking a leave from Stanford University and their doctoral studies to focus on making their algorithm work with more than a billion web pages.

## Explaining the nuts and bolts of PageRank

The innovation brought about by PageRank is that an inverted index of terms isn't enough to determine whether a page matches a user's information query. Matching words (or meaning, the semantic query match discussed at the end of the chapter) between a query and the page text is a prerequisite, but it isn't sufficient because hyperlinks are necessary to assess whether the page offers quality content and is authoritative.

When discussing sites, distinguishing between inbound and outbound links is important, and you shouldn't consider internal links that connect within the same site. The links you see on a page are *outbound* when they lead to another page on another site. The links that bring someone to your page from another page on another site are *inbound* links (backlinks). As the page creator, you use outbound links to provide additional information to the page content. You presumably won't use random links on your page (or links pointing to useless or bad content) because that would spoil the page quality. As you point to good content using links, other page creators use links on their pages to point to your page when your page is interesting and of high quality.

It's a chain of trust. Hyperlinks are like endorsements or recommendations for pages. Inbound links show that other page creators trust you, and you share part of that trust by adding outbound links on your pages to point to other pages.

# Implementing PageRank

Representing this chain of trust mathematically requires simultaneously determining how much authority your page has, as measured by inbound links, and how much it donates to other pages by outbound links. You can achieve such computations in two ways:

- **Simulation:** Uses the behavior of a web surfer who browses randomly on the web (a *random surfer*). This approach requires that you recreate the web structure and run the simulation.
- **Matrix computation:** Replicates the behavior of a random surfer using a *sparse matrix* (a matrix in which most data is zero) replicating the web structure. This approach requires some matrix operations, as explained in Chapter 5, and a series of computations that reach a result by successive approximation.

Even though it's more abstract, using the matrix computation for PageRank requires fewer programming instructions, and you can easily implement it using Python. (You can try the PageRank algorithm on real-world sites using an automatic PageRank checker, such as `http://checkpagerank.net/index.php`. Unfortunately, the program may produce inaccurate results for newer sites because they haven't been crawled properly yet, it can give you an idea of what PageRank is like in practice.)

## Implementing a Python script

PageRank is a function that scores the nodes in a graph with a number (the higher the number, the more important the node). When scoring a web page, the number could represent the probability of a random surfer visit. You express probabilities using a number from 0.0 to a maximum 1.0 and, ideally, when representing the probability of being on a particular site among all available sites, the sum of all the probabilities of the pages on the web should equal 1.0.

REMEMBER

Many versions of PageRank exist, each one changing its recipe a little to fit the kind of graph it has to score. The example in this section presents you with the original version for the web presented in the previously mentioned paper by Brin and Page and in the paper "PageRank: Bringing Order to the Web" (`http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf`).

The example creates three different web networks made of six nodes (web pages). The first one is a good working network, and the other two demonstrate problems that a random surfer may encounter because of the web structure or a web spammer's actions. This example also uses the NetworkX commands discussed in Chapter 8. (You can find this code in the `A4D; 11; PageRank.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline
```

```
Graph_A = nx.DiGraph()
Graph_B = nx.DiGraph()
Graph_C = nx.DiGraph()
Nodes = range(1,6)
Edges_OK = [(1,2),(1,3),(2,3),(3,1),(3,2),(3,4),(4,5),
            (4,6),(5,4),(5,6),(6,5),(6,1)]
Edges_dead_end = [(1,2),(1,3),(3,1),(3,2),(3,4),(4,5),
                  (4,6),(5,4),(5,6),(6,5),(6,1)]
Edges_trap = [(1,2),(1,3),(2,3),(3,1),(3,2),(3,4),(4,5),
              (4,6),(5,4),(5,6),(6,5)]
Graph_A.add_nodes_from(Nodes)
Graph_A.add_edges_from(Edges_OK)
Graph_B.add_nodes_from(Nodes)
Graph_B.add_edges_from(Edges_dead_end)
Graph_C.add_nodes_from(Nodes)
Graph_C.add_edges_from(Edges_trap)
```

This code displays the first network, the good one, as shown in Figure 11-1.

```
np.random.seed(2)
pos=nx.shell_layout(Graph_A)
nx.draw(Graph_A, pos, arrows=True, with_labels=True)
plt.show()
```

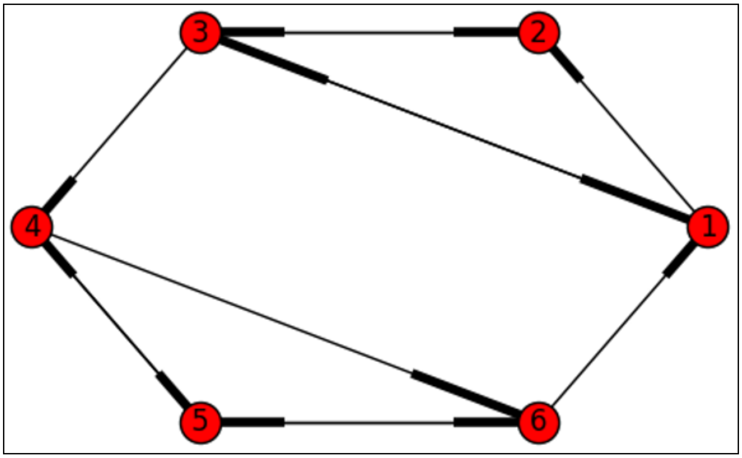


**FIGURE 11-1:** A strongly connected network.

TIP

All nodes connect with each other. This is an example of a strongly connected graph, which contains no isolated nodes or single nodes and enclaves that act as dead ends. A random surfer can freely run through it and never stop, and any node can reach any other node. In the NetworkX representation of a directed graph, there are no arrows, but the direction of an edge is represented by a thicker line entering a node. For example, a surfer can go from node 4 to node 6 because there is a thick line entering node 6 from node 4. However, the surfer can't go from node 6 to node 4 because the line entering node 4 from node 6 is thin.

The second graph isn't strongly connected. It presents a trap for a random surfer because the second node has no outbound links, and a user visiting the page could stop there and find no way out. This isn't an unusual event considering the structure of the web, but it could also show a spammer artifact, such that the spammer created a spam factory with many links that direct to a dead end site in order to trap web surfers. Figure 11-2 shows the output of the following code, which was used to display this graph.

```
np.random.seed(2)
pos=nx.shell_layout(Graph_B)
nx.draw(Graph_B, pos, arrows=True, with_labels=True)
plt.show()
```

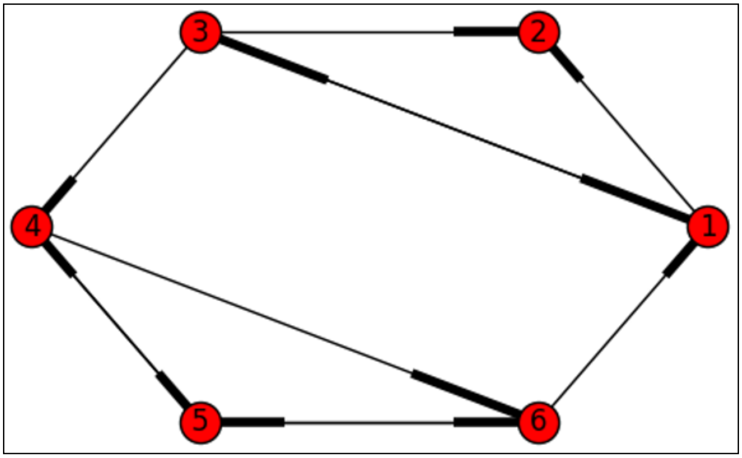


**FIGURE 11-2:** A dead end.

Another situation that may be natural or the result of a spammer's action is a spider trap. It's another dead end for a surfer, this time not on a single page but on a closed site that lacks links to an outside network of pages. Figure 11-3 shows the output of the following code, which was used to display this graph.

```
np.random.seed(2)
pos=nx.shell_layout(Graph_C)
nx.draw(Graph_C, pos, arrows=True, with_labels=True)
plt.show()
```

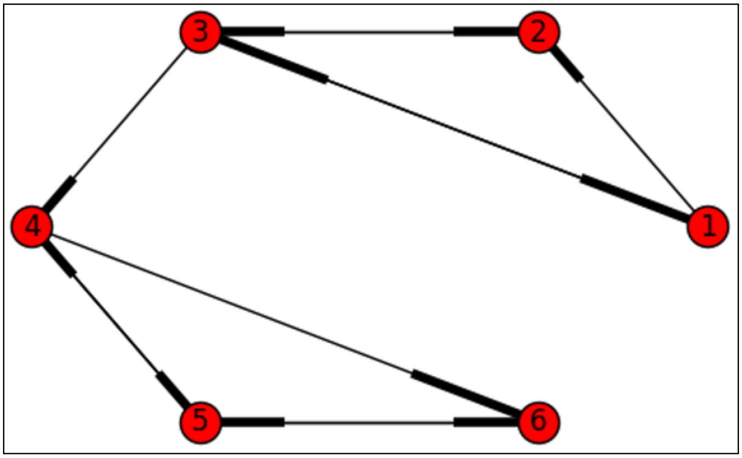


**FIGURE 11-3:** A spider trap.

REMEMBER

It’s called a *spider trap* because spammers devised it as a way to catch search engine software spiders in a loop and let them believe that the only websites were the ones inside the closed network.

## Struggling with a naive implementation

Given a graph made by using Python and NetworkX, you can extract its structure and render it as a *transition matrix,* a matrix that represents nodes in columns and rows:

- **Columns:** Contain the node a web surfer is on
- **Rows:** Contain the probability that the surfer will visit other nodes because of outbound links

In the real web, the transition matrix that feeds the PageRank algorithm is built by spiders’ continuous exploration of links.

```
def initialize_PageRank(graph):
    nodes = len(graph)
    M = nx.to_numpy_matrix(graph)
    outbound = np.squeeze(np.asarray(np.sum(M, axis=1)))
    prob_outbound = np.array(
        [1.0/count
         if count>0 else 0.0 for count in outbound])
    G = np.asarray(np.multiply(M.T, prob_outbound))
    p = np.ones(nodes) / float(nodes)
    if np.min(np.sum(G,axis=0)) < 1.0:
        print ('Warning: G is substochastic')
    return G, p
```

The Python code creates the function `initialize_PageRank` that extracts both the transition matrix and the initial vector of default PageRank scores.

```
G, p = initialize_PageRank(Graph_A)
print (G)

[[ 0.      0.      0.33333333  0.      0.      0.5 ]
 [ 0.5     0.      0.33333333  0.      0.      0.  ]
 [ 0.5     1.      0.          0.      0.      0.  ]
 [ 0.      0.      0.33333333  0.      0.5     0.  ]
 [ 0.      0.      0.          0.5     0.      0.5 ]
 [ 0.      0.      0.          0.5     0.5     0.  ]]
```

The printed transition matrix `G` represents the transition matrix of the network described in Figure 11-1. Each column represents a node in the sequence 1 through 6. For instance, the third column represents node 3. Each row in the column shows the connections with other nodes (outbound links toward nodes 1, 2, and 4) and values that define the probability of a random surfer using any of the outbound links (that is, 1/3, 1/3, 1/3).

TIP

The diagonal of the matrix is always zero unless a page has an outbound link toward itself (it is a possibility).

The matrix contains more zeros than values. This is also true in reality because estimates show that each site has only ten outbound links on average. Because billions of sites exist, the nonzero values in a transition matrix representing the web are minimal. In this case, it's helpful to use a data structure such as an adjacency list (explained in Chapter 8) to represent data without wasting disk or memory space with zero values:

```
from scipy import sparse
sG = sparse.csr_matrix(G)
print (sG)

  (0, 2)   0.333333333333
  (0, 5)   0.5
  (1, 0)   0.5
  (1, 2)   0.333333333333
  (2, 0)   0.5
  (2, 1)   1.0
  (3, 2)   0.333333333333
  (3, 4)   0.5
  (4, 3)   0.5
  (4, 5)   0.5
  (5, 3)   0.5
  (5, 4)   0.5
```

This example has just 12 links out of 30 possible (without counting links to *self*, which is the current site). Another particular aspect of the transition matrix to note is that if you sum the columns, the result should be 1.0. If it is a value less than 1.0, the matrix is *substochastic* (which means that the matrix data isn't representing probabilities properly because probabilities should sum to 1.0) and cannot work perfectly with PageRank estimations.

Accompanying G is a vector p, the initial estimate of the total PageRank score, equally distributed among the nodes. In this example, because the total PageRank is 1.0 (the probability of a random surfer being in the network, which is 100 percent), it's distributed as 1/6 among the six nodes:

```
print(p)

[ 0.16666667  0.16666667  0.16666667  0.16666667
  0.16666667  0.16666667]
```

To estimate the PageRank, take the initial estimate for a node in the vector p, multiply it by the corresponding column in the transition matrix, and determine how much of its PageRank (its authority) transfers to other nodes. Repeat for all nodes and you'll know how PageRank transfers between nodes because of the network structure. You can achieve this computation using a matrix-vector multiplication:

```
print(np.dot(G,p))

[ 0.13888889  0.13888889  0.25        0.13888889
  0.16666667  0.16666667]
```

After the first matrix-vector multiplication, you obtain another estimate of PageRank that you use for redistribution among the nodes. By redistributing multiple times, the PageRank estimate stabilizes (results won't change), and you'll have the score you need. Using a transition matrix containing probabilities and estimation by successive approximation using matrix-vector multiplication obtains the same results as a computer simulation with a random surfer:

```
def PageRank_naive(graph, iters = 50):
    G, p = initialize_PageRank(graph)
    for i in range(iters):
        p = np.dot(G,p)
    return np.round(p,3)

print(PageRank_naive(Graph_A))

[ 0.154  0.154  0.231  0.154  0.154  0.154]
```

The new function `PageRank_naive` wraps all the previously described operations and emits a vector of probabilities (the PageRank score) for each node in the network. The third node emerges as the one with most importance. Unfortunately, the same function doesn't work with the other two networks:

```
print(PageRank_naive(Graph_B))
Warning: G is substochastic
[ 0.  0.  0.  0.  0.  0.]

print(PageRank_naive(Graph_C))
[ 0.     0.     0.     0.222  0.444  0.333]
```

In the first case, the probabilities seem to drain out of the network — the effect of a dead-end website and the resulting substochastic transition matrix. In the second case, the bottom half of the network unfairly gets all the importance, leaving the top part as insignificant.

## Introducing boredom and teleporting

Both dead ends (*rank sinks*) and spider traps (*cycles*) are common situations on the web because of users' choices and spammers' actions. The problem, however, is easily solved by making the random surfer randomly jump to another network node (*teleporting*, as in the sci-fi devices that take you instantaneously from one place to another). The theory is that a surfer will get bored at one point or another and move away from deadlocking situations. Mathematically, you define an alpha value representing the probability of continuing the random journey on the graph by the surfer. The alpha value redistributes the probability of being on a node independently of the transition matrix.

TIP

The value originally suggested by Brin and Page for alpha (also called the damping factor) is 0.85, but you can change it according to your needs. For the web, it works the best between 0.8 and 0.9 and you can read why this is the best value range at `https://www.cise.ufl.edu/~adobra/DaMn/talks/damn05-santini.pdf`. The smaller the alpha value, the shorter the trip of the surfer on the network, on average, before restarting somewhere else.

```
def PageRank_teleporting(graph, iters = 50, alpha=0.85,
                         rounding=3):
    G, p = initialize_PageRank(graph)
    u = np.ones(len(p)) / float(len(p))
    for i in range(iters):
        p = alpha * np.dot(G,p) + (1.0 - alpha) * u
    return np.round(p / np.sum(p), rounding)
```

```
print('Graph A:', PageRank_teleporting(Graph_A,
                                       rounding=8))
print('Graph B:', PageRank_teleporting(Graph_B,
                                       rounding=8))
print('Graph C:', PageRank_teleporting(Graph_C,
                                       rounding=8))

Graph A: [ 0.15477863  0.15346061  0.22122243  0.15477863
           0.15787985  0.15787985]
Warning: G is substochastic
Graph B: [ 0.16502904  0.14922238  0.11627717  0.16502904
           0.20222118  0.20222118]
Graph C: [ 0.0598128   0.08523323  0.12286869  0.18996342
           0.30623677  0.23588508]
```

After applying the modifications to a new function, `PageRank_teleporting`, you can get similar estimates for the first graph and much more realistic (and useful) estimates for both the second and third graphs, without falling into the traps of dead ends or rank sinks. Interestingly, the function is equivalent to the one provided by NetworkX: `http://networkx.readthedocs.io/en/networkx-1.11/reference/generated/networkx.algorithms.link_analysis.pagerank_alg.pagerank.html`.

```
nx.pagerank(Graph_A, alpha=0.85)

{1: 0.15477892494151968,
 2: 0.1534602056628941,
 3: 0.2212224378270561,
 4: 0.15477892494151968,
 5: 0.1578797533135051,
 6: 0.15787975331350507}
```

## Looking inside the life of a search engine

Though it only reports on the web hyperlink structure, PageRank reveals how authoritative a page can become. However, Google isn't composed only of PageRank. The algorithm provides solid foundations for any query, and it initially bootstrapped Google's fame as a reliable search engine. Today, PageRank is just one of the many ranking factors that intervene when processing a query.

Specialized sources in SEO knowledge quote more than 200 factors as contributing to the results that Google provides. To see what other sorts of ranking factors Google considers, consult the list at `https://moz.com/search-ranking-factors` (made by MOZ, a U.S. company). You can also download the yearly reports from `http://www.searchmetrics.com/knowledge-base/ranking-factors/`, by Searchmetrics, a German company from Berlin specializing in SEO software.

You must also consider that the Google algorithm has received many updates, and at this point, it's more of an ensemble of different algorithms, each one named with a fantasy name (Caffeine, Panda, Penguin, Hummingbird, Pigeon, Mobile Update). Many of these updates have caused shake-ups of previous search rankings and were motivated by the need to fix spamming techniques or make the surfing the web more useful for users (for instance, the Mobile Update induced many sites to render their interfaces mobile-phone friendly).

## Considering other uses of PageRank

Although PageRank provides better search results, its applicability isn't limited to Google or search engines. You can use PageRank anywhere you can reduce your problem to a graph. Just modify and tune the algorithm to your needs. Cornell University has enumerated some other potential uses of PageRank in different sectors (`https://blogs.cornell.edu/info2040/2014/11/03/more-than-just-a-web-search-algorithm-googles-pagerank-in-non-internet-contexts/`), and surprising reports have emerged of the algorithm being successfully used in computational biology (`https://www.wired.com/2009/09/googlefoodwebs/`). By creating a teleportation tied to specific nodes that you want to explore, you see the algorithm shining at diverse applications such as the following:

- **Fraud detection:** Revealing how certain persons and facts are related in unexpected ways
- **Product recommendation:** Suggesting products that a person with a certain affinity might like

# Going Beyond the PageRank Paradigm

In recent years, Google has done more than introduce more ranking factors that modify the original PageRank algorithm. It has introduced some radical changes that leverage page content better (to avoid being fooled by the presence of certain keywords) and has adopted AI algorithms that rank the relevance of a page in a search result autonomously. These changes have led some search experts to declare that PageRank doesn't determine the position of a page in a search any longer (see `https://www.entrepreneur.com/article/269574`). They still debate the question, but it's most likely safe to assume that PageRank is still powering the Google engine as a ranking factor, albeit not sufficiently to enlist a page into the best results after a query.

## Introducing semantic queries

If you currently try to pose questions, not just chains of keywords, to Google, you'll notice that it tends to answer smartly and gets the sense of question. Since 2012, Google became better able to understand synonyms and concepts. However, after August 2013, with the Hummingbird update (`http://searchengineland.com/google-hummingbird-172816`), the search engine became capable of understanding conversational searches (queries in which you ask something as you would say it to another person) as well as the semantics behind queries and a page's contents.

Since this update, the Google algorithm works by disambiguating both users' intentions and the meanings expressed by pages, not just by the keywords. Now, the search engine works more in a semantic way, which means understanding what words imply on both sides: the query and resulting web pages. In this sense, it can't be tricked anymore by playing with keywords. Even without much support from PageRank, it can look at how a page is written and get a sense of whether the page contains good enough content for inclusion in the results of a query.

## Using AI for ranking search results

PageRank is still at the core, but the results have less weight because of the introduction of machine learning technology into ranking, the so-called RankBrain. According to some sources (see `https://www.bloomberg.com/news/articles/2015-10-26/google-turning-its-lucrative-web-search-over-to-ai-machines`), the machine learning algorithm now examines all Google queries and directly handles 15 percent of the volume of search queries it receives every day, specializing in

- Ambiguous and unclear search queries
- Queries expressed in slang or colloquial terms
- Queries expressed as though they were occurring in a conversation with the search engine

Even though RankBrain is still cloaked in secrecy, the algorithm seems to be capable of guessing, with much higher accuracy than performed by a human being, whether the contents of a page can appear in search results. It replaces all other ranking factors in cases that are difficult to judge. This is another example of an additional algorithm that limits the role played by the original PageRank algorithm.

# 4 Struggling with Big Data

**IN THIS PART . . .**

Interact with large datasets.

Work with streamed data to use even larger datasets.

Perform tasks in parallel to perform management and analysis tasks faster.

Encode data to reduce redundancies and to keep data safe.

Compress and decompress data using the LZW algorithm.

IN THIS CHAPTER

- **Realizing why big data is a driving force of our times**
- **Becoming familiar with Moore's Law and its implications**
- **Understanding big data and its 4 *V*s**
- **Discovering how to deal with infinite streaming data**
- **Leveraging sampling, hashing, and sketches for stream data**

# Chapter 12
# Managing Big Data

More than a buzzword used by vendors to propose innovative ways to store data and analyze it, big data is a reality and a driving force of our times. You may have heard it mentioned in many specialized scientific and business publications and even wondered what big data really means. From a technical point of view, big data refers to large and complex amounts of computer data, so large (as the name implies) and intricate that the data cannot be dealt with by making more storage available on your computers or by making new computers more powerful and faster in their computations. Big data implies a revolution in the way you store and deal with data.

However, this copious and sophisticated store of data didn't appear suddenly. It took time to develop the technology to store this amount of data. In addition, it took time to spread the technology that generates and delivers data, namely computers, sensors, smart mobile phones, the Internet, and its World Wide Web services. This chapter discusses what drives this huge data production.

Even though it took time to build this much data, technology changes in recent years have finally helped people realize the potential game changer that having huge amounts of data (of any nature) at hand represents. For centuries, humans have emphasized the power of the human intellect in determining the causes and forces driving the natural world using a few accurate observations (small data).

Humans also developed a method, the scientific method, that is at the foundation of our modern world based on scientific discovery. Suddenly people have found (with a certain surprise) that they can solve problems earlier and more successfully by learning the solution from large amounts of data rather than spending long years developing and elaborating theories using well-designed tests and experiments.

Simply having data won't find solutions to the many problems still afflicting civilization. However, having enough data, which actually equates to incredible amounts of it, and the right algorithm enables people to connect the dots between thousands and thousands of hints. Big data and algorithms enable access to wondrous new scientific (but also practical and business) discoveries.

# Transforming Power into Data

In 1965, Gordon Moore, cofounder of Intel and Fairchild Semiconductor (two giant companies that produce electronic components for electronics and computers), stated in an *Electronics* magazine paper entitled "Cramming More Components Onto Integrated Circuits" that the number of components found in integrated circuits would double every year for the next decade. At that time, transistors dominated electronics. Being able to stuff more transistors into a circuit using a single electronic component that gathered the functionalities of many of them (an integrated circuit), meant being able to make electronic devices more capable and useful. This process is *integration* and implies a strong process of electronics miniaturization (making the same circuit much smaller, which makes sense because the same volume should contain double the circuitry as the previous year).

As miniaturization proceeds, electronic devices, the final product of the process, become smaller or simply more powerful. For instance, today's computers aren't all that much smaller than computers of a decade ago, yet they are decisively more powerful. The same goes for mobile phones. Even though they're the same size as their predecessors, they have become able to perform more tasks. Other devices, such as sensors, are simply smaller, which means that you can put them everywhere.

## Understanding Moore's implications

What Moore stated in that article actually proved true for many years, and the semiconductor industry calls it Moore's Law (see `http://www.mooreslaw.org/` for details). Doubling did occur for the first ten years, as predicted. In 1975, Moore corrected his statement, forecasting a doubling every two years. Figure 12-1 shows the effects of this doubling. This rate of doubling is still valid, although now it's common opinion that it won't hold longer than the end of the present decade (up to about 2020). Starting in 2012, a mismatch occurs between the expectation of cramming more transistors into a component to make it faster and what semiconductor companies can achieve with regard to miniaturization. In truth, physical barriers exist to integrating more circuitry into an integrated circuit using the present silica components. (However, innovation will continue; you can readthearticleat`http://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338` for more details.) In addition, Moore's Law isn't actually a law. Physical laws, such as the law of universal gravitation (which explains why things are attracted to the ground as discovered by Newton), are based on proofs of various sorts that have received peer review for their accuracy. Moore's Law isn't anything more than mere observation, or even a tentative goal for the industry to strive to achieve (a self-fulfilling prophecy, in a certain sense).

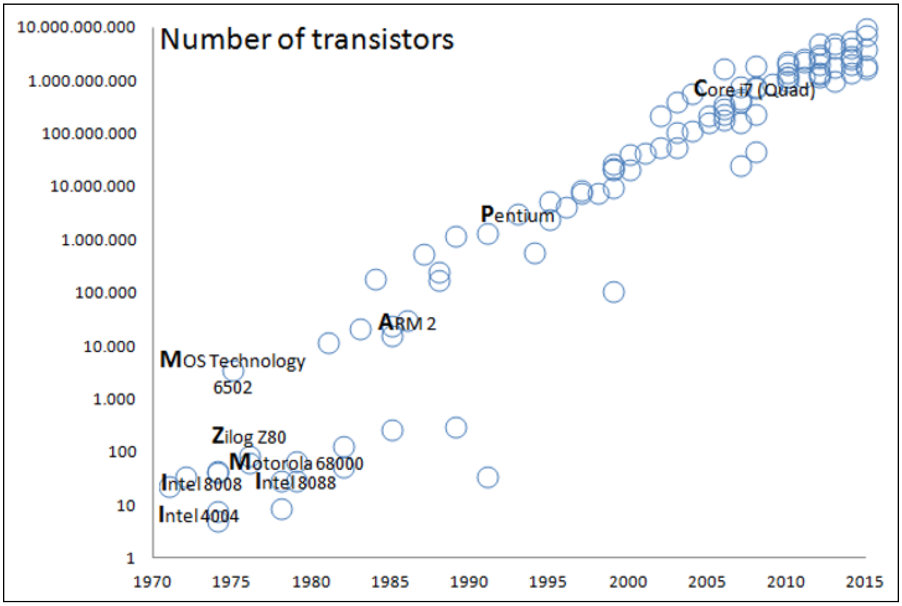


**FIGURE 12-1:** Stuffing more and more transistors into a CPU.

In the future, Moore's Law may not apply anymore because industry will switch to new technology (such as making components by using optical lasers instead of transistors; see the article at `http://www.extremetech.com/extreme/187746-by-2020-you-could-have-an-exascale-speed-of-light-optical-computer-on-your-desk` for details about optical computing). What matters is that since 1965, about every two years the computer industry experienced great advancements in digital electronics that had consequences.

### CONSIDERING THE POLITICAL ASPECTS OF VARIOUS LAWS

Depending on whom you talk with, the whole issue of whether a law will stand the test of time can look different because that person will have a different perspective. This book isn't here to convince you of one point of view or another, but simply reports the prevalent view. For example, it's possible to argue that Moore's Law is every bit proven as the laws of thermodynamics. If you look into conventional physics more, you can find many discrepancies with its laws and many of its assumptions. It's not a matter of devaluing science in any way — just pinpointing the fact that everything in science, including its laws, is a work in progress.

As to whether Moore's Law will cease to exist, generally speaking, laws don't stop applying; scientists refurbish them so that they are more general. Moore's Law may undergo the same transformation. Linear or overly simplistic laws rarely apply in a general sense because there are no straight lines anywhere in nature, including its temporal models. Therefore, the most likely scenario is that Moore's Law will change into a more sigmoidal function in an attempt to adhere to reality.

REMEMBER

Some advocate that Moore's Law already no longer holds. The chip industry has kept up the promise so far, but now it's lowering expectations. Intel has already increased the time between its generations of CPUs, saying that in five years, chip miniaturization will hit a wall. You can read this interesting story on the MIT Technology Review at `https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/`.

Moore's Law has a direct effect on data. It begins with smarter devices. The smarter the devices, the more diffusion (electronics are everywhere in our day and age). The greater the diffusion, the lower the price becomes, creating an endless loop that drove and is driving the use of powerful computing machines and small sensors everywhere. With large amounts of computer memory available and larger storage disks for data, the consequences are an expansion of the availability of data, such as websites, transaction records, a host of various measurements, digital images, and other sorts of data flooding from everywhere.

## Finding data everywhere

Scientists began fighting against impressive amounts of data for years before anyone coined the term *big data.* At this point, the Internet didn't produce the vast sums for data that it does today. It's useful to remember that big data is not just simply a fad created by software and hardware vendors but has a basis in many of the following fields:

- **Astronomy:** Consider the data received from spacecraft on a mission (such as Voyager or Galileo) and all the data received from radio telescopes, which are specialized antennas used to receive radio waves from astronomical bodies. A common example is the Search for Extraterrestrial Intelligence (SETI) project (`http://www.seti.org/`), which looks for extraterrestrial signals by observing radio frequencies arriving from space. The amount of data received and the computer power used to analyze a portion of the sky for a single hour is impressive (`http://www.setileague.org/askdr/howmuch.htm`). If aliens are out there, it's very hard to spot them. (The movie *Contact* `https://www.amazon.com/exec/obidos/ASIN/B002GHHHKQ/datacservip0f-20/` explores what could happen should humans actually intercept a signal.)
- **Meteorology:** Think about trying to predict weather for the near term given the large number of required measures, such as temperature, atmospheric pressure, humidity, winds, and precipitation at different times, locations, and altitudes. Weather forecasting is really one of the first problems in big data and quite a relevant one. According to Weather Analytics, a company that provides climate data, more than 33 percent of Worldwide Gross Domestic Product (GDP) is determined by how weather conditions affect agriculture, fishing, tourism, and transportation, just to name a few. Dating back to the 1950s, the first supercomputers of the time were used to crunch as much data as possible because, in meteorology, the more data, the more accurate the forecast. That's the reason everyone is amassing more storage and processing capacity, as you can read in this story regarding the Korean Meteorological Association `https://www.wired.com/insights/2013/02/how-big-data-can-boost-weather-forecasting/` for weather forecasting and studying climate change.
- **Physics:** Consider the large amounts of data produced by experiments using particle accelerators in an attempt to determine the structure of matter, space, and time. For example, the Large Hadron Collider (`https://home.cern/topics/large-hadron-collider`), the largest particle accelerator ever created, produces 15PB (petabytes) of data every year as a result of particle collisions (`http://home.web.cern.ch/about/computing`).
- **Genomics:** Sequencing a single DNA strand, which means determining the precise order of the many combinations of the four bases — adenine, guanine, cytosine, and thymine — that constitute the structure of the molecule, requires quite a lot of data. For instance, a single chromosome, a structure containing the DNA in the cell, may require from 50MB to 300MB. A human being has 46 chromosomes, and the DNA data for just one person consumes an entire DVD. Just imagine the massive storage required to document the DNA data of a large number of people or to sequence other life forms on earth (`https://www.wired.com/2013/10/big-data-biology/`).

- **Oceanography:** Because of the many sensors placed in the oceans to measure temperature, currents, and, using hydrophones, even sounds for acoustic monitoring for scientific purposes (discovering about fish, whales, and plankton) and military defense purposes (finding sneaky submarines from other countries). You can have a sneak peek at this old surveillance problem, which is turning more complex and digital, by reading this article: `http://www.theatlantic.com/technology/archive/2014/08/listening-in-the-navy-is-tracking-ocean-sounds-collected-by-scientists/378630/`.
- **Satellites:** Recording images from the entire globe and sending them back to earth in order to monitor the Earth's surface and its atmosphere isn't a new business (TIROS 1, the first satellite to send back images and data, dates back to 1960). Over the years, however, the world has launched more than 1,400 active satellites that provide earth observation. The amount of data arriving on earth is astonishing and serves both military (surveillance) and civilian purposes, such as tracking economic development, monitoring agriculture, and monitoring changes and risks. A single European Space Agency's satellite, Sentinel 1A, generates 5PB of data during two years of operation, as you can read at `https://spaceflightnow.com/2016/04/28/europes-sentinel-satellites-generating-huge-big-data-archive/`).

Accompanying these older data trends, new amounts of data are now generated or carried about by the Internet, creating new issues and requiring solutions in terms of both data storage and algorithms for processing:

- As reported by the National Security Agency (NSA), the amount of information flowing through the Internet every day from all over the world amounted to 1,826PB of data in 2013, and 1.6 percent of it consisted of e-mails and telephone calls. To assure national security, the NSA must verify the content of at least 0.025 percent of all emails and phone calls (looking for key words that could signal something like a terrorist plot). That still amounts to 25PB per year, which equates to 37,500 CD-ROMs every year of data stored and analyzed (and that's growing). You can read the full story at `http://www.business-standard.com/article/news-ani/nsa-claims-analysts-look-at-only-0-00004-of-world-s-internet-traffic-for-surveillance-113081100075_1.html`.
- The Internet of Things (IoT) is becoming a reality. You may have heard the term many times in the last 15 years, but now the growth of the stuff connected to the Internet is going to explode. The idea is to put sensors and transmitters on everything and use the data to both better control what happens in the world and to make objects smarter. Transmitting devices are getting tinier, cheaper and less power demanding; some are already so small that they can be put everywhere. (Just look at the ant-sized radio developed by

Stanford engineers at http://news.stanford.edu/news/2014/september/ant-radio-arbabian-090914.html.) Experts estimate that by 2020, there will be six times as many connected things on earth as there will be people, but many research companies and think tanks are already revisiting those figures (http://www.gartner.com/newsroom/id/3165317).

## Getting algorithms into business

The human race is now at an incredible intersection of unprecedented volumes of data, generated by increasingly smaller and powerful hardware, and analyzed by algorithms that this same process helped develop. It's not simply a matter of volume, which by itself is a difficult challenge. As formalized by the research company Gartner in 2001 and then reprised and expanded by other companies, such as IBM, big data can be summarized by four *V*s representing its key characteristics (you can read more on this topic at http://www.ibmbigdatahub.com/infographic/four-vs-big-data):

- **Volume:** The amount of data
- **Velocity:** The speed of data generation
- **Variety:** The number and types of data sources
- **Veracity:** The quality and authoritative voice of the data (quantifying errors, bad data, and noise mixed with signals), a measure of the uncertainty of the data

TIP

Each big data characteristic offers a challenge and an opportunity. For instance, volume considers the amount of useful data. What one organization considers big data could be small data for another one. The inability to process the data on a single machine doesn't make the data big. What differentiates big data from the business-as-usual data is that it forces an organization to revise its prevalent methods and solutions, and pushes present technologies and algorithms to look ahead.

Variety enables the use of big data to challenge the scientific method, as explained by this milestone and much discussed article written by Chris Anderson, *Wired*'s editor-in-chief at the time, on how large amounts of data can help scientific discoveries outside the scientific method: https://www.wired.com/2008/06/pb-theory/. The author relies on the example of Google in the advertising and translation business sectors, where the company could achieve prominence without using specific models or theories, but by applying algorithms to learn from data. As in advertising, science (physics, biology) data can support innovation that allows scientists to approach problems without hypotheses but by considering the variations found in large amounts of data and by discovery algorithms.

The veracity characteristic helps the democratization of data itself. In the past, organizations hoarded data because it was precious and difficult to obtain. At this point, various sources create data in such growing amounts that hoarding it is meaningless (90 percent of the world's data has been created in the last two years), so there is no reason to limit access. Data is turning into such a commodity that there are many open data programs going all around the world. (The United States has a long tradition of open access; the first open data programs date back to the 1970s when the National Oceanic and Atmospheric Administration, NOAA, started releasing weather data freely to the public.) However, because data has become a commodity, the uncertainty of that data has become an issue. You no longer know whether the data is completely true because you may not even know its source.

REMEMBER

Data has become so ubiquitous that its value is no longer in the actual information (such as data stored in a firm's database). The value of data exists in how you use it. Here algorithms come into play and change the game. A company like Google feeds itself from freely available data, such as the content of websites or the text found in publicly available texts and books. Yet, the value Google extracts from the data mostly derives from its algorithms. As an example, data value resides in the PageRank algorithm (illustrated in Chapter 11), which is the very foundation of Google's business. The value of algorithms is true for other companies as well. Amazon's recommendation engine contributes a significant part of the company's revenues. Many financial firms use algorithmic trading and robo-advice, leveraging freely available stock data and economic information for investments.

## Streaming Flows of Data

When data flows in huge amounts, storing it all may be difficult or even impossible. In fact, storing it all might not even be useful. Here are some figures of just some of what you can expect to happen within a single minute on the Internet:

- 150 million e-mails sent
- 350,000 new tweets sent on Twitter
- 2.4 million queries requested on Google
- 700,000 people logged in to their account on Facebook

Given such volumes, accumulating the data all day for incremental analysis might not seem efficient. You simply store it away somewhere and analyze it on the following or on a later day (which is the widespread archival strategy that's typical of databases and data warehouses). However, useful data queries tend to ask about the most recent data in the stream, and data becomes less useful when it ages (in some sectors, such as financial, a day can be a lot of time).

Moreover, you can expect even more data to arrive tomorrow (the amount of data increases daily) and that makes it difficult, if not impossible, to pull data from repositories as you push new data in. Pulling old data from repositories as fresh data pours in is akin to the punishment of Sisyphus. Sisyphus, as a Greek myth narrates, received a terrible punishment from the god Zeus: Being forced to eternally roll an immense boulder up on the top of a hill, only to watch it roll back down each time (see `http://www.mythweb.com/encyc/entries/sisyphus.html` for additional details).

Sometimes, rendering things even more impossible to handle, data can arrive so fast and in such large quantities that writing it to disk is impossible: New information arrives faster than the time required to write it to the hard disk. This is a problem typical of particle experiments with particle accelerators such as the Large Hadron Collider, requiring scientists to decide what data to keep (`http://home.cern/about/computing/processing-what-record`). Of course, you may queue data for some time, but not for too long, because the queue will quickly grow and become impossible to maintain. For instance, if kept in memory, queue data will soon lead to an out-of-memory error.

Because new data flows may render the previous processing on old data obsolete, and procrastination is not a solution, people have devised multiple strategies to deal instantaneously with massive and changeable data amounts. People use three ways to deal with large amounts of data:

- **Stored:** Some data is stored because it may help answer unclear questions later. This method relies on techniques to store it immediately and analyze it later very fast, no matter how massive it is.
- **Summarized:** Some data is summarized because keeping it all as it is makes no sense; only the important data is kept.
- **Consumed:** The remaining data is consumed because its usage is predetermined. Algorithms can instantly read, digest, and turn the data into information. After that, the system forgets the data forever.

The book deals with the first point in Chapter 13, which is about distributing data among multiple computers and understanding the algorithms used to deal with it (a divide-and-conquer strategy). The following sections address the second and third points, applying them to data that streams in systems.

TIP

When talking of massive data arriving into a computer system, you will often hear it compared to water: streaming data, data streams, data fire hose.

You discover how data streams is like consuming tap water: Opening the tap lets you store the water in cups or drinking bottles, or you can use it for cooking, scrubbing food, cleaning plates, or washing hands. In any case, most or all of the water is gone, yet it proves very useful and indeed vital.

## Analyzing streams with the right recipe

Streaming data needs streaming algorithms, and the key thing to know about streaming algorithms is that, apart a few measures that it can compute exactly, a streaming algorithm necessarily provides approximate results. The algorithm output is almost correct, guessing not the precisely right answer, but close to it.

When dealing with streams, you clearly have to concentrate only on the measures of interest and leave out many details. You could be interested in a statistical measurement, such as mean, minimum, or maximum. Moreover, you could want to count elements in the stream or distinguish old information from new. There are many algorithms to use, depending on the problem, yet the recipes always use the same ingredients. The trick of cooking the perfect stream is to use one or all of these algorithmic tools as ingredients:

- **Sampling:** Reduce your stream to a more manageable data size; represent the entire stream or the most recent observations using a shifting data window.
- **Hashing:** Reduce infinite stream variety to a limited set of simple integer numbers (as seen in the "Relying on Hashing" section of Chapter 7).
- **Sketching:** Create a short summary of the measure you need, removing the less useful details. This approach lets you leverage a simple working storage, which can be your computer's main memory or its hard disk.

Another characteristic to keep in mind about algorithms operating on streams is their simplicity and low computational complexity. Data streams can be quite fast. Algorithms that require too many calculations can miss essential data, which means that the data is gone forever. When you view the situation in this light, you can appreciate how hash functions prove useful because they're prompt in transforming inputs into something easier to handle and search because for both operations, complexity is O(1). You can also appreciate the sketching and sampling techniques, which bring about the idea of *lossy compression* (more on compression in Chapter 14). Lossy compression enables you to represent something complex by using a simpler form. You lose some detail but save a great deal of computer time and storage.

Sampling means drawing a limited set of examples from your stream and treating them as if they represented the entire stream. It is a well-known tool in statistics through which you can make inferences on a larger context (technically called the *universe* or the *population*) by using a small part of it.

## Reserving the right data

Statistics was born in a time when obtaining a census was impossible. A *census* is a systematic investigation on a population, counting it, and acquiring useful data from it. The government asks all the people in a country about where they live, their family, their daily life, and their work. The census has its origins in ancient times. In the Bible, a census occurs in the book of Numbers; the Israelite population is counted after the exodus from Egypt. For tax purposes, the ancient Romans periodically held a census to count the population of their large empire. Historical documents provide accounts of similar census activities in ancient Egypt, Greece, India, and China.

Statistics, in particular the branch of statistics called inferential statistics, can achieve the same outcome as a census, with an acceptable margin of error, by interrogating a smaller number of individuals (called a *sample*). Thus, by querying a few people, pollsters can determine the general opinion of a larger population on a variety of issues, such as who will win an election. In the United States, for instance, the statistician Nate Silver made news by predicting the winner of the 2012 presidential election in all 50 states, using data from samples (`https://www.cnet.com/news/obamas-win-a-big-vindication-for-nate-silver-king-of-the-quants/`).

Clearly, holding a census implies huge costs (the larger the population, the greater the costs) and requires a lot of organization (which is why censuses are infrequent), whereas a statistical sample is faster and cheaper. Reduced costs and lower organizational requirements also make statistics ideal for big data streaming: Users of big data streaming don't need every scrap of information and they can summarize the data's complexity.

However, there's a problem with using statistical samples. At the core of statistics is sampling, and sampling requires randomly picking a few examples from the pool of the entire population. The key element of the recipe is that every element from the population has exactly the same probability of being part of the sample. If a population consists of a million people and your sample size is one, each person's probability of being part of the sample is one out of a million. In mathematical terms, if you represent the population using the variable N and the sample size is n, the probability of being part of a sample is n/N, as shown in Figure 12-2. The represented sample is a *simple random sample.* (Other sample types have greater complexity; this is the simplest sample type and all the others build upon it.)

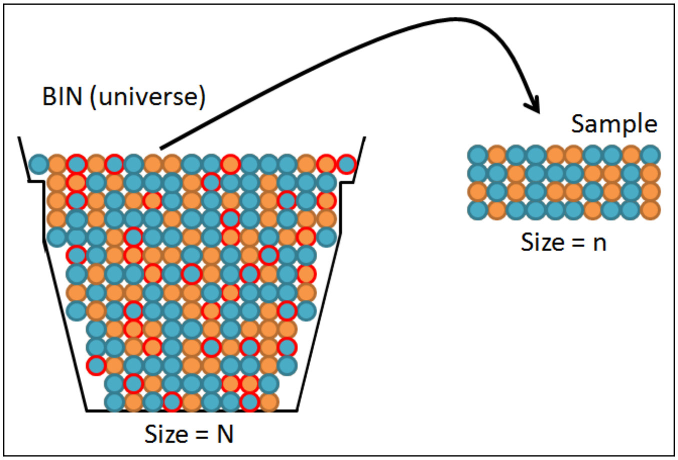


**FIGURE 12-2:** How sampling from an urn works.

Using a simple random sample is just like playing the lottery, but you need to have all the numbers inside an urn in order to extract a few to represent the whole. You can't easily put data streams into a repository from which you can extract a sample; instead, you have to extract your sample on the fly. What you really need is another sample type named *reservoir sampling.* Just as a reservoir retains water for later use, yet its water isn't still because some enters and some leaves, so this algorithm works by randomly choosing elements to keep as samples until other elements arrive to replace them.

The reservoir sampling algorithm is more sophisticated than another algorithmic strategy, called *windowing,* in which you create a queue and let new elements enter the queue (see Figure 12-3). Older elements leave the queue based on a trigger. This method applies when you want reports from the stream at exact time intervals. For instance, you may want to know how many pages users request from an Internet server each minute. Using windowing, you start queuing page requests a minute of time, count the elements in the queue, report the number, discard the content of the queue, and start queuing again.

Another motivation for using windowing is to have a fixed amount of the most recent data. In that case, every time you insert an element into the queue, the oldest element leaves. A queue is a First In, First Out (FIFO) structure discussed in Chapter 6.

Windowing looks at samples using a sliding window — it shows the elements under the window, which represent a certain time slice or a certain segment of the stream. Reservoir sampling represents the entire stream scope instead by offering a manageable amount of data, which is a statistical sample of the stream.

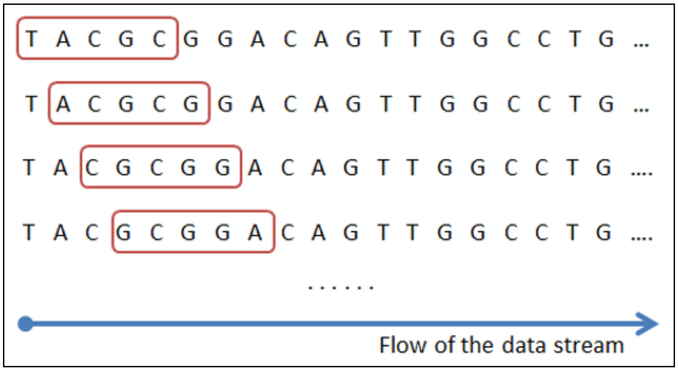

T A C G C G G A C A G T T G G C C T G ...
T A C G C G G A C A G T T G G C C T G ...
T A C G C G G A C A G T T G G C C T G ....
T A C G C G G A C A G T T G G C C T G ....
......
Flow of the data stream

**FIGURE 12-3:** An example of windowing a stream of DNA data.

Here is how reservoir sample works: Given a stream of data, containing many elements, you initialize the reservoir sample with elements taken from the stream until the sample is complete. For instance, if the sample contains 1,000 elements, a number that usually fits in the computer's internal memory, you start by picking the first 1,000 stream elements. The number of elements you want in the sample is k, and k implies a sample that fits into the computer's memory. At the point when you reserve the first k stream elements, the algorithm starts making its selections:

1. From the beginning of the stream, the algorithm counts every new element that arrives. It tracks the counting using the variable named as n. When the algorithm gets into action, the value of n is equivalent to k.
2. Now, new elements arrive, and they increment the value of n. A new element arriving from the stream has a probability of being inserted into the reservoir sample of k/n and probability of not being inserted equal to (1 – k/n).
3. The probability is verified for each new element arriving. It's like a lottery: If the probability is verified, the new element is inserted. On the other hand, if it isn't inserted, the new element is discarded. If it's inserted, the algorithm discards an old element in the sample according to some rule (the easiest being to pick an old element at random) and replaces it with the new element.

The following code shows a simple example in Python so that you can see this algorithm in action. The example relies on a sequence of alphabet letters (pretend that they are a data stream) and creates a sample of five elements. (You can find this code in the `A4D; 12; Managing Big Data.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
import string
datastream = list(string.ascii_uppercase) + list(
    string.ascii_lowercase)
print(datastream)
```

```
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
 'w', 'x', 'y', 'z']
```

Apart from strings, the example uses functions from the random package to create a seed (for stable and replicable solutions) and, drawing a random integer number, it checks whether it needs to change an element in the reservoir. Apart from the seed value, you can experiment with modifying the sample size or even feeding the algorithm a different stream (it should be in a Python list for the example to work correctly).

```
from random import seed, randint
seed(9) # change this value for different results
sample_size = 5
sample = []

for index, element in enumerate(datastream):
        # Until the reservoir is filled, we add elements
        if index < sample_size:
                sample.append(element)
        else:
                # Having filled the reservoir, we test a
                # random replacement based on the elements
                # seen in the data stream
                drawn = randint(0, index)
                # If the drawn number is less or equal the
                # sample size, we replace a previous
                # element with the one arriving from the
                # stream
                if drawn < sample_size:
                        sample[drawn] = element

print (sample)

['y', 'e', 'v', 'F', 'i']
```

This procedure assures you that, at any time, your reservoir sample is a good sample representing the overall data stream. In this implementation, the variable `index` plays the role of n and the variable `sample_size` acts as k. Note two particular aspects of this algorithm:

- As the variable `index` grows, because the stream floods with data, the probability of being part of the sample decreases. Consequently, at the beginning of the stream, many elements enter and leave the sample, but the rate of change decreases as the stream continues to flow.
- If you check the probability at which each element present in the sample enters, and you average them all, the average will approximate the probability of an element of a population's being picked into a sample, which is k/n.

# Sketching an Answer from Stream Data

Sampling is an excellent strategy for dealing with streams but it doesn't answer all the questions you may have from your data stream. For instance, a sample can't tell you when you've already seen a stream element because the sample doesn't contain all the stream information. The same holds true for problems such as counting the distinct number of elements in a stream or computing element frequency.

To achieve such results, you need hash functions (as seen in Chapter 7) and sketches, which are simple and approximate data summaries. The following sections start with hashes, and you discover how to be correct in finding when an arriving stream element has appeared before, even if your stream is infinite and you cannot keep exact memory of everything that flowed before.

## Filtering stream elements by heart

At the heart of many streaming algorithms are Bloom filters. Created almost 50 years ago by Burton H. Bloom, at a time when computer science was still quite young, the original intent of this algorithm's creator was to trade space (memory) and/or time (complexity) against what he called *allowable errors.* His original paper is entitled *Space/Time Trade-offs in Hash Coding with Allowable Errors* (see: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.2080&rank=2` for details).

You may wonder about the space and time that Bloom considers motivators for his algorithm. Imagine that you need to determine whether an element has already appeared in a stream using some previously discussed data structure. Finding something in a stream implies recording and searching are fast, thus a hash table seems an ideal choice. *Hash tables,* as discussed in Chapter 7, simply require adding the elements that you want to record and storing them. Recovering an element from a hash table is fast because the hash table uses easily manipulated values to represent the element, rather than the element itself (which could be quite complex). Yet, storing both elements and an index to those elements has limitations. If a hash table faces more elements than it can handle, such as the elements in a continuous and potentially infinite stream, you'll end up incurring memory problems at some point.

REMEMBER

An essential consideration for Bloom filters is that false positives can occur, but false negatives can't. For example, a data stream might contain real-time monitoring data for a power plant. When using a Bloom filter, the analysis of the data stream would show that expected readings are probably part of the set of allowed readings, with some errors allowed. However, when an error occurs in the system, the same analysis shows that the readings aren't part of the set of allowed readings. The false positives are unlikely to cause problems, but the absence of false negatives means that everyone remains safe. Because of the potential for false positives, filters such as the Bloom filter are probabilistic data structures — they don't provide a certain answer but a probable one.

REMEMBER

*Hashes*, the individual entries in a hash table, are fast because they act like the index of a book. You use a *hash function* to produce the hash; the input is an element containing complex data, and the output is a simple number that acts as an index to that element. A hash function is deterministic because it produces the same number every time you feed it a specific data input. You use the hash to locate the complex information you need. Bloom filters are helpful because they are a frugal way to record traces of many elements without having to store them away as a hash table does. They work in a simple way and use the following as main ingredients:

- **A bit vector:** A list of bit elements, where each bit in the element can have a value of 0 or 1. The list is a long number of bits called m. The greater m is, the better, though there are ways of optimally defining its size.
- **A series of hash functions:** Each hash function represents a different value. The hash functions can quickly crunch data and produce uniformly distributed results, which are results equally ranging from the minimum to the maximum output values of the hash.

## Adding elements to Bloom filters

Generally, you create Bloom filters of a fixed size (recently developed versions allow you to resize the filter). You operate them by adding new elements to the filter and looking them up when already present. It's not possible to remove an element from the filter after adding it (the filter has an indelible memory). When adding an element to a bit vector, the bit vector has some bits set to 1, as shown in Figure 12-4. In this case, the Bloom filter adds X to the bit vector.

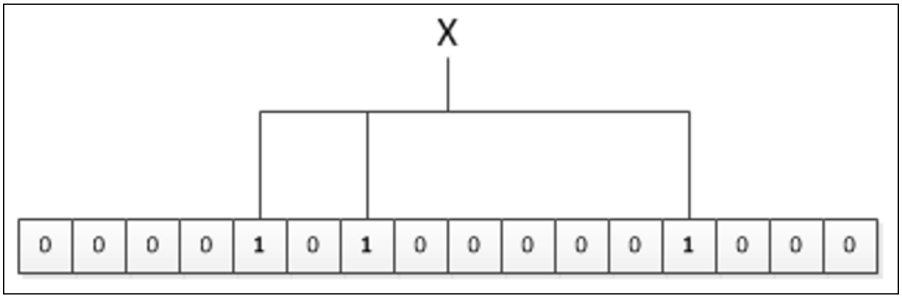


**FIGURE 12-4:** Adding a single element to a bit vector.

You can add as many elements as is necessary to the bit vector. For example, Figure 12-5 shows what happens when adding another element, Y, to the bit vector. Note that bit 7 is the same for both X and Y. Consequently, bit 7 represents a collision between X and Y. These collisions are the source of the potential false positives; because of them, the algorithm could say that an element is already added to the bit vector when it isn't. Using a larger bit vector makes collisions less likely and improves the performance of the Bloom filter, but does so at the cost of both space and time.

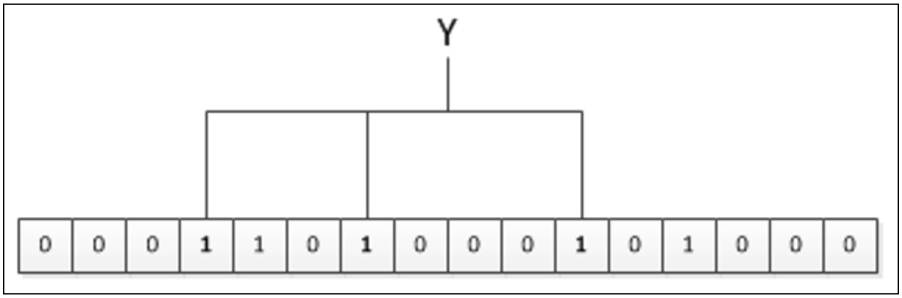


**FIGURE 12-5:** Adding a second element can cause collisions.

## Searching a Bloom filter for an element

Searching a Bloom filter lets you determine whether a particular element appears in the bit vector. During the search process, the algorithm looks for the presence of a 0 in the bit vector. For example, the previous section added elements X and Y to the bit vector. In searching for element Z, the algorithm finds a 0 in the second bit, as shown in Figure 12-6. The presence of a 0 means that Z isn't part of the bit vector.

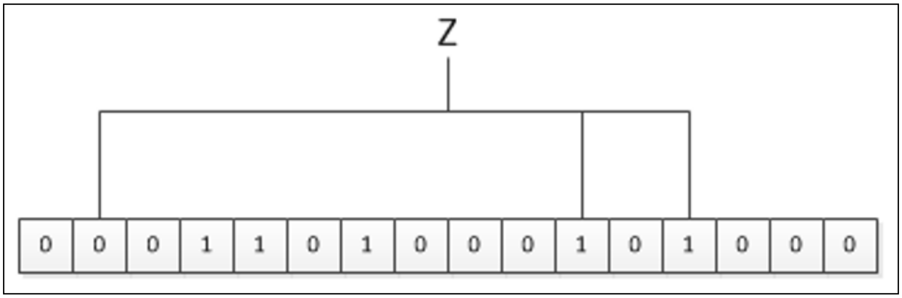


**FIGURE 12-6:** Locating an element and determining that it exists means searching for 0s in the bit vector.

## Demonstrating the Bloom filter

This example uses Python to demonstrate a Bloom filter and shows the result with a graphical visualization. Say that you're using a crawler, which is specialized software that journeys the web to check whether something has changed in the monitored websites (which may imply copying part of the website's data, an activity known as *scraping*). The example uses a short bit vector and three hash functions, which isn't the best setting for handling a large number of elements (the bit vector will get filled quickly), but enough for a working example.

```
hash_functions = 3
bit_vector_length = 10
bit_vector = [0] * bit_vector_length

from hashlib import md5, sha1

def hash_f(element, i, length):
    """ This is a magic function """
    h1 = int(md5(element.encode('ascii')).hexdigest(),16)
    h2 = int(sha1(element.encode('ascii')).hexdigest(),16)
    return (h1 + i*h2) % length

def insert_filter(website):
    result = list()
    for hash_number in range(hash_functions):
        position = hash_f(website, hash_number,
                          bit_vector_length)
        result.append(position)
        bit_vector[position] = 1
    print ('Inserted in positions: %s' % result)

def check_filter(website):
    result = list()
    for hash_number in range(hash_functions):
        position = hash_f(website, hash_number,
                          bit_vector_length)
        result.append((position,bit_vector[position]))
    print ('Bytes in positions: %s' % result)
```

The code begins by creating a bit vector and some functions that can do the following:

- Generate multiple hash functions (using the double hash trick mentioned in Chapter 7) based on the md5 and sha1 hashing algorithms
- Insert an object into the bit vector
- Check whether the bytes relative to an object in the bit vector are turned on

All these elements together constitute a Bloom filter (though the bit vector is the key part of it). This example has the crawler first visiting the website `wikipedia.org` to take some information from a few pages:

```
insert_filter('wikipedia.org')
print (bit_vector)

Inserted in positions: [0, 8, 6]
[1, 0, 0, 1, 0, 0, 1, 1, 1, 0]
```

That activity turns on the bits in positions 0, 6, and 8 of the bit vector. The example now crawls the `youtube.com` website (which has some new videos of kittens) and so it inserts the information of the visit into the Bloom filter:

```
insert_filter('youtube.com')
print (bit_vector)

Inserted in positions: [3, 0, 7]
[1, 0, 0, 1, 0, 0, 1, 1, 1, 0]
```

Here the Bloom filter is activated on positions 0, 3, and 7. Given the short length of the bit vector, there is already a collision on position 0, but positions 3 and 7 are completely new. At this point, because the algorithm can't remember what it visited before (but visited sites can be verified using the Bloom filter), the example verifies that it hasn't visited `yahoo.com` in order to avoid redoing things, as shown in Figure 12-7:

```
check_filter('yahoo.com')

Bytes in positions: [(7, 1), (5, 0), (3, 1)]
```

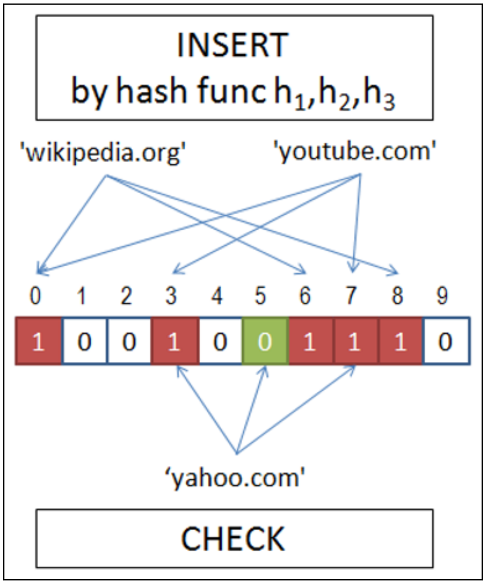


**FIGURE 12-7:** Testing membership of a website using a Bloom filter.

As graphically represented, in this case you can be sure that the example never visited `yahoo.com` because the Bloom filter reports at least one position, position 5, whose bit was never set on.

TIP

A crawler is often concerned with getting new content from websites and not copying data that it has already recorded and transmitted. Instead of hashing the domain or the address of a single page, you can directly populate a Bloom filter using part of the website content, and you can use it to check the same website for changes later.

There is a simple and straightforward way of decreasing the probability of having a false positive. You just increase the size of the bit vector that is the core of a Bloom filter. More addresses equate fewer chances for a collision by the hash functions' results. Ideally, the size m of the bit vector can be calculated estimating n, the number of distinct objects that you expect to add by keeping m much larger than n. The ideal number k of hash functions to use to minimize collisions can then be estimated using the following formula (ln is the natural logarithm):

k = (m/n)*ln(2)

After you have m, n, and k defined, this second formula helps you estimate the probability of a collision (a false positive rate) using a Bloom filter:

false positive rate = (1-exp(-kn/m))^k

If you can't determine n due to the variety of the data in the stream, you have to change m, the bit vector size (which equates to memory space), or k, the number of hash functions (which equates to time), to adjust the false positive rate. The trade-off reflects the relationships that Bloom considers in the original paper between space, time, and probability of an error.

## Finding the number of distinct elements

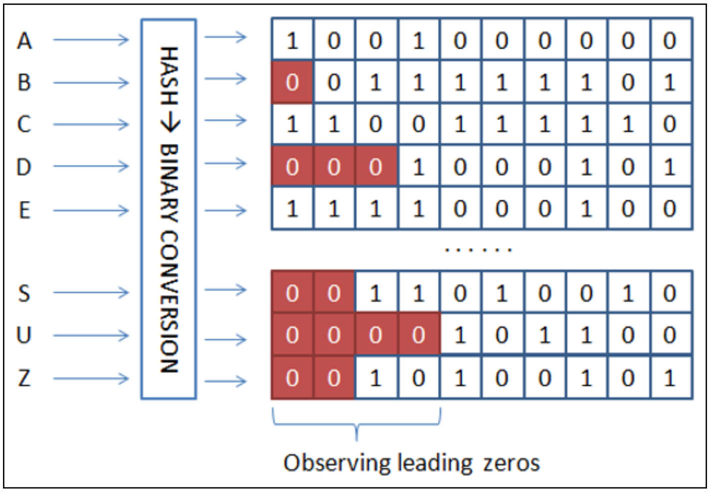
Even though a Bloom filter can track objects arriving from a stream, it can't tell how many objects are there. A bit vector filled by ones can (depending on the number of hashes and the probability of collision) hide the true number of objects being hashed at the same address.

Knowing the distinct number of objects is useful in various situations, such as when you want to know how many distinct users have seen a certain website page or the number of distinct search engine queries. Storing all the elements and finding the duplicates among them can't work with millions of elements, especially coming from a stream. When you want to know the number of distinct objects in a stream, you still have to rely on a hash function, but the approach involves taking a numeric sketch.

*Sketching* means taking an approximation, that is an inexact yet not completely wrong value as an answer. Approximation is acceptable because the real value is not too far from it. In this smart algorithm, *HyperLogLog*, which is based on probability and approximation, you observe the characteristics of numbers generated from the stream. HyperLogLog derives from the studies of computer scientists Nigel Martin and Philippe Flajolet. Flajolet improved their initial algorithm, *Flajolet–Martin* (or the LogLog algorithm), into the more robust HyperLogLog version, which works like this:

1. A hash converts every element received from the stream into a number.
2. The algorithm converts the number into binary, the base 2 numeric standard that computers use.
3. The algorithm counts the number of initial zeros in the binary number and tracks of the maximum number it sees, which is n.
4. The algorithm estimates the number of distinct elements passed in the stream using n. The number of distinct elements is 2^n.

For instance, the first element in the string is the word *dog.* The algorithm hashes it into an integer value and converts it to binary, with a result of 01101010. Only one zero appears at the beginning of the number, so the algorithm records it as the maximum number of trailing zeros seen. The algorithm then sees the words *parrot* and *wolf,* whose binary equivalents are 11101011 and 01101110, leaving n unchanged. However, when the word *cat* passes, the output is 00101110, so n becomes 2. To estimate the number of distinct elements, the algorithm computes 2^n, that is, 2^2=4. Figure 12-8 shows this process.



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| B | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| C | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| D | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| E | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| S | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| U | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| Z | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |

**FIGURE 12-8:** Counting only leading zeros.

The trick of the algorithm is that if your hash is producing random results, equally distributed (as in a Bloom filter), by looking at the binary representation, you can calculate the probability that a sequence of zeros appeared. Because the probability of a single binary number to be 0 is one in two, for calculating the probability of sequences of zeros, you just multiply that 1/2 probability as many times as the length of the sequence of zeros:

- 50 percent (1/2) probability for numbers starting with 0
- 25 percent (1/2 * 1/2 ) probability for numbers starting with 00
- 12.5 percent (1/2 * 1/2 * 1/2) probability for numbers starting with 000
- (1/2)^k probability for numbers starting with k zeros (you use powers for faster calculations of many multiplications of the same number)

TIP

The fewer the numbers that HyperLogLog sees, the greater the imprecision. Accuracy increases when you use the HyperLogLog calculation many times using different hash functions and average together the answers from each calculation, but hashing many times takes time, and streams are fast. As an alternative, you can use the same hash but divide the stream into groups (such as by separating the elements into groups as they arrive based on their arrival order) and for each group, you keep track of the maximum number of trailing zeros. In the end, you compute the distinct element estimate for each group and compute the arithmetic average of all the estimates. This approach is *stochastic averaging* and provides more precise estimates than applying the algorithm to the entire stream.

## Learning to count objects in a stream

This last algorithm in the chapter also leverages hash functions and approximate sketches. It does so after filtering duplicated objects and counting distinct elements that have appeared in the data stream. Learning to count objects in a stream can help you find the most frequent items or rank usual and unusual events. You use this technique to solve problems like finding the most frequent queries in a search engine, the bestselling items from an online retailer, the highly popular pages in a website, or the most volatile stocks (by counting the times a stock is sold and bought).

You apply the solution to this problem, *Count-Min Sketch*, to a data stream. It requires just one data pass and stores as little information as possible. This algorithm is applied in many real-world situations (such as analyzing network traffic or managing distributed data flows). The recipe requires using a bunch of hash functions, each one associated with a bit vector, in a way that resembles a Bloom filter, as shown in Figure 12-9:

1. Initialize all the bit vectors to zeroes in all positions.
2. Apply the hash function for each bit vector when receiving an object from a stream. Use the resulting numeric address to increment the value at that position.
3. Apply the hash function to an object and retrieve the value at the associated position when asked to estimate the frequency of an object. Of all the values received from the bit vectors, you take the smallest as the frequency of the stream.

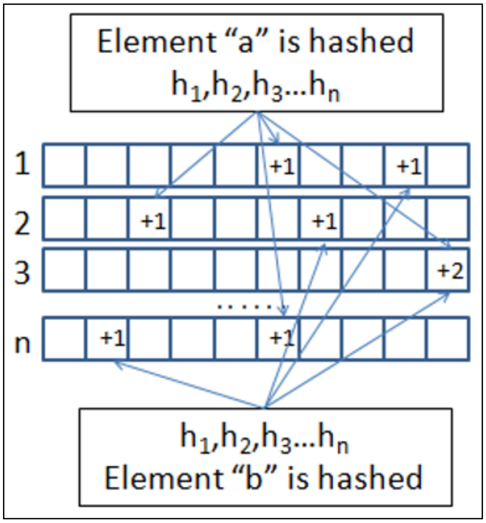


**FIGURE 12-9:** How values are updated in a Count-Min Sketch.

REMEMBER

Because collisions are always possible when using a hash function, especially if the associated bit vector has few slots, having multiple bit vectors at hand assures you that at least one of them keeps the correct value. The value of choice should be the smallest because it isn't mixed with false positive counts due to collisions.

IN THIS CHAPTER

- Understanding why simply bigger, larger, and faster isn't always the right solution
- Looking inside the storage and computational approaches of Internet companies
- Figuring out how using clusters of commodity hardware reduces costs
- Reducing complex algorithms into separable parallel operations by MapReduce

# Chapter 13 Parallelizing Operations

Managing immense amounts of data using streaming or sampling strategies has clear advantages (as discussed in Chapter 12) when you have to deal with massive data processing. Using streaming and sampling algorithms helps you obtain a result even when your computational power is limited (for instance, when using your own computer). However, some costs are associated with these approaches:

- **Streaming:** Handles infinite amounts of data. Yet your algorithms perform at low speed because they process individual pieces of data and the stream speed rules the pace.
- **Sampling:** Applies any algorithms on any machine. Yet the obtained result is imprecise because you have only a probability, not a certainty, of getting the right answer. Most often, you just get something plausible.

Some problems require that you handle great amounts of data in a both precise and timely fashion. Examples abound in the digital world, such as making a keyword query among billions of websites or processing multiple pieces of information (searching for an image in a video repository or a match in multiple DNA sequences). Doing such calculations sequentially would take a lifetime.

The solution is using *distributed computing,* which means interconnecting many computers in a network and using their computational capabilities together, combined with algorithms running on them in an independent, parallel manner.

# Managing Immense Amounts of Data

The use of the Internet to perform a wide range of tasks and the increase in popularity of its most successful applications, such as search engines or social networking, has required professionals in many fields to rethink how to apply algorithms and software solutions to cope with a deluge of data. Searching for topics and connecting people drive this revolution.

Just imagine the progression, in terms of available websites and pages, that has occurred in the last 15 years. Even if you use a smart algorithm, such as PageRank (discussed and explored in Chapter 11), coping with ever larger and changeable data is still hard. The same goes for social networking services offered by companies such as Facebook, Twitter, Pinterest, LinkedIn, and so on. As the number of users increases and their reciprocal relationships unfold, the underlying graph connecting them turns massive in scale. With large scale, handling nodes and links to find groups and connections becomes incredibly difficult. (Part 3 of the book discusses graphs in detail.)

In addition to communication-based data, consider online retailers that provide virtual warehouses of thousands and thousands of products and services (books, films, games, and so on). Even though you understand why you bought something, the retailer sees the items in your basket as small pieces of a purchase decision-making puzzle to solve to understand buying preferences. Solving the puzzle enables a retailer to suggest and sell alternative or supplementary products.

## Understanding the parallel paradigm

CPU makers found a simple solution when challenged to stuff more computing power into microprocessors (as forecasted and partially prescribed by Moore's law, discussed in Chapter 12). Yet, bigger, larger, and faster isn't always the right solution. When they found that power absorption and heat generation limited the addition of more CPUs to a single chip, engineers compromised by creating *multicore processing units,* which are CPUs made by stacking two or more CPUs together. The use of multicore technology gave raise to parallel computing to a larger audience.

Parallel computing has existed for a long time, but it mainly appeared in high-performance computers, such as the Cray super-computers created by Seymour Cray at Control Data Corporation (CDC) starting in the 1960s. Simply stated, the associative and commutative properties in math express the core idea of parallelism. In a math addition, for instance, you can group part of the sums together or you can add the parts in a different order than the one shown by the formulas:

```
Associative property
2 + (3 + 4) = (2 + 3) + 4

Commutative property
2 + 3 + 4 = 4 + 3 + 2
```

The same concepts apply to computing algorithms, regardless of whether you have a series of operations or a mathematical function. Most often, you can reduce the algorithm to a simpler form when you apply associative and commutative properties, as shown in Figure 13-1. Then you can split the parts and have different units perform atomic operations separately, summing the result at the end.

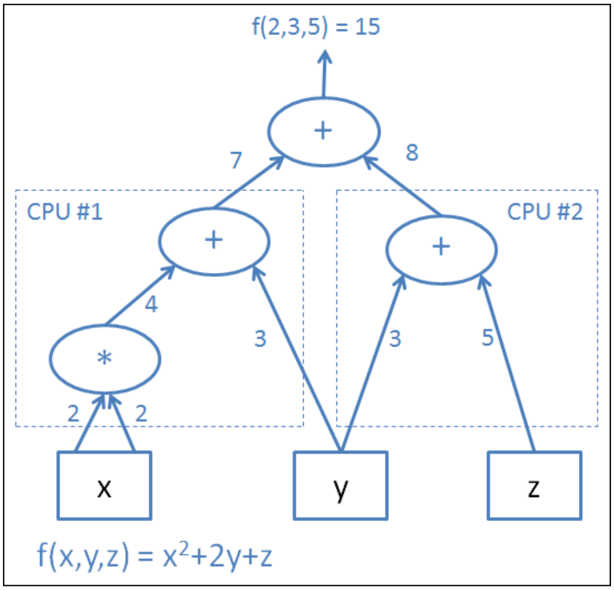


**FIGURE 13-1:** Associative and commutative properties allow parallelism.

In this example, two CPUs split a simple function with three inputs (x, y, and z) by leveraging both associative and commutative properties. The equation solution requires sharing common data (CPU1 needs x and y values; CPU2 needs y and z values instead), but the processing proceeds in parallel until the two CPUs emit their results, which are summed to obtain the answer.

Parallelism allows processing of large numbers of calculations simultaneously. The more processes, the higher the speed of computation execution, although the time spent is not linearly proportional to the number of parallel executors. (It is not completely true that two CPUs imply double speed, three CPUs imply three times the speed, and so on.) In fact, you can't expect the associative or commutative properties to work on every part of your algorithm or computer instructions. The algorithm simply can't make some parts parallel as stated by Amdahl's law, which helps determine the parallelism speed advantage of your computation (for details, see `http://home.wlu.edu/~whaleyt/classes/parallel/topics/amdahl.html`). In addition, other aspects can dampen the positive effect of parallelism:

- **Overhead:** You can't sum the results in parallel.
- **Housekeeping:** The underlying conversion from a human-readable language to machine language requires time. Keeping the processors working together increases the conversion costs, making it impossible to see a doubling effect from two processors even if you can perform every part of the task in parallel.
- **Asynchronous Outputs:** Because parallel executors don't perform tasks at the same exact speed, the overall speed is bound to the slowest one. (As with a fleet, the speed of the fleet is determined by the slowest boat.)

Even if not always as beneficial as expected, parallelism can potentially address the problem of handling a massive number of operations faster than using a single processor (if a massive number of executors can process them in parallel). Yet, parallelism can't address the massive amounts of data behind the computations without another solution: distributed computing on distributed systems.

REMEMBER

When you buy a new computer, the seller likely tells you about cores and threads. *Cores* are the CPUs that are stacked inside the single CPU chip and that work in a parallel fashion using multiprocessing. Because each core is independent, the tasks occur simultaneously. *Threads* refer instead to the capability of a single core to split its activity between multiple processes, in an almost parallel way. However, in this case, each thread takes its turn with the processor, so the tasks don't occur simultaneously. This is called *multithreading.*

## Distributing files and operations

Large graphs, huge amounts of text files, images, and videos, and immense adjacency relation matrices call forth the parallel approach. Fortunately, you no longer need a supercomputer to deal with them, but can instead rely on parallelism by a bunch of much less powerful computers. The fact that these large data sources keep growing means that you need a different approach than using a single computer specially designed to handle them. The data grows so fast that when you finish designing and producing a supercomputer for crunching the data, it may well no longer be suitable because the data has already grown too large.

The solution starts with an online service such as Google, Microsoft Azure, or Amazon Web Services (AWS). The first step to solve the problem is deciding where to put the data. The second step is to decide how to compute efficiently without moving the data around too much (because large data transfers take a lot of time to transit from one computer to another over a network or the Internet).

Most of these services work in a similar fashion. Engineers put many existing technological ideas together and create a Distributed File System (DFS) of some sort. When using a DFS, data isn't stored in a single powerful computer with a giant hard disk; instead, the DFS spreads it among multiple smaller computers, similar to a personal computer. The engineers arrange the computers into a *cluster,* a physical system of racks and cable connections. Racks are the true backbone of the network, in which multiple computers are stored next to each other. In a single rack of the network, you may find a variable number of computers, from eight to 64, each one connected to the other. Each rack connects to other racks by means of a cable network, created by interconnecting the racks not directly between themselves, but to various layers of *switches,* which are computer-networking devices able to efficiently handle and manage the data exchange between racks, as shown in Figure 13-2.

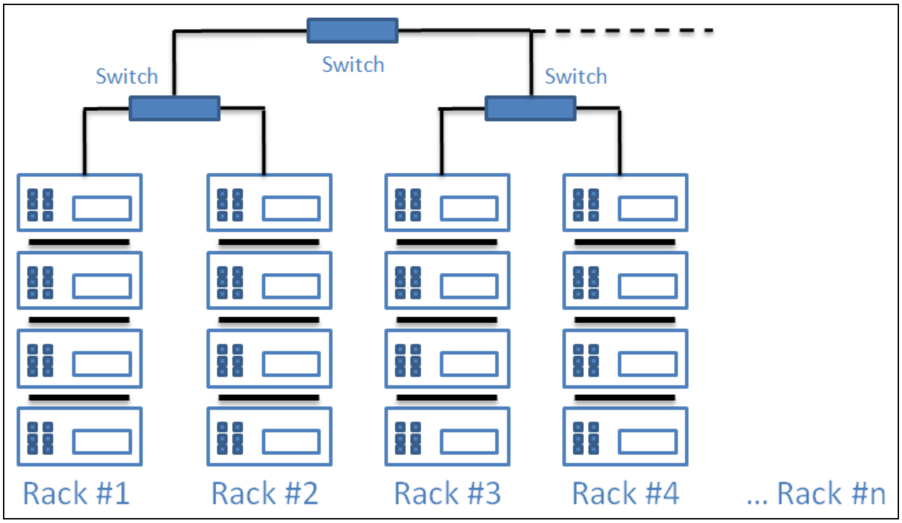


**FIGURE 13-2:** A schema representing a computing cluster.

You can find all this hardware at any computer store, yet it's exactly what makes the DFS infrastructure viable. Theoretically, you could find a million or more computers interconnected in a network. (You can read about the Google version of this setup at `http://www.datacenterknowledge.com/archives/2011/08/01/report-google-uses-about-900000-servers/`.) The interesting point is that these services increase computational power when needed by adding more computers, not by creating new networks.

In this system, as data arrives, the DFS splits it into chunks (each one up to 64MB in size). The DFS copies the chunks into multiple duplicates and then distributes each copy to a computer part on the network. The action of splitting data into chunks, duplicating it, and distributing it is quite fast, no matter how the data is structured (tidy and ordered information or messy ensemble). The only requirement pertains to the recording of the chunks' address in the DFS, which is achieved by an index for each file (itself replicated and distributed), called the master node. The DFS execution speed is due to how the DFS handles the data. Contrary to previous storage techniques (such as data warehouses), a DFS doesn't require any particular sorting, ordering, or cleaning operation on the data itself; on the contrary, it does the following:

- Handles data of any size because the data is split into manageable chunks
- Stores new data by piling it next to the old data; a DFS never updates any previously received data
- Replicates data redundantly so that you don't need to back it up; duplication is in itself a backup

REMEMBER

Computers fail in a number of ways: hard disk, CPU, power system, or some other component. Statistically, you can expect a computer serving in a network to work for about 1,000 days (about three years). Consequently, a service with a million computers, can expect 1,000 of its computers to fail every day. That is why the DFS spreads three or more copies of your data inside multiple computers in the network. Replication reduces the likelihood of losing data because of a failure. The probability of having a failure that involves only computers where the same chunk of data is stored is about one out of a billion (assuming that the DFS replicates the data three times), making this a tiny, acceptable risk.

## Employing the MapReduce solution

Even though distributed systems store data quickly, retrieving data is much slower, especially when performing analysis and applying algorithms. The same sort of problem occurs when you break a jigsaw puzzle into pieces and scatter the pieces around (easy). You must then pick the pieces up and recreate the original image (hard and time consuming). When working with data in a DFS:

## USING A PACKAGE SOLUTION FOR MAPREDUCE

Even though the book demonstrates how to create a MapReduce solution from scratch, you don't need to reinvent the wheel every time you want to perform this task. Packages such as MrJob (`https://pythonhosted.org/mrjob/`) enable you to perform MapReduce tasks quickly and easily. In addition, by using a package, you can make it easy to execute the task using cloud-based resources, such as Amazon Web Services using Elastic MapReduce (EMR) (`https://aws.amazon.com/emr/`) or with Hadoop (`http://hadoop.apache.org/`). The point is that you do need to know how the algorithm works, which is the point of this book, but having to write all the required code may be unnecessary in the right situation.

1. Get the master node and read it to determine the location of the file parts.
2. Dispatch a fetch order to the computers in the network to obtain the previously stored data chunks.
3. Gather the data chunks stored on multiple computers onto a single computer (if doing so is possible; some files may be too large to store on a single machine).

Obviously, this process can become complex, so web service engineers have decided that it's better not to recompose files before processing them. A smarter solution is to leave them in chunks on the source computer and let the hosting computer process them. Only a `reduce` version, which is already almost completely processed, would have to move across the network, limiting the data transmission. MapReduce is the solution that provides the means to process algorithms in parallel in a data-distributed system. As an algorithm itself, MapReduce consists of just two parts, `map` and `reduce`.

## Explaining map

The first phase of the MapReduce algorithm is the `map` part, a function found in many *functional programming languages* (a style of programming that treats computing as a mathematical function). `map` is straightforward: You begin with a one-dimensional array (which, in Python, could be a list) and a function. By applying the function on each array element, you obtain an identically shaped array whose values are transformed. The following example contains a list of ten numbers that the function transforms into their power equivalent:

```
L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
m = list(map(lambda x: x**2, L))
```

```
print(m)

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The `map` function applies the Python `lambda` function (a *lambda* function is a function defined on the fly) to transform each element in the initial list into a resulting element. Figure 13-3 shows the result of this mapping process.

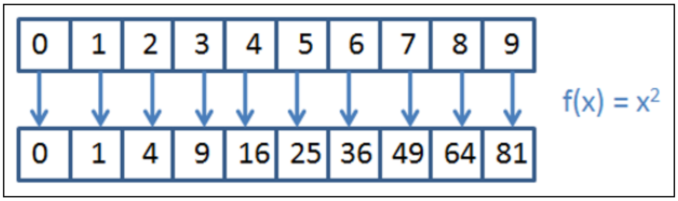


**FIGURE 13-3:** Mapping a list of numbers by a square function.

Note that each list element transformation is independent of the others. You can apply the function to the list elements in any order. (However, you must store the result in the right position in the final array.) The capability to process the list elements in any order creates a scenario that is naturally parallelized without any particular effort.

REMEMBER

Not all problems are naturally parallel, and some will never be. However, sometimes you can rethink or redefine your problem in order to achieve a set of computations that the computer can deal with in a parallel way.

## Explaining reduce

The second phase of the MapReduce algorithm is the `reduce` part (there is also an intermediate step, Shuffle and Sort, explained later but not important for now). When given a list, `reduce` applies a function in a sequence that cumulates the results. Thus, when using a summation function, `reduce` applies the summation to all the input list elements. `reduce` takes the first two array elements and combines them. Then it combines this partial result with the next array element and so on until it completes the array.

TIP

You can also supply a starting number. When you supply a starting number, `reduce` starts by combining the starting number with the first list element to obtain the first partial result. The following example uses the result from the mapping phase and reduces it using a summation function (as displayed in Figure 13-4):

```
from functools import reduce
L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
m = list(map(lambda x: x**2, L))
r = reduce(lambda x, y: x+y, m)
print(r)

285
```

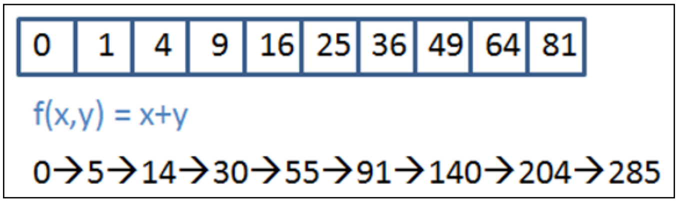


**FIGURE 13-4:** Reducing a list of numbers to its sum.

REMEMBER

The reduce function operates on the input array as if it were a data stream (as discussed in Chapter 12). It usually works with one element at a time and tracks the intermediate results.

## Distributing operations

Between the map and reduce phases is an intermediate phase, shuffling and sorting. As soon as a map task completes, the host redirects the resulting tuples of key and value pairs to the right computer on the network to apply the reduce phase. This is typically done by grouping matching key pairs into a single list and using a hash function on the key in a manner similar to Bloom filters (see Chapter 12). The output is an address in the computing cluster for transferring the lists.

On the other end of the transmission, the computer performing the reduce phase starts receiving lists of tuples of one or multiple keys. Multiple keys occur when a hash collision occurs, which happens when different keys result in the same hashed value, so they end up to the same computer. The computer performing the reduce phase sorts them into lists containing the same key before feeding each list into the reduce phase, as shown in Figure 13-5.

As shown in the figure, MapReduce takes multiple inputs at each computer in the computing cluster where they are stored, maps the data, and transforms it into tuples of key and value pairs. Arranged into lists, the host transmits these tuples to other computers over the network, where the receiving computers operate sort and reduce operations that lead to a result.

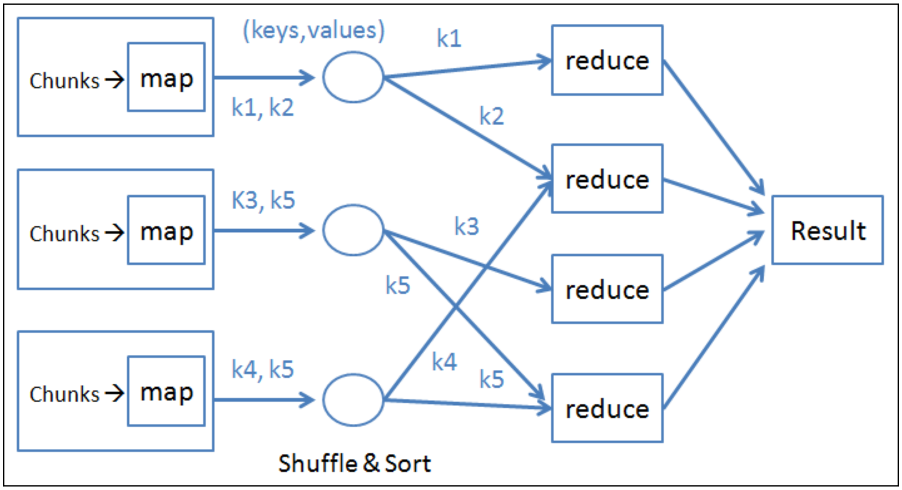


**FIGURE 13-5:** An overview of the complete MapReduce computation.

# Working Out Algorithms for MapReduce

Contrary to other examples in this book, you can think of MapReduce as more of a style of computing or a Big Data framework than an algorithm. As a framework, it enables you to combine different distributed algorithms (parallel algorithms that spread computations across different computers) and allow them to work efficiently and successfully with large amounts of data. You can find MapReduce algorithms in many applications, and you can read about them at the Apache wiki regarding Hadoop, detailing the company that uses it, how it is used, and on what kind of computing cluster: `http://wiki.apache.org/hadoop/PoweredBy`. Even though possibilities are many, you most often find MapReduce used to perform these tasks:

- Text algorithms for splitting the text into elements (tokens), creating indexes, and searching for relevant words and phrases
- Graph creation and graph algorithms
- Data mining and learning new algorithms from data (machine learning)

One of the most common MapReduce algorithm uses is to process text. The example in this section demonstrates how to solve a simple task, counting certain words in a text passage using a map and reduce approach, and leveraging multithreading or multiprocessing (depending on the operating system installed on your computer).

TECHNICAL STUFF

The Python programming language is not the ideal computer language for parallel operations. Technically, because of synchronization and shared memory access problems, the Python interpreter isn't thread safe, which means that it can experience errors when executing applications using multiple processes or threads on multiple cores. Consequently, Python limits multithreading to a single thread (code is distributed but no performance increase occurs) and multicore parallelism by multiple processes is indeed tricky to achieve, especially on computers running on Windows. You can learn more about the difference in threads and processes by reading the Microsoft article at https://msdn.microsoft.com/en-us/library/windows/desktop/ms681917(v=vs.85).aspx.

## Setting up a MapReduce simulation

This example processes text that is in the public domain, obtained from the non-profit Project Gutenberg organization (https://www.gutenberg.org/) site. The first text processed is the novel *War and Peace*, by Leo Tolstoy (who is also known by other names in other languages, such as Lev Tolstoj). The following code loads the data into memory:

```
import urllib.request
url = 'http://gutenberg.readingroo.ms/2/6/0/2600/2600.txt'
response = urllib.request.urlopen(url)
data = response.read()
text = data.decode('utf-8')[627:]

print (text[:37])

WAR AND PEACE

By Leo Tolstoy/Tolstoi
```

Be patient! Loading the book takes time (just try reading it in as short a time as the computer does). When complete, the code displays the first few lines with the title. The code stores the data in the text variable. Part of the process splits the text into words and stores them in a list, as shown in the following code:

```
words = text.split()
print ('Number of words: %i' % len(words))

Number of words: 566218
```

The `words` variable now contains individual words from the book. It's time to import the necessary Python packages and the functions for the example using the following code:

```
import os
if os.name == "nt":
    #Safer multithreading on Windows
    from multiprocessing.dummy import Pool
else:
    #Multiprocessing on Linux,Mac
    from multiprocessing import Pool

from multiprocessing import cpu_count
from functools import partial
```

Depending on your operating system, the example relies on multiprocessing or multithreading. Windows uses multithreading, which splits the task into multiple threads processed at the same time by the same core. On Linux and Mac systems, the code executes in parallel instead, and each operation is taken care of by a different computer core.

The code that comes next counts words from a list that corresponds to a set of keywords. After removing any punctuation, the code compares words, and if it finds any match with a keyword, the function returns a tuple consisting of a key, the matched keyword, and a unit value, which is a count. This output represents the core of the MapReduce map:

```
def remove_punctuation(text):
    return ''.join([l for l in text if l not in ['.',
            ',', '!', '?', '"']])

def count_words(list_of_words, keywords):
    results = list()
    for word in list_of_words:
        for keyword in keywords:
            if keyword == remove_punctuation(
                            word.upper()):
                results.append((keyword,1))
    return results
```

The functions that follow partition the data. This approach is similar to the way in which a distributed system partitions the data. The code distributes the computation and gathers the results:

```
def Partition(data, size):
    return [data[x:x+size] for x in range(0, len(data),
                                          size)]

def Distribute(function, data, cores):
    pool = Pool(cores)
    results = pool.map(function, data)
    pool.close()
    return results
```

Finally, the following functions shuffle and order the data to reduce the results. This step represents the last two phases of a MapReduce job:

```
def Shuffle_Sort(L):
    # Shuffle
    Mapping = dict()
    for sublist in L:
        for key_pair in sublist:
            key, value = key_pair
            if key in Mapping:
                Mapping[key].append(key_pair)
            else:
                Mapping[key] = [key_pair]
    return [Mapping[key] for key in Mapping]

def Reduce(Mapping):
  return (Mapping[0][0], sum([value for (key, value
                                ) in Mapping]))
```

## Inquiring by mapping

The following code simulates a distributed environment using multiple processor cores. It begins by requesting the number of available cores from the operating system. The number of cores you see will vary by the number of cores available in your computer. Most modern computers provide four or eight cores.

```
n = cpu_count()
print ('You have %i cores available for MapReduce' % n)

You have 4 cores available for MapReduce
```

REMEMBER

If you’re running the code on Windows, for technical reasons you work with a single core, so you won’t take advantage of the total number of available cores. The simulation still appears to work, but you won’t see any increase in speed.

In order to start, the code first defines the map operation. It then distributes the map function to threads, each of which processes a partition of the initial data (the list containing the words from *War and Peace*). The map operation finds the words *peace, war* (is there more war or peace in *War and Peace*?), *Napoleon,* and *Russia:*

```
Map = partial(count_words,
              keywords=['WAR', 'PEACE', 'RUSSIA',
                        'NAPOLEON'])
map_result = Distribute(Map,
                        Partition(
        words,len(words)//n+1), n)
print ('map_result is a list made of %i elements' %
       len(map_result))
print ('Preview of one element: %s]'% map_result[0][:5])

Map is a list made of 4 elements
Preview of one element: [('WAR', 1), ('PEACE', 1), ('WAR', 1),
          ('WAR', 1), ('RUSSIA', 1)]]
```

After a while, the code prints the results. In this case, the resulting list contains four elements because the host system has four cores (you may see greater or fewer elements, depending on the number of cores on your machine). Each element in the list is another list containing the results of the mapping on that part of the word data. By previewing one of these lists, you can see that it’s a sequence of coupled keys (depending on the keyword found) and unit values. The keys aren’t in order; they appear in the order in which the code generated them. Consequently, before passing the lists to the `reduce` phase for summing the total results, the code arranges the keys in order and sends them to the appropriate core for reducing:

```
Shuffled = Shuffle_Sort(map_result)
print ('Shuffled is a list made of %i elements' %
       len(Shuffled))
print ('Preview of first element: %s]'% Shuffled[0][:5])
print ('Preview of second element: %s]'% Shuffled[1][:5])
```

```
Shuffled is a list made of 4 elements
Preview of first element: [('RUSSIA', 1), ('RUSSIA', 1),
 ('RUSSIA', 1), ('RUSSIA', 1), ('RUSSIA', 1)]]
Preview of second element: [('NAPOLEON', 1), ('NAPOLEON',
 1), ('NAPOLEON', 1), ('NAPOLEON', 1), ('NAPOLEON', 1)]]
```

As shown in the example, the `Shuffle_Sort` function creates a list of four lists, each one containing tuples featuring one of the four keywords. In a cluster setting, this processing equates to having each mapping node pass through the emitted results, and, by using some kind of addressing (for instance, using a hash function, as seen in the bit vector of a Bloom filter Chapter 12), they send (shuffle phase) the tuple data to the appropriate reducing node. The receiving node places each key in the appropriate list (order phase):

```
result = Distribute(Reduce, Shuffled, n)
print ('Emitted results are: %s' % result)

Emitted results are: [('RUSSIA', 156), ('NAPOLEON', 469),
  ('WAR', 288), ('PEACE', 111)]
```

The reduce phase sums the distributed and ordered tuples and reports the total summation for each key, as seen in the result printed by the code that replicates a MapReduce. Reading the results, you can see that Tolstoy mentions war more than peace in *War and Peace,* but he mentions Napoleon even more often.

You can easily repeat the experiment on other texts or, even hack the map function in order to apply a different function to the text. For instance, you could choose to analyze some of the most famous novels by Sir Arthur Conan Doyle and try to discover how many times Sherlock Holmes used the phrase “Elementary, Watson”:

```
import urllib.request
url = "http://gutenberg.pglaf.org/1/6/6/1661/1661.txt"
text = urllib.request.urlopen(url).read().decode(
                                    'utf-8')[723:]
words = text.split()

print (text[:65])
print ('\nTotal words are %i' % len(words))
```

```
Map = partial(count_words,
              keywords=['WATSON', 'ELEMENTARY'])
result = Distribute(Reduce,
                    Shuffle_Sort(Distribute(Map,
                    Partition(words,len(words)//n), n)),
                    1)
print ('Emitted results are: %s' % result)


THE ADVENTURES OF SHERLOCK HOLMES
by
SIR ARTHUR CONAN DOYLE

Total words are 107431
Emitted results are: [('WATSON', 81), ('ELEMENTARY', 1)]
```

The result may be surprising! You never actually find that phrase in the novels; it’s a catchphrase that authors inserted later into the film screenplays: http://www.phrases.org.uk/meanings/elementary-my-dear-watson.html.

IN THIS CHAPTER

- **Understanding how computers can store information in order to save space**
- **Creating efficient and smart encodings**
- **Leveraging statistics and building Huffman trees**
- **Compressing and decompressing on the fly using the Lempel-Ziv-Welch (LZW) algorithm**

# Chapter 14

# Compressing Data

The last decade has seen the world flooded by data. In fact, data is the new oil, and specialists of all sorts hope to extract new knowledge and richness from it. As a result, you find data piled everywhere and often archived as soon as it arrives. The sometimes careless storage of data comes from an increased capacity to store information; it has become cheap to buy larger drives to store everything, useful or not. Chapters 12 and 13 discuss the drivers behind this data deluge, how to deal with massive data streams, methods used to distribute it over clusters of connected computers, and techniques you can use to process data rapidly and efficiently.

Data hasn't always been readily available, however. In previous decades, storing data required large investments in expensive mass-storage devices (hard disk, tapes, floppy disks, CDs) that couldn't store much data. Storing data required a certain efficiency (saving disk space meant saving money), and data compression algorithms offered the solution of compacting data to store more of it on a single device at the cost of computer processing time. Trading disk space for computer time reduced costs.

Compression algorithms have long been a topic of study and are now considered classics in computer knowledge. Even though storage disks are larger and cheaper today, these algorithms still play a role in mobile data transmission and see use anywhere there is a data stream bottleneck or high memory costs. Compression is also handy in situations in which data infrastructure growth doesn't match data growth, which is especially true of both wireless and mobile bandwidth in developing countries. In addition, compression helps deliver complex web pages faster, stream videos efficiently, store data on a mobile device, or reduce mobile phone data transmission costs. This chapter helps you understand how data compression works and when you need to use it.

# Making Data Smaller

Computer data is made of bits — sequences of zeros and ones. This chapter explains the use of zeros and ones to create data in more depth than previous chapters because compression leverages these zeros and ones in multiple ways. To understand compression, you must know how a computer creates and stores binary numbers. The following sections discuss the use of binary numbers in computers.

## Understanding encoding

Zeros and ones are the only numbers in the binary system. They represent the two possible states in an electric circuit: absence and presence of electricity. Computers started as simple circuits made of tubes or transistors; using the binary system instead of the human decimal system made things easier. Humans use ten fingers to count numbers from 0 to 9. When they have to count more, they add a unit number to the left. You may never have thought about it, but you can express counting by using powers of ten. Therefore a number such as 199 can be expressed as $10^2*1 + 10^1*9 +10^0*9 = 199$; that is, you can separate hundreds from tens and units by multiplying each figure by the power of ten relative to its position: $10^0$ for units, $10^1$ for tens, $10^2$ for hundreds, and so on.

REMEMBER

Knowing this information helps you understand binary numbers better because they actually work in exactly the same way. However, binary numbers use powers of two rather than powers of ten. For instance, the number 11000111 is simply

$$
\begin{aligned}
&2^7*1+2^6*1+2^5*0+2^4*0+2^3*0+2^2*1+2^1*1+2^0*1 = \\
&128*1+64*1+32*0+16*0+8*0+4*1+2*1+1*1 = \\
&128+64+4+2+1 = 199
\end{aligned}
$$

You can represent any number as a binary value in a computer. A value occupies the memory space required by its total length. For example, binary 199 is 8 figures, each figure is a bit, and 8 bits are called a byte. The computer hardware knows data only as bits because it has only circuitry to store bits. However, from a higher point of view, computer software can interpret bits as letters, ideograms, pictures, films, and sounds, which is where encoding comes into play.

*Encoding* uses a sequence of bits to represent something other than the number expressed by the sequence itself. For instance, you can represent a letter using a particular sequence of bits. Computer software commonly represents the letter A using the number 65, or binary 01000001 when working with the American Standard Code for Information Interchange (ASCII) encoding standard. You can see sequences used by ASCII system at `http://www.asciitable.com/`. ASCII uses just 7 bits for its encoding (8 bits, or a byte, in the extended version), which means that you can represent 128 different characters (the extended version has 256 characters). Python can represent the string “Hello World” using bytes:

```
print (''.join(['{0:08b}'.format(ord(l))
                for l in "Hello World"]))

0100100001100101011011000110110001101111001000000101011101
101111011100100110110001100100
```

When using extended ASCII, a computer knows that a sequence of exactly 8 bits represent a character. It can separate each sequence into 8-bit bytes and, using a conversion table called a *symbolic table*, it can turn these bytes into characters.

REMEMBER

ASCII encoding can represent the standard Western alphabet, but it doesn’t support the variety of accented European characters or the richness of non-European alphabets, such as the ideograms used by the Chinese and Japanese languages. Chances are that you’re using a robust encoding system such as UTF-8 or another form of Unicode encoding (see `http://unicode.org/` for more information). Unicode encoding is the default encoding in Python 3.

Using a complex encoding system requires that you use longer sequences than those required by ASCII. Depending on the encoding you choose, defining a character may require up to 4 bytes (32 bits). When representing textual information, a computer creates long bit sequences. It decodes each letter easily because encoding uses fixed-length sequences in a single file. Encoding strategies, such as Unicode Transformation Format 8 (UTF-8), can use variable numbers of bytes (1 to 4 in this case). You can read more about how UTF-8 works at `http://www.fileformat.info/info/unicode/utf8.htm`.

## Considering the effects of compression

The use of fixed-sized character sequences leaves a lot of room for improvement. You may not use all the letters in an alphabet, or you use some letters more than others. This is where compression comes into play. By using variable-length character sequences, you can greatly reduce the size of a file. However, the file also requires additional processing to turn it back into an uncompressed format that applications understand. Compression removes space in an organized and methodical manner; decompression adds the space back into the character strings. When it's possible to compress and decompress data in a manner that doesn't result in any data loss, you're using *lossless* compression.

The same idea behind compression goes for images and sounds that involve framing sequences of bits of a certain size in order to represent video details or to reproduce a second of a sound using the computer's speakers. Videos are simply sequences of bits, and each bit sequence is a *pixel,* which is composed of small points that constitute an image. Likewise, audio is composed of sequences of bits that represent an individual sample. Audio files store a certain number of samples per second to recreate a sound. The discussion at `http://kias.dyndns.org/comath/44.html` provides more information about both video and audio storage. Computers store data in many predefined formats of long sequences of bits (commonly called *bit streams*). Compression algorithms can exploit the way each format works to obtain the same result using a shorter, custom format.

You can compress data that represents images and sounds further by eliminating details that you can't process. Humans have both visual and aural limits, so they aren't likely to notice the loss of detail imposed by compressing the data in specific ways. You may have heard of MP3 compression that allows you to store entire collections of CDs on your computer or on a portable reader. The MP3 file format simplifies the original cumbersome WAV format used by computers. WAV files contain all the sound waves received by the computer, but MP3 saves space by removing and compacting waves that you can't hear. (For more more about MP3, see the article at `http://arstechnica.com/features/2007/10/the-audiofile-understanding-mp3-compression/`).

TIP

Removing details from data creates *lossy* compression. JPEG, DjVu, MPEG, MP3, and WMA are all lossy compression algorithms specialized in a particular kind of media data (images, video, sound), and there are many others. Lossy compression is fine for data meant for human input; however, by removing the details, you can't revert to the original data structure. Thus, you can get good digital photo compression and represent it in a useful way on a computer's screen. Yet when you print the compressed photo on paper, you may notice that the quality, though acceptable, is not as good as the original picture. The display provides output at 96 dots per inch (dpi), but a printer typically provides output at 300 to 1200 dpi (or higher). The effects of lossy compression become obvious because a printer is able to display them in a manner that humans can see.

## AN EXAMPLE OF LOSSY COMPRESSION BENEFITS

An example of the difference that lossy compression can make is in photography. A raw formatted picture file contains all the information originally provided by the camera's sensor, so it doesn't include any sort of compression. When working with a certain camera, you might find that this file consumes 29.8MB of hard drive space. A raw file often uses the .raw file extension to show that no processing has occurred. Opening the file and saving it as a lossy .jpeg might result in a file size of only 3.7 MB, but with a corresponding loss of detail. To save some of the detail but obtain some savings in file size as well, you might choose to use the .jpeg file format. In this case, the file size might be 12.4MB, which represents a good compromise in file size savings to loss of image data.

REMEMBER

Choosing between lossless and lossy compression is important. Discarding details is a good strategy for media, but it doesn't work so well with text, because losing words or letters may change the meaning of the text. (Discarding details doesn't work for programming languages or computer instructions for the same reason.) Even though lossy compression is an effective compression solution when details aren't as important, it doesn't work in situations in which precise meaning must be retained.

## Choosing a particular kind of compression

Lossless algorithms simply compress data to reduce its size and decompress it to its original state. Lossless algorithms have more general applications than lossy compression because you can use them for any data problem. (Even when using lossy compression, you remove some detail and further compress what remains using lossless compression.) Just as you can find many lossy algorithms that are specialized for use with different media, so can you find many lossless algorithms, each one adept at exploiting some data characteristics. (To get an idea of how large the lossless algorithm family is, read more details at http://ethw.org/History_of_Lossless_Data_Compression_Algorithms).

REMEMBER

It's essential to remember that the goal of both lossy and lossless compression is to reduce the redundancy contained in data. The more redundancies the data contains, the more effective the compression.

Chances are that you have many lossless data compression programs installed on your computer that output files such as ZIP, LHA, 7-Zip, and RAR, and you aren't sure which one is better. A "best" option may not exist, because you can use bit sequences in many different ways to represent information on a computer; also, different compression strategies work better with different bit sequences. This is the no-free-lunch problem discussed in Chapter 1. The option you choose depends on the data content you need to compress.

To see how compression varies by the sample you provide, you should try various text samples using the same algorithm. The following Python example uses the ZIP algorithm to compress the text of *The Adventures of Sherlock Holmes*, by Arthur Conan Doyle, and then to reduce the size of a randomly generated sequence of letters. (You can find the complete code for this example in the Compression Performances section of the `A4D; 14; Compression.ipynb` file of the downloadable source code for this book; see the Introduction for details).

```
import urllib.request
import zlib
from random import randint
url = "http://gutenberg.pglaf.org/1/6/6/1661/1661.txt"
sh = urllib.request.urlopen(url).read().decode('utf-8')
sh_length = len(sh)
rnd = ''.join([chr(randint(0,126)) for k in
               range(sh_length)])

def zipped(text):
    return len(zlib.compress(text.encode("ascii")))

print ("Original size for both texts: %s characters" %
       sh_length)
print ("The Adventures of Sherlock Holmes to %s" %
       zipped(sh))
print ("Random file to %s " % zipped(rnd))

Original size for both texts: 594941 characters
The Adventures of Sherlock Holmes to 226824
Random file to 521448
```

The output of the example is enlightening. Even though the example application can reduce the size of the short story to less than half of its original size, the size reduction for the random text is much less (both texts have the same original length). The output implies that the ZIP algorithm leverages the characteristics of the written text but doesn't do as well on random text that lacks a predictable structure.

TIP

When performing data compression, you can measure performance by calculating the compression ratio: Just divide the new compressed size of the file by the original size of the file. The compression ratio can tell you about algorithm efficiency in saving space, but high-performance algorithms also require time to perform the task. In case time is your concern, most algorithms let you trade some compression ratio for speedier compression and decompression. In the preceding example for the *Sherlock Holmes* text, the compression ratio is 226824 / 594941, that is, about 0.381. The `compress` method found in the example has a second optional parameter, `level`, which controls the level of compression. Changing this parameter controls the ratio between the time to perform the task and the amount of compression achieved.

## Choosing your encoding wisely

The example in the preceding section shows what happens when you apply the ZIP algorithm to random text. The results help you understand why compression works. Boiling down all the available compression algorithms, you discover four main reasons:

- **Shrinking character encoding:** Compression forces characters to use fewer bits by coding them according to some feature, such as commonality of use. For example, if you use only some of the characters in a character set, you can reduce the number of bits to reflect that level of usage. It's the same difference that occurs between ASCII, which uses 7 bits, and extended ASCII, which uses 8 bits. This solution is particularly effective with problems like DNA encoding, in which you can devise a better encoding than the standard one.
- **Shrinking long sequences of identical bits:** Compression uses a special code to identify multiple copies of the same bits and replaces those copies with just one copy, along with the number of times to repeat it. This option is very effective with images (it works fine with fax black and white images) or with any data that you can rearrange in order to group similar characters together (DNA data is one of this kind).
- **Leveraging statistics:** Compression encodes frequently used characters in a shorter way. For example, the letter *E* appears commonly in English, so if the letter *E* uses only 3 bits, rather than a full 8 bits, you save considerable space. This is the strategy used by Huffman encoding, in which you recreate the symbolic table and save space, on average, because common characters are shorter.
- **Encoding frequent long sequences of characters efficiently:** This is similar to shrinking long sequences of identical bits, but it works with character sequences rather than single characters. This is the strategy used by LZW, which learns data patterns on the fly and creates a short encoding for long sequences of characters.

To understand how rethinking encoding can help in compression, start with the first reason. Scientists working on the Genome Project around 2008 (`https://www.genome.gov/10001772/all-about-the--human-genome-project-hgp/`) managed to drastically reduce the size of their data using a simple encoding trick. Using this trick made the task of mapping the entire human DNA simpler, helping the scientists understand more about the life, disease, and death scripted into our body cells.

Scientists describe DNA using sequences of the letters A, C, T, and G (representing the four nucleotides present in all living beings). The human genome contains six billion nucleotides (you find them associated in couples, called bases) that add up to more than 50GB using ASCII encoding. In fact, you can represent A, C, T, and G in ASCII encoding as follows:

```
print (' '.join(['{0:08b}'.format(ord(l))
                 for l in "ACTG"]))

01000001 01000011 01010100 01000111
```

The sum of the preceding line is 32 bits, but because DNA maps just four characters, you can use 2 bits each, saving 75 percent of the previously used bits:

```
00 01 10 11
```

REMEMBER

Such a gain demonstrates the reason to choose the right encoding. The encoding works fine in this case because the DNA alphabet is made of four letters, and using a full ASCII table based on 8 bits is overkill. If a problem requires that you use the complete ASCII alphabet, you can't compress the data by redefining the encoding used. Instead, you have to approach the problem using Huffman compression.

If you can't shrink the character encoding (or you have already done it), you can still shrink long sequences, reducing them to a simpler encoding. Observe how binary data can repeat long sequences of ones and zeros:

```
00000000 00000000 01111111 11111111 10000011 11111111
```

In this case, the sequence starts from zero. You can therefore count the number of zeros, and then count the number of ones that follow, and then repeat with the next count of zeros, and so on. Because the sequence has only zeros and ones, you can count them and obtain sequence of counts to compress the data. In this case, the data compresses into values of 17 15 5 10. Translating these counts into bytes shortens the initial data in an easily reversible way:

```
00010001 00001111 00000101 00001010
```

Instead of using 6 bytes to represent the data, you now need only 4 bytes. To use this approach, you limit the maximum count to 255 consecutive values, which means:

- You can encode each sequence in a byte.
- The first value is a zero when the sequence starts from 1 instead of 0.
- When a block of values is longer than 255 elements, you insert a 0 value (so the decoder switches to the other value for 0 counts and then starts counting the first value again).

This algorithm, *run-length encoding* (RLE), is very effective if your data has many long repetitions. This algorithm enjoyed great success in the 1980s because it could reduce fax transmission times. Fax machines worked on just black-and-white images, and by land-line telephone, shrinking the long sequences of zeros and ones that made up images and text proved to be convenient. Though businesses seldom use fax machines now, scientists still use RLE for DNA compression in combination with the Burrows-Wheeler Transform (an advanced algorithm that you can read about at `http://marknelson.us/1996/09/01/bwt/`), which rearranges (in a reversible way) the genome sequence in long runs of the same nucleotide. You also find RLE used for compression of other data formats, such as JPEG and MPEG (see `http://motorscript.com/mpeg-jpeg-compression/` for additional details).

TIP

Data characteristics rule the success of a compression algorithm. By knowing how algorithms work and exploring your data characteristics, you can choose the best-performing algorithm or combine more algorithms in an effective way. Using multiple algorithms together creates an *ensemble of algorithms.*

## Encoding using Huffman compression

Redefining an encoding, such as when mapping nucleotides in DNA, is a smart move that works only when you use a part of the alphabet that the encoding represents. When you use all the symbols in the encoding, you can't use this solution. David A. Huffman discovered another way to encode letters, numbers, and symbols efficiently even when using all of them. He achieved this accomplishment when he was a student at MIT in 1952 as part of a term paper required by his professor, Prof. Robert M. Fano. His professor and another famous scientist, Claude Shannon (the father of information theory), had struggled with the same problem.

In his paper, "A Method for the Construction of Minimum-Redundancy Codes," Huffman describes in just three pages his mind-blowing encoding method.

It changed the way we stored data until the end of 1990s. You can read the details about this incredible algorithm in a September 1991 *Scientific American* article at `http://www.huffmancoding.com/my-uncle/scientific-american`. Huffman codes have three key ideas:

- **Encode frequent symbols with shorter sequences of bits.** For instance, if your text uses the letter *a* often, but rarely uses the letter *z,* you can encode *a* using a couple of bits and reserve an entire byte (or more) for *z*. Using shorter sequences for common letters means that overall your text requires fewer bytes than when you rely on ASCII encoding.
- **Encode shorter sequences using a unique series of bits.** When using variable length bit sequences, you have to ensure that you can't misinterpret a shorter sequence in place of a longer one because they are similar. For instance, if the letter *a* in binary is 110 and *z* is 110110, you could misinterpret the letter *z* as a series of two-letter *a* characters. Huffman encoding avoids this problem by using prefix-free codes: The algorithm never reuses shorter sequences as initial parts of longer sequences. For example, if *a* is 110, then *z* will be 101110 and not 110110.
- **Manage prefix-free coding using a specific strategy.** Huffman encoding manages prefix-free codes by using binary trees in a smart way. Binary trees are a data structure discussed in Chapters 6 and 7. The Huffman algorithm uses binary trees (called Huffman trees) in an advanced fashion. You can read more about the internals of the algorithm in the tutorial at `https://www.siggraph.org/education/materials/HyperGraph/video/mpeg/mpegfaq/huffman_tutorial.html`.

TECHNICAL STUFF

The algorithm used to perform Huffman encoding uses an iterative process that relies on *heaps*, which are specialized tree-based data structures (mentioned in Chapter 6). A heap is a complex data structure. Because of the manner in which you use a heap to arrange data, it's useful for achieving a *greedy* strategy. In the next chapter, which is devoted to greedy algorithms, you test Huffman encoding yourself, using the working examples in the downloadable code accompanying the book (the Huffman Compression example in the `A4D; 15; Greedy Algorithms.ipynb` file; see the Introduction for details).

For the moment, as an example of a Huffman encoding output, Figure 14-1 shows the Huffman encoding binary tree used to encode a long sequence of *ABCDE* letters distributed in a way that *A* is more frequent than *B*, *B* more than *C*, *C* more than *D*, and *D* more than *E.*

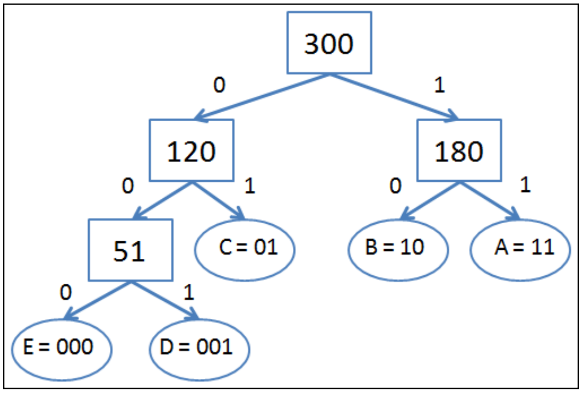


**FIGURE 14-1:** A Huffman tree and its symbolic table of conversion.

The square nodes represent *branch nodes,* where the algorithm places the number of the remaining letters it distributes to the *child nodes* (those that are below the branch nodes in the hierarchy). The round nodes represent *leaf nodes,* where you find the successfully encoded letters. The tree starts at the *root* with 300 letters left to distribute (the length of the text). It distributes the letters by branching the 0 and 1 bits, respectively, on the left and on the right branches until it reaches all the leaves necessary for encoding. By reading from the top of the sequence of branches to a specific letter, you determine the binary sequence representing that letter. Less frequent letters (*D* and *E*) get the longest binary sequences.

TIP

Following the Huffman tree from bottom to top lets you compress a symbol into a binary sequence. By following the tree from top to bottom, you can decompress a binary sequence into a symbol (as represented by the first leaf node you encounter).

REMEMBER

For decompression, you need to store both the compressed binary sequence and the Huffman tree that made the compression possible. When your text or data is too short, the Huffman tree could require more space than the compressed data, thus making compression ineffective. Huffman code works best on larger data files.

## Remembering sequences with LZW

Huffman encoding takes advantage of the most frequent characters, numbers, or symbols in data and shortens their encoding. The LZW algorithm performs a similar task but extends the encoding process to the most frequent sequences of characters. The LZW algorithm dates to 1984 and was created by Abraham Lempel, Jacob Ziv, and Terry Welch based on an earlier LZ78 algorithm (developed in 1978 by Lempel and Ziv alone). Both Unix compression and the GIF image format rely on this algorithm. LZW leverages repetitions, so it's also ideal for document and book text compression because humans often use the same words when writing. In addition, LZW can operate on streaming data, but Huffman can't; Huffman needs the full dataset to build its mapping table.

As the algorithm skims through the data-bit stream, it learns sequences of characters from it and assigns each sequence to a short code. Thus, when later reencountering the same series of characters, LZW can compress them using a simpler encoding. The interesting aspect of this algorithm is that it starts from a symbolic table made of single characters (usually the ASCII table) and then it enlarges that table using the character sequences it learns from the data it compresses.

Moreover, LZW doesn't need to store the learned sequences in a table for decompression; it can rebuild them easily by reading the compressed data. LZW can reconstruct the steps it took when compressing the original data and the sequences it encoded. This capability comes at a price; LZW isn't efficient at first. It works best with large pieces of data or text (a characteristic common to other compression algorithms).

LZW isn't a complex algorithm, but you need to see a number of examples to understand it fully. You can find quite a few good tutorials at `http://marknelson.us/2011/11/08/lzw-revisited/` and `http://www.matthewflickinger.com/lab/whatsinagif/lzw_image_data.asp`. The second tutorial explains how to use LZW to compress images. The following example shows a Python implementation. (You can find the complete code for this example in the LZW section of the `A4D; 14; Compression.ipynb` file of the downloadable source code for this book; see the Introduction for details).

```
def lzw_compress(text):
    dictionary = {chr(k): k for k in range(256)}
    encoded = list()
    s = text[0]
    for c in text[1:]:
        if s+c in dictionary:
            s = s+c
        else:
            print ('> %s' %s)
            encoded.append(dictionary[s])
            print ('found: %s compressed as %s' %
                   (s,dictionary[s]))
            dictionary[s+c] = max(dictionary.values()) + 1
            print ('New sequence %s indexed as %s' %
                   (s+c, dictionary[s+c]))
            s = c
    encoded.append(dictionary[s])
    print ('found: %s compressed as %s'
           %(s,dictionary[s]))
    return encoded
```

In this example, the algorithm scans the text by checking the text a character at a time. It begins by encoding characters using the initial symbolic table, which is actually the ASCII table in this case. The best way to see how this code works is to see a series of output messages and then analyze what has taken place, as shown here:

```
text = "ABABCABCABC"
compressed = lzw_compress(text)
print('\nCompressed: %s \n' % compressed)

> A
found: A compressed as 65
New sequence AB indexed as 256
> B
found: B compressed as 66
New sequence BA indexed as 257
> AB
found: AB compressed as 256
New sequence ABC indexed as 258
> C
found: C compressed as 67
New sequence CA indexed as 259
> ABC
found: ABC compressed as 258
New sequence ABCA indexed as 260
found: ABC compressed as 258
```

Here is a quick synopsis of what these output messages mean:

1. The first letter, *A,* appears in the initial symbolic table, so the algorithm encodes it as 65.
2. The second letter, *B,* is different from *A* but also appears in the initial symbolic table, so the algorithm encodes it as 66.
3. The third letter is another *A,* so the algorithm reads the next letter, which is a *B,* and encodes the two-letter combination, *AB,* as 256.
4. The fourth letter, a *C,* is different from any of the previous letters and also appears in the initial symbolic table, so the algorithm encodes it as 67.

5. The next letter has already appeared before; it's an *A.* The next letter is a *B,* which makes the *AB* letter combination; this also appears in the symbolic table. However, the next letter is a *C,* which makes a new sequence and which the algorithm now encodes as 258.
6. The final three letters are another set of *ABC,* so the code for them is 258 again. Consequently, the encoded output for *ABABCABCABC* is

   ```
   Compressed: [65, 66, 256, 67, 258, 258]
   ```

All the learning and encoding operations translated into final compression data consisting of just six numeric codes (costing 8 bits each) against the initial 11 test letters. The encoding results in a good compression ratio of about half the initial data: 6/11 = 0.55.

Retrieving the original text from the compressed data requires a different, inverse procedure, which accounts for the only situation when LZW decoding may fail to reconstruct the symbolic table when a sequence starts and ends with the same character. This particular case is taken care of by Python using an if-then-else command block, so you can safely use the algorithm to encode and decode anything:

```
def lzw_decompress(encoded):
    reverse_dictionary = {k:chr(k) for k in range(256)}
    current = encoded[0]
    output = reverse_dictionary[current]
    print ('Decompressed %s ' % output)
    print ('>%s' % output)
    for element in encoded[1:]:
        previous = current
        current = element
        if current in reverse_dictionary:
            s = reverse_dictionary[current]
            print ('Decompressed %s ' % s)
            output += s
            print ('>%s' % output)
            new_index = max(reverse_dictionary.keys()) + 1
            reverse_dictionary[new_index
            ] = reverse_dictionary[previous] + s[0]
            print ('New dictionary entry %s at index %s' %
                    (reverse_dictionary[previous] + s[0],
                     new_index))
```

```
        else:
            print ('Not found:',current,'Output:',
                   reverse_dictionary[previous
                    ] + reverse_dictionary[previous][0])
            s = reverse_dictionary[previous
                    ] + reverse_dictionary[previous][0]
            print ('New dictionary entry %s at index %s' %
                   (s, max(reverse_dictionary.keys())+1))
            reverse_dictionary[
                max(reverse_dictionary.keys())+1] = s
            print ('Decompressed %s' % s)
            output += s
            print ('>%s' % output)
    return output
```

Running the function on the previously compressed sequence recovers the original information by scanning through the symbolic table, as shown here:

```
print ('\ndecompressed string : %s' %
       lzw_decompress(compressed))
print ('original string was : %s' % text)

Decompressed A
> A
Decompressed B
> AB
New dictionary entry AB at index 256
Decompressed AB
> ABAB
New dictionary entry BA at index 257
Decompressed C
> ABABC
New dictionary entry ABC at index 258
Decompressed ABC
> ABABCABC
New dictionary entry CA at index 259
Decompressed ABC
> ABABCABCABC
New dictionary entry ABCA at index 260

decompressed string : ABABCABCABC
original string was : ABABCABCABC
```

# 5

# Challenging Difficult Problems

## IN THIS PART . . .

Use greedy programming techniques to obtain results faster.

Perform dynamic programming to perform tasks using a smart approach.

Randomize your results to solve problems where a straightforward approach doesn't work well.

Search locally to final solutions that are good enough in a short time.

Use linear programming techniques to perform scheduling and planning tasks.

Employ heuristics and interact with robots.

IN THIS CHAPTER

- Understanding how to design new algorithms and use solving paradigms
- Explaining how an algorithm can act greedy and get great results
- Drafting a greedy algorithm of your own
- Revisiting Huffman coding and illustrating some other classical examples

# Chapter 15
# Working with Greedy Algorithms

After taking your first steps into the world of algorithms by presenting what algorithms are and discussing ordering, searching, graphs, and big data, it's time to enter a more general part of the book. In this latter part of the book, you deal with some difficult examples and see general algorithmic approaches that you can use under different circumstances when solving real-world problems.

By taking new routes and approaches, this chapter goes well beyond the divide-and-conquer recursion approach that dominates in most sorting problems. Some of the discussed solutions aren't completely new; you've seen them in previous chapters. However, this chapter discusses those previous algorithms in greater depth, under the new *paradigms* (a consideration of application rules and conditions, general approach and steps to the solution of a problem, and analysis of problem complexity, limitations, and caveats) that the chapter illustrates.

Making some solutions general and describing them as widely applicable paradigms is a way to offer hints to solve new practical problems and is part of the analysis and design of algorithms. The remainder of this book discusses the following general approaches:

- Greedy algorithms (explained in this chapter)
- Dynamic programming
- Randomization, local search, and farsighted heuristics
- Linear programming and optimization problems

## Deciding When It Is Better to Be Greedy

When faced with difficult problems, you quickly discover that no magic potion exists for making wishes come true or silver bullets to dispel bad things. Similarly, no algorithmic technique saves the day every time. That's the *no-free-lunch* principle often quoted in the book. The good news is, you can arm yourself with different general techniques and test them all on your problem, because you have a good chance that something will work well.

Greedy algorithms come in handy for solving a wide array of problems, especially when drafting a global solution is difficult. Sometimes, it's worth giving up complicated plans and simply start looking for low-hanging fruit that resembles the solution you need. In algorithms, you can describe a shortsighted approach like this as *greedy*. Looking for easy-to-grasp solutions constitutes the core distinguishing characteristic of greedy algorithms. A greedy algorithm reaches a problem solution using sequential steps where, at each step, it makes a decision based on the best solution at that time, without considering future consequences or implications.

REMEMBER

Two elements are essential for distinguishing a greedy algorithm:

- At each turn, you always make the best decision you can at that particular instant.
- You hope that making a series of best decisions results in the best final solution.

Greedy algorithms are simple, intuitive, small, and fast because they usually run in *linear time* (the running time is proportional to the number of inputs provided). Unfortunately, they don't offer the best solution for all problems, but when they do, they provide the best results quickly. Even when they don't offer the top answers, they can give a nonoptimal solution that may suffice or that you can use as a starting point for further refinement by another algorithmic strategy.

Interestingly, greedy algorithms resemble how humans solve many simple problems without using much brainpower or with limited information. For instance, when working as cashiers and making change, a human naturally uses a greedy approach. You can state the *make-change* problem as paying a given amount (the change) using the least number of bills and coins among the available denominations. The following Python example demonstrates the make-change problem is solvable by a greedy approach. It uses the 1, 5, 10, 20, 50, and 100 USD bills, but no coins.

```
def change(to_be_changed, denomination):
    resulting_change = list()
    for bill in denomination:
        while to_be_changed >= bill:
            resulting_change.append(bill)
            to_be_changed = to_be_changed - bill
    return resulting_change, len(resulting_change)

currency = [100, 50, 20, 10, 5, 1]
amount = 367
print ('Change: %s (using %i bills)'
       % (change(amount, currency)))

Change: [100, 100, 100, 50, 10, 5, 1, 1] (using 8 bills)
```

The algorithm, encapsulated in the `change()` function, scans the denominations available, from the largest to the smallest. It uses the largest available currency to make change until the amount due is less than the denomination. It then moves to the next denomination and performs the same task until it finally reaches the lowest denomination. In this way, `change()` always provides the largest bill possible given an amount to deliver. (This is the greedy principle in action.)

Greedy algorithms are particularly appreciated for scheduling problems, optimal caching, and compression using Huffman coding. They also work fine for some graph problems. For instance, Kruskal's and Prim's algorithms for finding a minimum-cost spanning tree and Dijkstra's shortest-path algorithm are all greedy ones (see Chapter 9 for details). A greedy approach can also offer a nonoptimal, yet an acceptable first approximation, solution to the traveling salesman problem (TSP) and solve the knapsack problem when quantities aren't discrete. (Chapter 16 discusses both problems.)

## Understanding why greedy is good

It shouldn't surprise you that a greedy strategy works so well in the make-change problem. In fact, some problems don't require farsighted strategies: The solution is built using intermediate results (a sequence of decisions), and at every step the right decision is always the best one according to an initially chosen criteria.

Acting greedy is also a very human (and effective) approach to solving economic problems. In the 1987 film *Wall Street*, Gordon Gecko, the protagonist, declares that "Greed, for lack of a better word, is good" and celebrates greediness as a positive act in economics. Greediness (not in the moral sense, but in the sense of acting to maximize singular objectives, as in a greedy algorithm) is at the core of the neoclassical economy. Economists such as Adam Smith, in the eighteenth century, theorized that the individual's pursuit of self-interest (without a global vision or purpose) benefits society as a whole greatly and renders it prosperous in economy (it's the theory of the invisible hand: `https://plus.maths.org/content/adam-smith-and-invisible-hand`).

Detailing how a greedy algorithm works (and under what conditions it can work correctly) is straightforward, as explained in the following four steps:

1. You can divide the problem into partial problems. The sum (or other combination) of these partial problems provides the right solution. In this sense, a greedy algorithm isn't much different from a divide-and-conquer algorithm (like Quicksort or Mergesort, both of which appear in Chapter 7).
2. The successful execution of the algorithm depends on the successful execution of every partial step. This is the *optimal substructure characteristic* because an optimal solution is made only of optimal subsolutions.
3. To achieve success at each step, the algorithm considers the input data only at that step. That is, situation status (previous decisions) determines the decision the algorithm makes, but the algorithm doesn't consider consequences. This complete lack of a global strategy is the *greedy choice property* because being greedy at every phase is enough to offer ultimate success. As an analogy, it's akin to playing the game of chess by not looking ahead more than one move, and yet winning the game.
4. Because the greedy choice property provides hope for success, a greedy algorithm lacks a complex decision rule because it needs, at worst, to consider all the available input elements at each phase. There is no need to compute possible decision implications; consequently, the computational complexity is at worst linear O(n). Greedy algorithms shine because they take the simple route to solving highly complex problems that other algorithms take forever to compute because they look too deep.

## Keeping greedy algorithms under control

When faced with a new difficult problem, it's not hard to come up with a greedy solution using the four steps described in the previous section. All you have to do is divide your problems into phases and determine which greedy rule to apply at each step. That is, you do the following:

- Choose how to make your decision (determine which approach is the simplest, most intuitive, smallest, and fastest)
- Start solving the problem by applying your decision rule
- Record the result of your decision (if needed) and determine the status of your problem
- Repeatedly apply the same approach at every step until reaching the problem conclusion

No matter how you apply the previous steps, you must determine whether you're accomplishing your goal by relying on a series of myopic decisions. The greedy approach works for some problems and sometimes for specific cases of some problems, but it doesn't work for every problem. For instance, the make-change problem works perfectly with U.S. currency but produces less-than-optimal results with other currencies. For example, using a fictional currency (call it *credits*, using a term in many sci-fi games and fiction) with denominations of 1, 15, and 25 credits, the previous algorithm fails to deliver the optimal change for a due sum of 30 credits:

```
print ('Change: %s (using %i bills)'
       % (change(30, [25, 15, 1])))

Change: [25, 1, 1, 1, 1, 1] (using 6 bills)
```

Clearly, the optimal solution is to return two 15 credit bills, but the algorithm, being shortsighted, started with the highest denomination available (25 credits) and then used five 1 credit bills to make up the residual 5 credits.

TECHNICAL STUFF

Some complex mathematical frameworks called *matroids* (read the article at `https://jeremykun.com/2014/08/26/when-greedy-algorithms-are-perfect-the-matroid/` for details) can help verify whether you can use a greedy solution to optimally solve a particular problem. If phrasing a problem using a matroid framework is possible, a greedy solution will provide an optimal result. Yet there are problems that have optimal greedy solutions that don't abide by the matroid framework. (You can read about matroid structures being sufficient, but not necessary for an optimal greedy solution in the article found at `http://cstheory.stackexchange.com/questions/21367/does-every-greedy-algorithm-have-matroid-structure`.)

The greedy algorithms user should know that greedy algorithms do perform well but don't always provide the best possible results. When they do, it's because the problem consists of known examples or because the problem is compatible with matroid mathematical framework. Even when a greedy algorithm works best in one setting, changing the setting may *break the toy* and generate just good or acceptable solutions. In fact, the cases of just good or acceptable results are many, because greedy algorithms don't often outperform other solutions, as shown by

- The make-change problem solutions demonstrated earlier in this chapter show how a change in setting can cause a greedy algorithm to stop working.
- The scheduling problem (described in the "Finding Out How Greedy Can Be Useful" section, later in this chapter) illustrates how a greedy solution works perfectly with one worker, but don't expect it to work with more than one.
- The Dijkstra shortest-path algorithm works only with edges having positive weights. (Negative weights will cause the algorithm to loop around some nodes indefinitely.)

Demonstrating that a greedy algorithm works the best is a hard task, requiring a solid knowledge of mathematics. Otherwise, you can devise a proof in a more empirical way by testing the greedy solution against one of the following:

- Against a known optimal solution, when the greedy algorithm produces the optimal solution or you can change such a solution by exchanging its elements into an equivalent best solution (without any loss of performance or success). When a greedy solution matches the result of an optimal solution, you know that the greedy solution is equivalent and that it works best (this is the *exchange* proof).
- Against another algorithm when, as you see the greedy solution unfolding, you notice that the greedy solution *stays ahead* of the other algorithm; that is, the greedy solution always provides a better solution at every step than is provided by another algorithm.

TIP

Even considering that it's more the exception than a rule that a successful greedy approach will determine the top solution, greedy solutions often outperform other tentative solutions. You may not always get the top solution, but the solution will provide results that are good enough to act as a starting point (as a minimum), which is why you should start by trying greedy solutions first on new problems.

## Considering NP complete problems

Usually you think of a greedy algorithm because other choices don't compute the solution you need in a feasible time. The greedy approach suits problems for which you have many choices and have to combine them. As the number of possible combinations increases, complexity explodes and even the most powerful computer available fails to provide an answer in a reasonable time. For example, when attempting to solve a puzzle, you could try to solve it by determining all the possible ways you can fit the available pieces together. A more reasonable way is to start solving the problem by choosing a single location and then finding the best-fitting piece for it. Solving the puzzle this way means using time to find the best fitting piece, but you don't have to consider that location again, reducing the total number of pieces for each iteration.

Puzzle problems, in which the number of possible decisions can become huge, are more frequent than you expect. Some problems of this type have already been solved, but many others aren't, and we can't even transform them (yet) into versions we know how to solve. Until someone is smart enough to find a generic solution for these problems, a greedy approach may be the easiest way to approach them, provided that you accept that you won't always be getting the best solution but a roughly acceptable one instead (in many cases).

These difficult problems vary in characteristics and problem domain. Different examples of difficult problems are protein unfolding (which can help cure cancer) or breaking strong password encryption, such as the popular RSA cryptosystem (http://blogs.ams.org/mathgradblog/2014/03/30/rsa/). In the 1960s, researchers found a common pattern for all of them: They are all equally difficult to solve. This pattern is called the NP-completeness theory (NP stands for nondeterministic polynomial). In a sense, these problems distinguish themselves from others because it's not yet possible to find a solution in a reasonable time —that is, in polynomial time.

*Polynomial time* means that an algorithm runs in powers of the number of inputs (known as P problems). Linear time is polynomial time because it runs $O(n^1)$. Also quadratic $O(n^2)$ and cubic $O(n^3)$ complexities are polynomial time, and though they grow quite fast, they don't compare to NP-complete complexity, which is usually exponential time, that is, $O(c^n)$. *Exponential time* complexity makes it impossible to find a reasonable solution for any of these problems using brute force. In fact if *n* is large enough, you may easily have to try a number of solutions larger than the number of atoms present in the known universe. The hope of algorithm experts is that someone will find a way to solve any of these problems in the future, thus opening the door to solving all the NP-complete problems at one time. Solving NP-complete problems is one of the "Millennium Prize Problems" proposed by the Clay Mathematics Institute, which offers an award of one million USD to anyone who can devise a solution (http://www.claymath.org/millennium-problems/p-vs-np-problem).

REMEMBER

NP is a broad class of algorithmic problems that comprises both P and NP-complete problems. Generally, NP problems are difficult (the ones that require you to devise a smart algorithm). P problems are solvable in polynomial time; NP-complete problems are so hard to solve that the associated algorithms run in exponential time. Fortunately, if you have a solution for an NP-complete problem, you can easily check its validity.

TIP

Maybe you won't solve any NP-complete problem using an algorithm specifically designed to find an optimal solution. However, you can still find a reasonable solution using greedy algorithms.

# Finding Out How Greedy Can Be Useful

After discussing greedy algorithms generally, it's illuminating to describe some of them in detail, understanding how they work and determine how to reuse their strategies for solving other problems. The following sections review the Huffman coding algorithm to provide more insight on the way it works to create new efficient encoding systems. These sections also describe how a *computer cache* (an algorithm always found under the hood of any computer) works. In addition, you discover how to schedule tasks correctly to achieve deadlines and priorities. Production of material goods strongly relies on greedy algorithms to schedule resources and activities. Usually, activity algorithms appear at the core of Material Requirements Planning (MRP) software, and they help run a factory efficiently (`http://searchmanufacturingerp.techtarget.com/definition/Material-requirements-planning-MRP`).

## Arranging cached computer data

Computers are often processing the same data multiple times. Obtaining data from disk or the Internet requires times and costs computational time. Consequently, it's useful to store often-used data in local storage where it's easier to access (and maybe already preprocessed). A *cache,* which is usually a series of memory slots or space on disk reserved for that need, fulfills the purpose.

For instance, when reviewing your web browser's history, you likely notice that only a part of traffic is made of new websites, whereas you spend a large amount of time and page requests on sites you know well. Storing in cache some parts of commonly seen websites (such as the header, background, some pictures, and some pages that seldom change) can really improve your web experience because it reduces the need to download data again. All you need is the new data from the Internet because most of what you want to see is already somewhere in your computer. (The cache of a web browser is a disk directory.)

The problem isn't new. In the 1960s, László Bélády, a Hungarian computer scientist working at IBM Research, hypothesized that the best way to store information in a computer for prompt reuse is to know what data is needed in the future and for how long. It isn't possible to implement such forecasting in practice because computer usage can be unpredictable and not predetermined.

Yet, as a principle, the idea of anticipating the future can inspire an *optimal replacement strategy,* a greedy choice based on the idea of keeping the pages that you expect to use soon based on previous requests to the cache. Bélády's optimal page replacement policy (also known as the *clairvoyant replacement algorithm*) works on a greedy principle: to discard data from the cache whose next use will likely occur farthest in the future in order to minimize the chance of discarding something you need soon. To implement this idea, the algorithm follows these steps:

1. Fill the computer cache by recording data from every request made. Only when the cache is full do you start discarding past stuff to make room for new data.
2. Define a method for determining recent usage. This algorithm can use file date stamps or a system of memory flags (which flags recently used pages and clears all the flags after a certain time) to make the determination.
3. When you have to fit new data, you discard data that hasn't been used recently from the cache. The algorithm picks one piece of data randomly among the ones not used.

For instance, if your cache has only four memory slots, and it is filled by four alphabet letters that arrive in the following order:

| | | | |
|---|---|---|---|
| A | B | C | D |

when a new letter is processed, such as the letter *E,* the computer makes space for it by removing one of the letters that are less likely to be requested at this point. In this example, good candidates are *A, B,* or *C* (*D* is the most recent addition). The algorithm will pick one slot randomly and evict its data from the cache in order to let *E* in.

## Competing for resources

When you want to achieve an objective, such as to create a service or produce a material object, a common problem is scheduling several competing activities that require exclusive access to resources. Resources can include time or a production machine. Examples of such situations abound in the real world, ranging from scheduling your attendance at university courses to arranging the supplies of an army, or from assembling a complex product such as a car to organizing a computational job sequence in a data center. Invariably, common goals in such situations are to

- Achieve getting the most jobs done in a certain amount of time
- Manage jobs as quickly as possible, on average
- Respect some strict priorities (hard deadlines)
- Respect some priority indications (soft deadlines)

Job scheduling falls into two categories:

- Jobs that are hard to solve properly and require advanced algorithms to solve
- Jobs that are easier to deal with and that can be solved by simple greedy algorithms

Most of the scheduling you perform actually falls among those solvable by greedy algorithms. For instance, managing jobs as quickly as possible is a common requirement for industrial production or the service industry when each job serves the needs of a client and you want to do your best for all your clients. Here's how you can determine a context for such an algorithm:

- You have a single machine (or worker) that can work out orders.
- Orders arrive in batches, so you have many to choose from at a time.
- Orders differ in length, each requiring a different execution time.

For instance, you receive four jobs from four business customers requiring eight, four, 12, and three hours, respectively, to execute. Even though the total execution time remains the same, changing the job-execution order changes the time when you complete the jobs and dictates how long each business customer has to wait before having its job done. The following sections consider different methods for meeting business customer needs given specific goals.

## Addressing customer satisfaction

Business is about keeping customers happy. If you execute the jobs in the order presented, the work takes 8+4+12+3=27 hours to execute completely. Yet, the first customer will receive its job after eight hours, the last one after 27 hours. In fact, the first job completes in eight hours, the second in 8+4=12 hours, the third in 8+4+12=24 hours, the last in 8+4+12+3=27 hours.

If you aim at having all your customers happy and satisfied, you should strive to minimize the average waiting time for each of them. This measure is given by the average of the delivery times: (8+12+24+27)/4=17.75 hours on average to wait for a job. To reduce the average wait time, you could start simulating all the possible combinations of order execution and recalculate the estimate. This is feasible for a few jobs on a single machine, but if you have hundreds of them on multiple machines, that becomes a very heavy computational problem. A greedy algorithm can save the day without much planning: just execute the shortest first. The resulting average will be the smallest possible: (3+(3+4)+(3+4+8)+(3+4+8+12))/4= 13 hours on average.

REMEMBER

To obtain the average wait time, you take the average of the cumulated sums of runtimes. If you instead take the average of raw times, you obtain the average length of a task, which doesn't represent the customer's waiting time.

The greedy principle is simple: Because you sum cumulative times, if you start by running the longest tasks, you extend the longest run to all successive execution times (because it is a cumulative sum). If instead you start with the shortest jobs, you draw the smallest times first, positively affecting the average (and your customers' level of satisfaction).

## Meeting deadlines

Sometimes, more than just wanting to make your customers wait less time, you also have to respect their time requirements, meaning that you have deadlines. When you have deadlines, the greedy mechanism changes. Now you don't start from the shortest task but with the task that you must deliver the earliest, according to the principle *the earliest, the better*. This is the problem of hard deadlines, and it's a problem you can actually fail to solve. (Some deadlines are simply impossible to meet.)

TIP

If you try a greedy strategy and can't solve the problem, you can acknowledge that no solution to the required deadline exists. When hard deadlines can't work, you can try to solve the problem using soft deadlines instead, meaning that you have to respect a priority (executing certain tasks first).

In this example, you have both the lengths of the tasks, as discussed in the previous section, and you have a value (a *weight*) that defines the task importance (larger weights, higher priority). It's the same problem, but this time you must minimize the weighted average completion time. To achieve this goal, you create a priority score by dividing the time lengths by the weights, and you start with the tasks that have the lowest score. If a task has the lowest score, it's because it is high priority or very short.

For instance, reprising the previous example, you now have tuples of both weights and lengths: (40,8), (30,4), (20,12), (10,3), where 40 in the first tuple is a weight and 8 is a length. Divide each length by the weight and you get priority scores of: 0.20, 0.13, 0.60, and 0.30. Start from the lowest-priority score and, adding the lowest left priority score, you obtain a best schedule that assures that you both minimize times and respect priorities: (30,4), (40,8), (10,3), (20,12).

## Revisiting Huffman coding

As seen in the previous chapter, Huffman coding can represent data content in a more compact form by exploiting the fact that some data (for instance certain characters of the alphabet) appear more frequently in a data stream. By using encodings of different length (shorter for the most frequent characters, longer for the least frequent ones), the data consumes less space. Prof. Robert M. Fano (Huffman's professor) and Claude Shannon already envisioned such a compression strategy but couldn't find an efficient way to determine an encoding arrangement that would make it impossible to mistake one character for another.

REMEMBER

Prefix-free codes are necessary in order to avoid errors when decoding the message. It means that no previously used bit encoding should be used as the starting point of another bit encoding. Huffman found a simple and workable solution for implementing prefix-free codes using a greedy algorithm. The solution to the prefix-free problem found by Huffman is to transform the originally balanced tree (a data structure discussed in Chapter 6) containing the fixed-length encoding into an unbalanced tree, as shown in Figure 15-1.

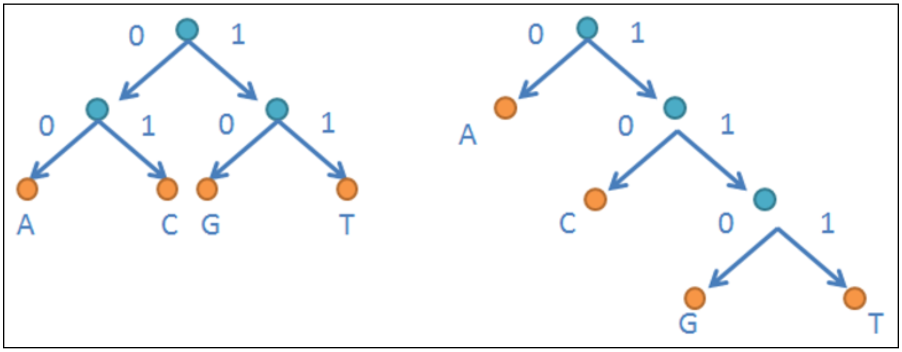


**FIGURE 15-1:** From a balanced tree (left) to an unbalanced tree (right).

An unbalanced tree has a special characteristic that each node has only one branch that keeps on developing in other nodes and branches, whereas the other branch terminates with an encoded character. This characteristic assures that no previously used encoding sequence can start a new sequence (graphically, a branch terminating with an encoded character is a dead end).

Apart from graphically drafting the unbalanced structure, a greedy algorithm can also construct an unbalanced tree. The idea is to build the structure up from the root, starting with the least frequently used characters. The algorithm creates the upper levels of the tree by aggregating less frequent characters in sequence until there are no more characters and you reach the top.

To demonstrate the greedy recipe behind the algorithm, this section provides a Python code example based on DNA. DNA is represented as a sequence of the letters *A*, *C*, *T*, and *G* (the four nucleotides present in all living beings). A good trick is to use just two bits in order to represent each of the four letters, which is already a good memory-saving strategy when compared to using a full ASCII encoding (which is at least 7 bits).

The nucleotides aren't uniformly distributed. The distribution varies depending on what genes you study. The following table shows a gene with an uneven distribution, allowing for a predominance of *A* and *C* nucleotides.

| Nucleotides | Percentage | Fixed Encoding | Huffman Encoding |
|---|---|---|---|
| A | 40.5% | 00 | 0 |
| C | 29.2% | 01 | 10 |
| G | 14.5% | 10 | 110 |
| T | 15.8% | 11 | 111 |
| Weighted bit average | | 2.00 | 1.90 |

By multiplying the number of bits of the two encodings by their percentage and summing everything, you obtain the weighted average of bits used by each of them. In this case, the result is 1.9 for the Huffman encoding versus 2.0 for the fixed encoding. It means that you obtain a five percent bit saving in this example. You could save even more space when having genes with an even more unbalanced distribution in favor of some nucleotide.

The following example generates a random DNA sequence and shows how the code systematically generates the encoding. (If you change the seed value, the random generation of the DNA sequences may lead to a different result, both in the distribution of nucleotides and Huffman encoding.)

```
from heapq import heappush, heappop, heapify
from collections import defaultdict, Counter
from random import shuffle, seed
```

```
generator = ["A"]*6+["C"]*4+["G"]*2+["T"]*2
text = ""
seed(4)
for i in range(1000):
    shuffle(generator)
    text += generator[0]

print(text)
frequencies = Counter(list(text))
print(frequencies)

CAACCCCGACACGCCTCCATAGCCACAACAAGCAAAAAAGGC ...
Counter({'A': 405, 'C': 292, 'T': 158, 'G': 145})
```

After making the data inputs ready to compress, the code prepares a heap data structure (see the "Performing specialized searches using a binary heap" section of Chapter 7 for details) to arrange the results efficiently along the steps the algorithm takes. The elements in the heap contain the frequency number of nucleotides, the nucleotide characters, and its encoding. With a log-linear time complexity, O(n*log(n)), a heap is the right structure to use to order the results and allow the algorithm to draw the two smallest elements quickly.

```
heap = ([[freq, [char, ""]] for char, freq in
         frequencies.items()])
heapify(heap)
print(heap)

[[145, ['G', '']], [158, ['T', '']], [405, ['A', '']],
  [292, ['C', '']]]
```

When you run the algorithm, it picks the nucleotides with fewer frequencies from the heap (the greedy choice). It aggregates these nucleotides into a new element, replacing the previous two. The process continues until the last aggregation reduces the number of elements in the heap to one.

```
iteration = 0
while len(heap) > 1:
    iteration += 1
    lo = heappop(heap)
    hi = heappop(heap)
    print ('Step %i 1st:%s 2nd:%s' % (iteration, lo,hi))
    for pair in lo[1:]:
        pair[1] = '0' + pair[1]
    for pair in hi[1:]:
        pair[1] = '1' + pair[1]
    heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

Step 1 1st:[145, ['G', '']] 2nd:[158, ['T', '']]
Step 2 1st:[292, ['C', '']] 2nd:[303, ['G', '0'],
 ['T', '1']]
Step 3 1st:[405, ['A', '']] 2nd:[595, ['C', '0'],
 ['G', '10'], ['T', '11']]
```

As the aggregations put the nucleotides together, which constitutes different levels of the unbalanced tree, their Huffman encoding is systematically modified; adding a zero in front of the encoding of the lowest frequent aggregate and adding one to the second-lowest frequent one. In this way, the algorithm efficiently replicates the unbalanced tree structure previously illustrated.

```
tree = sorted(heappop(heap)[1:], key=lambda p: (len(p[-
            1]), p))
print ("Symbol\tWeight\tCode")
for e in tree:
    print ("%s\t%s\t%s" % (e[0], frequencies[e[0]], e[1]))

Symbol   Weight   Code
A        405      0
C        292      10
G        145      110
T        158      111
```

The final step is to print the result, sorting it by the bit encoding and showing the final symbol table generated.

IN THIS CHAPTER

- » Understanding what *dynamic* means when used with programming
- » Using memoization effectively for dynamic programming
- » Discovering how the knapsack problem can be useful for optimization
- » Working with the NP-complete traveling salesman problem

Chapter **16**

# Relying on Dynamic Programming

Instead of using brute force, which implies trying all possible solutions to a problem, greedy algorithms provide an answer that is quick and often satisfying. In fact, a greedy algorithm can potentially solve the problem fully. Yet, greedy algorithms are also limited because they make decisions that don't consider the consequences of their choices. Chapter 15 shows that you can't always solve a problem using a greedy algorithm. Therefore, an algorithm may make an apparently optimal decision at a certain stage, which later appears limiting and suboptimal for achieving the best solution. A better algorithm, one that doesn't rely on the greedy approach, can revise past decisions or anticipate that an apparently good decision is not as promising as it might seem. This is the approach that dynamic programming takes.

*Dynamic programming* is an algorithm approach devised in the 1950s by Richard Ernest Bellman (an applied mathematician also known for other discoveries in the field of mathematics and algorithms, you can read more at `https://en.wikipedia.org/wiki/Richard_E._Bellman`) that tests more solutions than a corresponding greedy approach. Testing more solutions provides the ability to rethink and ponder the consequences of decisions. Dynamic programming avoids performing heavy computations thanks to a smart caching system (a *cache* is a storage system collecting data or information) called *memoization*, a term defined later in the chapter.

This chapter offers you more than a simple definition of dynamic programming. It also explains why dynamic programming has such a complicated name and how to transform any algorithm (especially recursive ones) into dynamic programming using Python and its function decorators (powerful tools in Python that allow changing an existing function without rewriting its code). In addition, you discover applications of dynamic programming to optimize resources and returns, creating short tours between places and comparing strings in an approximate way. Dynamic programming provides a natural approach to dealing with many problems you encounter while journeying through the world of algorithms.

# Explaining Dynamic Programming

Dynamic programming is as effective as an exhaustive algorithm (thus providing correct solutions), yet it is often as efficient as an approximate solution (the computational time of many dynamic programming algorithms is polynomial). It seems to work like magic because the solution you need often requires the algorithm to perform the same calculations many times. By modifying the algorithm and making it dynamic, you can record the computation results and reuse them when needed. Reusing takes little time when compared to recalculating, thus the algorithm finishes the steps quickly. The following sections discuss what dynamic programming involves in more detail.

## Obtaining a historical basis

You can boil *dynamic programming* down to having an algorithm remember the previous problem results where you'd otherwise have to perform the same calculation repeatedly. Even though dynamic programming might appear to be quite complex, the implementation is actually straightforward. However, it does have some interesting historical origins.

Bellman described the name as the result of necessity and convenience in his autobiography, *In the Eye of the Hurricane.* He writes that the name choice was a way to hide the true nature of his research at the RAND Corporation (a research and development institution funded by both the U.S. government and private financers) from Charles Erwin Wilson, the Secretary of Defense under the Eisenhower presidency. Cloaking the true nature of his research helped Bellman remain employed at the RAND Corporation. You can read his explanation in more detail in the excerpt at: `http://smo.sogang.ac.kr/doc/dy_birth.pdf`. Some researchers don't agree about the name source. For example, Stuart Russell and Peter Norvig, in their book *Artificial Intelligence: A Modern Approach*, argue that Bellman actually used the term *dynamic programming* in a paper dating 1952, which is before Wilson became Secretary in 1953 (and Wilson himself was CEO of General Motors before becoming an engineer involved in research and development).

Computer programming languages weren't widespread during the time that Bellman worked in operations research, a discipline that applies mathematics to make better decisions when approaching mainly production or logistic problems (but is also used for other practical problems). Computing was at the early stages and used mostly for planning. The basic approach of dynamic programming is the same as *linear programming*, another algorithmic technique (see Chapter 19) defined in those years when programming meant planning a specific process to find an optimal solution. The term *dynamic* reminds you that the algorithm moves and stores partial solutions. Dynamic programming is a complex name for a smart and effective technique to improve algorithm running times.

## Making problems dynamic

Because dynamic programming takes advantage of repeated operations, it operates well on problems that have solutions built around solving subproblems that the algorithm later assembles to provide a complete answer. In order to work effectively, a dynamic programming approach uses subproblems nested in other subproblems. (This approach is akin to greedy algorithms, which also require an optimal substructure, as explained in Chapter 15.) Only when you can break down a problem into nested subproblems can dynamic programming beat brute-force approaches that repeatedly rework the same subproblems.

REMEMBER

As a concept, dynamic programming is a huge umbrella covering many different applications because it isn't really a specific algorithm for solving a specific problem. Rather, it's a general technique that supports problem solving.

You can trace dynamic programming to two large families of solutions:

- **Bottom-up:** Builds an array of partial results that aggregate into a complete solution
- **Top-down:** Splits the problem into subproblems, starting from the complete solution (this approach is typical of recursive algorithms) and using memoization (defined in the next section) to avoid repeating computations more than once

Typically, the top-down approach is more computationally efficient because it generates only the subproblems necessary for the complete solution. The bottom-up approach is more explorative and, using trial and error, often obtains partial results that you won't use later. On the other hand, bottom-up approaches better reflect the approach that you'd take in everyday life when facing a problem (thinking recursively, instead, needs abstraction and training before application). Both top-down and bottom-up approaches aren't all that easy to understand at times. That's because using dynamic programming transforms the way you solve problems, as detailed in these steps:

1. Create a working solution using brute-force or recursion. The solution works but it takes a long time or won't finish at all.
2. Store the results of subproblems to speed your computations and reach a solution in a reasonable time.
3. Change the way you approach the problem and gain even more speed.
4. Redefine the problem approach, in a less intuitive but more efficient way to obtain the greatest advantage from dynamic programming.

Transforming algorithms using dynamic programming to make them work efficiently makes them harder to understand. In fact, you might look at the solutions and think they work by magic. Becoming proficient in dynamic programming requires repeated observations of existing solutions and some practical exercise. This proficiency is worth the effort, however, because dynamic programming can help you solve problems for which you have to systematically compare or compute solutions.

TIP

Dynamic programming is especially known for helping solve (or at least make less time demanding) combinatorial optimization problems, which are problems that require obtaining combinations of input elements as a solution. Examples of such problems solved by dynamic programming are the traveling salesman and the knapsack problems, described later in the chapter.

## Casting recursion dynamically

The basis of dynamic programming is to achieve something as effective as brute-force searching without actually spending all the time doing the computations required by a brute-force approach. You achieve the result by trading time for disk space or memory, which is usually done by creating a data structure (a hash table, an array, or a data matrix) to store previous results. Using lookup tables allows you to access results without having to perform a calculation a second time.

REMEMBER

The technique of storing previous function results and using them instead of the function is *memoization*, a term you shouldn't confuse with memorization. Memoization derives from *memorandum*, the Latin word for "to be remembered."

REMEMBER

Caching is another term that you find used when talking about memoization. *Caching* refers to using a special area of computer memory to serve data faster when required, which has more general uses than memoization.

To be effective, dynamic programming needs problems that repeat or retrace previous steps. A good example of a similar situation is using recursion, and the landmark of recursion is calculating Fibonacci numbers. The Fibonacci sequence is simply a sequence of numbers in which the next number is the sum of the previous two. The sequence starts with 0 followed by 1. After defining the first two elements, every following number in the sequence is the sum of the previous ones. Here are the first eleven numbers:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

As with indexing in Python, counting starts from the zero position, and the last number in the sequence is the tenth position. The inventor of the sequence, the Italian mathematician Leonardo Pisano, known as Fibonacci, lived in 1200. Fibonacci thought that the fact that each number is the sum of the previous two should have made the numbers suitable for representing the growth patterns of a group of rabbits. The sequence didn't work great for rabbit demographics, but it offered unexpected insights into both mathematics and nature itself because the numbers appear in botany and zoology. For instance, you see this progression in the branching of trees, in the arrangements of leaves in a stem, and of seeds in a sunflower (you can read about this arrangement at `https://www.goldennumber.net/spirals/`).

REMEMBER

Fibonacci was also the mathematician who introduced Hindu-Arabic numerals to Europe, the system we daily use today. He described both the numbers and the sequence in his masterpiece, the *Liber Abaci,* in 1202.

You can calculate a Fibonacci number sequence using recursion. When you input a number, the recursion splits the number into the sum of the previous two Fibonacci numbers in the sequence. After the first split, the recursion proceeds by performing the same task for each element of the split, splitting each of the two numbers into the previous two Fibonacci numbers. The recursion continues splitting numbers into their sums, until it finally finds the roots of the sequence, the numbers 0 and 1. Reviewing the two types of dynamic programming algorithm described in the previous paragraph, this solution uses a top-down approach. The following code shows the recursive approach in Python. (You can find this code in the `A4D; 16; Fibonacci.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
def fib(n, tab=0):
    if n==0:
        return 0
    elif n == 1:
        return 1
    else:
        print ("lvl %i, Summing fib(%i) and fib(%i)" %
               (tab, n-1, n-2))
        return fib(n-1,tab+1) + fib(n-2,tab+1)
```

The code prints the splits generated by each recursion level. The following output shows what happens when you call `fib()` with an input value of `7`:

```
fib(7)

lvl 0, Summing fib(6) and fib(5)
lvl 1, Summing fib(5) and fib(4)
lvl 2, Summing fib(4) and fib(3)
lvl 3, Summing fib(3) and fib(2)
lvl 4, Summing fib(2) and fib(1)
lvl 5, Summing fib(1) and fib(0)
lvl 4, Summing fib(1) and fib(0)
lvl 3, Summing fib(2) and fib(1)
lvl 4, Summing fib(1) and fib(0)
lvl 2, Summing fib(3) and fib(2)
lvl 3, Summing fib(2) and fib(1)
lvl 4, Summing fib(1) and fib(0)
lvl 3, Summing fib(1) and fib(0)
lvl 1, Summing fib(4) and fib(3)
lvl 2, Summing fib(3) and fib(2)
lvl 3, Summing fib(2) and fib(1)
lvl 4, Summing fib(1) and fib(0)
lvl 3, Summing fib(1) and fib(0)
lvl 2, Summing fib(2) and fib(1)
lvl 3, Summing fib(1) and fib(0)

13
```

The output shows 20 splits. Some numbers appear more than once as part of the splits. It seems like an ideal case for applying dynamic programming. The following code adds a dictionary, called `memo`, which stores previous results. After the recursion splits a number, it checks whether the result already appears in the dictionary before starting the next recursive branch. If it finds the result, the code uses the precomputed result, as shown here:

```
memo = dict()
def fib_mem(n, tab=0):
    if n==0:
        return 0
    elif n == 1:
        return 1
    else:
        if (n-1, n-2) not in memo:
            print ("lvl %i, Summing fib(%i) and fib(%i)" %
                   (tab, n-1, n-2))
            memo[(n-1,n-2)] = fib_mem(n-1,tab+1
                          ) + fib_mem(n-2,tab+1)
        return memo[(n-1,n-2)]
```

Using memoization, the recursive function doesn't compute 20 additions but rather uses just six, the essential ones used as building blocks to solve the initial requirement for computing a certain number in the sequence:

```
fib_mem(7)

lvl 0, Summing fib(6) and fib(5)
lvl 1, Summing fib(5) and fib(4)
lvl 2, Summing fib(4) and fib(3)
lvl 3, Summing fib(3) and fib(2)
lvl 4, Summing fib(2) and fib(1)
lvl 5, Summing fib(1) and fib(0)

13
```

Looking inside the `memo` dictionary, you can find the sequence of sums that define the Fibonacci sequence starting from `1`:

```
memo
{(1, 0): 1, (2, 1): 2, (3, 2): 3, (4, 3): 5, (5, 4): 8,
 (6, 5): 13}
```

## Leveraging memoization

Memoization is the essence of dynamic programming. You often find the need to use it when scripting an algorithm yourself. When creating a function, whether recursive or not, you can easily transform it using a simple command, a *decorator*, which is a special Python function that transforms functions. To see how to work with a decorator, start with a recursive function, stripped of any print statement:

```
def fib(n):
    if n==0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

When using Jupyter, you use IPython built-in magic commands, such as `timeit`, to measure the execution time of a command on your computer:

```
%timeit -n 1 -r 1 print(fib(36))

14930352
1 loop, best of 1: 15.5 s per loop
```

The output shows that the function requires about 15 seconds to execute. However, depending on your machine, function execution may require more or less time. No matter the speed of your computer, it will certainly take a few seconds to complete, because the Fibonacci number for 36 is quite huge: 14930352. Testing the same function for higher Fibonacci numbers takes even longer.

Now it's time to see the effect of decorating the function. Using the `lru_cache` function from the `functools` package can radically reduce execution time. This function is available only when using Python3. It transforms a function by automatically adding a cache to hold its results. You can also set the cache size by using the `maxsize` parameter (`lru_cache` uses a cache with an optimal replacement strategy, as explained in the Chapter 15). If you set `maxsize=None`, the cache uses all the available memory, without limits.

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fib(n):
    if n==0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Note that the function is the same as before. The only addition is the imported `lru_cache` (`https://docs.python.org/3.5/library/functools.html`), which you call by putting an @ symbol in front of it. Any call with the @ symbol in front is an *annotation* and calls the `lru_cache` function as a decorator of the following function.

TECHNICAL STUFF

Using decorators is an advanced technique in Python. Decorators don't need to be explained in detail in this book, but you can still take advantage of them because they are so easy to use. (You can find additional information about decorators at `http://simeonfranklin.com/blog/2012/jul/1/python-decorators-in-12-steps/` and `https://www.learnpython.org/en/Decorators`.) Just remember that you call them using annotations (@ + *decorator function's name*) and that you put them in front of the function you want to transform. The original function is fed into the decorator and comes out transformed. In this example of a simple recursive function, the decorator outputs a recursion function enriched by memoization.

It's time to test the function speed, as before:

```
%timeit -n 1 -r 1 print(fib(36))

14930352
1 loop, best of 1: 60.6 µs per loop
```

Even if your execution time is different, it should decrease from seconds to milliseconds. Such is the power of memoization. You can also explore how your function uses its cache by calling the `cache_info` method from the decorated function:

```
fib.cache_info()

CacheInfo(hits=34, misses=37, maxsize=None, currsize=37)
```

The output tells you that there are 37 function calls that don't find an answer in the cache. However, 34 other calls did find a useful answer in the cache.

TIP

Just by importing `lru_cache` from `functools` and using it in annotations in front of your heavy-duty algorithms in Python, you will experience a great increase in performance (unless they are greedy algorithms).

# Discovering the Best Dynamic Recipes

Even dynamic programming has limitations. The biggest limitation of all relates to its main strength: If you keep track of too many partial solutions to improve running time, you may run out of memory. You may have too many partial solutions in store because the problem is complex, or simply because the order you use to produce partial solutions is not an optimal one and too many of the solutions don't fit the problem requirements.

The order used to solve subproblems is something you must track. The order you choose should make sense for the efficient progression of the algorithm (you solve something that you're going to reuse immediately) because the trick is in smart reuse of previously built building blocks. Therefore, using memoization may not provide enough benefit. Rearranging your problems in the right order can improve the results. You can learn how to correctly order your subproblems by learning directly from the best dynamic programming recipes available: knapsack, traveling salesman, and approximate string search, as described in the sections that follow.

## Looking inside the knapsack

The knapsack problem has been around since at least 1897 and is likely the work of Tobias Dantzig (`https://www.britannica.com/biography/Tobias-Dantzig`). In this case, you have to pack up your knapsack with as many items as possible. Each item has a value, so you want maximize the total value of the items you carry. The knapsack has a threshold capacity or you have a limit of weight you can carry, so you can't carry all the items.

The general situation fits any problem that involves a budget and resources, and you want to allocate them in the smartest way possible. This problem setting is so common that many people consider the knapsack problem to be one of the most popular algorithmic problems. The knapsack problem finds applications in computer science, manufacturing, finance, logistics, and cryptography. For instance, real-world applications of the knapsack problem are how to best load a cargo ship with goods or how to optimally cut raw materials, thus creating the least waste possible.

REMEMBER

Even though it's such a popular problem, this book doesn't explore the knapsack problem again because the dynamic approach is incontestably one of the best solving approaches. It's important to remember, though, that in specific cases, for such as when the items are quantities, other approaches, such as using greedy algorithms, may work equally well (or even better).

This section shows how to solve the *1-0 knapsack problem*. In this case, you have a finite number of items and can put each of them into the knapsack (the one status) or not (the zero status). It's useful to know there are other possible variants of the problem:

- **Fractional knapsack problem:** Deals with quantities. For example, an item could be kilograms of flour, and you must pick the best quantity. You can solve this version using a greedy algorithm.
- **Bounded knapsack problem:** Puts one or more copies of the same item into the knapsack. In this case, you must deal with minimum and maximum number requirements for each item you pick.
- **Unbounded knapsack problem:** Puts one or more copies of the same item into the knapsack without constraints. The only limit is that you can't put a negative number of items into the knapsack.

The 1-0 knapsack problem relies on a dynamic programming solution and runs in pseudo-polynomial time (which is worse than just polynomial time) because the running time depends on the number of items (n) multiplied by the number of fractions of the knapsack capacity (W) that you use on building your partial solution. When using big-O notation, you can say that the running time is `O(nW)`. The brute-force version of the algorithm instead runs in $O(2^n)$. The algorithm works like this:

1. Given the knapsack capacity, test a range of smaller knapsacks (subproblems). In this case, given a knapsack capable of carrying 20 kilograms, the algorithm tests a range of knapsacks carrying from 0 kilograms to 20 kilograms.
2. For each item, test how it fits in each of the knapsacks, from the smallest knapsack to the largest. At each test, if the item can fit, choose the best value from the following:
   a. The solution offered by the previous smaller knapsack
   b. The test item, plus you fill the residual space with the best valued solution previously that filled that space

The code runs the knapsack algorithm and solves the problem with a set of six items of different weight and value combinations as well as a 20-kg knapsack:

| Item | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Weight in kg | 2 | 3 | 4 | 4 | 5 | 9 |
| Profit in 100 USD | 3 | 4 | 3 | 5 | 8 | 10 |

Here is the code to execute the dynamic programming procedure described. (You can find this code in the `A4D; 16; Knapsack.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
import numpy as np

values = np.array([3,4,3,5,8,10])
weights = np.array([2,3,4,4,5,9])
items = len(weights)
capacity = 20

memo = dict()
for size in range(0, capacity+1, 1):
    memo[(-1, size)] = ([], 0)

for item in range(items):
    for size in range(0, capacity+1, 1):
        # if the object doesn't fit in the knapsack
        if weights[item] > size:
            memo[item, size] = memo[item-1, size]
        else:
        # if the objcts fits, we check what can best fit
        # in the residual space
            previous_row, previous_row_value = memo[
                        item-1, size-weights[item]]
            if memo[item-1, size][1] > values[item
                            ] + previous_row_value:
                memo[item, size] = memo[item-1, size]
            else:
                memo[item, size] = (previous_row + [item
                    ], previous_row_value + values[item])
```

The best solution is the cached result when the code tests inserting the last item with the full capacity (20 kg) knapsack:

```
best_set, score = memo[items-1, capacity]
print ('The best set %s weights %i and values %i'
       % (best_set, np.sum((weights[best_set])), score))

The best set [0, 3, 4, 5] weights 20 and values 26
```

You may be curious to know what happened inside the memoization dictionary:

```
print (len(memo))

147

print (memo[2, 10])

([0, 1, 2], 10)
```

It contains 147 subproblems. In fact, six items multiplied by 21 knapsacks is 126 solutions, but you have to add another 21 naive solutions to allow the algorithm to work properly (*naive* means leaving the knapsack empty), which increases the number of subproblems to 147.

You may find solving 147 subproblems daunting (even though they're blazingly fast to solve). Using brute force alone to solve the problem means solving fewer subproblems in this particular case. Solving fewer subproblems requires less time, a fact you can test by solving the accounts using Python and the `comb` function:

```
from scipy.misc import comb
objects = 6
np.sum([comb(objects,k+1) for k in range(objects)])
```

It takes testing 63 combinations to solve this problem. However, if you try using more objects, say, 20, running times look much different because there are now 1,048,575 combinations to test. Contrast this huge number with dynamic programming, which requires solving just 20*21+21 = 441 subproblems.

REMEMBER

This is the difference between quasi-polynomial and exponential time. (As a reminder, the book discusses exponential complexity in Chapter 2 when illustrating the Big O Notation. In Chapter 15, you discover polynomial time as part of the discussion about NP complete problems.) Using dynamic programming becomes fruitful when your problems are complex. Toy problems are good for learning but they can't demonstrate the full extent of employing smart algorithm techniques such as dynamic programming. Each solution tests what happens after adding a certain item when the knapsack has a certain size. The preceding example adds item 2 (weight=4, value=3) and outputs a solution that puts items 0, 1, and 2 into the knapsack (total weight 9 kg) for a value of 10. This intermediate solution leverages previous solutions and is the basis for many of the following solutions before the algorithm reaches its end.

TIP

You may wonder whether the result offered by the script is really the best one achievable. Unfortunately, the only way to be sure is to know the right answer, which means running a brute-force algorithm (when feasible in terms of running time on your computer). This chapter doesn't use brute force for the knapsack problem but you'll see a brute-force approach used in the traveling salesman problem that follows.

## Touring around cities

The traveling salesman problem (TSP for short) is at least as widely known as the knapsack problem. You use it mostly in logistics and transportation (such as the derivative Vehicle Routing Problem shown at `http://neo.lcc.uma.es/vrp/vehicle-routing-problem/`), so it doesn't see as much use as the knapsack problem. The TSP problem asks a traveling salesperson to visit a certain number of cities and then come back to the initial starting city (because it's circular, it's called a tour) using the shortest path possible.

TSP is similar to graph problems, but without the edges because the cities are all interconnected. For this reason, TSP usually relies on a distance matrix as input, which is a table listing the cities on both rows and columns. The intersections contain the distance from a row city to a column city. TSP problem variants may provide a matrix containing time or fuel consumption instead of distances.

TSP is an NP-hard problem, but you can solve the problem using various approaches, some approximate (heuristic) and some exact (dynamic programming). The problem, as with any other NP-hard problem, is the running time. Although you can count on solutions that you presume optimally solve the problem (you can't be certain except when solving short tours), you can't know for sure with problems as complex as touring the world: `http://www.math.uwaterloo.ca/tsp/world/`. The following example tests various algorithms, such as brute force, greedy, and dynamic programming, on a simple tour of six cities, represented as a weighted graph (see Figure 16-1). (You can find this code in the `A4D; 16; TSP.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

D = np.array([[0,20,16,25,24],[20,0,12,12,27],
              [16,12,0,10,14],[25,12,10,0,20],
              [24,27,14,20,0]])
```

```
Graph = nx.Graph()
Graph.add_nodes_from(range(D.shape[0]))
for i in range(D.shape[0]):
    for j in range(D.shape[0]):
        Graph.add_edge(i,j,weight=D[i,j])

np.random.seed(2)
pos=nx.shell_layout(Graph)
nx.draw(Graph, pos, with_labels=True)
labels = nx.get_edge_attributes(Graph,'weight')
nx.draw_networkx_edge_labels(Graph,pos,
                             edge_labels=labels)
plt.show()
```

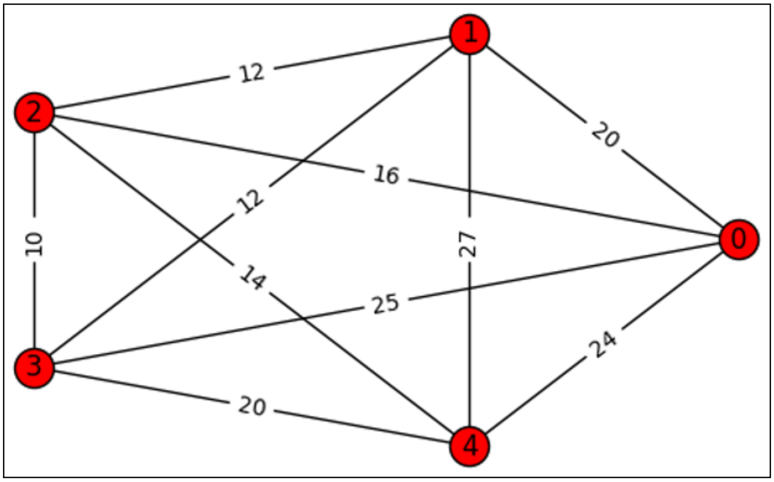


**FIGURE 16-1:** Cities represented as nodes in a weighted graph.

After defining the D (distance) matrix, the example tests the first, simplest solution to determine the shortest tour starting and ending from city zero. This solution relies on brute force, which generates all the possible order permutations between the cities, leaving out zero. The distance from zero to the first city and from the last city of the tour to zero is added after the total distance of each solution is calculated. When all the solutions are available, you simply choose the shortest.

```
from itertools import permutations
best_solution = [None, np.sum(D)]
for solution in list(permutations(range(1,D.shape[0]))):
    start, distance = (0,0)
    for next_one in solution:
        distance += D[start, next_one]
        start = next_one
    distance += D[start,0]
    if distance <= best_solution[1]:
        best_solution = [[0]+list(solution)+[0], distance]
        print ('Best solution so far: %s kms' %
               str(best_solution)[1:-1])

Best solution so far: [0, 1, 2, 3, 4, 0], 86 kms
Best solution so far: [0, 1, 3, 2, 4, 0], 80 kms
Best solution so far: [0, 4, 2, 3, 1, 0], 80 kms
```

The brute-force algorithm quickly determines the best solution and its symmetric path. However, as a result of the small problem size, you obtain a prompt answer because, given four cities, just 24 possible solutions exist. As the number of cities increases, the number of permutations to test becomes intractable, even after removing the symmetric paths (which halves the permutations) and using a fast computer. For example, consider the number of computations when working with 13 cities plus the starting/ending point:

```
from scipy.special import perm
print (perm(13,13)/2)

3113510400.0
```

Dynamic programming can simplify the running time. The Held–Karp algorithm (also known as the Bellman–Held–Karp algorithm because Bellman published it in 1962, the same year as Michael Held and Richard Karp) can cut time complexity to $O(2^n n^2)$. It's still exponential complexity, yet it requires less time than applying the exhaustive enumeration of all tours by brute force.

TIP

Approximate and heuristic algorithms can provide fast and useful results (even though the result may not always reflect the optimal solution, it's usually good enough). You see TSP again later in the book (see Chapters 18 and 20), when dealing with local search and heuristics.

To find the best TSP solution for *n* cities, starting and ending from city 0, the algorithm proceeds from city 0 and keeps records of the shortest path possible, considering different settings. It always uses a different ending city and touches only a city subset. As the subsets become larger, the algorithm learns how to solve the problem efficiently. Therefore, when solving TSP for five cities, the algorithm first considers solutions involving two cities, then three cities, then four, and finally five (sets have dimensions 1 to *n*). Here are the steps the algorithm uses:

1. Initialize a table to track the distances from city 0 to all other cities. These sets contain only the initial city and a destination city because they represent the initial step.
2. Consider every possible set size, from two to the number of tour cities. This is a first iteration, the outer loop.
3. Inside the outer loop, for each set size, consider every possible combination of cities of that size, not containing the initial city. This is an inner iteration.
4. Inside the inner iteration (Step 3), for every available combination, consider each city inside the combination as the ending city. This is another inner iteration.
5. Inside the inner iteration (Step 4), given a different destination city, determine the shortest path connecting the cities in the set from the city that starts the tour (city 0). In finding the shortest path, use any useful, previously stored information, thus applying dynamic programming. This step saves computations and provides the rationale for working by growing subsets of cities. Reusing previously solved subproblems, you find the shorter tours by adding to a previous shortest path the distance necessary to reach the destination city. Given a certain set of cities, a specific initial city, and a specific destination city, the algorithm stores the best path and its length.
6. When all the iterations end, you have as many different shortest solutions as `n–1` cities, with each solution covering all the cities but ending at a different city. Add a closing point, the city 0, to each one to conclude the tour.
7. Determine the shortest solution and output it as the result.

TECHNICAL STUFF

The Python implementation of this algorithm isn't very simple because it involves some iterations and manipulating sets. It's an exhaustive search reinforced by dynamic programming and relies on an iterative approach with subsets of cities and with candidates to add to them. The following commented Python example explores how this solution works. You can use it to calculate customized tours (possibly using cities in your region or county as entries in the distance matrix). The script uses advanced commands such as `frozenset` (a command that makes a set usable as a dictionary key) and operators for sets in order to achieve the solution.

```
from itertools import combinations

memo = {(frozenset([0, idx+1]), idx+1): (dist, [0,idx+1])
```

```
        for idx,dist in enumerate(D[0][1:])}
cities = D.shape[0]
for subset_size in range(2, cities):
    # Here we define the size of the subset of cities
    new_memo = dict()
    for subset in [frozenset(comb) | {0} for comb in
                   combinations(range(1, cities),
                                subset_size)]:
        # We enumerate the subsets having a certain subset
        # size
        for ending in subset - {0}:
            # We consider every ending point in the subset
            all_paths = list()
            for k in subset:
                # We check the shortest path for every
                # element in the subset
                if k != 0 and k!=ending:
                    length = memo[(subset-{ending},k)][0
                                        ] + D[k][ending]
                    index  = memo[(subset-{ending},k)][1
                                            ] + [ending]
                    all_paths.append((length, index))
            new_memo[(subset, ending)] = min(all_paths)
    # In order to save memory, we just record the previous
    # subsets since we won't use shorter ones anymore
    memo = new_memo
# Now we close the cycle and get back to the start of the
# tour, city zero
tours = list()
for distance, path in memo.values():
    distance += D[path[-1],0]
    tours.append((distance, path+[0]))
# We can now declare the shortest tour
distance, path = min(tours)
print ('Shortest dynamic programming tour is: %s, %i kms'
       % (path, distance))
Shortest dynamic programming tour is:
 [0, 1, 3, 2, 4, 0], 80 kms
```

## Approximating string search

Determining when one word is similar to another isn't always simple. Words may differ slightly because of misspelling or different ways of writing the word itself, thus rendering any exact match impossible. This isn't just a problem that raises interesting issues during a spell check, though. For example, putting similar text strings together (such as names, addresses, or code identifiers) that refer to the same person may help create a one-customer view of a firm's customer base or help a national security agency locate a dangerous criminal.

REMEMBER

Approximating string searches has many applications in machine translation, speech recognition, spell checking and text processing, computational biology, and information retrieval. Thinking about the manner in which sources input data into databases, you know there are many mismatches between data fields that a smart algorithm must solve. Matching a similar, but not precisely equal, series of letters is an ability that finds uses in fields such as genetics when comparing DNA sequences (expressed by letters representing nucleotides G,A,T, and C ) to determine whether two sequences are similar and how they resemble each other.

Vladimir Levenshtein, a Russian scientist expert in information theory (see `http://ethw.org/Vladimir_I._Levenshtein` for details), devised a simple measure (named after him) in 1965 that computes the grade of similarity between two strings by counting how many transformations it takes to change the first string into the second. The Levenshtein distance (also known as edit distance) counts how many changes are necessary in a word:

- **Deletion:** Removing a letter from a word
- **Insertion:** Inserting a letter into a word and obtaining another word
- **Substitution:** Replacing one letter with another, such as changing the *p* letter into an *f* letter and obtaining *fan* from *pan*

TIP

Each edit has a cost, which Levenshtein defines as 1 for each transformation. However, depending on how you apply the algorithm, you could set the cost differently for deletion, insertion, and substitution. For example, when searching for similar street names, misspellings are more common than outright differences in lettering, so substitution might incur only a cost of 1, and deletion or insertion might incur a cost of 2. On the other hand, when looking for monetary amounts, similar values quite possibly will have different numbers of numbers. Someone could enter $123 or $123.00 into the database. The numbers are the same, but the number of numbers is different, so insertion and deletion might cost less than substitution (a value of $124 is not quite the same as a value of $123, so substituting 3 for 4 should cost more).

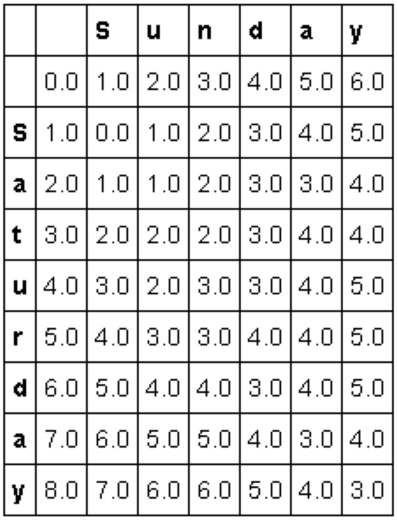
You can render the counting algorithm as a recursion or an iteration. However, it works much faster using a bottom-up dynamic programming solution, as described in the 1974 paper "The String-to-string Correction Problem," by Robert A. Wagner and Michael J. Fischer (`http://www.inrg.csie.ntu.edu.tw/algorithm2014/homework/Wagner-74.pdf`). The time complexity of this solution is `O(mn)`, where *n* and *m* are the lengths in letter of the two words being compared. The following code computes the number of changes required to turn the word *Saturday* into *Sunday* by using dynamic programming with a matrix (see Figure 16-2) to store previous results (the bottom-up approach). (You can find this code in the `A4D; 16; Levenshtein.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
import numpy as np
import pandas as pd

s1 = 'Saturday'
s2 = 'Sunday'
m = len(s1)
n = len(s2)
D = np.zeros((m+1,n+1))
D[0,:] =  list(range(n+1))
D[:,0] = list(range(m+1))

for j in range(1, n+1):
    for i in range(1, m+1):
        if s1[i-1] == s2[j-1]:
            D[i, j] = D[i-1, j-1]
        else:
                D[i, j] = np.min([
                D[i-1, j]   + 1,  # a deletion
                D[i, j-1]   + 1,  # an insertion
                D[i-1, j-1] + 1   # a substitution
                ])
print ('Levenshtein distance is %i' % D[-1,-1])

Levenshtein distance is 3
```

You can plot or print the result using the following command:

```
pd.DataFrame(D,index=list(' '+s1), columns=list(' '+s2))
```

The algorithm builds the matrix, placing the best solution in the last cell. After building the matrix using the letters of the first string as rows and the letters of the second one as columns, it proceeds by columns, computing the differences between each letter in the rows compared to those in the columns. In this way, the algorithm makes a number of comparisons equivalent to the multiplication of the number of the letters in the two strings. As the algorithm continues, it accounts for the result of previous comparisons by looking at the solutions present in the previous matrix cells and choosing the solution with the least number of edits.



|  |  | S | u | n | d | a | y |
|---|---|---|---|---|---|---|---|
|  | 0.0 | 1.0 | 2.0 | 3.0 | 4.0 | 5.0 | 6.0 |
| S | 1.0 | 0.0 | 1.0 | 2.0 | 3.0 | 4.0 | 5.0 |
| a | 2.0 | 1.0 | 1.0 | 2.0 | 3.0 | 3.0 | 4.0 |
| t | 3.0 | 2.0 | 2.0 | 2.0 | 3.0 | 4.0 | 4.0 |
| u | 4.0 | 3.0 | 2.0 | 3.0 | 3.0 | 4.0 | 5.0 |
| r | 5.0 | 4.0 | 3.0 | 3.0 | 4.0 | 4.0 | 5.0 |
| d | 6.0 | 5.0 | 4.0 | 4.0 | 3.0 | 4.0 | 5.0 |
| a | 7.0 | 6.0 | 5.0 | 5.0 | 4.0 | 3.0 | 4.0 |
| y | 8.0 | 7.0 | 6.0 | 6.0 | 5.0 | 4.0 | 3.0 |

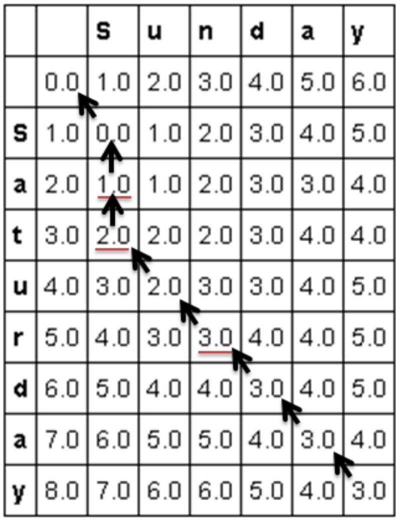
**FIGURE 16-2:** Transforming Sunday into Saturday.

When the matrix iteration completes, the resulting number represents the minimum number of edits necessary for the transformation to occur — the smaller the number, the more similar the two strings. Retracing from the last cell to the first one by moving to the previous cell with the least value (if more directions are available, it prefers to move diagonally) hints at what transformations to execute (see Figure 16-3):

- A diagonal backward movement hints at a substitution in the first string if the letters on the row and column differ (otherwise, no edit needs to be done)
- An upward movement dictates a deletion of a letter in the first string
- A left backward move indicates that an insertion of a new letter should be done on the first string

In this example, the backtracking points out the following transformations (two deletions and one substitution):

```
Saturday => Sturday => Surday => Sunday
```



| | | S | u | n | d | a | y |
|---|---|---|---|---|---|---|---|
| | 0.0 | 1.0 | 2.0 | 3.0 | 4.0 | 5.0 | 6.0 |
| S | 1.0 | 0.0 | 1.0 | 2.0 | 3.0 | 4.0 | 5.0 |
| a | 2.0 | 1.0 | 1.0 | 2.0 | 3.0 | 3.0 | 4.0 |
| t | 3.0 | 2.0 | 2.0 | 2.0 | 3.0 | 4.0 | 4.0 |
| u | 4.0 | 3.0 | 2.0 | 3.0 | 3.0 | 4.0 | 5.0 |
| r | 5.0 | 4.0 | 3.0 | 3.0 | 4.0 | 4.0 | 5.0 |
| d | 6.0 | 5.0 | 4.0 | 4.0 | 3.0 | 4.0 | 5.0 |
| a | 7.0 | 6.0 | 5.0 | 5.0 | 4.0 | 3.0 | 4.0 |
| y | 8.0 | 7.0 | 6.0 | 6.0 | 5.0 | 4.0 | 3.0 |

**FIGURE 16-3:** Highlighting what transformations are applied.

IN THIS CHAPTER

- » **Understanding how randomness can prove smarter than more reasoned ways**
- » **Introducing key ideas about probability and its distributions**
- » **Discovering how a Monte Carlo simulation works**
- » **Learning about Quickselect and revisiting Quicksort algorithms**

# Chapter 17

# Using Randomized Algorithms

Random number generators are a key function in computing and play an important role in the algorithmic techniques discussed in this part of the book. Randomization isn't just for gaming or for gambling, but people employ it to solve a large variety of problems. Randomization sometimes proves more effective during optimization than other techniques and in obtaining the right solution than more reasoned ways. It helps different techniques work better, from local search and simulated annealing to heuristics, cryptography, and distributed computing (with cryptography for concealing information being the most critical).

You can find randomization embedded into unexpected everyday tools. The robot vacuum cleaner Roomba (designed by a company founded by the Massachusetts Institute of Technology [MIT]) cleans rooms without having a precise plan and a blueprint of the place. The tool works most of the time by wandering randomly around the room and, according to the original patent, after hitting an obstacle, it turns a random number of degrees and starts in a new direction. Yet Roomba always completes its cleaning chores. (If you are curious about how it operates, you can consult `http://www.explainthatstuff.com/how-roomba-works.html`.)

From a historical perspective, randomized algorithms are a recent innovation, because the first algorithm of this kind, the closest-pair algorithm (which determines the pair of points, among many on a geometric plane, with the smallest distance between the points without having to compare them all) was developed by Michael Rabin in 1976. That first algorithm was followed the next year by the randomized primality test (an algorithm for determining whether a number is a composite or a probable prime number), by Robert M. Solovay and Volker Strassen. Soon after, applications in cryptography and distributed computing made randomization more popular and the subject of intense research, although the field is still new and uncharted.

Randomization makes finding a solution simpler, trading time against complexity. Simplifying tasks isn't its only advantage: Randomization saves resources and operates in a distributed way with a reduced need for communication and coordination. This chapter introduces you to the information needed to understand how enriching your algorithms with randomness can help solve problems (the chapter uses the term *injecting randomness,* as if it were a cure). Even more applications wait in the following chapters, so this chapter also discusses key topics such as probability basics, probability distributions, and Monte Carlo simulations.

# Defining How Randomization Works

Randomization relies on the capability by your computer to generate random numbers, which means creating the number without a plan. Therefore, a random number is unpredictable, and as you generate subsequent random numbers, they shouldn't relate to each other.

REMEMBER

However, randomness is hard to achieve. Even when you throw dice, the result can't be completely unexpected because of the way you hold the dice, the way you throw them, and the fact that the dice aren't perfectly shaped. Computers aren't good at creating random numbers, either. They generate randomness by using algorithms or pseudorandom tables (which work by using a *seed* value as a starting point, a number equivalent to an index) because a computer can't create a truly random number. Computers are deterministic machines; everything inside them responds to a well-defined response pattern, which means that it imitates randomness in some way.

## Considering why randomization is needed

Even if a computer can't create true randomness, streams of pseudorandom numbers (numbers that appear as random but that are somehow predetermined) can still make the difference in many computer science problems. Any algorithm that employs randomness in its logic can appear as a randomized algorithm, no matter whether randomness determines its results, improves performance, or mitigates the risk of failing by providing a solution in certain cases.

Usually you find randomness employed in selecting input data, the start point of the optimization, or the number and kind of operations to apply to the data. When randomness is a core part of the algorithm logic and not just an aid to its performance, the expected running time of the algorithm and even its results may become uncertain and subject to randomness, too; for instance, an algorithm may provide different, though equally good, results during each run. It's therefore useful to distinguish between kinds of randomized solutions, each one named after iconic gambling locations:

- **Las Vegas:** These algorithms are notable for using random inputs or resources to provide the correct problem answer every time. Obtaining a result may take an uncertain amount of time because of its random procedures. An example is the Quicksort algorithm.
- **Monte Carlo:** Because of their use of randomness, Monte Carlo algorithms may not provide a correct answer or even an answer at all, although these outcomes seldom happen. Because the result is uncertain, a maximum number of trials in their running time may bind them. Monte Carlo algorithms demonstrate that algorithms do not necessarily always successfully solve the problems they are supposed to. An example is the Solovay–Strassen primality test.
- **Atlantic City:** These algorithms run in polynomial time, providing a correct problem answer at least 75 percent of the time. Monte Carlo algorithms are always fast but not always correct, and Las Vegas algorithms are always correct but not always fast. People therefore think of Atlantic City algorithms as halfway between the two because they are usually both fast and correct. This class of algorithms was introduced in 1982 by J. Finn in an unpublished manuscript entitled *Comparison of Probabilistic Test for Primality.* Created for theoretical reasons to test for prime numbers, this class comprises hard-to-design solutions, thus very few of them exist today.

## Understanding how probability works

Probability tells you the likelihood of an event, which you normally express as a number. In this book, and generally in the field of probabilistic studies, the probability of an event is measured in the range between 0 (no probability that an event will occur) and 1 (certainty that an event will occur). Intermediate values, such as 0.25 or 0.75, indicate that the event will happen with a certain frequency under conditions that should lead to that event (referred to as *trials*). Even if a numeric range from 0 to 1 doesn't seem intuitive at first, working with probability over time makes the reason for using such a range easier to understand. When an event occurs with probability 0.25, you know that out of 100 trials, the event will happen 0.25 * 100 = 25 times.

For instance, when the probability of your favorite sports team winning is 0.75, you can use the number to determine the chances of success when your team plays a game against another team. You can even get more specific information, such as the probability of winning a certain tournament (your team has a 0.65 probability of winning a match in this tournament) or conditioned by another event (when a visitor, the probability of winning for your team decreases to 0.60).

Probabilities can tell you a lot about an event, and they're helpful for algorithms, too. In a randomized algorithmic approach, you may wonder when to stop an algorithm because it should have reached a solution. It's good to know how long to look for a solution before giving up. Probabilities can help you determine how many iterations you may need. The discussion of the 2-satisfiability (o 2-SAT) algorithm in Chapter 18 provides a working example of using probabilities as stopping rules for an algorithm.

REMEMBER

You commonly hear about probabilities as percentages in sports and economics, telling you that an event occurs a certain number of times after 100 trials. It's exactly the same probability no matter whether you express it as 0.25 or 25 percent. That's just a matter of conventions. In gambling, you even hear about odds, which is another way of expressing probability, where you compare the likelihood of an event (for example, having a certain horse win the race) against not having the event happen at all. In this case, you express 0.25 as 25 against 75 or in any other way resulting in the same ratio.

You can multiply a probability for a number of trials and get an estimated number of occurrences of the event, but by doing the inverse, you can empirically estimate a probability. Perform a certain number of trials, observe each of them, and count the number of times an event occurs. The ratio between the number of occurrences and the number of trials is your probability estimate. For instance, the probability 0.25 is the probability of picking a certain suit when choosing a card randomly from a deck of cards. French playing cards (the most widely used deck; it also appears in America and Britain) provide a classic example for explaining probabilities. (The Italians, Germans, and Swiss, for example, use decks with different suits, which you can read about at http://healthy.uwaterloo.ca/museum/VirtualExhibits/Playing%20Cards/decks/index.html.) The deck contains 52 cards equally distributed into four suits: clubs and spades, which are black, and diamonds and hearts, which are red. If you want to determine the probability of picking an ace, you must consider that, by picking cards from a deck, you will observe four aces. Your trials at picking the cards are 52 (the number of cards), therefore the answer in terms of probability is 4/52 = 0.077.

TIP

You can get a more reliable estimate of an empirical probability by using a larger number of trials. When using a few trials, you may not get a correct estimate of the event probability because of the influence of chance. As the number of trials grows, event observations will get nearer to the true probability of the event itself. The principle there is a generating process behind events. To understand how the generating process works, you need many trials. Using trials in such a way is also known as *sampling* from a probabilistic distribution.

## Understanding distributions

Probability distribution is another idea that is important for working out better algorithms. A *distribution* is a table of values or a mathematical function that links every possible value of an input to the probability that such values could occur. Probability distributions are usually (but not solely) represented in charts whose abscissa axis represents the possible values of an input and whose ordinal axis represents the probability of occurrence. Most statistical models rely on the normal distributions, a distribution which is symmetric and has a characteristic bell shape. Representing a normal distribution in Python (as shown in Figure 17-1) requires a few lines of code. (You can find this code in the `A4D; 17; Probability.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
import numpy as np
from numpy.random import normal, uniform
import matplotlib.pyplot as plt
%matplotlib inline

normal_distribution = normal(size=10000) * 25 + 100
weights = np.ones_like(normal_distribution
                      ) / len(normal_distribution)
plt.hist(normal_distribution, bins=20, weights=weights)
plt.xlabel("Value")
plt.ylabel("Probability")
plt.show()
```

The plotted distribution represents an input of 10,000 numbers whose average is about 100. Each bar in the histogram represents the probability that a certain range of values will appear in the input. If you sum all the bars, you obtain the value of 1, which comprises all the probabilities expressed by the distribution.

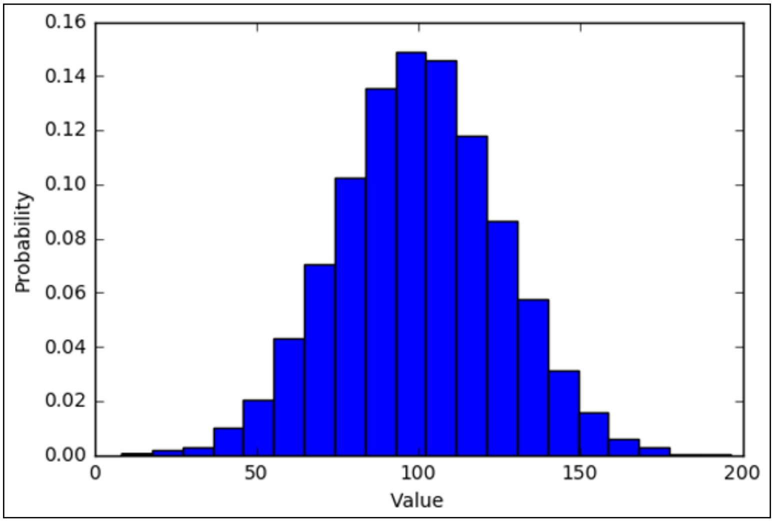


**FIGURE 17-1:** A histogram of a normal distribution.

In a normal distribution, most of the values are around the mean value. Therefore, if you pick a random number from the input, you most likely get a number around the center of the distribution. However; though less likely, you may also draw a number far from the center. If your algorithm works better by using the mean than it does with any other number, picking a number at random makes sense and may be less trouble than devising a smarter way to draw values from your input.

Another important distribution mentioned in this chapter is the uniform distribution. You can represent it using some Python code (the output appears in Figure 17-2), too:

```
uniform_distribution = uniform(size=10000) * 100
weights = np.ones_like(uniform_distribution
                      ) / len(uniform_distribution)
plt.hist(uniform_distribution, bins=20, weights=weights)
plt.xlabel("Value")
plt.ylabel("Probability")
plt.show()
```

The uniform distribution is noticeably different from the normal distribution because each number has the same probability of being in the input as any other. Consequently, the histogram bars are all roughly of the same size, and picking a number in a uniform distribution means giving all the numbers the same chance to appear. It's a way to avoid systematically picking the same groups of numbers when your algorithm works better with varied inputs. For instance, uniform distributions work well when your algorithm works fine with certain numbers, so-so with most, and badly with a few others, and you prefer to pick numbers randomly to avoid picking a series of bad numbers. This is the strategy used by the Quickselect and randomized Quicksort algorithms, described later in the chapter.

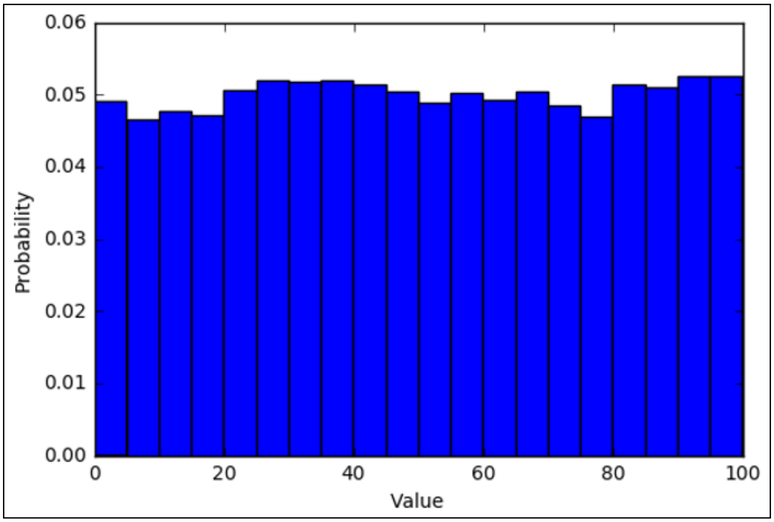


**FIGURE 17-2:** A histogram of a uniform distribution.

Because algorithms need numeric inputs, knowing their distribution can help make them work smarter. It's not just the initial distribution that counts. You can also take advantage of how data distribution changes as the algorithm proceeds. As an example of how a changing distribution can improve your algorithm, the following code shows how to guess a card in a French deck by random choice:

```
numbers = ['Ace','2','3','4','5','6','7','8','9','10',
                                'Jack','Queen','King']
seeds = ['Clubs','Spades','Diamonds','Hearts']
deck = [s+'_'+n for n in numbers for s in seeds]

from random import choice
my_cards = deck.copy()
guessed = 0
for card in deck:
    if card == choice(my_cards):
        guessed += 1
print ('Guessed %i card(s)' % guessed)

Guessed 1 card(s)
```

This strategy brings few results, and on average, you'll guess a single card in all 52 trials. In fact, for each trial, you have a 1/52 probability of guessing the correct card, which amounts to 1 after picking all the cards: (1/52) * 52 = 1. Instead, you can change this simple random algorithm by discarding the cards that you've seen from your possible choices:

```
from random import choice
my_cards = deck.copy()
guessed = 0
for card in deck:
    if card == choice(my_cards):
        guessed += 1
    else:
        my_cards.pop(my_cards.index(card))
print ('Guessed %i card(s)' % guessed)

Guessed 1 card(s)
```

Now, on average, you'll guess the right card more often because as the deck decreases, your chances of guessing increases and you'll likely guess correctly more often when nearing the end of the game. (Your chances are 1 divided by the number of cards left in the deck).

TIP

Counting cards can provide an advantage in card games. A team of MIT students used card counting and probability estimates to win huge amounts in Las Vegas until the practice was banned from Casinos. The story even inspired a 2008 film entitled *21*, starring Kevin Spacey. You can read more about the story at: `http://www.bbc.com/news/magazine-27519748`.

## Simulating the use of the Monte Carlo method

Calculating probabilities, apart from the operations discussed earlier in this chapter, is beyond the scope of this book. Understanding how an algorithm incorporating randomness works is not an easy task, even when you know how to compute probabilities because it may be the result of blending many different probability distributions. However, a discussion of the Monte Carlo method casts light on the results of the most complex algorithms and helps you understand how they work. This method sees use in both mathematics and physics to solve many problems. For instance, scientists such as Enrico Fermi and Edward Teller used Monte Carlo simulations on specially devised supercomputers during the Manhattan project (which developed the atomic bomb during World War II) to accelerate their experiments. You can read more about this use at `http://www.atomicheritage.org/history/computing-and-manhattan-project`.

REMEMBER

Don't confuse the Monte Carlo method with the Monte Carlo algorithm. The Monte Carlo method is a way to understand how a probability distribution affects a problem, whereas, as discussed previously, the Monte Carlo algorithm is a randomized algorithm that isn't guaranteed to reach a solution.

In a Monte Carlo simulation, you repeatedly sample the algorithm results. You store a certain number of results and then calculate statistics, such as the mean, and visualize them as a distribution. For instance, if you want to understand better how reducing the size of the deck you're drawing from can help you achieve better results (as in the previous Python script), you iterate the algorithm a few times and record the success rate:

```
import numpy as np
samples = list()
for trial in range(1000):
    my_cards = deck.copy()
    guessed = 0
    for card in deck:
        if card == choice(my_cards):
            guessed += 1
        else:
            my_cards.pop(my_cards.index(card))
    samples.append(guessed)
```

Running a Monte Carlo simulation may take a few seconds. The time required depends on the speed of the algorithm, the size of the problem, and the number of trials. However, when sampling from distributions, the more trials you make, the more stable the result. This example performs 1,000 trials. You can both estimate and visualize the expected result (see Figure 17-3) using the following code:

```
plt.hist(samples, bins=8)
plt.xlabel("Guesses")
plt.ylabel("Frequency")
plt.show()
print ('On average you can expect %0.2f guesses each run'
       % np.mean(samples))

On average you can expect 3.15 guesses each run
```

Observing the resulting histogram, you can determine that you get a result of three in about 300 runs out of the 1,000 trials, which gives three the highest probability of happening. Interestingly you never got a result of zero, but it is also rare to score seven or more hits. Later examples in the chapter use Monte Carlo simulations to understand how more sophisticated randomized algorithms work.

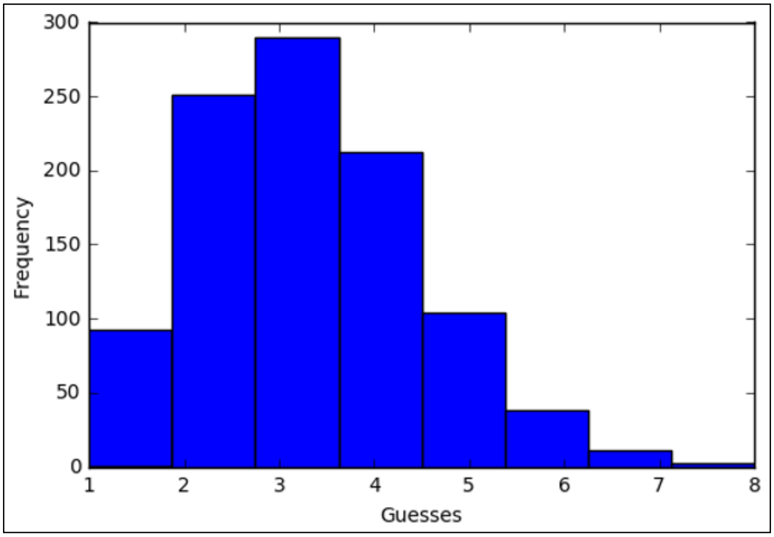


**FIGURE 17-3:** Displaying the results of a Monte Carlo simulation.

# Putting Randomness into your Logic

Here are some of many reasons to include randomness in the logic of your algorithm:

- It makes algorithms work better and provide smarter solutions.
- It requires fewer resources, in terms of memory and computations.
- It creates algorithms that have a distributed output with little or no supervision.

In the next chapter, which is dedicated to local search, you see how randomization and probability can prove helpful when it's difficult to determine what direction your algorithm should take. The examples in the sections that follow demonstrate how randomization helps to quickly find values in a certain position in your data input and how relying on randomness can speed up sorting.

## Calculating a median using Quickselect

Calculating a statistical measure, the median, can prove challenging when you work on unsorted input lists. In fact, a median relies on the position of your data when it is ordered:

- If the data inputs have an odd number of elements, the median is exactly the middle value.
- If the data inputs have an even number of elements, the median is the average of the pair of middle numbers in the ordered input list.

REMEMBER

A median is like a mean, a single value that can represent a distribution of values. The median, based on the input vector element order, isn't influenced much by the values present in your list. It's simply the middle value. Instead, the values present at the head and tail of the input can influence the mean when they're extremely small or large. This robustness makes the median very helpful in many situations when using statistics. A simple example of a median calculation using Python functions helps you understand this measure. (You can find this code in the `A4D; 17; Quickselect.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
from random import randint, random, choice
import numpy as np
import sys
sys.setrecursionlimit(1500)

n = 501
series = [randint(1,25) for i in range(n)]
print ('Median is %0.1f' % np.median(series))

Median is 14.0
```

The code creates a list of 501 elements and obtains the list median using the `median` function from the NumPy package. The reported median is actually the middle point of the ordered list, which is the 251st element:

```
print ('251st element of the ordered series is %0.1f' %
       sorted(series)[250])

251st element of the ordered series is 14.0
```

Ordering the list and extracting the necessary element demonstrates how `median` works. Because ordering is involved in calculating a median, you can expect a best running time of `O(n*log(n))`. By using randomization provided by the Quickselect algorithm, you can get an even better result, a running time of `O(n)`. Quickselect works recursively, which is why you must set a higher recursion limit in Python, given a list and the position of the value needed from an ordered list.

The value index is called *k*, and the algorithm is also known as the largest *kth value algorithm.* It uses the following steps to obtain a result:

1. Determine a pivot number in the data list and split the list into two parts, a left list whose numbers are less than the pivot number, and a right list whose numbers are higher.
2. Determine the length of each list. When the length of the left list is larger than the kth position, the median value is inside the left part. The algorithm applies itself recursively to just that list.
3. Compute the number of pivot number duplicates in the list (subtract from the length of the list the length of the left and right sides).
4. Determine whether the number of duplicates is more than k.

    a. When this condition is true, it means that the algorithm has found the solution because the kth position is contained in the duplicates (it's the pivot number).

    b. When this condition isn't true, remove the number of duplicates from k and apply the result recursively to the right side, which must contain the value of the kth position.

Now that you understand the process, you can look at some code. The following example shows how to implement a Quickselect algorithm.

```
def quickselect(series, k):
    pivot = choice(series)

    left, right = list(),list()
    for item in series:
        if item < pivot:
            left.append(item)
        if item > pivot:
            right.append(item)

    length_left = len(left)
    if length_left > k:
        return quickselect(left, k)
    k -= length_left

    duplicates = len(series) - (length_left + len(right))
    if duplicates > k:
        return float(pivot)
    k -= duplicates
    return quickselect(right, k)

quickselect(series, 250)

14.0
```

The algorithm works well because it keeps reducing the problem size. It works best when the random pivot number is drawn nearer to the kth position (the stopping rule is that the pivot number is the value in the kth position). Unfortunately, because you can't know the kth position in the unordered list, drawing randomly by using a uniform distribution (each element in the list has the same chance of being chosen) is the best solution because the algorithm eventually finds the right solution. Even when random chance doesn't work in the algorithm's favor, the algorithm keeps on reducing the problem, thus getting more chances to find the solution, as demonstrated earlier in the chapter when guessing the cards randomly picked from a deck. As the deck gets smaller, guessing the answer gets easier. The following code shows how to use Quickselect to determine the median of a list of numbers:

```
def median(series):
    if len(series) % 2 != 0:
        return quickselect(series, len(series)//2)
    else:
        left  = quickselect(series, (len(series)-1) // 2)
        right = quickselect(series, (len(series)+1) // 2)
        return (left + right) / 2

median(series)

14.0
```

## Doing simulations using Monte Carlo

As part of understanding the Quickselect algorithm, it pays to know how it works internally. By setting a counter inside the `quickselect` function, you can check performance under different conditions using a Monte Carlo simulation:

```
def quickselect(series, k, counter=0):
    pivot = choice(series)

    left, right = list(),list()
    for item in series:
        if item < pivot:
            left.append(item)
        if item > pivot:
            right.append(item)

    counter += len(series)

    length_left = len(left)
    if length_left > k:
        return quickselect(left, k, counter)
    k -= length_left

    duplicates = series.count(pivot)
    if duplicates > k:
        return float(pivot), counter
    k -= duplicates

    return quickselect(right, k, counter)
```

The first experiment tries to determine how many operations the algorithm needs, on average, to find the median of an input list of 1001 numbers:

```
results = list()
for run in range(1000):
    n = 1001
    series = [randint(1,25) for i in range(n)]
    median,count = quickselect(series, n//2)
    assert(median==np.median(series))
    results.append(count)

print ("Mean operations: %i" % np.mean(results))

Mean operations: 2764
```

Displaying the results on a histogram (see Figure 17-4) reveals that the algorithm computes from two to four times the size of the input, with three times being the most likely number of processed computations.

```
import matplotlib.pyplot as plt

%matplotlib inline

plt.hist(results, bins='auto')
plt.xlabel("Computations")
plt.ylabel("Frequency")
plt.show()
```

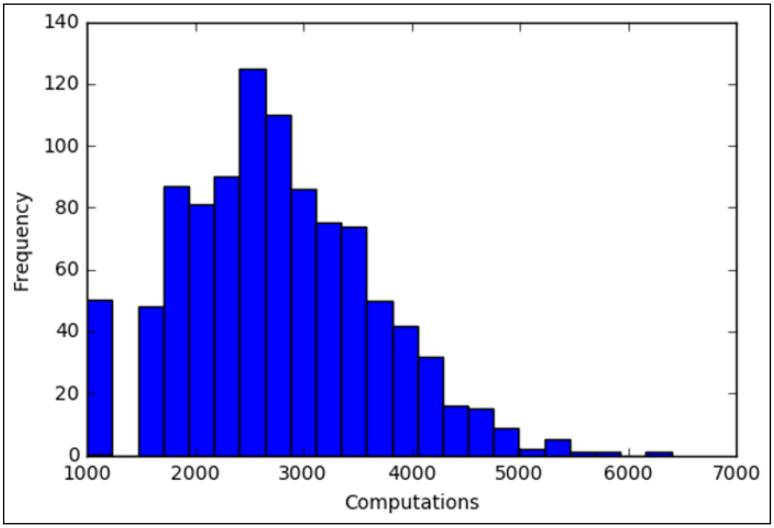


**FIGURE 17-4:** Displaying the results of a Monte Carlo simulation on Quickselect.

If on average it takes about three times the size of the input, Quickselect is providing good performance. However, you may wonder whether the proportion between inputs and computations will hold when the input size grows. As seen when studying NP-complete problems, many problems explode when the input size grows. You can prove this theory by using another Monte Carlo simulation on top of the previous one and plotting the output, as shown in Figure 17-5.

```
input_size = [501, 1001, 5001, 10001, 20001, 50001]
computations = list()
for n in input_size:
    results = list()
    for run in range(1000):
        series = [randint(1, 25) for i in range(n)]
        median,count = quickselect(series, n//2)
        assert(median==np.median(series))
        results.append(count)
    computations.append(np.mean(results))

plt.plot(input_size, computations, '-o')
plt.xlabel("Input size")
plt.ylabel("Number of computations")
plt.show()
```

WARNING

Completing the computations from this example may take up to ten minutes (some Monte Carlo simulations may be quite time consuming), but the result helps you visualize what it means to work with an algorithm that works with linear time. As the input grows (represented in abscissa), the computations (represented on the ordinal axis) grow proportionally, making the growth curve a perfect line.

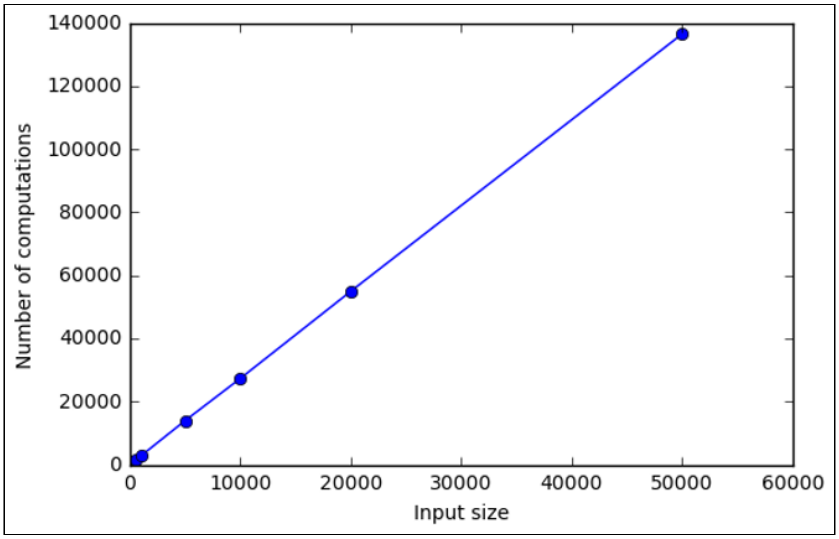


**FIGURE 17-5:** Displaying Monte Carlo simulations as input grows.

## Ordering faster with Quicksort

Chapter 7 explains ordering algorithms, the true foundations of all the modern computer-based algorithmic knowledge. The Quicksort algorithm, which can run in logarithmic time but sometimes fails and produces results in quadratic time under ill-conditioned inputs, will surely amaze you. This section explores the reasons why this algorithm may fail and provides an effective solution by injecting randomness into it. You start by examining the following code:

```
def quicksort(series, get):

    try:
        global operations
        operations += len(series)
    except:pass

    if len(series) <= 3:
        return sorted(series)

    pivot = get(series)
    duplicates = series.count(pivot)

    left, right = list(),list()
    for item in series:
        if item < pivot:
            left.append(item)
        if item > pivot:
            right.append(item)

    return quicksort(left, get) + [pivot
            ] * duplicates + quicksort(right, get)
```

This is another implementation of the algorithm from Chapter 7. However, this time the code extracts the function that decides the pivot the algorithm uses to recursively split the initial list. The algorithm decides the split by taking the first list value. It also tracks how many operations it takes to complete the ordering using the `operations` global variable, which is defined, reset, and accessed as a counter outside the function. The following code tests the algorithm, under unusual conditions, requiring it to process an already ordered list. Note its performance:

```
series = list(range(25))
operations = 0
sorted_list = quicksort(series, choose_leftmost)
print ("Operations: %i" % operations)

Operations: 322
```

In this case, the algorithm takes 322 operations to order a list of 25 elements, which is horrid performance. Using an already ordered list causes the problem because the algorithm splits the list into two lists: an empty one and another one with the residual values. It has to repeat this unhelpful split for all the unique values present in the list. Usually the Quicksort algorithm works fine because it works with unordered lists, and picking the leftmost element is equivalent to randomly drawing a number as the pivot. To avoid this problem, you can use a variation of the algorithm that provides a true random draw of the pivot value.

```
def choose_random(l): return choice(l)

series = [randint(1,25) for i in range(25)]
operations = 0
sorted_list = quicksort(series, choose_random)
print ("Operations: %i" % operations)

Operations: 81
```

Now the algorithm performs its task using a lower number of operations, which is exactly the running time `n * log(n)` that is `25 * log(25) = 80.5`.

IN THIS CHAPTER

- **Determining how to perform a local search on an NP-hard problem**
- **Working with heuristics and neighboring solutions**
- **Solving the 2-SAT problem with local search and randomization**
- **Discovering that you have many tricks to apply to a local search**

# Chapter 18

# Performing Local Search

When dealing with an NP-hard problem, a problem for which no known solution has a running complexity less than exponential (see the NP-completeness theory discussion in Chapter 15), you have a few alternatives worth trying. Based on the idea that NP-class problems require some compromise (such as accepting partial or nonoptimal results), the following options offer a solution to this otherwise intractable problem:

- Identify special cases under which you can solve the problem efficiently in polynomial time using an exact method or a greedy algorithm. This approach simplifies the problem and limits the number of solution combinations to try.
- Employ dynamic programming techniques (described in Chapter 16) that improve on brute-force search and reduce the complexity of the problem.
- Compromise and sketch an approximate algorithm that finds a *partial,* close-to-optimal solution. When you're satisfied with a partial solution, you cut the algorithm's running time short. Approximate algorithms can be
  - Greedy algorithms (as discussed in Chapter 15)
  - Local search using randomization or some other heuristic technique (the topic of the present chapter)
  - Linear programming (the topic of Chapter 19)

- Choose a heuristic or a *meta-heuristic* (a heuristic that helps you determine which heuristic to use) that works well for your problem in practice. However, it has no theoretical guarantee and tends to be empiric.

# Understanding Local Search

*Local search* is a general approach to solving problems that comprises a large range of algorithms, which will help you escape the exponential complexities of many NP problems. A local search starts from an imperfect problem solution and moves away from it, a step at a time. It determines the viability of nearby solutions, potentially leading to a perfect solution, based on random choice or an astute heuristic (no exact method is involved).

REMEMBER

A *heuristic* is an educated guess about a solution, such as a rule of thumb that points to the direction of a desired outcome but can't tell exactly how to reach it. It's like being lost in an unknown city and having people tell you to go a certain way to reach your hotel (but without precise instructions) or just how far you are from it. Some local search solutions use heuristics, so you find them in this chapter. Chapter 20 delves into the full details of using heuristics to perform practical tasks.

You have no guarantee that a local search will arrive at a problem solution, but your chances do improve from the starting point when you provide enough time for the search to run its computations. It stops only after it can't find any further way to improve the solution it has reached.

## Knowing the neighborhood

Local search algorithms iteratively improve from a starting solution, moving one step at a time through neighboring solutions until they can't improve the solution any further. Because local search algorithms are as simple and intuitive as greedy algorithms, designing a local search approach to an algorithmic problem is not difficult. The key is defining the correct procedure:

1. Start with an existing solution (usually a random solution or a solution from another algorithm).
2. Search for a set of possible new solutions within the current solution's neighborhood, which constitutes the *candidates' list*.

3. Determine which solution to use in place of the current solution based on the output of a heuristic that accepts the candidates' list as input.
4. Continue performing Steps 2 and 3 until you see no further improvement on the solution, which means that you have the best solution available.

REMEMBER

Although easy to design, local search solutions may not find a solution in a reasonable time (you can stop the process and use the current solution) or produce a minimum quality solution. You can employ some tricks of the trade to ensure that you get the most out of this approach.

At the start of the local search, you pick an initial solution. If you decide on a random solution, it's helpful to wrap the search in repeated iterations in which you generate different random start solutions. Sometimes, arriving at a good final solution depends on the starting point. If you start from an existing solution to be refined, plugging the solution into a greedy algorithm may prove to be a good compromise in fitting a solution that doesn't take too long to produce.

After choosing a starting point, define the neighborhood and determine its size. Defining a neighborhood requires figuring the smallest change you can impose on your solution. If a solution is a set of elements, all the neighboring solutions are the sets in which one of the elements mutates. For instance, in the traveling salesman problem (TSP), neighboring solutions could involve changing the ending cities of two (or more) trips, as shown in Figure 18-1.

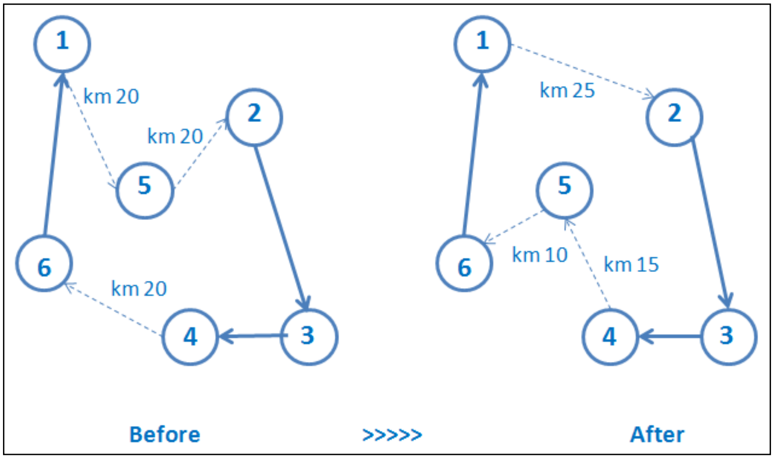


**FIGURE 18-1:** Switching ending trips in a TSP problem may bring better results.

Based on how you create the neighborhood, you may have a smaller or a larger candidates' list. Larger lists require more time and computations but, contrary to short lists, may offer more opportunities for the process to end earlier and better.

List length involves a trade-off that you refine by using experimentation after each test to determine whether enlarging or shrinking the candidate list brings an advantage or a disadvantage in terms of time to complete and solution quality.

Base the choice of the new solution on a heuristic and, given the problem, decide on the best solution. For instance, in the TSP problem, use the trip switches that shorten the total tour length the most. In certain cases, you can use a random solution in place of a heuristic (as you discover in the SAT-2 problem in this chapter). Even when you have a clear heuristic, the algorithm could find multiple best solutions. Injecting some randomness could make your local search more efficient. When faced with many solutions, you can safely choose one randomly.

TIP

Ideally, in a local search, you get the best results when you run multiple searches, injecting randomness as much as you can into the start solution and along the way as you decide the next process step. Let the heuristic decide only when you see a clear advantage to doing so. Local search and randomness are good friends.

Your search has to stop at a certain point, so you need to choose stopping rules for the local search. When your heuristic can't find good neighbors anymore or it can't improve solution quality (for instance, computing a cost function, as it happens in TSP, by measuring the total length of the tour). Depending on the problem, if you don't create a stopping rule, your search may go on forever or take an unacceptably long time. In case you can't define a stopping, just limit the time spent looking for solutions or count the number of trials. When counting trials, you can decide that it's not worth going on because you calculate the probability of success and at a certain point, the probability of success becomes too small.

## Presenting local search tricks

Local search tracks the current solution and moves to neighboring solutions one at a time until it finds a solution (or can't improve on the present solution). It presents some key advantages when working on NP-hard problems because it

- Is simple to devise and execute
- Uses little memory and computer resources (but searches require running time)
- Finds acceptable or even good problem solutions when starting from a less-than-perfect solution (neighboring solutions should create a path to the final solution)

TIP

You can see the problems that a local search can solve as a graph of interconnected solutions. The algorithm traverses the graph, moving from node to node looking for the node that satisfies the task requirements. Using this perspective, a local search takes advantage of graph exploration algorithms such as depth-first search (DFS) or breadth-first search (BFS), both discussed in Chapter 9.

Local search provides a viable way to find acceptable solutions to NP-hard problems. However, it can't work properly without the right heuristic. Randomization can provide a good match with local search, and it helps by using

- **Random sampling:** Generating solutions to start
- **Random walk:** Picking a random solution that neighbors the current one. (You find more on random walks in the "Solving 2-SAT using randomization" section, later in this chapter.)

Randomization isn't the only heuristic available. A local search can rely on a more reasoned exploration of solutions using an objective function to get directions (as in *hill-climbing* optimization) and avoid the trap of so-so solutions (as in *simulated annealing* and *Tabu Search*). An *objective function* is a computation that can assess the quality of your solution by outputting a score number. If you need higher scores in hill climbing, you have a maximization problem; if you are looking for smaller score numbers, you have a problem of minimization.

## Explaining hill climbing with n-queens

You can easily find analogies of the techniques employed by local search because many phenomena imply a gradual transition from one situation to another. Local search isn't just a technique devised by experts on algorithms but is actually a process that you see in both nature and human society. In society and science, for instance, you can view innovation as a local search of the next step among the currently available technologies: `https://www.technologyreview.com/s/603366/mathematical-model-reveals-the-patterns-of-how-innovations-arise/`. Many heuristics derive from the physical world, taking inspiration from the force of gravity, the fusion of metals, the evolution of DNA in animals, and the behavior of swarms of ants, bees, and fireflies (the paper at `https://arxiv.org/pdf/1003.1464.pdf` explains the Lévy-Flight Firefly algorithm).

Hill climbing takes inspiration from the force of gravity. It relies on the observation that as a ball rolls down a valley, it takes the steepest descent, and when it climbs a hill, it tends to take the most direct upward direction to reach the top. Gradually, one step after the other, no matter whether it's climbing up or down, the ball arrives at its destination, where proceeding higher or lower isn't possible.

In local search, you can mimic the same procedure successfully using an *objective function,* a measurement that evaluates the neighboring solutions and determines which one improves on the current one. Using the hill-climbing analogy, having an objective function is like feeling the inclination of the terrain and determining the next best move. From the current position, a hiker evaluates each direction to determine the terrain's slope. When the goal is to reach the top, the hiker chooses the direction with the greatest upward slope to reach the top. However, that's just the ideal situation; hikers often encounter problems during a climb and must use other solutions to circumnavigate them.

An objective function is similar to a greedy criterion (see Chapter 5). It's blind with respect to its final destination, so it can determine direction but not detect obstacles. Think about the effect of blindness when climbing the mountains — it's difficult to say when a hiker reaches the top. Flat terrain that lacks any opportunities for upward movement could indicate that the hiker reached the top. Unfortunately, a flat spot can also be a plain, an elbow, or even a hole the hiker happened to fall into. You can't be sure because the hiker can't see.

The same problem happens when using a local search guided by a hill-climbing heuristic: It pursues progressively better neighbor solutions until it can't find a better solution by checking the solutions that exist around the current one. At this point, the algorithm declares it found the solution. It also says that it has found a global solution, even though, as illustrated in Figure 18-2, it may have simply found a local *maximum,* a solution that's the best around because it's surrounded by worse solutions. It's still possible to find a better solution through further exploration.

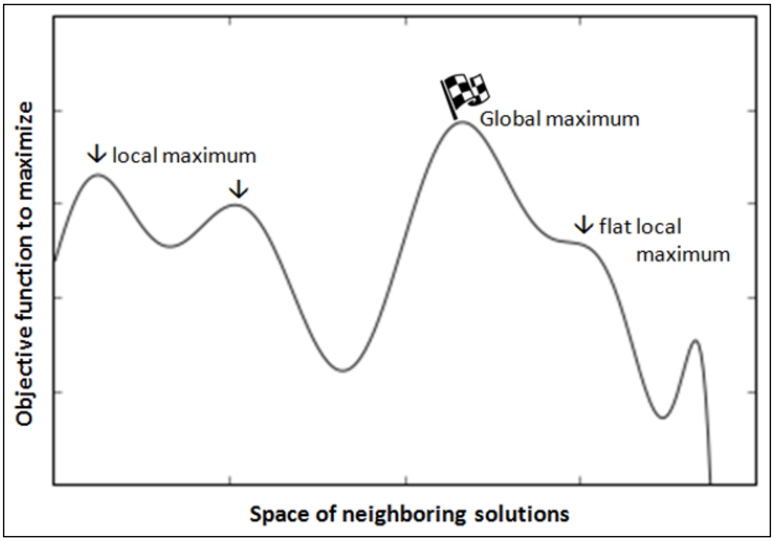


**FIGURE 18-2:** Local search explores the landscape by hill climbing.

An example of hill climbing in action (and of the risks of getting stuck in a local maximum or in a local minimum when you're descending, as in this example) is the n-queens puzzle, first created by the chess expert Max Bezzel, in 1848, as a challenge for chess lovers. In this problem, you have a number of queens (this number is *n*) to place on a chessboard of *n* x *n* dimensions. You must place them so that no queen is threatened by any other. (In chess, a queen can attack by any direction by row, column, or diagonal.)

TIP

This is really a NP-hard problem. If you have eight queens to place on a 8 x 8 chessboard, there are 4,426,165,368 different ways to place them but only 92 configurations solve the problem. Clearly, you can't solve this problem using brute force or luck alone. Local search solves this problem in a very simple way using hill climbing:

1. Place the *n* queens randomly on the chessboard so that each one is on a different column (no two queens on the same column).
2. Evaluate the next set of solutions by moving each queen one square up or down in its column. This step requires 2**n* moves.
3. Determine how many queens are attacking each other after each move.
4. Determine which solution has the fewest queens attacking each other and use that solution for the next iteration.
5. Perform Steps 4 and 5 until you find a solution.

Unfortunately, this approach works only about 14 percent of the time because it gets stuck in a chessboard configuration that won't allow any further improvement 86 percent of the time. (The number of queens under attack won't diminish for all 2**n* moves available as next solutions.) The only way you get away from such a block is to restart the local search from scratch by choosing another random starting configuration of the queens on the chessboard. Figure 18-3 shows a successful solution.

In spite of this weakness, hill-climbing algorithms are used everywhere, especially in artificial intelligence and machine learning. Neural networks that recognize sounds or images, power mobile phones, and motivate self-driving cars mostly rely on a hill-climbing optimization called *gradient descent.* Randomized starts and random injections in the hill-climbing procedure make it possible to escape any local solution and reach the global maximum. Both simulated annealing and Tabu Search are smart ways to use random decisions in hill climbing.

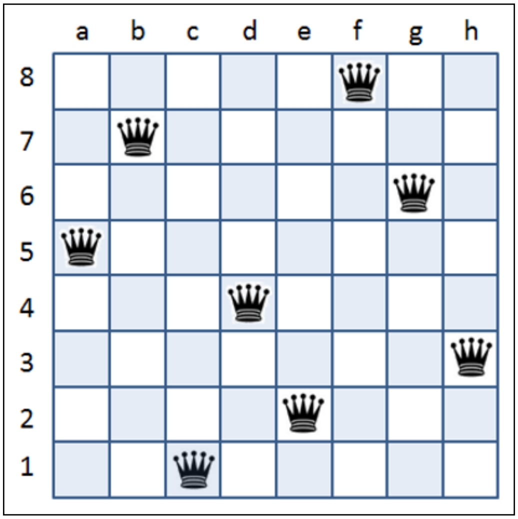


**FIGURE 18-3:** An 8-queen puzzle solved.

## Discovering simulated annealing

At a certain point in the search, if your objective function stops giving you the right indications, you can use another heuristic to control the situation and try to find a better path to a better task solution. This is how both simulated annealing and Tabu Search work: They provide you with an emergency exit when needed.

Simulated annealing take its name from a technique in metallurgy, which heats the metal to a high temperature and then slowly cools it to soften the metal for cold working and to remove internal crystalline defects (see `http://www.brighthubengineering.com/manufacturing-technology/30476-what-is-heat-treatment/` for details on this metal-working process). Local search replicates this idea by viewing the solution search as an atomic structure that changes to improve its workability. The temperature is the game changer in the optimization process. Just as high temperatures make the structure of a material relax (solids melt and liquid evaporate at high temperatures), so high temperatures in a local search algorithm induce relaxation of the objective function, allowing it to prefer worse solutions to better ones. Simulated annealing modifies the hill-climbing procedure, keeping the objective function for neighbor solution evaluation but allowing it to determine the search solution choice in a different way:

1. Obtain a temperature expressed as probability. (The Gibbs-Boltzmann physics function is a formula that converts temperature to probability. An explanation of this function is beyond the scope of this book, but you can explore it at: `http://www.iue.tuwien.ac.at/phd/binder/node87.html`.)
2. Set a temperature schedule. The temperature decreases at a certain rate as time passes and the search runs.

3. Define a starting solution (using random sampling or another algorithm) and start a loop. As the loop proceeds, the temperature decreases.
4. Stop the optimization when the temperature is zero.
5. Propose the current result as the solution.

At this point, you must iterate the search for solutions. For each step in the previous iteration, between the preceding Steps 3 and 4, do the following:

1. List the neighboring solutions and choose one at random.
2. Set the neighboring solution as the current solution when the neighboring solution is better than the current one.
3. Otherwise, pick a random number between 0 and 1 based on a threshold probability associated with the actual temperature and determine whether it's less than the threshold probability:
    - If it's less, set the neighboring solution as the current solution (even if it's worse than the current solution, according to the objective function).
    - If it's more, keep the current solution.

Simulated annealing is a smart way to improve hill climbing because it avoids having the search stopped at a local solution. When the temperature is high enough, the search might use a random solution and find another way to a better optimization. Because the temperature is higher at the beginning of the search, the algorithm has a chance of injecting randomness into the optimization. As the temperature cools to zero, less and less chance exists for picking a random solution, and the local search proceeds as in hill climbing. In TSP, for instance, the algorithm achieves simulated annealing by challenging the present solution at high temperatures by

» Choosing a segment of the tour randomly and traversing it in the opposite direction

» Visiting a city earlier or afterward in the tour, leaving the order of visit to the other cities the same

If the resulting adjustments worsen the tour's length, the algorithm keeps or rejects them according to the temperature in the simulated annealing process.

## Avoiding repeats using Tabu Search

Tabu is an ancient word from Polynesian Tongan that says certain things can't be touched because they're sacred. The word *tabu* (which is spelled as taboo in

English) passed from anthropological studies to the everyday language to indicate something that is prohibited. In local search optimization, it's common to become stuck in a neighborhood of solutions that don't offer any improvement; that is, it's a local solution that appears as the best solution but is far from being the solution you want. Tabu search relaxes some rules and enforces others to offer a way out of local minima and help you reach better solutions.

The Tabu Search heuristics wraps objective functions and works its way along many neighboring solutions. It intervenes when you can't proceed because the next solutions don't improve on your objective. When such happens, Tabu Search does the following:

- Allows use of a pejorative solution for a few times to see whether moving away from the local solution can help the search find a better path to the best solution.
- Remembers the solutions that the search tries and forbids it from using them anymore, thus assuring that the search doesn't loop between the same solutions around the local solution without finding an escape route.
- Creates a long-term or short-term memory of Tabu solutions by modifying the length of the queue used to store past solutions. When the queue is full, the heuristic drops the oldest Tabu to make space for the new one.

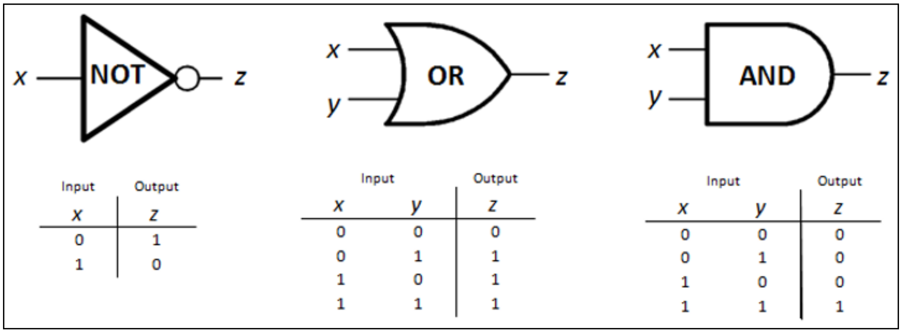
You can relate Tabu Search to caching and memoization (see Chapter 16) because it requires the algorithm to track its steps to save time and avoid retracing previously used solutions. In the TSP, it can help when you try optimizing your solution by swapping the visit order of two or more cities by avoiding repeat solution sets.

## Solving satisfiability of Boolean circuits

As a practical view of how a local search works, this example delves into circuit satisfiability, a classical NP-complete problem. It uses a randomization and Monte Carlo algorithm approach. As seen in Chapter 17, a Monte Carlo algorithm relies on random choices during its optimization process and isn't guaranteed to succeed in its task, although it has a high likelihood of completing the task successfully. The problem isn't merely theoretical, though, because it tests how electronic circuits work, optimizing them by removing circuits that can't transport electric signals. Moreover, the solving algorithm sees use in other applications: automatic labeling on maps and charts, discrete tomography, scheduling with constraints, data clustering into groups, and other problems for which you have to make conflicting choices.

Computer circuits are composed of a series of connected components, each one opening or closing a circuit based on its inputs. Such elements are called *logic gates* (physically, their role is played by transistors) and if you build a circuit with many logic gates, you need to understand whether electricity can pass through it and under what circumstances.

Chapter 14 discusses the internal representation of a computer, based on zeros (absence of electricity in the circuit) or ones (presence of electricity). You can render this 0/1 representation from a logical perspective, turning signals into `False` (there isn't electricity in the circuit) or `True` (there is indeed electricity) conditions. Chapter 4 examines the Boolean operators (`AND`, `OR`, and `NOT`), as shown in Figure 18-4, which work on `True` and `False` conditions as inputs and transform them into a different output. All these concepts help you represent a physical electric circuit as a sequence of Boolean operators defining logic gates. The combination of all their conditions determines whether the circuit can carry electricity.



| Input | Output |
|---|---|
| x | z |
| 0 | 1 |
| 1 | 0 |

| Input | | Output |
|---|---|---|
| x | y | z |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| Input | | Output |
|---|---|---|
| x | y | z |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**FIGURE 18-4:** Symbols and truth tables of logic operators `AND`, `OR`, and `NOT`.

This logic representation is a *Boolean combinational circuit,* and the test to verify its functionality is the *circuit satisfiability.* In the easiest scenario, the circuit consists of only `NOT` conditions (called *inverters*) that accept one wire input, and `OR` conditions that accept two wires as inputs. This is a satisfiability-two (2-SAT) scenario, and if the algorithm were to go through it using an exhaustive search, it would take at worst $2^k$ trials (having *k* as the number of input wires) to find a set of conditions that makes electricity pass through the whole circuit. There are even more complex versions of the problem, accepting more inputs for each OR logic gate and using AND gates, but they are beyond the scope of this book.

## Solving 2-SAT using randomization

No matter the electronic circuit you have to test using a Boolean representation, you can render it as a vector of Boolean variables. You can also create another vector to contain the *clauses*, the set of conditions the circuit needs to satisfy (for example, that wire A and wire B should both be True). This isn't the only way to represent the problem; in fact, there are other solutions involving the use of graphs. However, for this example, these two vectors are enough.

You solve the problem using a randomized local search in polynomial time. Professor Christos H. Papadimitriou, teaching at the University of California at Berkeley (https://people.eecs.berkeley.edu/~christos/), devised this algorithm, called *RandomWalkSAT*. He presented it in his paper "On Selecting a Satisfying Truth Assignment," published in 1991 on the Proceedings of the 32nd IEEE Symposium on the Foundations of Computer Science. The algorithm is competitive when compared to more reasoned ways, and it is an exemplary local search approach because it makes just one change at a time on the current solution. It uses two nested loops, one for trying the starting solution multiple times and one for randomly amending the initial random solution. Repeat the outer loop $\log_2(k)$ times (where *k* is the number of wires). The inner loop uses the following steps:

1. Pick a random problem solution.
2. Repeat the following steps $2*k^2$ times:
   a. Determine whether the current solution is the right one. When it is the correct solution, break free of all the loops and report the solution.
   b. Pick an unsatisfied clause at random. Choose one of the conditions in it at random and amend it.

## Implementing the Python code

To solve the 2-SAT problem using Python and the RandomWalkSAT algorithm, you need to set a few helpful functions. The create_clauses and signed functions help generate a circuit problem to solve by handling the OR and NOT gates, respectively. Using these functions, you specify the number of OR gates and provide a seed number that guarantees that you can recreate the resulting problem later (allowing you to try the problem multiple times and on different computers).

The create_random_solutions function provides a cold problem start by providing a random solution that sets the inputs. The chances of finding the right solution using random luck is slim (one out of the power of two to the number of gates), but on average, you can expect that three quarters of the gates are correctly set (because, as seen using the truth table for the OR function, three inputs out of four possible are True). The check_solution function determines when the circuit is satisfied (indicating a correct solution). Otherwise, it outputs what conditions aren't satisfied. (You can find this code in the A4D; 18; Local Search.ipynb file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
import numpy as np
import random
from math import log2

import matplotlib.pyplot as plt
% matplotlib inline

def signed(v):
    return v if np.random.random()<0.5 else -v

def create_clauses(i, seed=1):
    np.random.seed(seed)
    return [(signed(np.random.randint(i)), signed(
                np.random.randint(i))) for j in range(i)]

def create_random_solution(i, *kwargs):
    return {j:signed(1)==1 for j in range(i)}

def check_solution(solution, clauses):
    violations = list()
    for k,(a,b) in enumerate(clauses):
        if not (((solution[abs(a)]) == (a>0)) |
                ((solution[abs(b)]) == (b>0))):
            violations.append(k)
    return violations
```

After setting these functions, you have all the building blocks for a sat2 function to solve the problem. This solution uses two nested iterations: The first replicates many starts; the second picks unsatisfied conditions at random and makes them true. The solution runs in polynomial time. The function isn't guaranteed to find a solution, if one exists, but chances are, it will provide a solution when one exists. In fact, the internal iteration loop makes $2*k^2$ random attempts to solve the circuit, which usually proves enough for a random walk on a line to reach its destination.

REMEMBER

A random walk is a series of computations representing an object that moves away from its initial position by taking a random direction at every step. You might imagine a random walk as the journey of a drunken person from one light pole to the next. Random walks are useful for representing a mathematical model of many real-world aspects. They find applications in biology, physics, chemistry, computer science, and economics, especially in stock market analysis. If you want to know more about random walks, go to http://www.mit.edu/~kardar/teaching/projects/chemotaxis(AndreaSchmidt)/random.htm.

A random walk on a line is the easiest example of a random walk. On average, $k^2$ steps of a random walk are required to arrive at a *k* distance from the starting point. This expected effort explains why RandomWalkSAT requires $2*k^2$ random chances to amend the starting solution. The number of chances provides a high probability that the algorithm will fix the *k* clauses. Moreover, it works as the random card guessing game discussed in the previous chapter. As the algorithm proceeds, choosing the right answer becomes easier. The external replications guarantee an escape from unfortunate internal-loop random choices that may stop the process in a local solution.

```
def sat2(clauses, n, start=create_random_solution):
    for external_loop in range(round(log2(n))):
        solution = start(n, clauses)
        history = list()
        for internal_loop in range(2*n**2):
            response = check_solution(solution, clauses)
            unsatisfied = len(response)
            history.append(unsatisfied)
            if unsatisfied==0:
                print ("Solution in %i external loops," %
                       (external_loop+1), end=" ")
                print ("%i internal loops" %
                       (internal_loop+1))
                break
            else:
                r1 = random.choice(response)
                r2 = np.random.randint(2)
                clause_to_fix = clauses[r1][r2]
                solution[abs(clause_to_fix)] = (
                 clause_to_fix>0)
        else:
            continue
        break
    return history, solution
```

Now that all the functions are correctly set, you can run the code to solve a problem. Here's the first example, which tries the circuit created by seed 0 and uses 1,000 logic gates.

```
n = 1000
# Solvable seeds with n=1000 : 0,1,2,3,4,5,6,9,10
# Unsolvable seeds with n=1000 : 8
clauses = create_clauses(n, seed=0)
history, solution = sat2(clauses, n,
        start=create_random_solution)

Found solution in 1 external loops, 1360 internal loops
```

Plotting the solution, as a chart representing the number of steps on the abscissa (random emendations of the solution) and the clauses left to fix on the ordinal axis, you can verify that the algorithm tends to find the correct solution in the long run, as shown in Figure 18-5.

```
plt.plot(np.array(history), 'b-')
plt.xlabel("Random adjustments")
plt.ylabel("Unsatisfied clauses")
plt.grid(True)
plt.show()
```

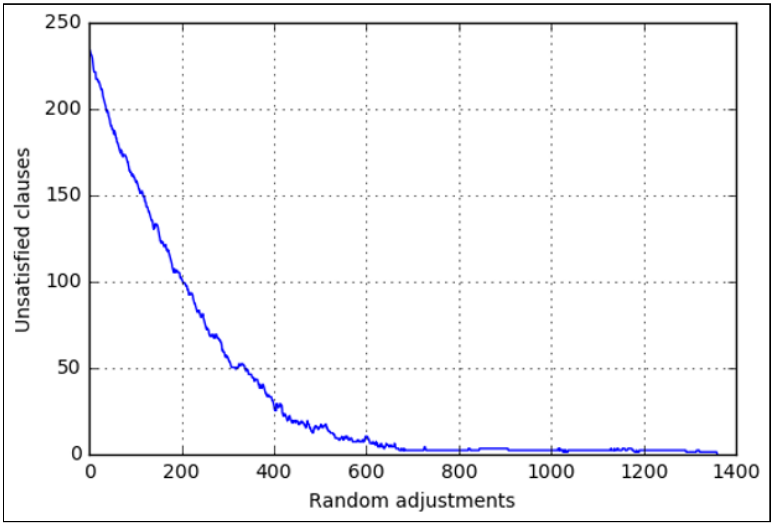


**FIGURE 18-5:** The number of unsatisfiable clauses decreases after random adjustments.

If you try the circuit with 1,000 gates and seed equal to 8, you will notice that it seems to never end. This is because the circuit is not solvable, and making all the random choices and attempts takes a long time. In the end, the algorithm won't provide you with any solution.

## Realizing that the starting point is important

Even though the RandomWalkSAT algorithm has a runtime complexity of $O(\log_2 k * k^2)$ at worst, with *k* the number of inputs, you can speed it up by hacking the starting point. In fact, even though starting with a random configuration means that a quarter of the clauses remains unsatisfied at the start on average, you can fix many of them using a pass over the data.

The problem with clauses is that many require a true input, and simultaneously, many others require a false input. When all clauses require an input to be true or false, you can set it to the required condition, which satisfies a large number of clauses and makes solving the residual ones easier. The following new RandomWalkSAT implementation includes a start phase that immediately solves the situations in which an input requires a specific true or false setting by all the clauses they interact with:

```
def better_start(n, clauses):
    clause_dict = dict()
    for pair in clauses:
        for clause in pair:
            if abs(clause) in clause_dict:
                clause_dict[abs(clause)].add(clause)
            else:
                clause_dict[abs(clause)] = {clause}

    solution = create_random_solution(n)

    for clause, value in clause_dict.items():
        if len(value)==1:
            solution[clause] = value.pop() > 0
    return solution
```

The code defines a new function for the cold start where, after generating a random solution, it scans through the solution and finds all the inputs associated with a single state (true or false). By setting them immediately to the required state, you can reduce the number of clauses requiring amendment, and have the local search do less work and complete earlier.

```
n = 1000
# Solvable seeds = 0,1,2,3,4,5,6,9,10
# Unsolvable seeds = 8
clauses = create_clauses(n, seed=0)
```

```
history, solution = sat2(clauses, n, start=better_start)

Found solution in 1 external loops, 393 internal loops
```

By providing this new, simplified starting point, after charting the results you can immediately see an improvement because on average, fewer operations are needed to complete the task.

TIP

In a local search, always consider that the starting point is important to allow the algorithm to complete earlier and more successfully, as shown in Figure 18-6. In sum, try to provide the best-quality start for your search as possible.

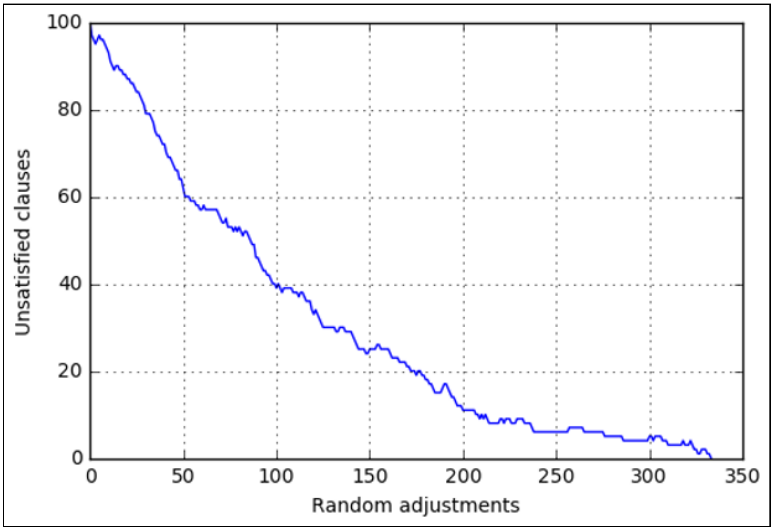


**FIGURE 18-6:** Execution is speedier because the starting point is better.

IN THIS CHAPTER

- Discovering how optimization happens using linear programming
- Transforming real-world problems into math and geometry ones
- Learning how to use Python to solve linear programming problems

# Chapter 19 Employing Linear Programming

Linear programming made a first appearance during World War II when logistics proved critical in maneuvering armies of millions of soldiers, weapons, and supplies across geographically variegated battlefields. Tanks and airplanes needed to refuel and rearm, which required a massive organizational effort to succeed in spite of limitations in time, resources, and actions from the enemy.

You can express most of these military problems in mathematical form. Mathematician George Bernard Dantzig, who was employed in the U.S. Air Force Office of Statistical Control, devised a smart way to solve these problems using the *simplex* algorithm. Simplex is the core idea that created interest in numerical optimization after the war and gave birth to the promising field of linear programming. The availability of the first performing computers of the time also increased interest, rendering complex computations solvable in a new and fast way. You can view the early history of computing in the 1950s and 1960s as a quest to optimize logistical problems using the simplex method and applying both high-speed computers and specialized programming languages.

Dantzig died in 2005, and the field he inaugurated is still under constant development. In the recent years, fresh ideas and methods related to linear programming continue to make successful appearances, such as the following:

- **Constrain programming:** Expresses the relationships between the variables in a computer program as constraints in linear programming.
- **Genetic algorithms:** Considers the idea that math formulas can replicate and mutate in order to solve problems in the same manner as DNA does in nature by evolution. Genetic algorithms also appear in Chapter 20 because of their heuristic approach to optimization.

This chapter helps you understand linear programming. In addition, you see how to apply linear programming to real-world problems by using Python as the tool to express those problems in code.

## Using Linear Functions as a Tool

This section shows how to address a problem where someone transforms the *objective* (the representation of cost, profit, or some other quantity to maximize or minimize subject to the constraints) and *constraints* (linear inequalities derived from the application, such as the limit of a 40-hour work week) of that problem into linear functions. The purpose of linear programming is to provide an optimum numeric solution, which could be a maximum or a minimum value, and the set of conditions to obtain it.

This definition may sound a little bit tricky because both math and some abstraction is involved (objective and constraints as linear functions), but things become clearer after considering what a function is and when we can determine whether a function is linear or not. Beyond the math jargon, linear programming is just a different point of view when dealing with algorithmic problems, where you trade operations and manipulations of data inputs with mathematical functions and you perform calculations using a software program called an *optimizer*.

You can't use linear programming to solve all problems, but a large number of them fit linear programming requirements, especially problems requiring optimization using previously defined limits. Previous chapters discuss how dynamic programming is the best approach when you need to optimize problems subject to constraints. Dynamic programming works with problems that are discrete, that is the numbers you work with are whole numbers. Linear programming mainly works with decimal numbers, although special optimization algorithms are available that provide solutions as integer numbers (for instance you can solve the traveling salesman problem using integer linear programming). Linear programming has a wider scope, because it can cope with almost any polynomial time problem.

Linear programming sees use for needs such as manufacturing, logistics, transportation (especially for airlines, for defining routes, timetables, and the cost of tickets), marketing, finance, and telecommunications. All these applications require that you obtain a maximum economic result and minimum cost while optimizing available resource allocation and satisfying all constraints and limitations. In addition, you can apply linear programming to common applications such as video games and computer visualization, because games require dealing with bidimensional and tridimensional complex shapes, and you need to determine whether any shapes collide as well as ensure that they respect the rules of the game. You achieve these aims via the convex hull algorithm powered by linear programming (see `http://www.tcs.fudan.edu.cn/rudolf/Courses/Algorithms/Alg_ss_07w/Webprojects/Chen_hull/applications.htm`). Finally, linear programming is at work in search engines for document-retrieval problems; you can transform words, phrases, and documents into functions and determine how to maximize your search result (getting the documents you need in order to answer your query) when you look for documents with certain mathematical characteristics.

## Grasping the basic math you need

In computer programming, functions provide the means for packaging code that you intend to use more than once. Functions turn code into a black box, an entity to which you provide inputs and expect certain outputs. Chapter 4 discusses how to create functions in Python. Mathematics uses functions in a similar manner to programming; they are set of mathematical operations that transform some input into an output. The input can include one or more variables, resulting in a unique output based on the input. Usually a function has this form:

```
f (x) = x*2
```

- **f:** Determines the function name. It can be anything; you can use any letter of the alphabet or even a word.
- **(x):** Specifies the input. In this example, the input is the variable x, but you can use more inputs and of any complexity, including multiple variables or matrices.
- **x*2:** Defines the set of operations that the function performs after receiving the input. The result is the function output in the form of a number.

If you plug the input 2 as x in this example, you obtain:

```
f(2) = 4
```

In math terms, by calling this function, you mapped the input 2 to the output 4.

REMEMBER

Functions can be simple or complex, but every function has one and only one result for every set of inputs that you provide (even when the input is made of multiple variables).

Linear programming leverages functions to render the objectives it has to reach in a mathematical way to solve the problem at hand. When you turn objectives into a math function, the problem translates into determining the input to the function that maps the maximum output (or the minimum, depending on what you want to achieve). The function representing the optimization objective is the *objective function.* In addition, linear programming uses functions and inequalities to express constraints or bounds that keep you from plugging just any input you want into the objective function. For instance, inequalities are

```
0 <= x <= 4
y + x < 10
```

The first of these inequalities translates into limiting the input of the objective function to values between 0 and 4. Inequalities can involve more input variables at a time. The second of these inequalities ties the values of an input to other values because their sum can't exceed 10.

REMEMBER

*Bounds* imply an input limitation between values, as in the first example. *Constraints* always involve a math expression comprising more than one variable, as in the second example.

The final linear programming requirement is for both the objective function and the inequalities to be linear expressions. This means that the objective function and inequalities can't contain variables that multiply each other, or contain variables raised to a power (squared or cubed, for instance).

TECHNICAL STUFF

All the functions in an optimization should be linear expressions because the procedure represents them as lines in a Cartesian space. (If you need to review the concept of a Cartesian space, you can find useful information at `http://www.mathsisfun.com/data/cartesian-coordinates.html`.) As explained in the "Using Linear Programming in Practice" section, later in this chapter, you can imagine working with linear programming more as solving a geometric problem than a mathematical one.

## Learning to simplify when planning

The problems that the original simplex algorithm solved were all of the kind that you usually read as math problems in a textbook. In such problems, all the data, information, and limitations are stated clearly, there is no irrelevant or redundant information, and you clearly have to apply a math formula (and most likely the one you just studied) to solve the problem.

In the real world, solutions to problems are never so nicely hinted at. Instead, they often appear in a confused way, and any necessary information isn't readily available for you to process. Yet, you can analyze the problem and locate required data and other information. In addition, you can discover limitations such as money, time, or some rule or order that you must consider. When solving the problem, you gather the information and devise the means to simplify it.

Simplification implies some loss of realism but renders things simpler, which can highlight the underlying processes that make things move, thereby helping you decide what happens. A simpler problem allows you to develop a model representing the reality. A model can approximate what happens in reality, and you can use it for both managing simulations and linear programming.

For instance, if you work in a factory and have to plan a production schedule, you know that the more people you add, the speedier production will be. However, you won't always obtain the same gain with the same addition of people. For example, the skills of the operators you add to the job affects results. In addition, you may find that adding more people to the job brings decreasing results when those people spend more time communicating and coordinating between themselves than performing useful work. Yet, you can make the model easier by pretending that every person you add to the task will produce a certain amount of final or intermediate goods.

## Working with geometry using simplex

Classic examples of linear programming problems imply production of goods using limited resources (time, workers, or materials). As an example for depicting how linear programming approaches such challenges, imagine a factory that assembles two or more products that it must deliver in a certain time. The factory workers produce two products, x and y, during an eight-hour shift. For each product, they get a different profit (that you compute by subtracting costs from revenue), different hourly production rates, and different daily demands from the market:

- **Revenue in USD for each product:** x=15, y=25
- **Production rate per hour:** x=50, y=40
- **Daily demand per product:** x=300, y=200

In essence, the business problem is to decide whether to produce more x, which is easier to assemble but pays less, or y, which guarantees more revenue but less production. To solve the problem, first determine the objective function. Express it as the sum of the quantities of the two products, multiplied by their expected unit revenue, which you know you have to maximize (only if the problem is about costs do you have to minimize the objective function):

```
f(x,y) = 15 * x + 25 * y
```

This problem has inequalities, which are bounded by x and y values that have to hold true to obtain a valid result from the optimization:

```
0 <= x <= 300
0 <= y <= 200
```

In fact, you can't produce a negative number of products, nor does it make sense to produce more products than the market demands. Another important limitation is available time, because you can't exceed eight hours for each work shift. This means calculating the time to produce both x and y products and constraining the total time to less than or equal to eight hours.

```
x/40 + y/50 <= 8
```

You can represent functions on a Cartesian plane. (For a refresher on plotting functions, consult `http://www.mathplanet.com/education/pre-algebra/graphing-and-functions/linear-equations-in-the-coordinate-plane`.) Because you can express everything using functions in this problem, you can also solve the linear programming problems as geometry problems on a Cartesian coordinate space. If the problem doesn't involve more than two variables, you can plot the two functions and their constraints as lines on a plane, and determine how they delimit a geometric shape. You'll discover that the lines delimit an area, shaped as a polygon, called the *feasible region.* This region is where you find the solution, which contains all the valid (according to constraints) inputs for the problem.

REMEMBER

When the problem deals with more than two variables, you can still imagine it using lines intersecting in a space, but you can't represent this visually because each input variable needs a dimension in the graph, and graphs are bound to the three dimensions of the world we live in.

At this point, the linear programming algorithm explores the delimited feasible region in a smart way and reports back with the solution. In fact, you don't need to check every point in the delimited area to determine the best problem solution. Imagine the objective function as another line that you represent on the plane (after all, even the objective function is a linear function). You can see that the solution you are looking for is the coordinate points where the feasible area and the objective function line first touch each other (see Figure 19-1). When the objective function line descends from above (arriving from outside the feasible region, where results occur that you can't accept because of the constraints), at a certain point it will touch the area. This contact point is usually a vertex of the area, but it could be an entire side of the polygon (in which case each point on that side is an optimal solution).

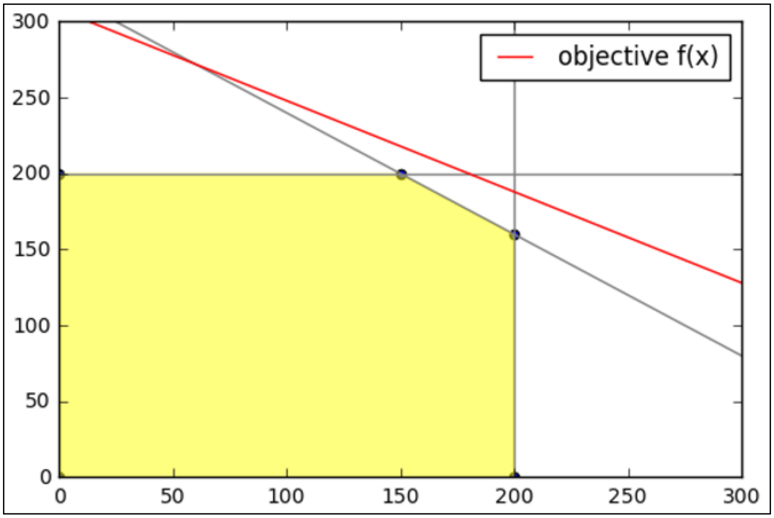


**FIGURE 19-1:** Looking where the objective function is going to touch the feasible area.

As a practical matter, the simplex algorithm can't make lines visually descend, as in this example. Instead, it walks along the border of the feasible area (by enumerating the vertexes) and tests the resulting objective function values at each vertex until it finds the solution. Consequently, the effective running time depends on the number of vertexes, which for its part depends on the number of constraints and variables involved in the solution. (More variables mean more dimensions and more vertexes.)

## Understanding the limitations

As you gain more confidence with linear programming and the problems become more challenging, you require more complex approaches than the basic simplex algorithm presented in this chapter. In fact, the simplex isn't used anymore because more sophisticated algorithms have replaced it —algorithms that geometrically cut through the interior of the feasible region instead of walking along it. These newer algorithms take a shortcut when the algorithm is clearly looking for the solution at the wrong side of the region.

You can also find working with floating-point numbers limiting because many problems require a binary (1/0) or integer answer. Moreover, other problems may require using curves, not lines, to represent the problem space and feasible region correctly. You find integer linear programming and nonlinear programming algorithms implemented in commercial software. Just be aware that both integer and nonlinear programming are NP-complete problems and may require as much, if not more, time than other algorithms you know.

# Using Linear Programming in Practice

The best way to start in linear programming is to use predefined solutions, rather than create custom applications on your own. The first section that follows helps you install a predefined solution used for the examples that follow.

WARNING

When working with a software product, you may find significant differences between open source software and commercial packages. Although open source software offers a wide spectrum of algorithms, performance could be disappointing on large and complex problems. Much art is still involved in implementing linear programming algorithms as part of working software, and you can't expect open source software to run as fast and smoothly as commercial offerings.

Even so, open source provides some nice options for learning linear program. The following sections use an open source Python solution named PuLP that allows you to create linear programming optimizations after defining a cost function and constraints as Python functions. It's mostly a didactical solution, suitable to help you test how linear programming works on some problems and get insight on formulating problems in math terms.

TECHNICAL STUFF

PuLP provides an interface to the underlying solver programs. Python comes with a default, open source, solver program that PuLP helps you access. The performance (speed, accuracy, and scalability) that PuLP provides depends almost entirely on the solver and optimizer that the user chooses. The best solvers are commercial products, such as CPLEX (`https://en.wikipedia.org/wiki/CPLEX`), XPRESS (`https://en.wikipedia.org/wiki/FICO_Xpress`), and GuRoBi (`https://en.wikipedia.org/wiki/Gurobi`), which provide a huge speed advantage when compared to open source solvers.

## Setting up PuLP at home

PuLP is a Python open source project created by Jean-Sebastien Roy, later modified and maintained by Stuart Antony Mitchell. The PuLP package helps you define linear programming problems and solve them using the internal solver (which relies on the simplex algorithm). You can also use other solvers that are available on public domain repositories or by paying for a license. The project repository (containing all the source code and many examples) is at `https://github.com/coin-or/pulp` . The complete documentation is located at `https://pythonhosted.org/PuLP/`.

PuLP isn't readily available as part of the Anaconda distribution, thus you have to install it yourself. You must use the Anaconda3 (or above) command prompt to install PuLP because the older versions of the Anaconda command prompt won't work. Open a command-line shell, type **pip install pulp** and press Enter. If you have Internet access, the pip command downloads the PuLP package and installs it in Python. (The version used by the examples in this chapter is PuLP 1.6.1, but later versions should provide the same functionality.)

## Optimizing production and revenue

The problem in this section is another optimization related to production. You work with two products (because this implies just two variables that you can represent on a bidimensional chart), product A and B, which have to undergo a series of transformations through three stages. Each stage requires a number of operators (the value *n*), which could be workers or robots, and each stage is operative at most for a number of days in the month (represented by the value *t*). Each stage operates differently on each product, requiring a different number of days before completion. For instance, a worker in the first stage (called 'res_1') takes two days to finish product A but three days for product B. Finally, each product has a different profit: product A brings $3,000 USD each and product B $2,500 USD each. The following table summarizes the problem:

| Production Stage | Time for Product A per Worker (Days) | Time for Product B per Worker (Days) | Uptime (Days) | Workers |
|---|---|---|---|---|
| res_1 | 2 | 3 | 30 | 2 |
| res_2 | 3 | 2 | 30 | 2 |
| res_3 | 3 | 3 | 22 | 3 |

To find the objective function, compute the sum of each product quantity multiplied by its profit. It has to be maximized. Although not stated explicitly by the problem, some constraints exist. First is the fact that uptime limits productivity at each stage. Second is the number of workers. Third is productivity relative to the processed product type. You can restate the problem as the sum of the time used to process each product at each stage, which can't exceed the uptime multiplied by the number of workers available. The number of workers multiplied by number of working days provides you the time resources you can use. These resources can't be less than the time it takes to produce all the products you plan to deliver. Here are the resulting formulations with constraints for each stage:

```
objective = 3000 * qty_A + 2500 * qty_B
   production_rate_A * qty_A + production_rate_B * qty_B
     <=  uptime_days * workers
```

You can express each constraint using the quantity of one product to determine the other (in fact, if you produce A, you can't produce B when A's production leaves no time):

```
qty_B <= ((uptime_days * workers) -
   (production_rate_A * qty_A) ) / production_rate_B
```

You can record all the values relative to each stage for `production_rate_A`, `production_rate_B`, `uptime_days`, and `workers` for easier access into a Python dictionary. Keep profits in variables instead. (You can find this code in the `A4D; 19; Linear Programming.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
import numpy as np
import matplotlib.pyplot as plt
import pulp

%matplotlib inline

res_1 = {'A':2, 'B':3, 't':30, 'n':2}
res_2 = {'A':3, 'B':2, 't':30, 'n':2}
res_3 = {'A':3, 'B':3, 't':22, 'n':3}
res = {'res_1':res_1, 'res_2':res_2, 'res_3':res_3}
profit_A = 3000
profit_B = 2500
```

Having framed the problem in a suitable data structure, try to visualize it using the Python plotting functions. Set product A as the abscissa and, because you don't know the solution, represent the production of product A as a vector of quantities ranging from 0 to 30 (quantities can't be negative). As for product B (as seen in the formulations above), derive it from the production remaining after A is done. Formulate three functions, one for each stage, so that as you decide the quantity for A, you get the consequent quantity of B — considering the constraints.

```
a = np.linspace(0, 30, 30)
c1 = ((res['res_1']['t'] * res['res_1']['n'])-
      res['res_1']['A']*a) / res['res_1']['B']
c2 = ((res['res_2']['t'] * res['res_2']['n'])-
      res['res_2']['A']*a) / res['res_2']['B']
c3 = ((res['res_3']['t'] * res['res_3']['n'])-
      res['res_3']['A']*a) / res['res_3']['B']

plt.plot(a, c1, label='constrain #1')
plt.plot(a, c2, label='constrain #2')
plt.plot(a, c3, label='constrain #3')

axes = plt.gca()
axes.set_xlim([0,30])
axes.set_ylim([0,30])
plt.xlabel('qty model A')
plt.ylabel('qty model B')

border = np.array((c1,c2,c3)).min(axis=0)

plt.fill_between(a, border, color='yellow', alpha=0.5)
plt.scatter(*zip(*[(0,0), (20,0),
                   (0,20), (16,6), (6,16)]))
plt.legend()
plt.show()
```

The constraints turn into three lines on a chart, as shown in Figure 19-2. The lines intersect among themselves, showing the feasible area. This is the area delimited by the three lines whose A and B values are always inferior or equal compared to the values on any of the constraint lines. (The constraints represent a frontier; you can't have A or B values beyond them.)

According to the simplex method, the optimal solution is one of the five vertexes of the polygon (which are (0,0), (20,0), (0,20), (16,6), and (6,16)). You can discover which one is the solution by setting the necessary functions from the PuLP package. First, define the problem and call it *model.* By doing so, you determine that it's a maximization problem and that both A and B should be positive.

```
model = pulp.LpProblem("Max profit", pulp.LpMaximize)
A = pulp.LpVariable('A', lowBound=0)
B = pulp.LpVariable('B', lowBound=0)
```

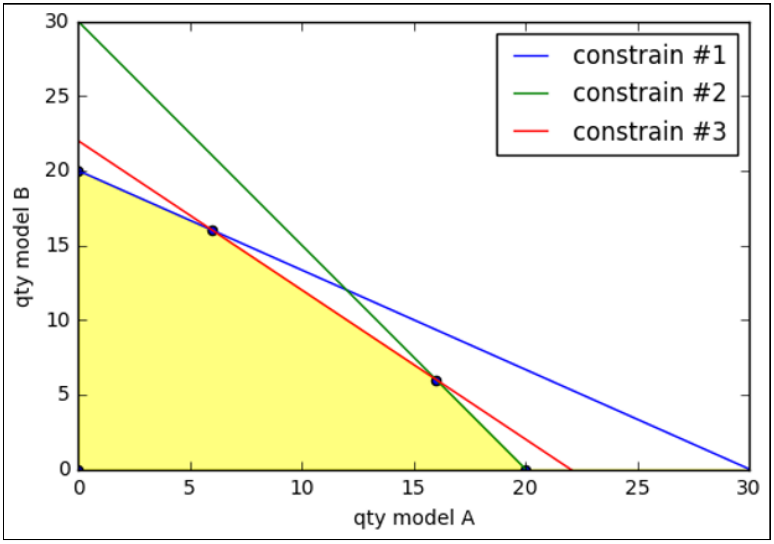


**FIGURE 19-2:** Wondering which vertex is the right one.

TIP

The PuLP solver can also look for integer solutions, something the original simplex can't do. Just add `cat='Integer'` as a parameter when defining a variable: `A = pulp.LpVariable('A', lowBound=0, cat='Integer')`, and you get only whole numbers as a solution. Be aware that in certain problems, integer number results may prove less optimal than the decimal number results; therefore, use an integer solution only if it makes sense for your problem (for instance, you can't produce a fraction of a product).

Next, add the objective function by summing the sum of the two variables defined by `pulp.LpVariable` and representing the ideal quantities of products A and B, multiplied by each unit profit value.

```
model += profit_A * A + profit_B * B
```

Finally, add the constraints, in exactly the same way as the objective function. You create the formulation by using the appropriate values (taken from the data dictionary) and the predefined A and B variables.

```
model += res['res_1']['A'] * A + res['res_1']['B'
              ] * B <= res['res_1']['t'] * res['res_1']['n']
model += res['res_2']['A'] * A + res['res_2']['B'
              ] * B <= res['res_2']['t'] * res['res_2']['n']
model += res['res_3']['A'] * A + res['res_3']['B'
              ] * B <= res['res_3']['t'] * res['res_3']['n']
```

The model is ready to optimize (it has ingested both the objective function and the constraints). Call the `solve` method and then check its status. (Sometimes a solution may prove impossible to find or may not be optimal.)

```
model.solve()
print ('Completion status: %s'
       % pulp.LpStatus[model.status])

Completion status: Optimal
```

Having received confirmation that the optimizer found the optimal solution, you print the related quantities of product A and B.

```
print ("Production of model A = %0.1f" % A.varValue)
print ("Production of model B = %0.1f" % B.varValue)

Production of model A = 16.0
Production of model B = 6.0
```

In addition, you print the resulting total profit achievable by this solution.

```
print ('Maximum profit achieved: %0.1f'
       % pulp.value(model.objective))

Maximum profit achieved: 63000.0
```

IN THIS CHAPTER

- » **Understanding when heuristics are useful to algorithms**
- » **Discovering how pathfinding can be difficult for a robot**
- » **Getting a fast start using the Best-first search**
- » **Improving on Dijkstra's algorithm and taking the best heuristic route using A***

# Chapter 20
# Considering Heuristics

As a concluding topic, this chapter completes the overview of heuristics started in Chapter 18 that describes heuristics as an effective means of using a local search to navigate neighboring solutions. Chapter 18 defines heuristics as educated guesses about a solution — that is, they are sets of rules of thumb pointing to the desired outcome, thus helping algorithms take the right steps toward it; however, heuristics alone can't tell you exactly how to reach the solution.

There are shades of heuristics, just as there can be shades to the truth. Heuristics touch the fringes of algorithm development today. The AI revolution builds on the algorithms presented so far in the book that order, arrange, search, and manipulate data inputs. At the top of the hierarchy are heuristic algorithms that power optimization, as well as searches that determine how machines learn from data and become capable of solving problems autonomously from direct intervention.

Heuristics aren't silver bullets; no solution solves every problem. Heuristic algorithms have serious drawbacks, and you need to know when to use them. In addition, heuristics can lead to wrong conclusions, both for computers and humans. As for humans, biases that save time when evaluating a person or situation can often prove wrong, and even rules of conduct taken from experience obtain the right solution only under certain circumstances. For instance, consider the habit of hitting electric appliances when they don't work. If the problem is a loose connection, hitting the appliance may prove beneficial by reestablishing the electric connection, but you can’t make it a general heuristic because in other cases, that “solution” may prove ineffective or even cause serious damage to the appliance.

# Differentiating Heuristics

The word *heuristic* comes from the ancient Greek *heuriskein*, which meant to invent or discover. Its original meaning underlines the fact that employing heuristics is a practical means of finding a solution that isn’t well defined, but that is found through exploration and an intuitive grasp of the general direction to take. Heuristics rely on the lucky guess or a trial-and-error approach of trying different solutions. A *heuristic algorithm*, which is an algorithm powered by heuristics, solves a problem faster and more efficiently in terms of computational resources by sacrificing solution precision and completeness, in contrast to most algorithms, which have certain output guarantees. When a problem becomes too complex, a heuristic algorithm can represent the only way to obtain a solution.

## Considering the goals of heuristics

Heuristics can speed the long, exhaustive searches performed by other solutions, especially with NP-hard problems that require an exponential number of attempts based on the number of their inputs. For example, consider the traveling salesman problem or variants of the SAT problem, such as the MAX-3SAT (both problems appear in Chapter 18). Heuristics determine the search direction using an estimation, which eliminates a large number of the combinations it would have to test otherwise.

Because a heuristic is an estimate or a guess, it can lead the algorithm that relies on it to a wrong conclusion, which could be an inexact solution or just a suboptimal solution, which is a solution that works but is far from being the best possible. For example, in a numerical estimation, a heuristic might answer that the solution is 41 instead of 42. Other problems often associated with heuristics are the impossibility of finding all best solutions and the variability of time and computations required to reach a solution.

A heuristic provides a perfect match when working with algorithms that would otherwise incur a higher cost when running using other algorithmic techniques. For instance, you can’t solve certain problems without heuristics because of the poor quality and overwhelming number of data inputs. The traveling salesman problem (TSP) is one of these: If you have to tour a large number of cities, you cannot use any exact method. TSP and other problems exclude any exact solution. AI applications fall into this category because many AI problems, such as recognizing spoken words or the content of an image, aren't solvable in an exact sequence of steps and rules.

## Going from genetic to AI

The Chapter 18 local search discussion presents heuristics such as simulated annealing and tabu search, which helps with hill-climbing optimization (not getting stuck with solutions that are less than ideal). Apart from these, the family of heuristics comprises many different applications, among which are the following:

- **Swarm intelligence:** A set of heuristics based on the study of the behavior of insect swarms (such as bees, ants, or fireflies) or particles. The method uses multiple attempts to find a solution using agents (such as running several instances of the same algorithm) that interact cooperatively between themselves and the problem setting. Professor Marco Dorigo, one of the top experts and contributors on the study of swarm intelligence algorithms, provides more information on this topic at `http://www.aco-metaheuristic.org/`.
- **Metaheuristics:** Heuristics that help you determine (or even generate) the right heuristic for your problem. Among metaheuristics, the most widely known are *genetic algorithms,* inspired by natural evolution. Genetic algorithms start with a pool of possible problem solutions and then generate new solutions using mutation (they add or remove something in the solution) and cross-over (they mix parts of different solutions when a solution is divisible). For instance, in the n-Queen problem (Chapter 18), you see that you can split a checkerboard vertically into parts because the Queens do not move horizontally, making it a problem suitable for cross-over. When the pool is enough large, genetic algorithms select the surviving solutions by ruling out those that don't work or lack promise. The selected pool then undergoes another iteration of mutation, cross-over, and selection. After enough time and iterations, genetic algorithms can find solutions that perform better and are completely different from the initial ones.
- **Machine learning:** Approaches such as neuro-fuzzy systems, support vector machines, and neural networks, which are the foundation of how a computer learns to estimate and classify from training examples that are provided as part of datasets of data. Similar to how a child learns by experience, machine learning algorithms determine how to deliver the most plausible answer without using precise rules and detailed rules of conduct. (See *Machine Learning For Dummies,* by John Paul Mueller and Luca Massaron [Wiley], for details on how machine learning works.)
- **Heuristic routing:** A set of heuristics that helps robots (but also found in network telecommunications and logistic transportations) to choose the best path to avoid obstacles when moving around.

# Routing Robots Using Heuristics

Guiding a robot in an unknown environment means avoiding obstacles and reaching a specific target. It's both a fundamental and challenging task in artificial intelligence. Robots can rely on *laser rangefinder, lidar* (devices that allow you to determine the distance to an object by means of a laser ray), or *sonar arrays* (devices that use sounds to see their environment) to navigate their surroundings. Yet, no matter the sophisticated hardware they are equipped with, robots still need proper algorithms to

- Find the shortest path to a destination (or at least a reasonably short one)
- Avoid obstacles on the way
- Perform custom behaviors such as minimizing turning or braking

A *pathfinding* (also called *path planning* or simply *pathing*) algorithm helps a robot start in one location and reach a goal using the shortest path between the two, anticipating and avoiding obstacles along the way (it isn't enough to react after hitting a wall). Pathfinding is also useful when moving any other device to a target in space, even a virtual one, such as in a video game or the pages in the World Wide Web.

REMEMBER

Routing autonomously is a key capability of self-driving cars (SDC), vehicles that can sense the road environment and drive to the destination without any human intervention. (You still need to tell the car where to go; it can't read minds.) This recent article from the *Guardian* newspaper provides a good overview about the developments and expectations for self-driving cars: `https://www.theguardian.com/technology/2015/sep/13/self-driving-cars-bmw-google-2020-driving`.

## Scouting in unknown territories

Pathfinding algorithms achieve all the previously discussed tasks to achieve shortest routing, obstacle avoidance, and other desired behaviors. Algorithms work by using basic schematic maps of their surroundings. These maps are of two kinds:

- **Topological maps:** Simplified diagrams that remove every unnecessary detail. The maps retain key landmarks, correct directions, and some scale proportions for distances. Real-life examples of topological maps include subway maps of Tokyo (`http://www.tokyometro.jp/en/subwaymap/`) and London (`https://tfl.gov.uk/maps/track/tube`).

- **Occupancy grid maps:** These maps divide the surroundings into small, empty squares or hexagons, filling them in when the robot's sensors find an obstacle on the spot they represent. You can see an example of such a map at the Czech Technical University in Prague: `http://cmp.felk.cvut.cz/cmp/demos/Omni/mobil/`. In addition, check out the videos showing how a robot builds and visualizes such a map at `https://www.youtube.com/watch?v=zjl7NmutMIc` and `https://www.youtube.com/watch?v=RhPlzIyTT58`.

You can visualize both topological and occupancy grid maps as graphic diagrams. However, they're best understood by algorithms when rendered into an appropriate data structure. The best data structure for this purpose is the graph because vertexes can easily represent squares, hexagons, landmarks, and waypoints. Edges can connect vertexes in the same way that road, passages, and paths do.

REMEMBER

Your GPS navigation device operates using graphs. Underlying the continuous, detailed, colorful map that the device displays on screen, road maps are elaborated behind the scenes as sets of vertexes and edges traversed by algorithms helping you find the way while avoiding traffic jams.

Representing the robot's territory as a graph re-introduces problems discussed in Chapter 9, which examines how to travel from one vertex to another using the shortest path. The shortest path can be the path that touches the fewest vertexes or the path that costs less (given the sum of the cost of the crossed edge weights, which may represent the length of the edge or some other cost). As when driving your car, you decide not only on the distance driven to reach your destination but also on traffic (roads crowded with traffic or blocked by traffic jams), road conditions, and speed limits that may influence the quality of your journey.

When finding the shortest path to a destination in a graph, the simplest and most basic algorithms in graph theory are depth-first search and Dijkstra's algorithm (described in Chapter 9). Depth-first search explores the graph by going as far as possible from the start and then retracing its steps to explore other paths until it finds the destination. Dijkstra's algorithm explores the graph in a smart and greedy way, considering only the shortest paths. Despite their simplicity, both algorithms are extremely effective when evaluating a simple graph, as in a bird's-eye view, with complete knowledge of the directions you must take to reach the destination and little cost in evaluating the various possible paths.

The situation with a robot is slightly different because it can't perceive all the possible paths at one time, being limited in both visibility and range of sight (obstacles may hide the path or the target may be too far). A robot discovers its environment as it moves and, at best, can assess the distance and direction of its final destination. It's like solving a maze, though not as when playing in a puzzle maze but more akin to immersion in a hedge maze, where you can feel the direction you are taking or you can spot the destination in the distance.

TIP

Hedges are found everywhere in the world. Some of the most famous were built in Europe from the mid-sixteenth century to eighteenth century. In a hedge maze, you can't see where you're going because the hedges are too high. You can perceive direction (if you can see the sun) and even spot the target (see `https://www.venetoinside.com/hidden-treasures/post/maze-of-villa-pisani-in-stra-venice/` as an example). There are also famous hedge mazes in films such as *The Shining* by Stanley Kubrick and in *Harry Potter and the Goblet of Fire*.

## Using distance measures as heuristics

When you can't solve real-life problems in a precise algorithmic way because their input is confused, missing, or unstable, using heuristics can help. When performing path finding using coordinates in a Cartesian plane (flat maps that rely on a set of horizontal and vertical coordinates), two simple measures can provide the distances between two points in that plane: the Euclidean distance and the Manhattan distance.

People commonly use the Euclidean distance because it derives from the Pythagorean Theorem on triangles. If you want to know the distance in line of sight between two points in a plane, say, A and B, and you know their coordinates, you can pretend they're the extremes of the hypotenuse (the longest side in a triangle). As depicted in Figure 20-1, you calculate distance based on the length of the other two sides by creating a third point, C, whose horizontal coordinate is derived from B and whose vertical coordinate is from A.

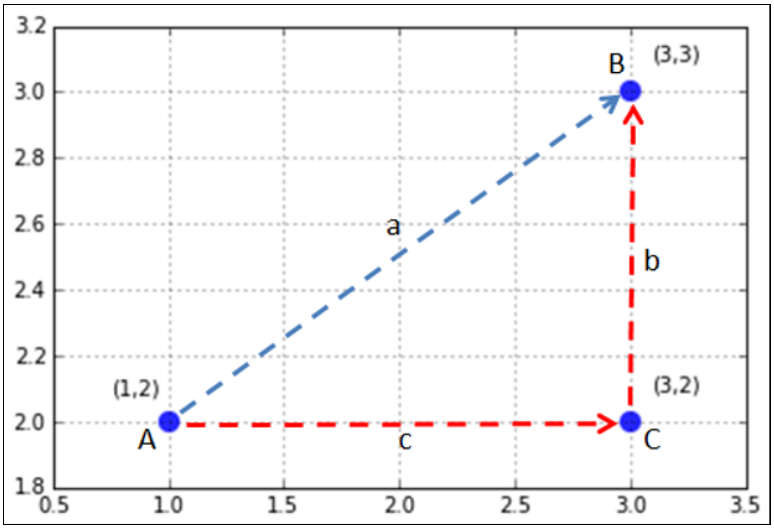


**FIGURE 20-1:** A and B are points on a map's coordinates.

This process translates into taking the difference between the horizontal and vertical coordinates of your two points, squaring both the differences (so that they both become positive), sum them, and finally taking the square root of the result. In this example, going from A to B uses coordinates of (1,2) and (3,3):

```
sqrt((1-3)^2 + (2-3)^2) = sqrt(2^2+1^2) = sqrt(5) = 2.236
```

The Manhattan distance works differently. You begin by summing the lengths of the sides B and C, which equates summing the absolute value of the differences between the horizontal and vertical coordinates of the points A and B.

```
|(1-3)| + |(2-3)| = 2 + 1 = 3
```

The Euclidean distance marks the shortest route, and the Manhattan distance provides the longest yet most plausible route if you expect obstacles when taking a direct route. In fact, the movement represents the trajectory of a taxi in Manhattan (hence the name), moving along a city block to reach its destination (taking the short route through buildings would never work). Other names for this approach are the city block distance or the taxicab distance. Consequently, if you have to move from A to B but you don't know whether you'll find obstacles between them, taking a detour through point C is a good heuristic because that's the distance you expect at worst.

# Explaining Path Finding Algorithms

This last part of the chapter concentrates on explaining two algorithms, best-first search and A* (read as "A star"), both based on heuristics. The following sections demonstrate that both these algorithms provide a fast solution to a maze problem representing a topological or occupancy grid map that's represented as a graph. Both algorithms are widely used in robotics and video gaming.

## Creating a maze

A topological or occupancy grid map resembles a hedge maze, as mentioned previously, especially if obstacles exist between the start and the end of the route. There are specialized algorithms for both creating and processing mazes, most notably the Wall Follower (known since antiquity: You place your hand on a wall and never pull it away until you get out of the maze) or the Pledge algorithm (read more about the seven maze classifications at `http://www.astrolog.org/labyrnth/algrithm.htm`). However, pathfinding is fundamentally different from maze solving because in pathfinding, you know where the target should be, whereas maze-solving algorithms try to solve the problem in complete ignorance of where the exit is.

Consequently, the procedure for simulating a maze of obstacles that a robot has to navigate takes a different and simpler approach. Instead of creating a riddle of obstacles, you create a graph of vertexes arranged in a grid (resembling a map) and randomly remove connections to simulate the presence of obstacles. The graph is undirected (you can traverse each edge in both directions) and weighted because it takes time to move from one vertex to another. In particular, it takes longer to move diagonally than to move upward/downward or left/right.

The first step is to import the necessary Python packages. The code defines the Euclidean and the Manhattan distance functions next. (You can find this code in the `A4D; 20; Heuristic Algorithms.ipynb` file on the Dummies site as part of the downloadable code; see the Introduction for details.)

```
import numpy as np
import string
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

def euclidean_dist(a, b, coord):
    (x1, y1) = coord[a]
    (x2, y2) = coord[b]
    return np.sqrt((x1-x2)**2+(y1-y2)**2)

def manhattan_dist(a, b, coord):
    (x1, y1) = coord[a]
    (x2, y2) = coord[b]
    return abs(x1 - x2) + abs(y1 - y2)

def non_informative(a,b):
    return 0
```

The next step creates a function to generate random mazes. It's based on an integer number seed of your choice that allows you to recreate the same maze every time you provide the same number. Otherwise, maze generation is completely random.

```
def create_maze(seed=2, drawing=True):
    np.random.seed(seed)
    letters = [l for l in string.ascii_uppercase[:25]]
    checkboard = np.array(letters[:25]).reshape((5,5))
    Graph = nx.Graph()
    for j, node in enumerate(letters):
        Graph.add_nodes_from(node)
        x, y = j // 5, j % 5
        x_min = max(0, x-1)
        x_max = min(4, x+1)+1
        y_min = max(0, y-1)
        y_max = min(4, y+1)+1
        adjacent_nodes = np.ravel(
            checkboard[x_min:x_max,y_min:y_max])
        exits = np.random.choice(adjacent_nodes,
            size=np.random.randint(1,4), replace=False)
        for exit in exits:
            if exit not in Graph.edge[node]:
                Graph.add_edge(node, exit)
    spacing = np.arange(0.0, 1.0, 0.2)
    coordinates = [[x,y] for x in spacing \
                   for y in spacing]
    position  = {l:c for l,c in zip(letters, coordinates)}

    for node in Graph.edge:
        for exit in Graph.edge[node]:
            length = int(round(
                    euclidean_dist(
                        node, exit, position)*10,0))
            Graph.add_edge(node,exit,weight=length)

    if drawing:
        nx.draw(Graph, position, with_labels=True)
        labels = nx.get_edge_attributes(Graph,'weight')
        nx.draw_networkx_edge_labels(Graph, position,
                                     edge_labels=labels)
        plt.show()

    return Graph, position
```

The functions return a NetworkX graph (Graph), a favorite data structure for representing graphs, which contains 25 vertexes (or nodes, if you prefer) and a Cartesian map of points (position). The vertexes are placed on a 5 x 5 grid, as shown in Figure 20-2. The output also applies distance functions and calculates the position of vertexes.

```
graph, coordinates = create_maze(seed=3)
```

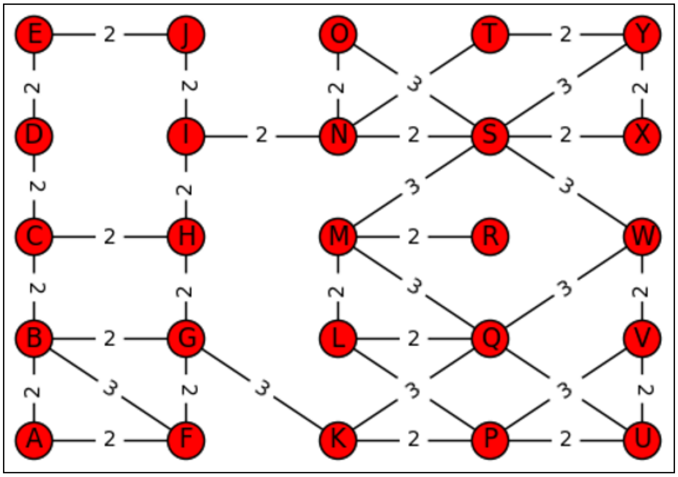


**FIGURE 20-2:** A maze representing a topological map with obstacles.

TIP

In the maze generated by a seed value of 2, every vertex connects with the others. Because the generation process is random, some maps may contain disconnected vertexes, which precludes going between the disconnected vertexes. To see how this works, try a seed value of 13. This actually happens in reality; for example, sometimes a robot can't reach a particular destination.

## Looking for a quick best-first route

The depth-first search algorithm explores the graph by moving from vertex to vertex and adding directions to a stack data structure. When it's time to move, the algorithm moves to the first direction found on the stack. It's like moving through a maze of rooms by taking the first exit you see. Most probably, you arrive at a dead end, which isn't your destination. You then retrace your steps to the previously visited rooms to see whether you encounter another exit, but it takes a long time when you're far from your target.

Heuristics can greatly help with the repetition created by a depth-first search strategy. Using heuristics can tell you whether you're getting nearer or farther from your target. This combination is called the *best-first search* algorithm (BFS). In this case, the *best* in the name hints at the fact that, as you explore the graph, you don't take the first edge in sight, but rather evaluate which edge to take and choose the one that, based on the heuristic, should take you closer to your desired outcome. This behavior resembles greedy optimization (the best first), and some people also call this algorithm *greedy best-first search*. BFS will probably miss the target at first, but because of heuristics, it won't end up far from target and will retrace less than it would if using depth-first search alone.

REMEMBER

You use the BFS algorithm principally in web crawlers that look for certain information on the web. In fact, BFS allows a software agent to move in a mostly unknown graph, using heuristics to detect how closely the content of the next page resembles the initial one (to explore for better content). The algorithm is also widely used in video games, helping characters controlled by the computer move in search of enemies and bounties, thereby resembling a greedy, target-oriented behavior.

Demonstrating BFS in Python using the previously built maze illustrates how a robot can move in a space by seeing it as a graph. The following code shows some general functions, which are also used for the next algorithm in this section. These two functions provide the directions to take from a vertex (`node_neighbors`) and determines the cost of going from one vertex to another (`graph_weight`). Weight represents distance or time.

```
def graph_weight(graph, a, b):
    return graph.edge[a][b]['weight']

def node_neighbors(graph, node):
    return graph.edge[node]
```

The path-planning algorithm simulates robot movement in a graph. When it found a solution, the plan translates into movement. Therefore, path-planning algorithms provide an output telling you how to best move from one vertex to another, you still need a function to translate the information and determine the route to take and calculate trip length. The functions `reconstruct_path` and `compute_path` provide the plan in terms of steps and expected cost when provided with the result from the path-planning algorithm.

```
def reconstruct_path(connections, start, goal):
    if goal in connections:
        current = goal
        path = [current]
        while current != start:
            current = connections[current]
            path.append(current)
        return path[::-1]

def compute_path_dist(path, graph):
    if path:
        run = 0
        for step in range(len(path)-1):
            A = path[step]
            B = path[step+1]
            run += graph_weight(graph, A, B)
        return run
    else:
        return 0
```

Having prepared all the basic functions, the example creates a maze using a seed value of 30. This maze presents two main routes going from vertex A to vertex Y because there are some obstacles in the middle of the map (as shown in Figure 20-3). There are also some dead ends on the way (such as vertexes E and O).

```
graph, coordinates = create_maze(seed=30)
start = 'A'
goal  = 'Y'
scoring=manhattan_dist
```

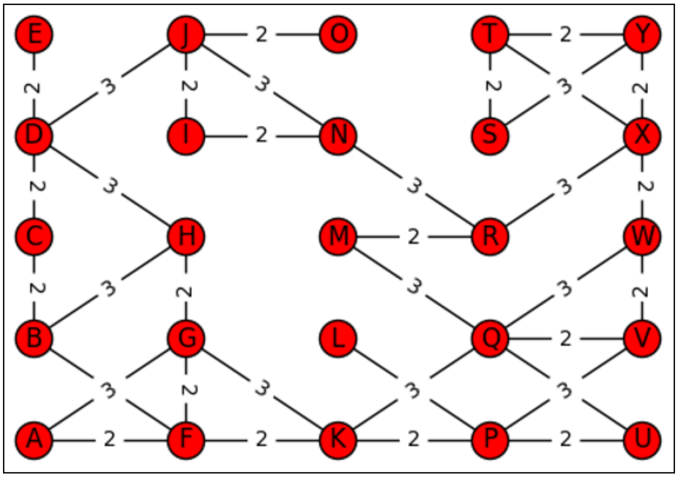


**FIGURE 20-3:** An intricate maze to be solved by heuristics.

The BFS implementation is a bit more complex than the depth-first search code found in Chapter 9. It uses two lists: one to hold the never-visited vertexes (called open_list), and another to hold the visited ones (closed_list). The open_list list acts as a priority queue, one in which a priority determines the first element to extract. In this case, the heuristic provides the priority, thus the priority queue provides a direction that's closer to the target. The Manhattan distance heuristic works best because of the obstacles obstructing the way to the destination:

```
# Best-first search
path = {}
open_list = set(graph.nodes())
```

```
closed_list = {start: manhattan_dist(start, goal,
                                     coordinates)}

while open_list:

    candidates = open_list&closed_list.keys()
    if len(candidates)==0:
        print ("Cannot find a way to the goal %s" % goal)
        break
    frontier = [(closed_list[node],
                 node) for node in candidates]
    score, min_node =sorted(frontier)[0]

    if min_node==goal:
        print ("Arrived at final vertex %s" % goal)
        print ('Unvisited vertices: %i' % (len(
                    open_list)-1))
        break
    else:
        print("Processing vertex %s, " % min_node, end="")

    open_list = open_list.difference(min_node)
    neighbors = node_neighbors(graph, min_node)
    to_be_visited = list(neighbors-closed_list.keys())

    if len(to_be_visited) == 0:
        print ("found no exit, retracing to %s"
               % path[min_node])
    else:
        print ("discovered %s" % str(to_be_visited))

    for node in neighbors:
        if node not in closed_list:
            closed_list[node] = scoring(node, goal,
                                        coordinates)
            path[node] = min_node

print ('\nBest path is:', reconstruct_path(
        path, start, goal))
print ('Length of path: %i' % compute_path_dist(
        reconstruct_path(path, start, goal), graph))
```

```
Processing vertex A, discovered ['F', 'G']
Processing vertex G, discovered ['K', 'H']
Processing vertex H, discovered ['B', 'D']
Processing vertex D, discovered ['E', 'J', 'C']
Processing vertex J, discovered ['O', 'I', 'N']
Processing vertex O, found no exit, retracing to J
Processing vertex N, discovered ['R']
Processing vertex R, discovered ['M', 'X']
Processing vertex X, discovered ['T', 'W', 'Y']
Arrived at final vertex Y
Unvisited vertices: 15

Best path is: ['A', 'G', 'H', 'D', 'J', 'N', 'R', 'X',
 'Y']
Length of path: 22
```

The verbose output from the example tells you how the algorithm works. BFS keeps moving until it runs out of vertexes to explore. When it exhausts the vertexes without reaching the target, the code tells you that it can't reach the target and the robot won't budge. When the code does find the destination, it stops processing vertexes, even if `open_list` still contains vertexes, which saves time.

Finding a dead end, such as ending up in vertex O, means looking for a previously unused route. The best alternative immediately pops up thanks to the priority queue, and the algorithm takes it. In this example, BFS efficiently ignores 15 vertexes and takes the upward route in the map, completing its journey from A to Y in 22 steps.

TIP

You can test other mazes by setting a different seed number and comparing the BFS results with the A* algorithm described in the next section. You'll find that sometimes BFS is both fast and accurate in choosing the best way, sometimes not. If you need a robot that searches quickly, BFS is the best choice.

## Going heuristically around by A*

The A* algorithm speedily produces best shortest paths in a graph by combining the Dijikstra greedy search discussed in Chapter 9 with an early stop (the algorithm stops when it reaches its destination vertex) and a heuristic estimate (usually based on the Manhattan distance) that hints at the graph area to explore first. A* was developed at the Artificial Intelligence Center of Stanford Research Institute (now called SRI International) in 1968 as part of the Shakey the robot project. Shakey was the first mobile robot to autonomously decide how to go somewhere (although it was limited to wandering around a few rooms in the labs). To render

Shakey fully autonomous, its developers came up with the A* algorithm, the Hough transform (an image-processing transformation to detect the edges of an object), and the visibility graph method (a way to represent a path as a graph). The article at `http://www.ai.sri.com/shakey/` describes Shakey in more detail and even shows it in action. It is still surprising watching what Shakey was capable of doing; go to `https://www.youtube.com/watch?v=qXdn6ynwpiI` to take a look. The A* algorithm is currently the best available algorithm when you're looking for the shortest route in a graph and you must deal with partial information and expectations (as captured by the heuristic function guiding the search). A* is able to

- **Find the shortest path solution every time:** The algorithm can do this if such a path exists and if A* is properly informed by the heuristic estimate. A* is powered by the Dijkstra algorithm, which guarantees always finding the best solution.
- **Find the solution faster than any other algorithm:** A* can do this if given access to a fair heuristic — one that provides the right directions to reach the target proximity in a similar, though even smarter, way as BFS.
- **Computes weights when traversing edges:** Weights account for the cost of moving in a certain direction. For example, turning may take longer than going straight, as in the case of Shakey the robot.

REMEMBER

A proper, fair, *admissible* heuristic provides useful information to A* about the distance to the target by never overestimating the cost of reaching it. Moreover, A* makes greater use of its heuristic than BFS, therefore the heuristic must perform calculations quickly or the overall processing time will be too long.

The Python implementation in this example uses the same code and data structures used for BFS, but there are differences between them. The main differences are that as the algorithm proceeds, it updates the cost of reaching from the start vertex to each of the explored vertexes. In addition, when it decides on a route, A* considers the shortest path from the start to the target, passing by the current vertex, because it sums the estimate from the heuristic with the cost of the path computed to the current vertex. This process allows the algorithm to perform more computations than BFS when the heuristic is a proper estimate and to determine the best path possible.

TIP

Finding the shortest path possible in cost terms is the core Dijkstra algorithm function. A* is simply a Dijkstra algorithm in which the cost of reaching a vertex is enhanced by the heuristic of the expected distance to the target. Chapter 9 describes the Dijkstra algorithm in detail. Revisiting the Chapter 9 discussion will help you better understand how A* operates in leveraging heuristics.

```
# A*
open_list = set(graph.nodes())
closed_list = {start: manhattan_dist(
        start, goal, coordinates)}
visited = {start: 0}
path = {}

while open_list:

    candidates = open_list&closed_list.keys()
    if len(candidates)==0:
        print ("Cannot find a way to the goal %s" % goal)
        break
    frontier = [(closed_list[node],
                 node) for node in candidates]
    score, min_node =sorted(frontier)[0]

    if min_node==goal:
        print ("Arrived at final vertex %s" % goal)
        print ('Unvisited vertices: %i' % (len(
                open_list)-1))
        break
    else:
        print("Processing vertex %s, " % min_node, end="")

    open_list = open_list.difference(min_node)
    current_weight = visited[min_node]
    neighbors = node_neighbors(graph, min_node)
    to_be_visited = list(neighbors-visited.keys())

    for node in neighbors:
        new_weight = current_weight + graph_weight(
                     graph, min_node, node)
        if node not in visited or \
        new_weight < visited[node]:
            visited[node] = new_weight
            closed_list[node] = manhattan_dist(node, goal,
                        coordinates) + new_weight
            path[node] = min_node

    if to_be_visited:
        print ("discovered %s" % to_be_visited)
    else:
        print ("getting back to open list")
print ('\nBest path is:', reconstruct_path(
        path, start, goal))
print ('Length of path: %i' % compute_path_dist(
        reconstruct_path(path, start, goal), graph))

Processing vertex A, discovered ['F', 'G']
Processing vertex F, discovered ['B', 'K']
Processing vertex G, discovered  ['H']
Processing vertex K, discovered  ['Q', 'P']
Processing vertex H, discovered  ['D']
Processing vertex B, discovered  ['C']
Processing vertex P, discovered  ['L', 'U', 'V']
Processing vertex Q, discovered  ['M', 'W']
Processing vertex C, getting back to open list
Processing vertex U, getting back to open list
Processing vertex D, discovered  ['E', 'J']
Processing vertex V, getting back to open list
Processing vertex L, getting back to open list
Processing vertex W, discovered  ['X']
Processing vertex E, getting back to open list
Processing vertex M, discovered  ['R']
Processing vertex J, discovered  ['O', 'I', 'N']
Processing vertex X, discovered ['T', 'Y']
Processing vertex R, getting back to open list
Processing vertex O, getting back to open list
Processing vertex I, getting back to open list
Arrived at final vertex Y
Unvisited vertices: 3

Best path is: ['A', 'F', 'K', 'Q', 'W', 'X', 'Y']
Length of path: 14
```

When the A* has completed analyzing the maze, it outputs a best path that's much shorter than the BFS solution. This solution comes at a cost: A* explores almost all the present vertexes, leaving just three vertexes unconsidered. As with Dijkstra, its worst running time is $O(v^2)$, where v is the number of vertexes in the graph; or O(e + v*log(v)), where e is the number of edges, when using min-priority queues, an efficient data structure when you need to obtain the minimum value for a long list. The A* algorithm is not different in its worst running time than Dijkstra's, though on average, it performs better on large graphs because it first finds the target vertex when correctly guided by the heuristic measurement (in the case of a routing robot, the Manhattan distance).

# 6

# The Part of Tens

## IN THIS PART . . .

Consider how algorithms are changing the world.

Discover the future of algorithms.

Define problems that algorithms haven’t solved.

Learn how games help people solve algorithms.

IN THIS CHAPTER

- » Considering sort and search routines
- » Using random numbers
- » Making data smaller
- » Ensuring that data remains secret, and more . . .

# Chapter 21

# Ten Algorithms That Are Changing the World

It's hard to imagine an algorithm doing much of anything, much less changing the world. However, algorithms today appear everywhere, and you might not even realize just how much effect they have on your life.

Most people realize that online stores and other sales venues rely on algorithms to determine which add-on products to suggest based on previous purchases. However, most people are unaware of the uses of algorithms in medicine, many of which help a doctor decide what diagnosis to provide.

Algorithms appear in the oddest places. The timing of traffic lights often depends on the calculations of algorithms. Algorithms will help your smartphone talk to you today, and you see algorithms at work in making your television do more than any television has done in the past. Consequently, it's not all that impossible to believe that algorithms are poised to change the world. This chapter highlights ten of them.

REMEMBER

For algorithm purists, you can say that the algorithm has changed the world throughout the centuries, so nothing has really changed for thousands of years. The Babylonians used algorithms to perform factorization and find square roots as early as 1600 BC. Al-Khawarizmi described algorithms to solve both linear and quadratic equations around 820. This chapter focuses on computer-based algorithms, but algorithms have been around for a long time.

## Using Sort Routines

Without ordered data, most of the world would come to a stop. To use data, you must be able to find it. You can find hundreds of sort algorithms explained on sites such as `https://betterexplained.com/articles/sorting-algorithms/` and as part of this book (see Chapter 7).

However, the three most common sort routines are Mergesort, Quicksort, and Heapsort because of the superior speed they provide (see the time comparisons at `http://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html`). The sort routine that works best for your application depends on the following:

- What you expect the application to do
- The kind of data you work with
- The computing resources you have available

The point is that the capability to sort data into whatever form an application needs to accomplish a task makes the world run, and this capability is changing how the world works.

Some businesses today thrive as a result of the sort algorithm. For example, consider the fact that Google exists because it helps people find things, and this ability resides substantially in the capability to sort data to make it readily accessible. Consider just how hard it would be to find an item on Amazon without the sort routine. Even that recipe application on your computer at home relies heavily on sort routines to keep the data it contains in order. In fact, it probably wouldn't be too much of a stretch to say that any substantial application relies heavily on sort routines.

## Looking for Things with Search Routines

As with sort routines, search routines appear in nearly every application of any size today. The applications appear everywhere, even in places that you might not think too much about, such as your car. Finding information quickly is an essential part of daily life. For example, imagine being late for an appointment and suddenly discovering that your GPS can't find the address you need. As with sort routines, search routines come in all shapes and sizes, and you can find them described on sites such as `https://tekmarathon.com/2012/10/05/best-searching-algorithm-2/` and `http://research.cs.queensu.ca/home/cisc121/2006s/webnotes/search.html`. In fact, if anything, there are more search routines than sort routines because search requirements are often more strenuous and complex. You find a lot of search routines discussed in this book as well (see Chapter 7).

## Shaking Things Up with Random Numbers

All sorts of things would be a lot less fun without randomness. For example, imagine starting Solitaire and seeing precisely the same game every time you start it. No one would play such a game. Consequently, random number generation is an essential part of the gaming experience. In fact, as expressed in a number of chapters in this book, some algorithms actually require some level of randomness to work properly (see the "Arranging caching computer data" section of Chapter 15 as an example). You also find that testing works better when using random values in some cases (see the "Choosing a particular kind of compression" section of Chapter 14 as an example).

REMEMBER

The numbers that you obtain from an algorithm are actually pseudo-random, which means that you can potentially predict the next number in a series by knowing the algorithm and the seed value used to generate the number. That's why this information is so closely guarded.

TECHNICAL STUFF

Not all applications and not all computers rely on pseudo-random numbers generated by algorithms (the vast majority do, however). Computer hardware-based methods of creating random numbers exist, such as relying on atmospheric noise or temperature changes (see `http://engineering.mit.edu/ask/can-computer-generate-truly-random-number` for details). In fact, you can get a hardware-based random number solution, such as ChaosKey (`http://altusmetrum.org/ChaosKey/`) and plug it into your USB slot to generate what likely are true random numbers. The interesting thing about the ChaosKey site is that it provides you with a schematic to show how it collects random noise and changes it into a random number.

# Performing Data Compression

Chapter 14 discusses data compression techniques and uses the kind of compression that you normally find used for files. However, data compression affects every aspect of computing today. For example, most graphics, video, and audio files rely on data compression. Without data compression, you couldn't possibly obtain the required level of throughput to make tasks such as streamed movies work.

However, data compression finds even more uses than you might expect. Just about every Database Management System (DBMS) relies on data compression to make data fit in a reasonable amount of space on disk. Cloud computing wouldn't work without data compression because it downloading items from the cloud to local machines would take too long. Even web pages often rely on data compression to get information from one place to another.

# Keeping Data Secret

The concept of keeping data secret isn't new. In fact, it's one of the oldest reasons to use an algorithm of some sort. The word cryptography actually comes from two Greek words: *kryptós* (hidden or secret) and *graphein* (writing). In fact, the Greeks were probably the first users of cryptography, and ancient texts report that Julius Caesar used encrypted missives to communicate with his generals. The point is, keeping data secret is one of the longest running battles in history. The moment one party finds a way to keep a secret, someone else finds a way to make the secret public by breaking the cryptography. General uses for computer-driven cryptography today include:

- **Confidentiality:** Ensuring that no one can see information exchanged between two parties.
- **Data integrity:** Reducing the likelihood that someone or something can change the content of data passed between two parties.
- **Authentication:** Determining the identity of one or more parties.
- **Nonrepudiation:** Reducing the ability of a party to say he or she didn't commit a particular act.

TIP

Because keeping a secret when using computers, the history of computer-based cryptographic algorithms is long and interesting. You can find a list of commonly used algorithms (both present and historical) at `http://www.cryptographyworld.com/algo.htm` and `https://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/crypto.html`. The guide at `https://www.owasp.org/index.php/Guide_to_Cryptography` provides additional details on how cryptography works.

## Changing the Data Domain

The Fourier Transform and Fast Fourier Transform (FFT) make a huge difference in how applications perceive data. These two algorithms transform data from the frequency domain (how fast a signal oscillates) to the time domain (the time differential between signal changes). In fact, it's impossible to get any sort of computer hardware degree without having spent time working with these two algorithms extensively. Timing is everything.

REMEMBER

By knowing how often something changes, you can figure out the time interval between changes and therefore know how long you have to perform a task before a change in state requires that you do something else. These algorithms commonly see use in filters of all sorts. Without the filtering effects of these algorithms, reproducing video and audio faithfully through a streamed connection would be impossible. All these applications sound rather advanced, and they are, but some amazing tutorials give you a better sense of how these algorithms work (see the tutorial at `http://w.astro.berkeley.edu/~jrg/ngst/fft/fft.html` as an example). The tutorial at `https://betterexplained.com/articles/an-interactive-guide-to-the-fourier-transform/` is possibly the most interesting and especially entertaining if you like smoothies.

## Analyzing Links

The capability to analyze relationships is something that has made modern computing unique. In fact, the capability to first create a representation of these relationships and then analyze them is the subject of Part III of this book. The whole idea of the web, in fact, is to create connections, and connectivity was a consideration at the start of what has become a worldwide phenomenon. Without the capability to analyze and utilize links, applications such as databases and e-mail wouldn't work. You couldn't communicate well with friends on Facebook.

As the web has matured and people have become more in tune with devices that make connectivity both simpler and ubiquitous, applications such as Facebook and sales sites such as Amazon have made greater use of link analysis to do things like sell you more products. Of course, some of this connectivity has a negative outcome (see `http://www.pcmag.com/commentary/351623/facebook-a-tool-for-evil` as an example), but for the most part, link analysis does make it possible for people to remain better informed and in better contact with the world around them.

Of course, link analysis does more than inform in a connected sort of way. Consider the use of link analysis to provide driving directions or to find casual links between human activity and disease. Link analysis enables you to see the connection between things that you might not ordinarily consider but that do have an impact on your daily live. Because of link analysis, you might live longer because a doctor can advise you on which habits to change to correct issues that could become problems later. The point is that connections exist everywhere, and link analysis offers a method to determine where these connections exist and whether they're actually important.

## Spotting Data Patterns

Data doesn't exist in a vacuum. All sorts of factors affect data, including biases that color how humans perceive data. Chapter 10 starts a discussion of how data tends to cluster in certain environments and how analysis of these clusters can tell you all sorts of things about the data.

TIP

Pattern analysis is at the forefront of some of the more amazing uses of computers today. For example, the Viola–Jones object detection framework makes real-time facial recognition possible. This algorithm could enable people to create better security in places like airports where nefarious individuals currently ply their trade. Similar algorithms could help your doctor detect cancers of various sorts long before the cancer is actually visible to the human eye. Earlier detection makes a full recovery a higher probability. The same holds true for all sorts of other medical problems (such as finding bone fractures that are currently too small to see but cause pain nonetheless).

You also find pattern recognition used for more mundane purposes. For example, pattern analysis lets people detect potential traffic problems before they occur. It's also possible to use pattern analysis to help farmers grow more food at a lower cost by applying water and fertilizer only when necessary. The use of pattern recognition can also help move drones around fields so that the farmer becomes more time efficient and can work more land at a lower cost. Without algorithms, these sorts of patterns, which have such a high impact on daily life, can't be recognized.

## Dealing with Automation and Automatic Responses

The proportional integral derivative algorithm is quite a mouthful. Just try saying it three times fast! However, it's one of the most important secret algorithms you've never heard about, yet rely on every day. This particular algorithm relies on a control loop feedback mechanism to minimize the error between the desired output signal and the real output signal. You see it used all over the place to control automation and automatic responses. For example, when your car goes into a skid because you break too hard, this algorithm helps ensure that the Automatic Breaking System (ABS) actually works as intended. Otherwise, the ABS could overcompensate and make matters worse.

Just about every form of machinery today uses the proportional integral derivative algorithm. In fact, robotics wouldn't be possible without it. Imagine what would happen to a factory if all of the robots constantly overcompensated for every activity in which they engaged. The resulting chaos would quickly convince owners to stop using machines for any purpose whatsoever.

## Creating Unique Identifiers

It seems as if we're all just a number. Actually, not just one number — lots and lots of numbers. Each of our credit cards has a number, as does our driver license, as does our government identifier, as do all sorts of other businesses and organizations. People actually have to keep lists of all of the numbers because they simply have too many to track. Yet, each of these numbers must identify the person uniquely to some party. Behind all of this uniqueness are various kinds of algorithms.

Chapter 7 discusses hashes, which are one way to ensure uniqueness. Underlying both hashes and cryptography is integer factorization, a kind of algorithm that breaks really large numbers into prime numbers. In fact, integer factorization is one of the hardest kinds of problems to solve with algorithms, but people are working on the problem all the time. So much of society today depends on your ability to identify yourself uniquely that the hidden secrets of creating these identifiers is an essential part of a modern world.

IN THIS CHAPTER

- Performing text searches easily
- Detecting differences in individual words
- Considering the feasibility of hypercomputers
- Employing one-way functions, and more . . .

# Chapter 22

# Ten Algorithmic Problems Yet to Solve

Algorithms have indeed been around for centuries, so you'd think that scientists would have discovered and solved every algorithm by now. Unfortunately, the opposite is true. Solving a particular algorithm often presents a few more questions that the algorithm doesn't solve and that didn't seem apparent until someone did come up with the solution. In addition, changes in technologies and lifestyle often present new challenges that call for yet more algorithms. For example, the connected nature of society and the use of robots have both increased the need for new algorithms.

As presented in Chapter 1, algorithms are a series of steps used to solve a problem, and you shouldn't confuse them with other entities, such as equations. An algorithm is never a solution in search of a problem. No one would create a series of steps to solve a problem that doesn't yet exist (or may never exist). In addition, many problems are interesting but have no pressing need for a solution. Consequently, even though everyone knows about the problem and understands that someone might want a solution for it, no one is in a hurry to create the solution.

This chapter is about algorithmic problems that would serve a purpose should someone find a solution for them. In short, the reason you need to care about this chapter is that you might find a problem that you'd really like to solve and might even decide to become part of the team that solves it.

## Dealing with Text Searches

Many text searches involve the use of regular expressions — a sort of shorthand that tells the computer what to find. The grammar used for the regular expression depends on the language or application, but you find regular expressions used in a number of places, including word processors, email applications, search dialogs, and in all sorts of other places where you need to provide precise search terms for a range of text items. You can read more about regular expressions at `http://www.regular-expressions.info/`.

REMEMBER

One of the current problems with regular expressions is that it seems as if every application environment has a similar set of rules, but with just enough differences to make creating a search term hard. The generalized star-height problem seeks to discover whether a generalized regular expression syntax exists. If so, the resulting algorithm would make it possible for someone to learn just one method of creating regular expressions to perform searches. You can read more about this problem at `https://www.irif.fr/~jep/Problemes/starheight.html`.

## Differentiating Words

When working with characters, a computer sees numbers, not letters. The numbers are actually just a series of 0s and 1s to the computer and don't actually have any meaning. Combining characters into strings just makes the series of 0s and 1s longer. Consequently, comparing two strings, something that a human can do at a glance, can take time within a computer, and confusion is likely between conjugates. For example, unless you're careful in constructing the algorithm, a computer could confuse *enlist* and *listen.* More important, the computer would require time to discern the difference between the two. The separating words problem seeks to find the smallest (and fastest) possible algorithm (a deterministic finite automaton, DFN, in this case) to perform word separation. The goal is to accept one word and reject another, given two words of a particular length.

## Determining Whether an Application Will End

One of the problems that Alan Turing proposed in 1936 is the issue of whether an algorithm, given a description of a program and an input, could determine whether the program would eventually halt (the *halting problem*). When working with a simple application, it's easy to determine in many cases whether the program will halt or continue running in an endless loop. However, as program complexity increases, determining the result of running the program with any given input becomes harder. A Turing machine can't make this determination; the result is buggy code with infinite loops. No amount of testing that uses current technology can solve this issue.

REMEMBER

A *hypercomputer* is a computing model that goes beyond the Turing machine to solve problems such as the halting problem. However, such machines aren't possible using current technology. If they were possible, you would be able to ask them all kinds of imponderables that computers can't currently answer. The article at `https://www.newscientist.com/article/mg22329781-500-what-will-hypercomputers-let-us-do-good-question/` provides you with a good idea of what would happen if someone were able to solve this problem.

## Creating and Using One-Way Functions

A one-way function is a function that is easy to use to obtain an answer in one direction, but nearly impossible to use with the inverse of that answer. In other words, you use a one-way function to create something like a hash that would appear as part of a solution for cryptography, personal identification, authentication, or other data security needs.

REMEMBER

The existence of a one-way function is less mystery and more a matter of proof. Many telecommunications, e-commerce, and e-banking systems currently rely on functions that are purportedly one way, but no one truly knows whether they really are one way. The existence of a one-way function is currently a hypothesis, not a theory (see an explanation of the difference between the two at `http://www.diffen.com/difference/Hypothesis_vs_Theory`). If someone were able to prove that a one-way function exists, data security issues would be easier to solve from a programming perspective.

## Multiplying Really Large Numbers

Really large numbers exist in many places. For example, consider performing the calculations involving distances to Mars, or perhaps Pluto. Methods currently do exist for performing multiplication on really large numbers, but they tend to be slow because they require multiple operations to complete. The problem occurs when the numbers are too large to fit in the processor's registers. At that point, the multiplication must occur in more than one step, which slows things considerably. The current solutions include:

- Gauss's complex multiplication algorithm
- Karatsuba multiplication
- Toom-Cook
- Fourier transform methods

Even though many of the methods currently available produce acceptable results, they all take time, and when you have a lot of calculations to perform, the time problem can become critical. Consequently, large number multiplication is one of those problems that requires a better solution than those available today.

## Dividing a Resource Equally

Dividing resources equally may not seem hard, but humans, being the envious sort, might see the resource as being unequally divided unless you can find a way to assure everyone that the division is indeed fair. This is the envy-free cake-cutting problem. Of course, when you cut a cake, no matter how fairly you attempt to do it, there is always the perception that the division is unfair. Creating a fair division of resources is important in daily life to minimize strife between stakeholders in any organization, making everyone more efficient.

TIP

Two solutions already exist for the envy-free cake-cutting problem with a specific number of people, but no general solution exists. When there are two people involved, the first cuts the cake and the second chooses the first piece. In this way, both parties are assured of an equal division. The problem becomes harder with three people, but you can find the Selfridge-Conway solution for the problem at `https://ochronus.com/cutting-the-pie` (even though the site discusses pie, the process is the same). However, after you get to four people, no solution exists.

# Reducing Edit Distance Calculation Time

The *edit distance* between two strings is the number of operations required to transform one string into the other string. The distance calculation revolves around the Levenshtein distance operations, which are the removal, insertion, or substitution of a character in the string. This particular technique sees use in natural language interfaces, DNA sequence quantification, and all sorts of other places where you can have two similar strings that require some sort of comparison or modification.

A number of solutions for this problem currently exist, all of them quite slow. In fact, most of them take exponential time, so the time required to perform a transformation quickly adds up to the point where humans can see pauses in the processing of input. The pause isn't quite so bad when using a word processor that performs automatic word checks and changes a misspelled word into the correct one. However, when using voice interfaces, the pause can become quite noticeable and cause the human operator to make mistakes. The current goal is to allow edit distance calculation in subquadratic time: $O(n^{2-\epsilon})$.

# Solving Problems Quickly

As machine learning takes off and we count more and more on computers to solve problems, the issue of how quickly a computer can solve a problem becomes critical. The P versus NP problem simply asks whether a computer can solve a problem quickly when it can verify the solution to the problem quickly. In other words, if the computer can reasonably ascertain that a human response to a problem is correct in polynomial time or less, can it also solve the problem itself in polynomial time or less?

This question was originally discussed in the 1950s by John Nash in letters to the National Security Agency (NSA) and again in letters between Kurt Gödel and John von Neumann. In addition to machine learning (and AI in general), this particular problem is a concern to many other fields, including mathematics, cryptography, algorithm research, game theory, multimedia processing, philosophy, and economics.

## Playing the Parity Game

At first, solving a game might not seem all that useful in real life. Yes, games are fun and interesting, but they don't really provide a background for doing anything useful — at least, that's the general theory. However, game theory does come into play in a large number of real-life scenarios, many of which involve complex processes that someone can understand more easily as games than as actual processes. In this case, the game helps people understand automated verification and controller synthesis, among other things. You can read more about the parity game at `http://www.sciencedirect.com/science/article/pii/S0890540115000723`. In fact, you can play it if you'd like at `https://www.abefehr.com/parity/`.

## Understanding Spatial Issues

To put this particular problem into context, think about moving boxes around in a warehouse or some other situations in which you need to consider the space in which things move. Obviously, if you have many boxes in a big warehouse and they all require a forklift to pick up, you don't want to try to figure out how to store them optimally by physically rearranging them. This is where you need to work through the problem by visualizing a solution.

However, the question is whether all spatial problems have a solution. In this case, think about one of those kids' puzzles that has you putting a picture together by sliding the little tiles around. It seems as if a solution should exist in all cases, but in some situations, a bad starting point can result in a situation that has no solution. You can find a discussion of such a problem at `http://math.stackexchange.com/questions/754827/does-a-15-puzzle-always-have-a-solution`.

TIP

Mathematicians such as Sam Loyd (see `https://www.mathsisfun.com/puzzles/sam-loyd-puzzles-index.html`) often use puzzles to demonstrate complex math problems, some of which have no solution today. Visiting these sites is fun because you not only get some free entertainment but also, food for thought. The issues that these puzzles raise do have practical applications, but they're presented in a fun way.

# Index

# About the Authors

**John Mueller** is a freelance author and technical editor. He has writing in his blood, having produced 102 books and more than 600 articles to date. The topics range from networking to artificial intelligence and from database management to heads-down programming. Some of his current books include a book on Python for beginners, Python for data science, and a book about MATLAB. He has also written books about Amazon Web Services for Administrators, web application security, HTML5 development with JavaScript, and CSS3. His technical editing skills have helped more than 63 authors refine the content of their manuscripts. John has provided technical editing services to various technical magazines as well. It was during his time with *Data Based Advisor* that John was first exposed to MATLAB, and he has continued to follow the progress in MATLAB development ever since. During his time at Cubic Corporation, John was exposed to reliability engineering and has continued his interest in probability. Be sure to read John's blog at `http://blog.johnmuellerbooks.com/`.

When John isn't working at the computer, you can find him outside in the garden, cutting wood, or generally enjoying nature. John also likes making wine, baking cookies, and knitting. You can reach John on the Internet at `John@JohnMuellerBooks.com`. John is also setting up a website at `http://www.johnmuellerbooks.com/`. Feel free to take a look and make suggestions on how he can improve it.

**Luca Massaron** is a data scientist and marketing research director who is specialized in multivariate statistical analysis, machine learning, and customer insight, with more than a decade of experience in solving real-world problems and generating value for stakeholders by applying reasoning, statistics, data mining, and algorithms. Passionate about everything regarding data and analysis and about demonstrating the potentiality of data-driven knowledge discovery to both experts and non-experts, Luca is the coauthor of *Python for Data Science For Dummies* and *Machine Learning For Dummies.* Favoring simplicity over unnecessary sophistication, he believes that a lot can be achieved by understanding in simple terms and practicing the essentials of any discipline.

# John's Dedication

This book is dedicated to Julie Thrond, a friend who is both faithful and kind. May she always win out against the struggles in life.

# Luca's Dedication

I dedicate this book to my wife, Yukiko, whom I always find curious and ready to marvel at the wonders of this amazing and baffling world. Stay curious. Enjoy the surprise of both little and great things alike. Do not have fear of anything or let foolish details bother you. Stay young at heart forever. With love.

# John’s Acknowledgments

Thanks to my wife, Rebecca. Even though she is gone now, her spirit is in every book I write, in every word that appears on the page. She believed in me when no one else would.

Russ Mullen deserves thanks for his technical edit of this book. He greatly added to the accuracy and depth of the material you see here. Russ worked exceptionally hard helping with the research for this book by locating hard-to-find URLs and also offering a lot of suggestions. Matt Wagner, my agent, deserves credit for helping me get the contract in the first place and taking care of all the details that most authors don’t really consider. I always appreciate his assistance. It’s good to know that someone wants to help.

A number of people read all or part of this book to help me refine the approach, test scripts, and generally provide input that all readers wish they could have. These unpaid volunteers helped in ways too numerous to mention here. I especially appreciate the efforts of Eva Beattie, Glenn A. Russell, Alberto Boschetti, Cristian Mastrofrancesco, Diego Paladini, Dimitris Papadopoulos, Matteo Malosetti, Sebastiano Di Paola, Warren B, and Zacharias Voulgaris, who provided general input, read the entire book, and selflessly devoted themselves to this project.

Finally, I would like to thank Katie Mohr, Susan Christophersen, and the rest of the Wiley editorial and production staff for their unparalleled support of this writing effort.

# Luca’s Acknowledgments

My first greatest thanks to my family, Yukiko and Amelia, for their support, sacrifices, and loving patience during the long days/nights, weeks, and months I’ve been involved in working on this book.

I thank all the editorial and production staff at Wiley, in particular Katie Mohr and Susan Christophersen, for their great professionalism and support in all the phases of writing this book of the *For Dummies* series.

### Publisher’s Acknowledgments

**Acquisitions Editor:** Katie Mohr
**Project and Copy Editor:** Susan Christophersen
**Technical Editor:** Russ Mullen
**Editorial Assistant:** Serena Novosel
**Sr. Editorial Assistant:** Cherie Case
**Production Editor:** Selvakumaran Rajendiran
**Cover Image:** © Nobi_Prizue/iStockphoto