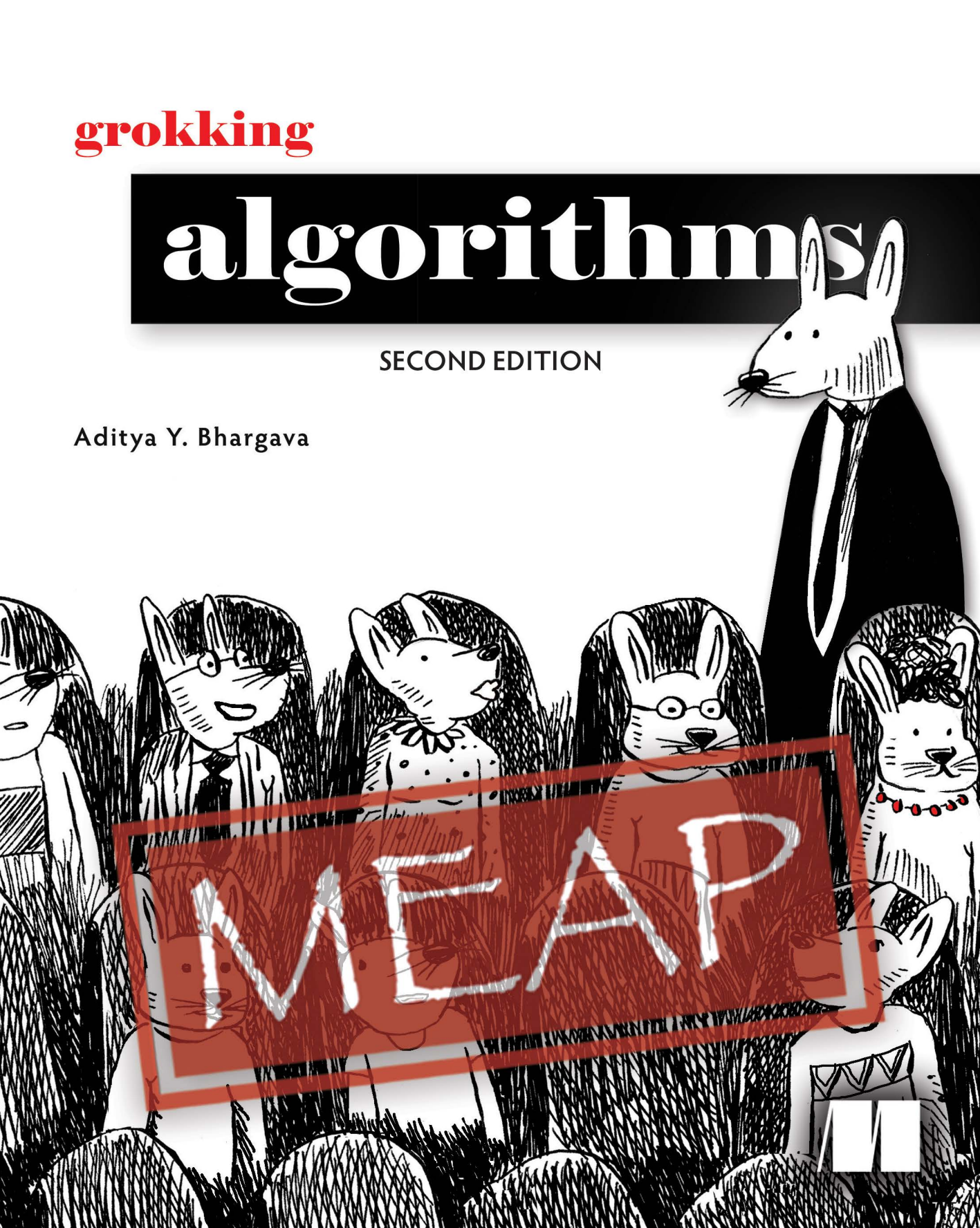
grokking algorithms

SECOND EDITION

Aditya Y. Bhargava



MEAP Edition
Manning Early Access Program
Grokking Algorithms
Second Edition
Version 1

For more information on this and other Manning titles go to
manning.com

welcome

Dear Reader,

Welcome to the second edition of *Grokking Algorithms*, an introductory book on algorithms that will teach you the fundamental concepts in a friendly and easy-to-understand manner.

When I had written the first edition of this book back in 2016, my goal was to write in an easy-to-read manner and make the learning journey as smooth as possible. Since then, almost 100,000 people have read this book.

With this second edition, my goal remains the same. In this book, I use illustrations and memorable examples to make concepts stick. The book is designed for readers who know how to code and want to learn more about algorithms without any math knowledge required.

I believe that understanding algorithms is essential for every programmer, whether you are building websites, mobile apps, or working on complex data analysis. Algorithms are everywhere, from search engines to self-driving cars, and they provide us with powerful tools to solve problems and create efficient solutions.

Throughout this book, you will learn about data structures, graphs, recursion, and a lot more. I'll show you techniques for solving problems programmers face every day, with practical examples. In addition to some tweaks throughout, and some additions to existing chapters, this second edition adds two chapters on trees, a topic many readers have requested. Trees are a fundamental data structure. I cover binary search trees, AVL trees, and B-trees. There are also new notes on performance. For example, the first edition discusses trade-offs between arrays and linked lists. There I had said linked lists waste less memory, but the reality is more nuanced. Arrays are often more memory efficient. I dive deeper into nuances like this in the second edition. I have also extended the section on NP completeness, and added an explanation on NP hard versus NP complete.

This book is meant to be enjoyable to read. I hope that you will find it interesting and engaging. Please feel free to provide your feedback, comments, and questions in the liveBook discussion forum. Your input is essential in making this book the best possible resource for learning algorithms.

Thank you for choosing *Grokking Algorithms* (second edition), and I wish you happy learning.

Best regards,
—Aditya Bhargava

brief contents

1 Introduction to algorithms

In this chapter

- You get a foundation for the rest of the book
- You write your first search algorithm (binary search)
- You learn how to talk about the running time of an algorithm (Big O notation)

Introduction

An *algorithm* is a set of instructions for accomplishing a task. Every piece of code could be called an algorithm, but this book covers the more interesting bits. I chose the algorithms in this book for inclusion because they're fast, or they solve interesting problems, or both. Here are some highlights:

- Chapter 1 talks about binary search and shows how an algorithm can speed up your code. In one example, the number of steps needed goes from 4 billion down to 32!
- A GPS device uses graph algorithms (as you'll learn in chapters 6, 7, and 8) to calculate the shortest route to your destination.
- You can use dynamic programming (discussed in chapter 9) to write an AI algorithm that plays checkers.

In each case, I'll describe the algorithm and give you an example. Then I'll talk about the running time of the algorithm in Big O notation. Finally, I'll explore what other types of problems could be solved by the same algorithm.

What you'll learn about performance

The good news is, an implementation of every algorithm in this book is probably available in your favorite language, so you don't have to write each algorithm yourself! But those implementations are useless if you don't understand the trade-offs. In this book, you'll learn to compare trade-offs between different algorithms: Should you use merge sort or quicksort? Should you use an array or a list? Just using a different data structure can make a big difference.

What you'll learn about solving problems

You'll learn techniques for solving problems that might have been out of your grasp until now. For example:

- If you like making video games, you can write an AI system that follows the user around using graph algorithms.
- You'll learn to make a recommendations system using k-nearest neighbors.
- Some problems aren't solvable in a timely manner! The part of this book that talks about NP-complete problems shows you how to identify those problems and come up with an algorithm that gives you an approximate answer.

More generally, by the end of this book, you'll know some of the most widely applicable algorithms. You can then use your new knowledge to learn about more specific algorithms for AI, databases, and so on. Or you can take on bigger challenges at work.

What you need to know

You'll need to know basic algebra before starting this book. In parti-cular, take this function: f(x) = x × 2. What is f(5)? If you answered 10, you're set.

Additionally, this chapter (and this book) will be easier to follow if you're familiar with one programming language. All the examples in this book are in Python. If you don't know any programming languages and want to learn one, choose Python—it's great for beginners. If you know another language, like Ruby, you'll be fine.

Binary search

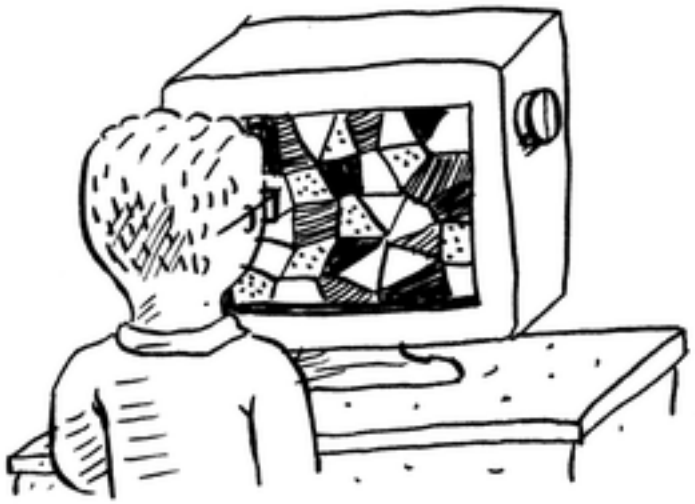
Suppose you're searching for a person in the phone book (what an old-fashioned sentence!). Their name starts with *K*. You could start at the beginning and keep flipping pages until you get to the *K*s. But you're more likely to start at a page in the middle, because you know the *K*s are going to be near the middle of the phone book.

Or suppose you're searching for a word in a dictionary, and it starts with *O*. Again, you'll start near the middle.

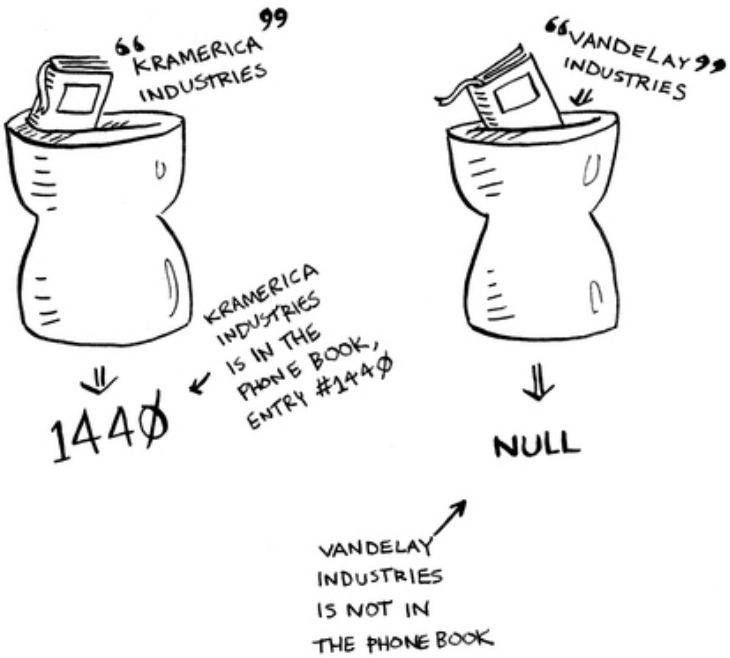
Now suppose you log on to Facebook. When you do, Facebook has to verify that you have an account on the site. So, it needs to search for your username in its database. Suppose your username is karlmageddon. Facebook could start from the *A*s and search for your name—but it makes more sense for it to begin somewhere in the middle.

This is a search problem. And all these cases use the same algorithm to solve the problem: *binary search*.

Binary search is an algorithm; its input is a sorted list of elements (I'll explain later why it needs to be sorted). If an element you're looking for is in that list, binary search returns the position where it's located. Otherwise, binary search returns `null`.

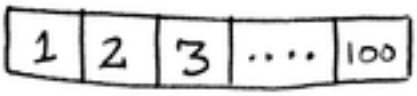


For example:



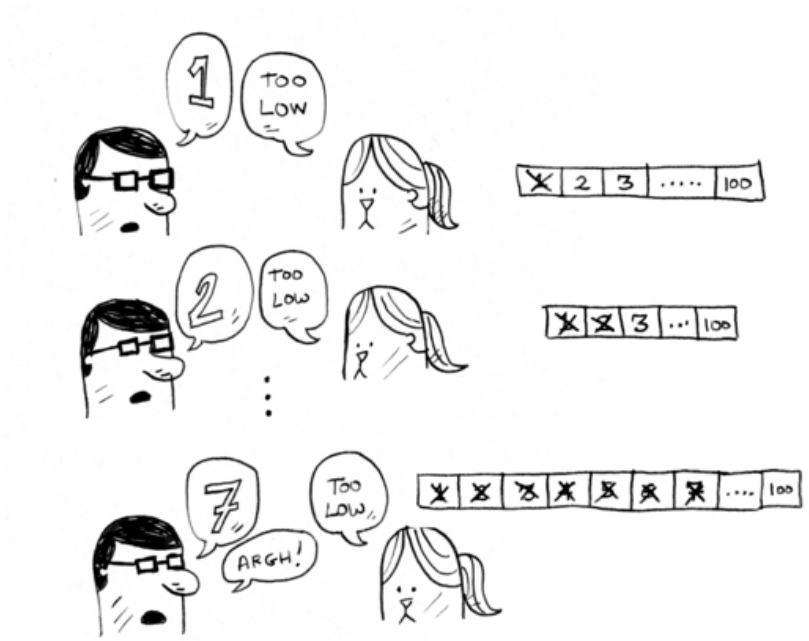
Looking for companies in a phone book with binary search

Here's an example of how binary search works. I'm thinking of a number between 1 and 100.



You have to try to guess my number in the fewest tries possible. With every guess, I'll tell you if your guess is too low, too high, or correct.

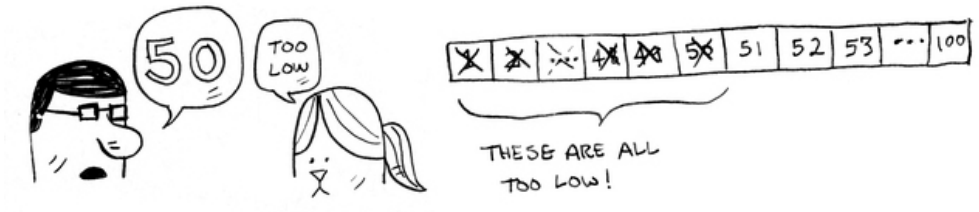
Suppose you start guessing like this: 1, 2, 3, 4 …. Here's how it would go.



A bad approach to number guessing

This is *simple search* (maybe *stupid search* would be a better term). With each guess, you're eliminating only one number. If my number was 99, it could take you 99 guesses to get there!

A better way to search

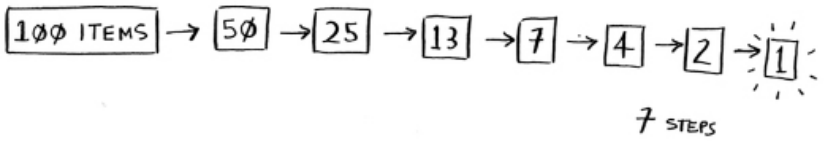


Here's a better technique. Start with 50.

Too low, but you just eliminated *half* the numbers! Now you know that 1–50 are all too low. Next guess: 75.

Too high, but again you cut down half the remaining numbers! *With binary search, you guess the middle number and eliminate half the remaining numbers every time*. Next is 63 (halfway between 50 and 75).

This is binary search. You just learned your first algorithm! Here's how many numbers you can eliminate every time.

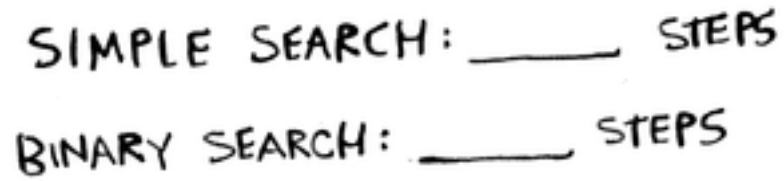


Eliminate half the numbers every time with binary search.

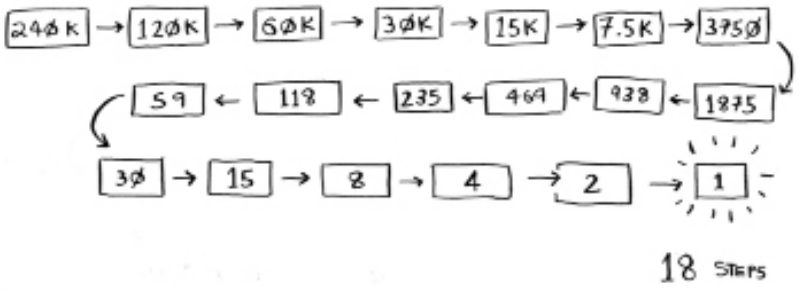
Whatever number I'm thinking of, you can guess in a maximum of seven guesses—because you eliminate so many numbers with every guess!

Suppose you're looking for a word in the dictionary. The dictionary has 240,000 words. *In the worst case*, how many

steps do you think each search will take?



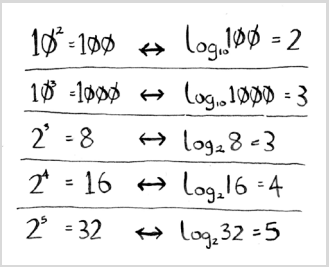
Simple search could take 240,000 steps if the word you're looking for is the very last one in the book. With each step of binary search, you cut the number of words in half until you're left with only one word.



So binary search will take 18 steps—a big difference! In general, for any list of *n*, binary search will take $\log_2 n$ steps to run in the worst case, whereas simple search will take *n* steps.

Logarithms

You may not remember what logarithms are, but you probably know what exponentials are. $\log_{10} 100$ is like asking, "How many 10s do we multiply together to get 100?" The answer is 2: 10 × 10. So $\log_{10} 100 = 2$. Logs are the inverse of exponentials.



Logs are the inverse of exponentials.

In this book, when I talk about running time in Big O notation (explained a little later), log always means $\log_2$. When you search for an element using simple search, in the worst case you might have to look at every single element. So for a list of 8 numbers, you'd have to check 8 numbers at most. For binary search, you have to check log *n* elements in the worst case. For a list of 8 elements, log 8 == 3, because $2^3 == 8$. So for a list of 8 numbers, you would have to check 4 numbers at most. For a list of 1,024 elements, log 1,024 = 10, because $2^{10} == 1{,}024$. So for a list of 1,024 numbers, you'd have to check 10 numbers at most.

> **NOTE** I'll talk about log time a lot in this book, so you should understand the concept of logarithms. If you don't, Khan Academy (khanacademy.org) has a nice video that makes it clear.

> **NOTE** Binary search only works when your list is in sorted order. For example, the names in a phone book are sorted in alphabetical order, so you can use binary search to look for a name. What would happen if the names weren't sorted?

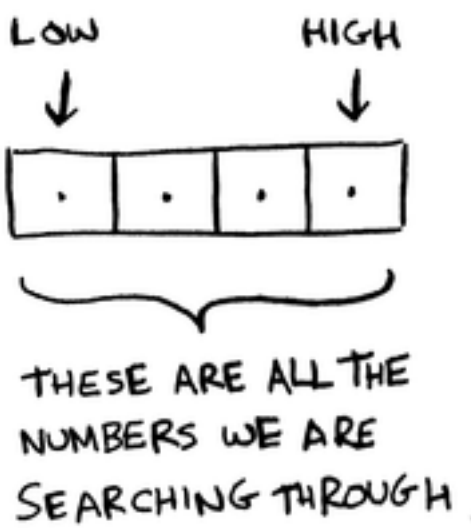
Let's see how to write binary search in Python. The code

sample here uses arrays. If you don't know how arrays work, don't worry; they're covered in the next chapter. You just need to know that you can store a sequence of elements in a row of consecutive buckets called an array. The buckets are numbered starting with 0: the first bucket is at position #0, the second is #1, the third is #2, and so on.

> **NOTE** You will see me use the terms list and array interchangeably in the code. This is because in Python, arrays are called lists.

The `binary_search` function takes a sorted array and an item. If the item is in the array, the function returns its position. You'll keep track of what part of the array you have to search through. At the beginning, this is the entire array:

```
low = 0
high = len(arr) - 1
```



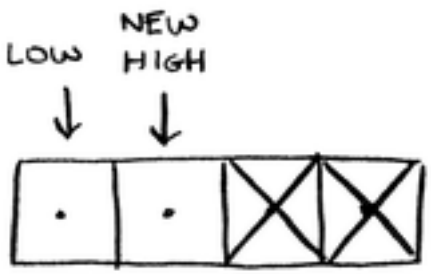
Each time, you check the middle element:

```
mid = (low + high) // 2     #A
guess = arr[mid]
```

#A mid is rounded down by Python automatically if (low + high) isn't an even number.

If the guess is too low, you update `low` accordingly:

```
if guess < item:
  low = mid + 1
```



And if the guess is too high, you update `high`. Here's the full code:

```
def binary_search(arr, item):
  low = 0                          #A
  high = len(arr)-1                #A

  while low <= high:               #B
    mid = (low + high) // 2        #C
    guess = arr[mid]
    if guess == item:              #D
      return mid
    elif guess > item:             #E
      high = mid - 1
    else:                          #F
      low = mid + 1
  return None                      #G

my_list = [1, 3, 5, 7, 9]   #H

print(binary_search(my_list, 3)) # => 1          #I
print(binary_search(my_list, -1)) # => None      #J
```

#A low and high keep track of which part of the list you'll search in.

#B While you haven't narrowed it down to one element …

#C … check the middle element.

#D Found the item.

```
#E The guess was too high.
#F The guess was too low.
#G The item doesn't exist.
#H Let's test it!
#I Remember, lists start at 0. The second slot has index 1.
#J "None" means nil in Python. It indicates that the item wasn't found.
```

Exercises

1. 1.1 Suppose you have a sorted list of 128 names, and you're searching through it using binary search. What's the maximum number of steps it would take?
2. 1.2 Suppose you double the size of the list. What's the maximum number of steps now?

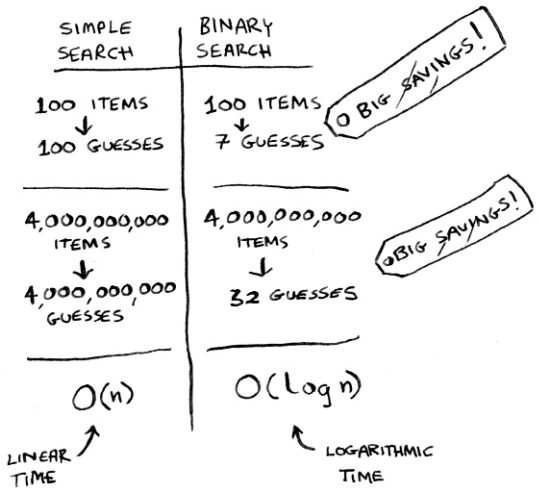
Running time

Any time I talk about an algorithm, I'll discuss its running time. Generally you want to choose the most efficient algorithm—whether you're trying to optimize for time or space.

Back to binary search. How much time do you save by using it? Well, the first approach was to check each number, one by one. If this is a list of 100 numbers, it takes up to 100 guesses. If it's a list of 4 billion numbers, it takes up to 4 billion guesses. So the maximum number of guesses is the same as the size of the list. This is called *linear time*.

Binary search is different. If the list is 100 items long, it takes at most 7 guesses. If the list is 4 billion items, it takes at most 32 guesses. Powerful, eh? Binary search runs in

logarithmic time (or *log time*, as most people call it). Here's a table summarizing our findings today.



Run times for search algorithms

Big O notation

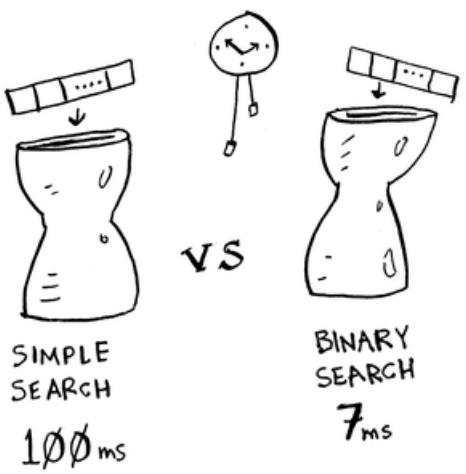
Big O notation is special notation that tells you how fast an algorithm is. Who cares? Well, it turns out that you'll use other people's algorithms often—and when you do, it's nice to understand how fast or slow they are. In this section, I'll explain what Big O notation is and give you a list of the most common running times for algorithms using it.

Algorithm running times grow at different rates

Bob is writing a search algorithm for NASA. His algorithm will kick in when a rocket is about to land on the Moon, and it will help calculate where to land.

This is an example of how the run time of two algorithms can grow at different rates. Bob is trying to decide between simple search and binary search. The algorithm needs to be both fast and correct. On one hand, binary search is faster. And Bob has only *10 seconds* to figure out where to land—otherwise, the rocket will be off course. On the other hand, simple search is easier to write, and there is less chance of bugs being introduced. And Bob *really* doesn't want bugs in the code to land a rocket! To be extra careful, Bob decides to time both algorithms with a list of 100 elements.

Let's assume it takes 1 millisecond to check one element. With simple search, Bob has to check 100 elements, so the search takes 100 ms to run. On the other hand, he only has to check 7 elements with binary search ($\log_2$ 100 is roughly 7), so that search takes 7 ms to run. But realistically, the list will have more like a billion elements. If it does, how long will simple search take? How long will binary search take? Make sure you have an answer for each question before reading on.



Running time for simple search vs. binary search, with a list of 100 elements

Bob runs binary search with 1 billion elements, and it takes 30 ms ($\log_2$ 1,000,000,000 is roughly 30). "30 ms!" he thinks. "Binary search is about 15 times faster than simple search, because simple search took 100 ms with 100 elements, and binary search took 7 ms. So simple search will take 30 × 15 = 450 ms, right? Way under my threshold of 10 seconds." Bob decides to go with simple search. Is that the right choice?

No. Turns out, Bob is wrong. Dead wrong. The run time for simple search with 1 billion items will be 1 billion ms, which is 11 days! The problem is, the run times for binary search and simple search *don't grow at the same rate*.

	SIMPLE SEARCH	BINARY SEARCH
100 ELEMENTS	100 ms	7 ms
10,000 ELEMENTS	10 seconds	14 ms
1,000,000,000 ELEMENTS	11 days	30 ms

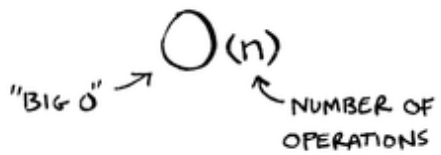
Run times grow at very different speeds!

That is, as the number of items increases, binary search

takes a little more time to run. But simple search takes a *lot* more time to run. So as the list of numbers gets bigger, binary search suddenly becomes a *lot* faster than simple search. Bob thought binary search was 15 times faster than simple search, but that's not correct. If the list has 1 billion items, it's more like 33 million times faster. That's why it's not enough to know how long an algorithm takes to run—you need to know how the running time increases as the list size increases. That's where Big O notation comes in.

Big O notation tells you how fast an algorithm is. For example, suppose you have a list of size *n*. Simple search needs to check each element, so it will take *n* operations. The run time in Big O notation is O(*n*). Where are the seconds? There are none—Big O doesn't tell you the speed in seconds. *Big O notation lets you compare the number of operations*. It tells you how fast the algorithm grows.

Here's another example. Binary search needs log *n* operations to check a list of size *n*. What's the running time in Big O notation? It's O(log *n*). In general, Big O notation is written as follows.



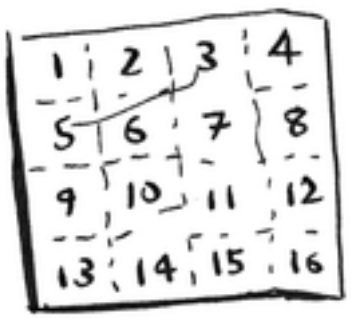
What Big O notation looks like

This tells you the number of operations an algorithm will make. It's called Big O notation because you put a "big O" in front of the number of operations (it sounds like a joke, but it's true!).

Now let's look at some examples. See if you can figure out the run time for these algorithms.

Visualizing different Big O run times

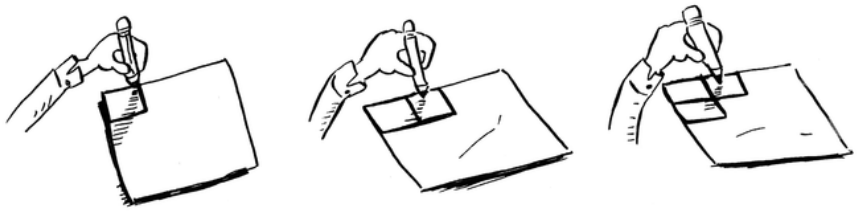
Here's a practical example you can follow at home with a few pieces of paper and a pencil. Suppose you have to draw a grid of 16 boxes.



What's a good algorithm to draw this grid?

ALGORITHM 1

One way to do it is to draw 16 boxes, one at a time. Remember, Big O notation counts the number of operations. In this example, drawing one box is one operation. You have to draw 16 boxes. How many operations will it take, drawing one box at a time?



Drawing a grid one box at a time

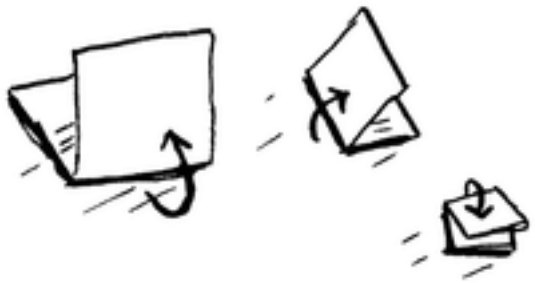
It takes 16 steps to draw 16 boxes. What's the running time for this algorithm?

ALGORITHM 2

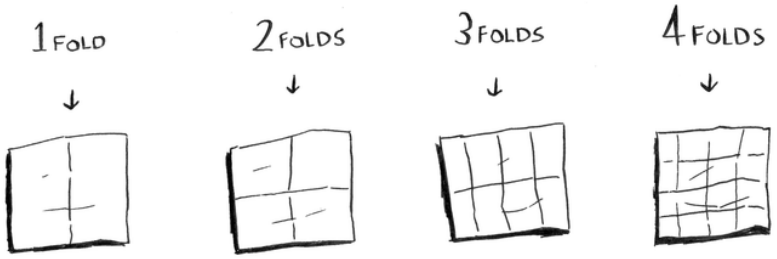
Try this algorithm instead. Fold the paper.

In this example, folding the paper once is an operation. You just made two boxes with that operation!

Fold the paper again, and again, and again.



Unfold it after four folds, and you'll have a beautiful grid! Every fold doubles the number of boxes. You made 16 boxes with 4 operations!



Drawing a grid in four folds

You can "draw" twice as many boxes with every fold, so you can draw 16 boxes in 4 steps. What's the running time for this algorithm? Come up with running times for both algorithms before moving on.

Answers: Algorithm 1 takes O(*n*) time, and algorithm 2 takes O(log *n*) time.

Big O establishes a worst-case run time

Suppose you're using simple search to look for a person in the phone book. You know that simple search takes O(*n*) time to run, which means in the worst case, you'll have to look through every single entry in your phone book. In this case, you're looking for Adit. This guy is the first entry in

your phone book. So you didn't have to look at every entry—you found it on the first try. Did this algorithm take O(*n*) time? Or did it take O(1) time because you found the person on the first try?

Simple search still takes O(*n*) time. In this case, you found what you were looking for instantly. That's the best-case scenario. But we are using Big O notation for *worst-case* scenario analysis. So you can say that, in the *worst case*, you'll have to look at every entry in the phone book once. That's O(*n*) time. It's a reassurance—you know that simple search will never be slower than O(*n*) time.

> **NOTE** Along with the worst-case run time, it's also important to look at the average-case run time. Worst case versus average case is discussed in chapter 4.

Some common Big O run times

Here are five Big O run times that you'll encounter a lot, sorted from fastest to slowest:

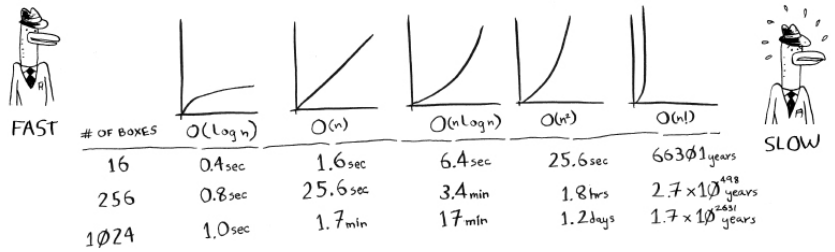
- O(log *n*), also known as *log time*. Example: Binary search.
- O(*n*), also known as *linear time*. Example: Simple search.
- O(*n* * log *n*). Example: A fast sorting algorithm, like quicksort (coming up in chapter 4).
- O(n^2). Example: A slow sorting algorithm, like selection sort (coming up in chapter 2).
- O(*n*!). Example: A really slow algorithm, like the traveling salesperson (coming up next!).

Suppose you're drawing a grid of 16 boxes again, and you can choose from 5 different algorithms to do so. If you use the first algorithm, it will take you O(log *n*) time to draw the grid. You can do 10 operations per second. With O(log *n*) time, it will take you 4 operations to draw a grid of 16 boxes (log 16 is 4). So it will take you 0.4 seconds to draw

the grid. What if you have to draw 1,024 boxes? It will take you log 1,024 = 10 operations, or 1 second to draw a grid of 1,024 boxes. These numbers are using the first algorithm.

The second algorithm is slower: it takes O(*n*) time. It will take 16 operations to draw 16 boxes, and it will take 1,024 operations to draw 1,024 boxes. How much time is that in seconds?

Here's how long it would take to draw a grid for the rest of the algorithms, from fastest to slowest:



There are other run times, too, but these are the five most common.

This is a simplification. In reality you can't convert from a Big O run time to a number of operations this neatly, but this is good enough for now. We'll come back to Big O notation in chapter 4, after you've learned a few more algorithms. For now, the main takeaways are as follows:

- Algorithm speed isn't measured in seconds, but in growth of the number of operations.
- Instead of seconds, we talk about how quickly the run time of an algorithm increases as the size of the input increases.
- Run time of algorithms is expressed in Big O notation.
- O(log *n*) is faster than O(*n*), but it gets a lot faster as the list of items you're searching grows.

Exercises

Give the run time for each of these scenarios in terms of

Big O.

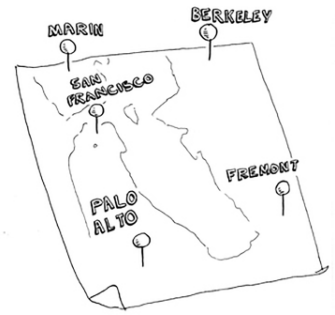
1. 1.3 You have a name, and you want to find the person's phone number in the phone book.
2. 1.4 You have a phone number, and you want to find the person's name in the phone book. (Hint: You'll have to search through the whole book!)
3. 1.5 You want to read the numbers of every person in the phone book.
4. 1.6 You want to read the numbers of just the As. (This is a tricky one! It involves concepts that are covered more in chapter 4. Read the answer—you may be surprised!)

The traveling salesperson

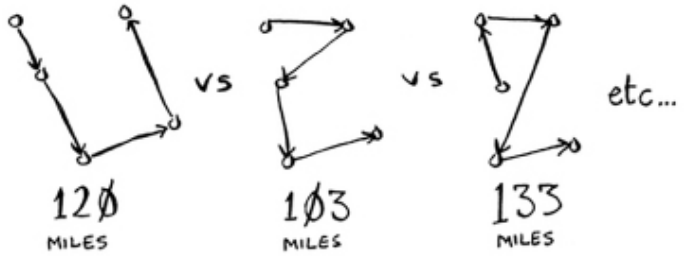
You might have read that last section and thought, "There's no way I'll ever run into an algorithm that takes O(*n*!) time." Well, let me try to prove you wrong! Here's an example of an algorithm with a really bad running time. This is a famous problem in computer science, because its growth is appalling and some very smart people think it can't be improved. It's called the *traveling salesperson* problem.

You have a salesperson.

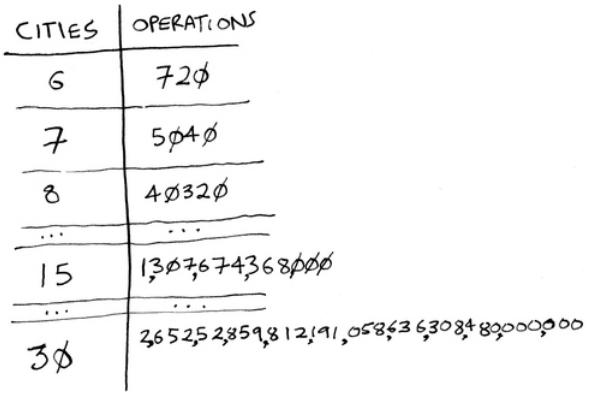
The salesperson has to go to five cities.



This salesperson, whom I'll call Opus, wants to hit all five cities while traveling the minimum distance. Here's one way to do that: look at every possible order in which he could travel to the cities.



He adds up the total distance and then picks the path with the lowest distance. There are 120 permutations with 5 cities, so it will take 120 operations to solve the problem for 5 cities. For 6 cities, it will take 720 operations (there are 720 permutations). For 7 cities, it will take 5,040 operations!



The number of operations increases drastically.

In general, for *n* items, it will take *n*! (*n* factorial) operations to compute the result. So this is O(*n*!) time, or *factorial time*. It takes a lot of operations for everything except the smallest numbers. Once you're dealing with 100+ cities, it's impossible to calculate the answer in time—the Sun will collapse first.

This is a terrible algorithm! Opus should use a different one, right? But he can't. This is one of the unsolved problems in computer science. There's no fast known algorithm for it, and some smart people think it's *impossible* to have a smart algorithm for this problem. The best we can do is come up with an approximate solution; see chapter 10 for more.

Recap

- Binary search is a lot faster than simple search as your array gets bigger.
- O(log *n*) is faster than O(*n*), but it gets a lot faster once the list of items you're searching through grows.smat
- Algorithm speed isn't measured in seconds.
- Algorithm times are measured in terms of *growth* of an algorithm.
- Algorithm times are written in Big O notation.

Selection sort

2

In this chapter

- You learn about arrays and linked lists—two of the most basic data structures. They're used absolutely everywhere. You already used arrays in chapter 1, and you'll use them in almost every chapter in this book. Arrays are a crucial topic, so pay attention! But sometimes it's better to use a linked list instead of an array. This chapter explains the pros and cons of both so you can decide which one is right for your algorithm.

- You learn your first sorting algorithm. A lot of algorithms only work if your data is sorted. Remember binary search? You can run binary search only on a sorted list of elements. This chapter teaches you selection sort. Most languages have a sorting algorithm built in, so you'll rarely need to write your own version from scratch. But selection sort is a stepping stone to quicksort, which I'll cover in chapter 4. Quicksort is an important algorithm, and it will be easier to understand if you know one sorting algorithm already

What you need to know

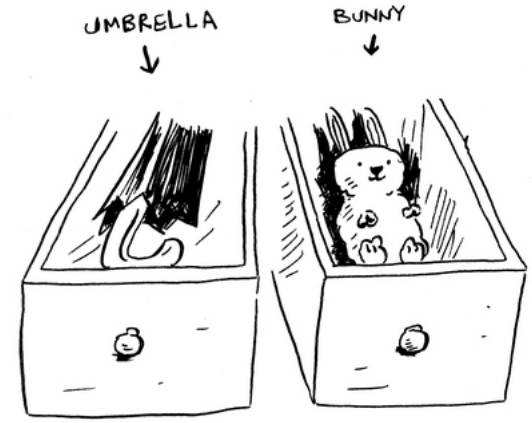
To understand the performance analysis bits in this chapter, you need to know Big O notation and logarithms. If you don't know those, I suggest you go back and read chapter 1. Big O notation will be used throughout the rest of the book.

How memory works

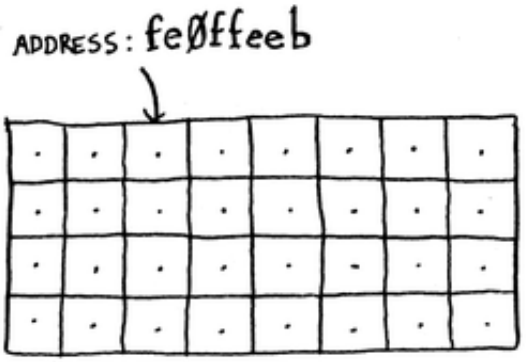
Imagine you go to a show and need to check your things. A chest of drawers is available.

Each drawer can hold one element. You want to store two things, so you ask for two drawers.

You store your two things here.



And you're ready for the show! This is basically how your computer's memory works. Your computer looks like a giant set of drawers, and each drawer has an address.

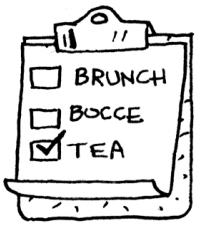


`fe0ffeeb` is the address of a slot in memory.

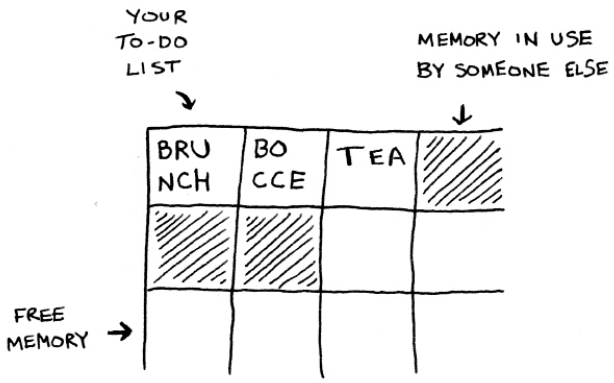
Each time you want to store an item in memory, you ask the computer for some space, and it gives you an address where you can store your item. If you want to store multiple items, there are two basic ways to do so: arrays and linked lists. I'll talk about arrays and lists next, as well as the pros and cons of each. There isn't one right way to store items for every use case, so it's important to know the differences.

Arrays and linked lists

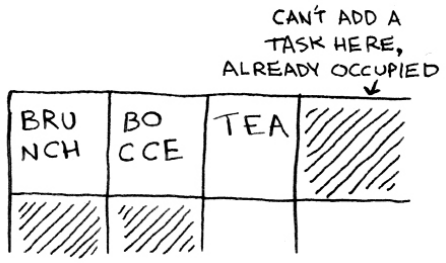
Sometimes you need to store a list of elements in memory. Suppose you're writing an app to manage your todos. You'll want to store the todos as a list in memory.



Should you use an array, or a linked list? Let's store the todos in an array first, because it's easier to grasp. Using an array means all your tasks are stored contiguously (right next to each other) in memory.



Now suppose you want to add a fourth task. But the next drawer is taken up by someone else's stuff!



It's like going to a movie with your friends and finding a place to sit—but another friend joins you, and there's no place for them. You have to move to a new spot where you all fit. In this case, you need to ask your computer for a different chunk of memory that can fit four tasks. Then you need to move all your tasks there.

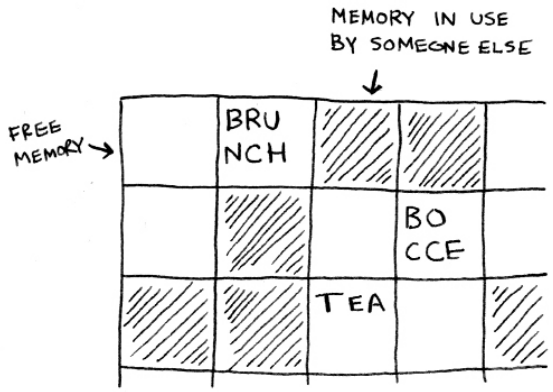
If another friend comes by, you're out of room again—and you all have to move a second time! What a pain. Similarly, adding new items to an array can be a big pain. If you're out of space and need to move to a new spot in memory every time, adding a new item will be really slow. One easy fix is to "hold seats": even if you have only 3 items in your task list, you can ask the computer for 10 slots, just in case. Then you can add 10 items to your task list without having to move. This is a good workaround, but you should be aware of a couple of downsides:

- You may not need the extra slots that you asked for, and then that memory will be wasted. You aren't using it, but no one else can use it either.
- You may add more than 10 items to your task list and have to move anyway.

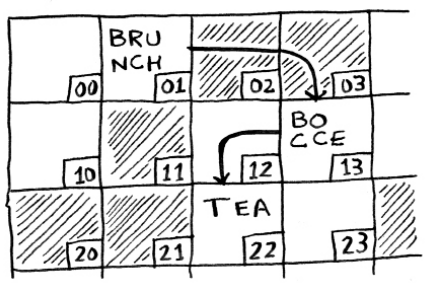
So it's a good workaround, but it's not a perfect solution. Linked lists solve this problem of adding items.

Linked lists

With linked lists, your items can be anywhere in memory.



Each item stores the address of the next item in the list. A bunch of random memory addresses are linked together.



Linked memory addresses

It's like a treasure hunt. You go to the first address, and it says, "The next item can be found at address 123." So you go to address 123, and it says, "The next item can be found at address 847," and so on. Adding an item to a linked list is easy: you stick it anywhere in memory and store the address with the previous item.

With linked lists, you never have to move your items. You also avoid another problem. Let's say you go to a popular movie with five of your friends. The six of you are trying to find a place to sit, but the theater is packed. There aren't six

seats together. Well, sometimes this happens with arrays. Let's say you're trying to find 10,000 slots for an array. Your memory has 10,000 slots, but it doesn't have 10,000 slots together. You can't get space for your array! A linked list is like saying, "Let's split up and watch the movie." If there's space in memory, you have space for your linked list.

If linked lists are so much better at inserts, what are arrays good for?

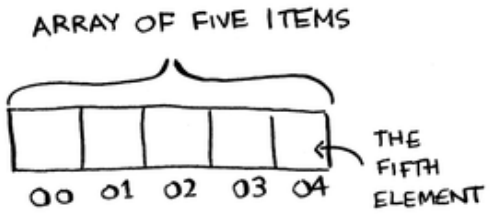
Arrays

Websites with top-10 lists use a scummy tactic to get more page views. Instead of showing you the list on one page, they put one item on each page and make you click Next to get to the next item in the list. For example, Top 10 Best TV Villains won't show you the entire list on one page. Instead, you start at #10 (Newman), and you have to click Next on each page to reach #1 (Gustavo Fring). This technique gives the websites 10 whole pages on which to show you ads, but it's boring to click Next 9 times to get to #1. It would be much better if the whole list was on one page and you could click each person's name for more info.

Linked lists have a similar problem. Suppose you want to read the last item in a linked list. You can't just read it, because you don't know what address it's at. Instead, you have to go to item #1 to get the address for item #2. Then you have to go to item #2 to get the address for item #3. And so on, until you get to the last item. Linked lists are great if you're going to read all the items one at a time: you can read one item, follow the address to the next item, and

so on. But if you're going to keep jumping around, linked lists are terrible.

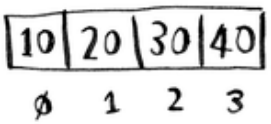
Arrays are different. You know the address for every item in your array. For example, suppose your array contains five items, and you know it starts at address 00. What is the address of item #5?



Simple math tells you: it's 04. Arrays are great if you want to read random elements, because you can look up any element in your array instantly. With a linked list, the elements aren't next to each other, so you can't instantly calculate the position of the fifth element in memory—you have to go to the first element to get the address to the second element, then go to the second element to get the address of the third element, and so on until you get to the fifth element.

Terminology

The elements in an array are numbered. This numbering starts from 0, not 1. For example, in this array, 20 is at position 1.

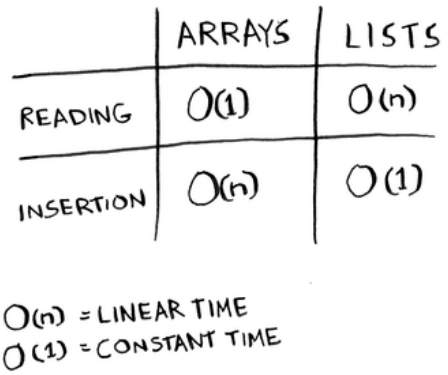


And 10 is at position 0. This usually throws new programmers for a spin. Starting at 0 makes all kinds of array-based code easier to write, so programmers have stuck with it. Almost every programming language you use will number array elements starting at 0. You'll soon get

used to it.

The position of an element is called its *index*. So instead of saying, "20 is at *position* 1," the correct terminology is, "20 is at *index* 1." I'll use *index* to mean *position* throughout this book.

Here are the run times for common operations on arrays and lists.



Question: Why does it take O(*n*) time to insert an element into an array? Suppose you wanted to insert an element at the beginning of an array. How would you do it? How long would it take? Find the answers to these questions in the next section!

Exercise

1. 2.1 Suppose you're building an app to keep track of your finances.

   Every day, you write down everything you spent money on. At the end of the month, you review your expenses

and sum up how much you spent. So, you have lots of inserts and a few reads. Should you use an array or a list?

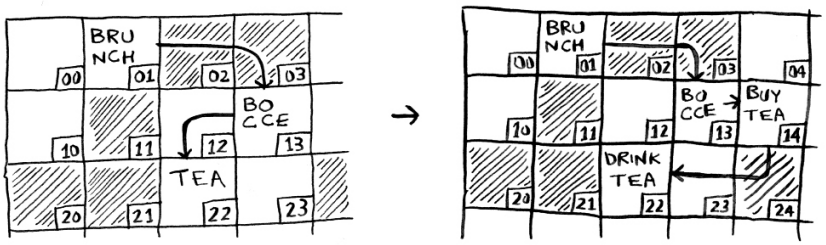
Inserting into the middle of a list

Suppose you want your todo list to work more like a calendar. Earlier, you were adding things to the end of the list.

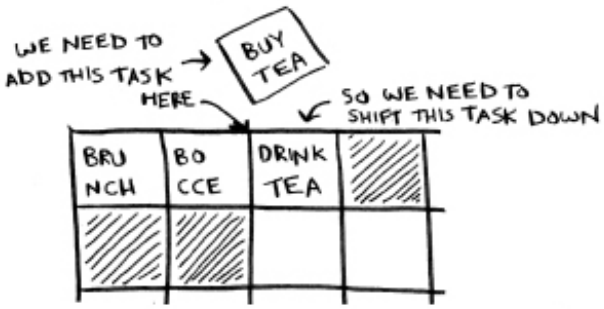
Now you want to add them in the order in which they should be done.

Unordered Ordered

What's better if you want to insert elements in the middle: arrays or lists? With lists, it's as easy as changing what the previous element points to.

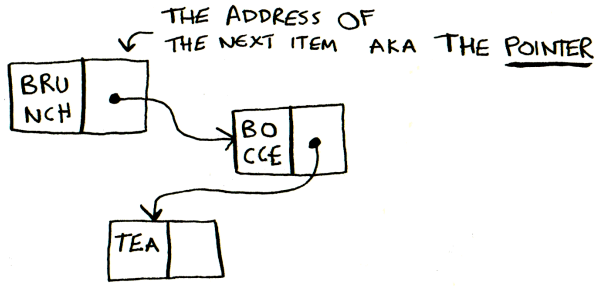


But for arrays, you have to shift all the rest of the elements down.



And if there's no space, you might have to copy everything to a new location! Lists are better if you want to insert elements into the middle.

I have talked a lot about how in a linked list, each item points to the next item in the list. But how does it do that exactly? By using pointers.



With each item in your linked list, you use a little bit of memory to store the address of the next item. This is called a pointer.

You will hear the word pointers come up sometimes, especially if you write using a lower level language like C. So it's good to know what it means.

Deletions

What if you want to delete an element? Again, lists are better, because you just need to change what the previous element points to. With arrays, everything needs to be moved up when you delete an element.

Unlike insertions, deletions will always work. Insertions can fail sometimes when there's no space left in memory. But you can always delete an element.

Here are the run times for common operations on arrays and linked lists.

	ARRAYS	LISTS
READING	O(1)	O(n)
INSERTION	O(n)	O(1)
DELETION	O(n)	O(1)

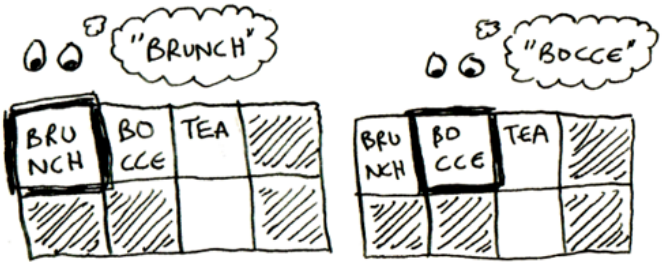
It's worth mentioning that insertions and deletions are O(1) time only if you can instantly access the element to be deleted. It's a common practice to keep track of the first and last items in a linked list, so it would take only O(1) time to delete those.

Which is used more, arrays or linked lists? Arrays are often used because they have a lot of advantages over linked lists. First of all, they are better at reads. Arrays provide random access.

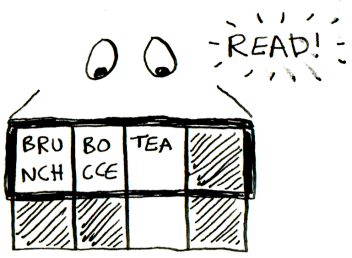
There are two different types of access: random access and sequential access. Sequential access means reading the elements one by one, starting at the first element. Linked lists can only do sequential access. If you want to read the 10th element of a linked list, you have to read the first 9 elements and follow the links to the 10th element. Random access means you can jump directly to the 10th element. Arrays provide random access. A lot of use cases require

random access, so arrays are used a lot.

Even beyond random access, though, arrays are faster because they can use caching. Maybe you are picturing reads like this, reading one item at a time:



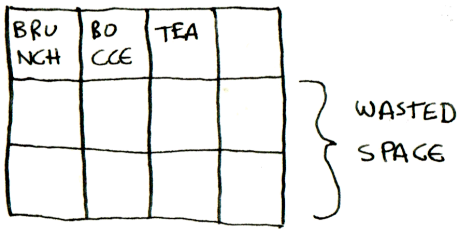
But in reality, computers read a whole section at a time, because that makes it a lot faster to go to the next item:



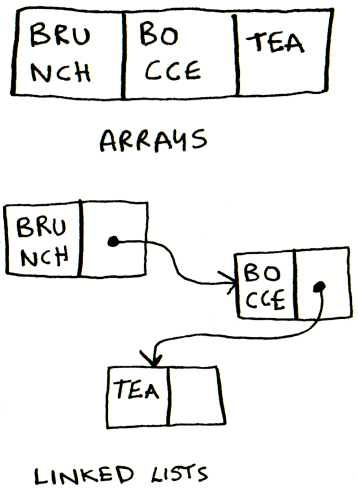
This is something you can do with arrays. With an array, you can read a whole section of items. But you can't do this with a linked list! You don't know where the next item is. You need to read an item, find out where the next item is, then read the next item.

So not only do arrays give you random access, they also provide faster sequential access!

Arrays are better for reads. What about memory efficiency? Remember earlier I had said that with arrays, you typically request more space than you need, and if you don't end up using that extra memory you requested, it is wasted?



Well, in reality, there is not much wasted space like this. On the other hand, when you use a linked list, you are using extra memory per item, because you need to store the address of the next item. So linked lists will take up more space if each item is pretty small. Here's the same information as an array and a linked list. You can see the linked list takes up more space:

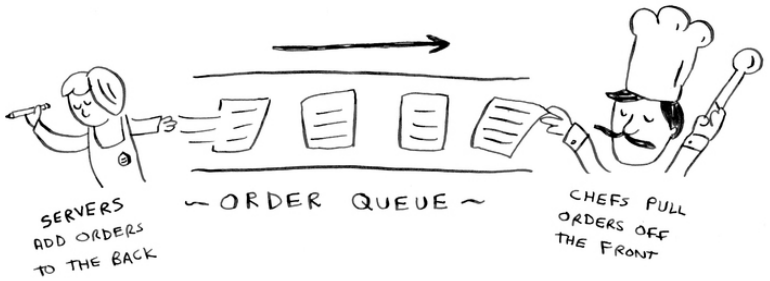


Of course, if each item is big, then even a single slot of wasted space can be a big deal, and that extra memory you're using to store the pointers can feel pretty small by comparison.

So arrays are used more often than linked lists except in specific use cases.

Exercises

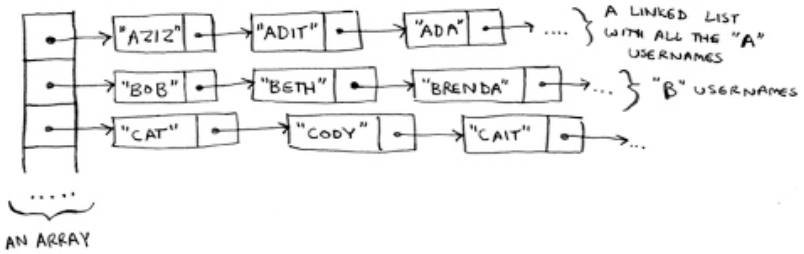
1. 2.2 Suppose you're building an app for restaurants to take customer orders. Your app needs to store a list of orders. Servers keep adding orders to this list, and chefs take orders off the list and make them. It's an order queue: servers add orders to the back of the queue, and the chef takes the first order off the queue and cooks it.



   Would you use an array or a linked list to implement this queue? (Hint: Linked lists are good for inserts/deletes, and arrays are good for random access. Which one are you going to be doing here?)

2. 2.3 Let's run a thought experiment. Suppose Facebook keeps a list of usernames. When someone tries to log in to Facebook, a search is done for their username. If their name is in the list of usernames, they can log in. People log in to Facebook pretty often, so there are a lot of searches through this list of usernames. Suppose Facebook uses binary search to search the list. Binary search needs random access—you need to be able to get to the middle of the list of usernames instantly. Knowing this, would you implement the list as an array or a linked list?

3. 2.4 People sign up for Facebook pretty often, too. Suppose you decided to use an array to store the list of users. What are the downsides of an array for inserts? In particular, suppose you're using binary search to search for logins. What happens when you add new users to an array?
4. 2.5 In reality, Facebook uses neither an array nor a linked list to store user information. Let's consider a hybrid data structure: an array of linked lists. You have an array with 26 slots. Each slot points to a linked list. For example, the first slot in the array points to a linked list containing all the usernames starting with a. The second slot points to a linked list containing all the usernames starting with b, and so on.



Suppose Adit B signs up for Facebook, and you want to add them to the list. You go to slot 1 in the array, go to the linked list for slot 1, and add Adit B at the end. Now, suppose you want to search for Zakhir H. You go to slot 26, which points to a linked list of all the Z names. Then you search through that list to find Zakhir H.

Compare this hybrid data structure to arrays and linked lists. Is it slower or faster than each for searching and inserting? You don't have to give Big O run times, just whether the new data structure would be faster or slower.

Selection sort

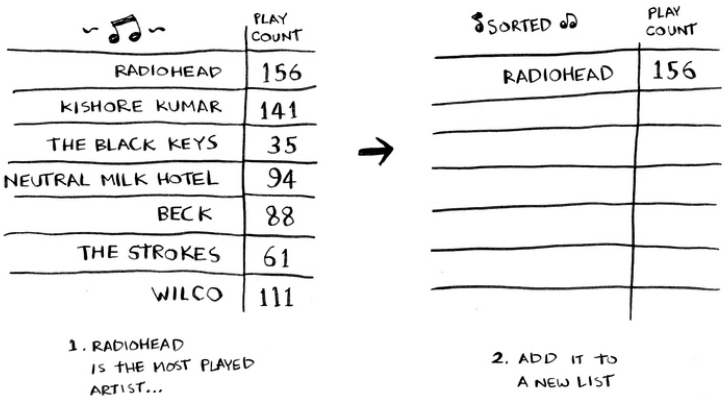
Let's put it all together to learn your second algorithm: selection sort. To follow this section, you need to understand arrays, as well as Big O notation.

Suppose you have a bunch of music on your computer. For each artist, you have a play count.

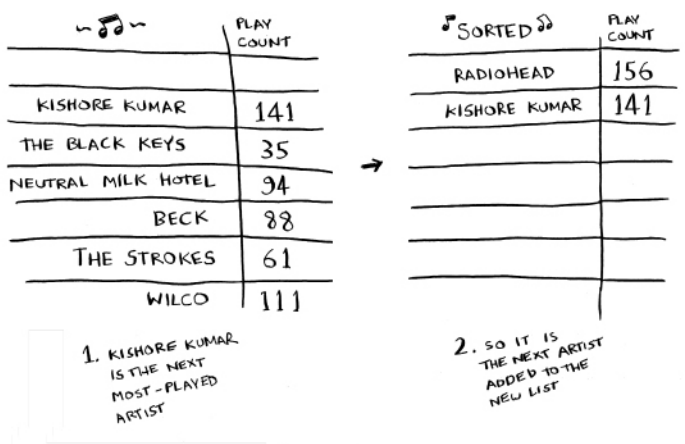
	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

You want to sort these artists from most to least played, so that you can rank your favorite artists. How can you do it?

One way is to go through the list and find the most-played artist. Add that artist to a new list.



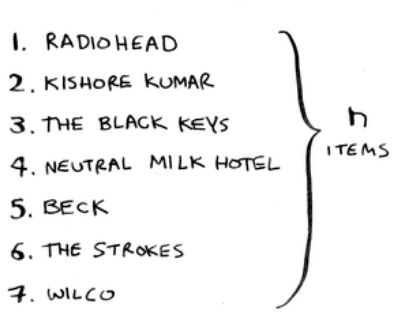
Do it again to find the next-most-played artist.



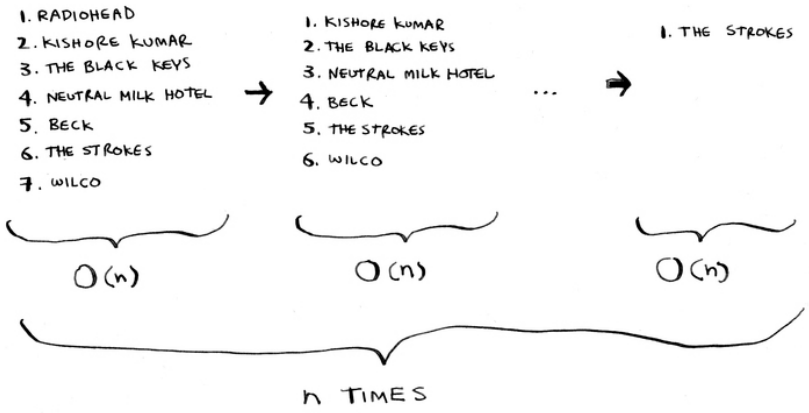
Keep doing this, and you'll end up with a sorted list.

	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
WILCO	111
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
THE BLACK KEYS	35

Let's put on our computer science hats and see how long this will take to run. Remember that O(*n*) time means you touch every element in a list once. For example, running simple search over the list of artists means looking at each artist once.



To find the artist with the highest play count, you have to check each item in the list. This takes O(*n*) time, as you just saw. So you have an operation that takes O(*n*) time, and you have to do that *n* times:



This takes O($n \times n$) time or O(n^2) time.

Sorting algorithms are very useful. Now you can sort

- Names in a phone book
- Travel dates
- Emails (newest to oldest)

Checking fewer elements each time

Maybe you're wondering: as you go through the operations, the number of elements you have to check keeps decreasing. Eventually, you're down to having to check just one element. So how can the run time still be O(n^2)? That's a good question, and the answer has to do with constants in Big O notation. I'll get into this more in chapter 4, but here's the gist.

You're right that you don't have to check a list of n elements each time. You check n elements, then n – 1, n - 2 ... 2, 1. On average, you check a list that has 1/2 × n elements. The runtime is O($n \times 1/2 \times n$). But constants like 1/2 are ignored in Big O notation (again, see chapter 4 for the full discussion), so you just write O($n \times n$) or O(n^2).

Selection sort is a neat algorithm, but it's not very fast. Quicksort is a faster sorting algorithm that only takes O(n log n) time. It's coming up in chapter 4!

Example code listing

We didn't show you the code to sort the music list, but the following is some code that will do something very similar: sort an array from smallest to largest. Let's write a function to find the smallest element in an array:

```
def findSmallest(arr):
  smallest = arr[0]    #A
  smallest_index = 0    #B
  for i in range(1, len(arr)):
    if arr[i] < smallest:
      smallest = arr[i]
      smallest_index = i
  return smallest_index
```

#A Stores the smallest value

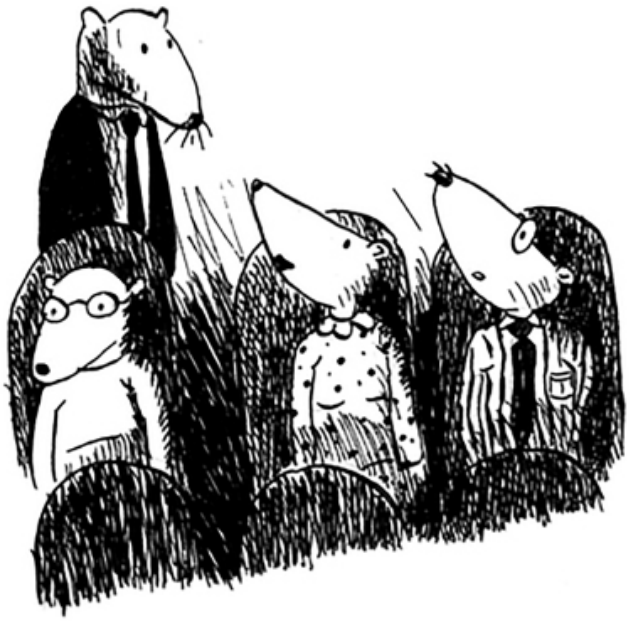
#B Stores the index of the smallest value

Now you can use this function to write selection sort:

```
def selectionSort(arr):         #A
  newArr = []
  copiedArr = list(arr) // copy array before mutating
  for i in range(len(copiedArr)):
      smallest = findSmallest(copiedArr)        #B
      newArr.append(copiedArr.pop(smallest))
  return newArr

print(selectionSort([5, 3, 6, 2, 10]))
```

#A Sorts an array

#B Finds the smallest element in the array, and adds it to the new array



Recap

- Your computer's memory is like a giant set of drawers.
- When you want to store multiple elements, use an array or a linked list.
- With an array, all your elements are stored right next to each other.
- With a linked list, elements are strewn all over, and one element stores the address of the next one.
- Arrays allow fast reads.
- Linked lists allow fast inserts and deletes.

Recursion 3

In this chapter

- You learn about recursion. Recursion is a coding technique used in many algorithms. It's a building block for understanding later chapters in this book.
- You learn what a base case and a recursive case is. The divide-and-conquer strategy (chapter 4) uses this simple concept to solve hard problems.

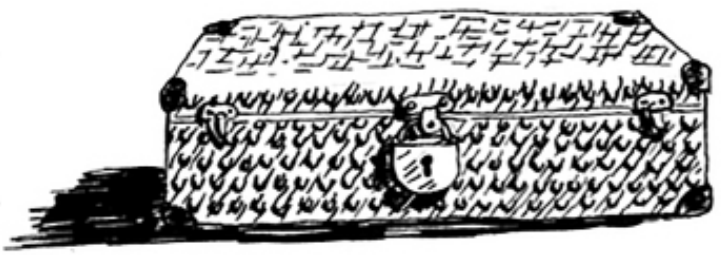
I'm excited about this chapter because it covers *recursion*, an elegant way to solve problems. Recursion is one of my favorite topics, but it's divisive. People either love it or hate it, or hate it until they learn to love it a few years later. I personally was in that third camp. To make things easier for you, I have some advice:

- This chapter has a lot of code examples. Run the code for yourself to see how it works.
- I'll talk about recursive functions. At least once, step through a recursive function with pen and paper: something like, "Let's see, I pass 5 into `factorial`, and then I return 5 times passing 4 into `factorial`, which is …," and so on. Walking through a function like this will teach you how a recursive function works.

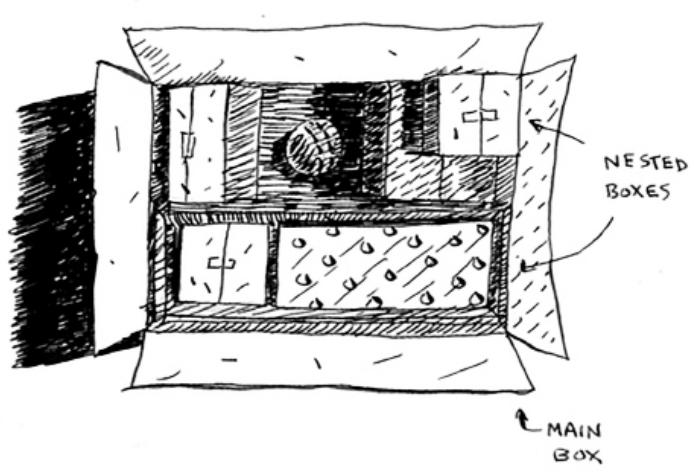
This chapter also includes a lot of pseudocode. *Pseudocode* is a high-level description of the problem you're trying to solve, in code. It's written like code, but it's meant to be closer to human speech.

Recursion

Suppose you're digging through your grandma's attic and come across a mysterious locked suitcase.

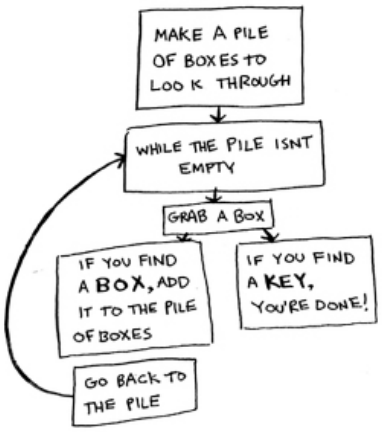


Grandma tells you that the key for the suitcase is probably in this other box.



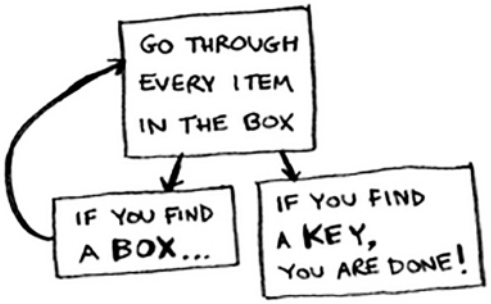
This box contains more boxes, with more boxes inside those boxes. The key is in a box somewhere. What's your algorithm to search for the key? Think of an algorithm before you read on.

Here's one approach.



1. Make a pile of boxes to look through.
2. Grab a box, and look through it.
3. If you find a box, add it to the pile to look through later.
4. If you find a key, you're done!
5. Repeat.

Here's an alternate approach.



1. Look through the box.
2. If you find a box, go to step 1.
3. If you find a key, you're done!

Which approach seems easier to you? The first approach uses a `while` loop. While the pile isn't empty, grab a box and look through it. Here's some pseudocode:

```
def look_for_key(main_box):
  pile = main_box.make_a_pile_to_look_through()
  while pile is not empty:
    box = pile.grab_a_box()
    for item in box:
      if item.is_a_box():
        pile.append(item)
      elif item.is_a_key():
        print(“found the key!”)
```

The second way uses recursion. *Recursion* is where a function calls itself. Here’s the second way in pseudocode:

```
def look_for_key(box):
  for item in box:
    if item.is_a_box():
      look_for_key(item)      #A
    elif item.is_a_key():
      print(“found the key!”)
```

#A Recursion!

Both approaches accomplish the same thing, but the second approach is clearer to me. Recursion is used when it makes the solution clearer. There’s no performance benefit to using recursion; in fact, loops are sometimes better for performance. I like this quote by Leigh Caldwell on Stack Overflow: “Loops may achieve a performance gain for your program. Recursion may achieve a performance gain for your programmer. Choose which is more important in your situation!”[1]

Many important algorithms use recursion, so it’s important to understand the concept.

[1] http://stackoverflow.com/a/72694/139117

Base case and recursive case

Because a recursive function calls itself, it's easy to write a function incorrectly that ends up in an infinite loop. For example, suppose you want to write a function that prints a countdown, like this:

```
> 3...2...1
```

You can write it recursively, like so:

```
def countdown(i):
  print(i)
  countdown(i-1)

countdown(3)
```

Write out this code and run it. You'll notice a problem: this function will run forever!

Infinite loop

```
> 3...2...1...0...-1...-2...
```

(Press Ctrl-C to kill your script.)

When you write a recursive function, you have to tell it when to stop recursing. That's why *every recursive function has two parts: the base case, and the recursive case*. The

recursive case is when the function calls itself. The base case is when the function doesn't call itself again … so it doesn't go into an infinite loop.

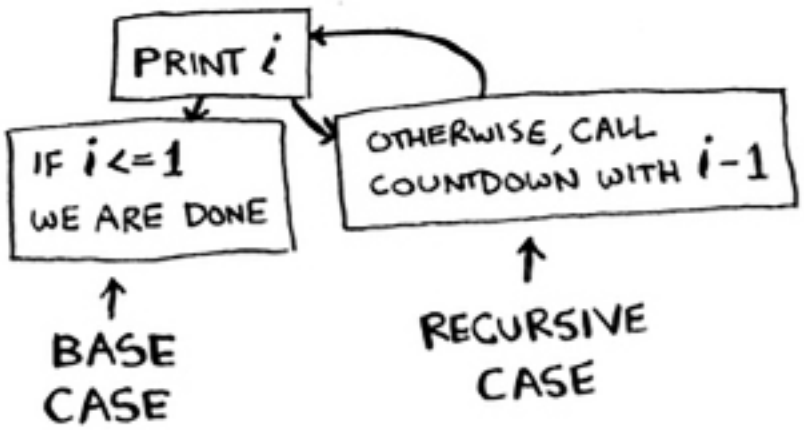
Let's add a base case to the countdown function:

```
def countdown(i):
  print(i)
  if i <= 1:      #A
    return
  else:          #B
    countdown(i-1)

countdown(3)
```

#A Base case

#B Recursive case

Now the function works as expected. It goes something like this.

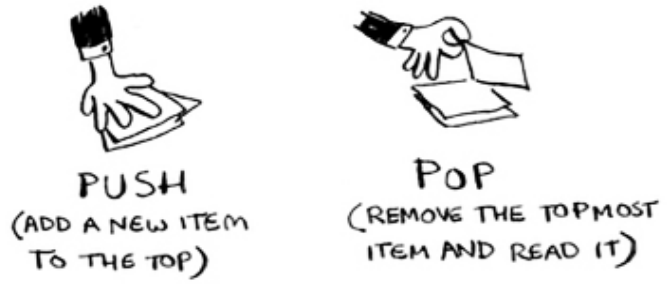


The stack

This section covers the *call stack*. It's an important concept in programming. The call stack is an important concept in general programming, and it's also important to understand when using recursion.

Suppose you're throwing a barbecue. You keep a todo list for the barbecue, in the form of a stack of sticky notes.

Remember back when we talked about arrays and lists, and you had a todo list? You could add todo items anywhere to the list or delete random items. The stack of sticky notes is much simpler. When you insert an item, it gets added to the top of the list. When you read an item, you only read the topmost item, and it's taken off the list. So your todo list has only two actions: *push* (insert) and *pop* (remove and read).



Let's see the todo list in action.

This data structure is called a *stack*. The stack is a simple data structure. You've been using a stack this whole time without realizing it!

The call stack

Your computer uses a stack internally called the *call stack*. Let's see it in action. Here's a simple function:

```
def greet(name):
    print("hello, " + name + "!")
    greet2(name)
    print("getting ready to say bye...")
    bye()
```

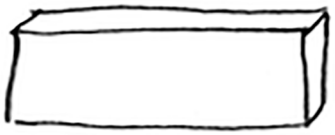
This function greets you and then calls two other functions. Here are those two functions:

```
def greet2(name):
    print(“how are you, “ + name + “?”)

def bye():
    print(“ok bye!”)
```

Let’s walk through what happens when you call a function.

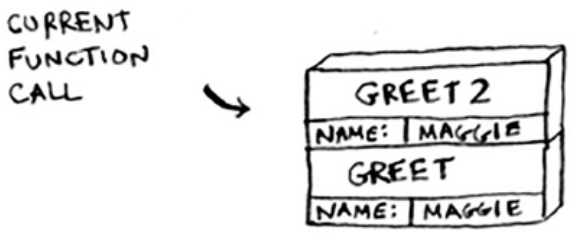
> **NOTE** To keep things simple, I’m only showing the calls to `greet` and `greet2`, and not showing the calls to the `print` function.

Suppose you call `greet(“maggie”)`. First, your computer allocates a box of memory for that function call.



Now let’s use the memory. The variable `name` is set to “maggie”. That needs to be saved in memory.

Every time you make a function call, your computer saves the values for all the variables for that call in memory like this. Next, you print `hello, maggie!` Then you call `greet2(“maggie”)`. Again, your computer allocates a box of memory for this function call.



Your computer is using a stack for these boxes. The second box is added on top of the first one. You print `how are you, maggie?` Then you return from the function call. When this happens, the box on top of the stack gets popped off.

Now the topmost box on the stack is for the `greet` function, which means you returned back to the `greet` function. When you called the `greet2` function, the `greet` function was *partially completed*. This is the big idea behind this section: *when you call a function from another function, the calling function is paused in a partially completed state*. All the values of the variables for that function are still stored on the call stack (i.e. in memory). Now that you're done with the `greet2` function, you're back to the `greet` function, and you pick up where you left off. First you print `getting ready to say bye...`. You call the `bye` function.

A box for that function is added to the top of the stack. Then you print `ok bye!` and return from the function call.

And you're back to the `greet` function. There's nothing else to be done, so you return from the `greet` function too. This stack, used to save the variables for multiple functions, is called the *call stack*.

Exercise

1. 3.1 Suppose I show you a call stack like this.

What information can you give me, just based on this call stack?

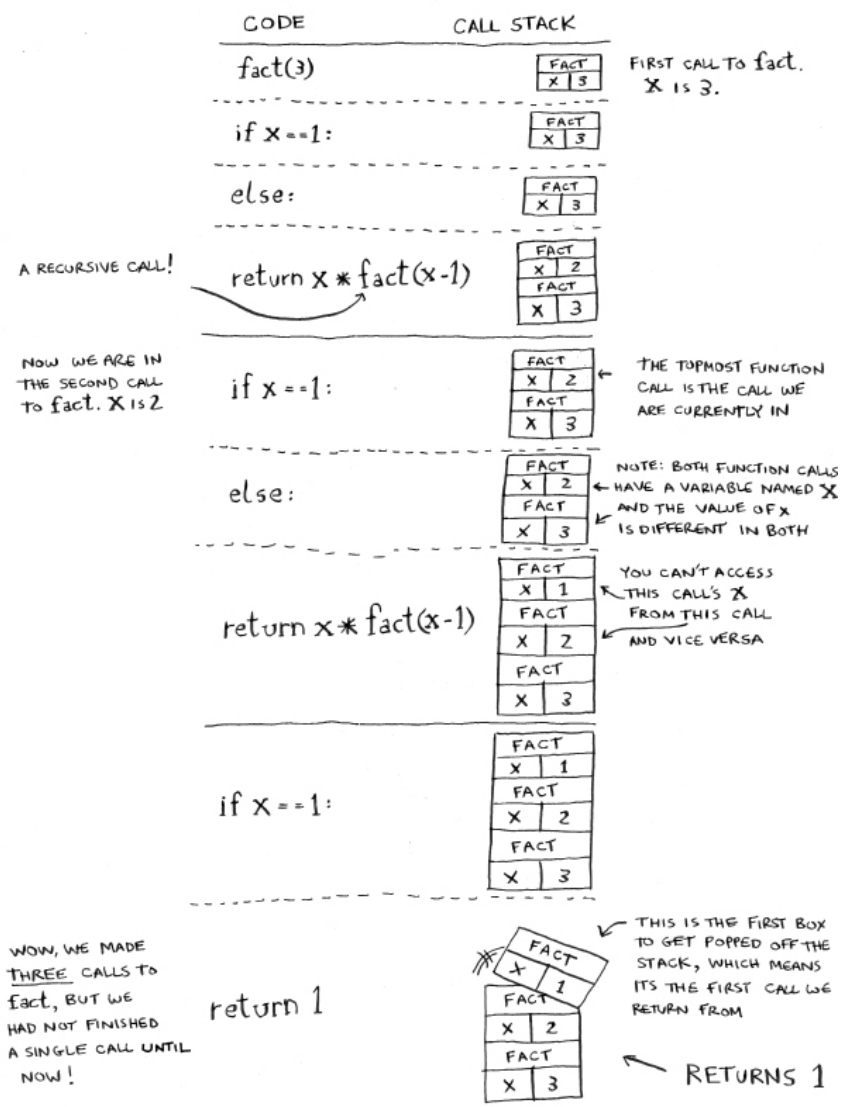
Now let's see the call stack in action with a recursive function.

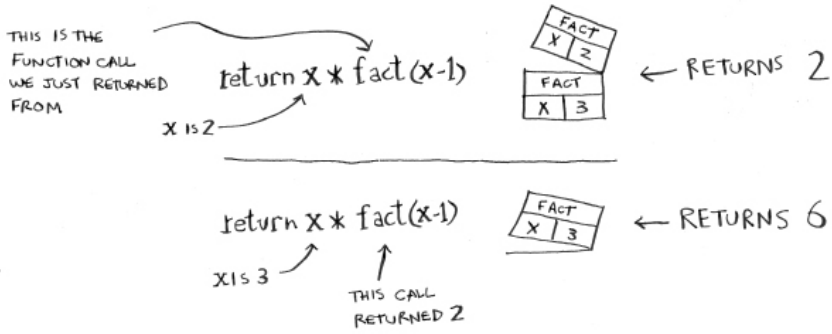
The call stack with recursion

Recursive functions use the call stack too! Let's look at this in action with the `factorial` function. `factorial(5)` is written as 5!, and it's defined like this: 5! = 5 * 4 * 3 * 2 * 1. Similarly, `factorial(3)` is 3 * 2 * 1. Here's a recursive function to calculate the factorial of a number:

```
def fact(x):
  if x == 1:
    return 1
  else:
    return x * fact(x-1)
```

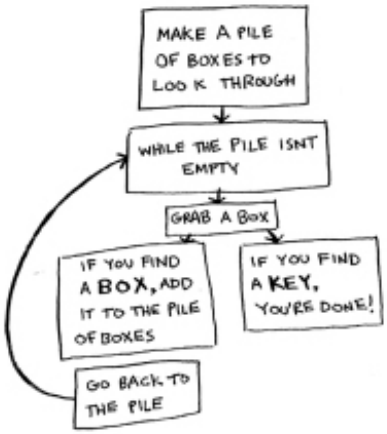
Now you call `fact(3)`. Let's step through this call line by line and see how the stack changes. Remember, the topmost box in the stack tells you what call to `fact` you're currently on.


CODE
CALL STACK
fact(3)
FACT x 3
FIRST CALL TO fact. X IS 3.
if x==1:
else:
A RECURSIVE CALL!
return x * fact(x-1)
FACT x 2
FACT x 3
NOW WE ARE IN THE SECOND CALL TO fact. X IS 2
THE TOPMOST FUNCTION CALL IS THE CALL WE ARE CURRENTLY IN
NOTE: BOTH FUNCTION CALLS HAVE A VARIABLE NAMED X AND THE VALUE OF X IS DIFFERENT IN BOTH
FACT x 1
YOU CAN'T ACCESS THIS CALL'S X FROM THIS CALL AND VICE VERSA
WOW, WE MADE THREE CALLS TO fact, BUT WE HAD NOT FINISHED A SINGLE CALL UNTIL NOW!
return 1
THIS IS THE FIRST BOX TO GET POPPED OFF THE STACK, WHICH MEANS IT'S THE FIRST CALL WE RETURN FROM
RETURNS 1

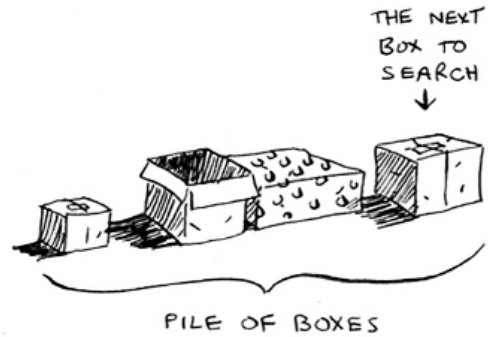


Notice that each call to `fact` has its own copy of `x`. You can't access a different function's copy of `x`.

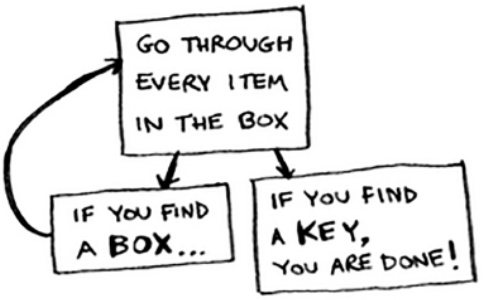
The stack plays a big part in recursion. In the opening example, there were two approaches to find the key. Here's the first way again.



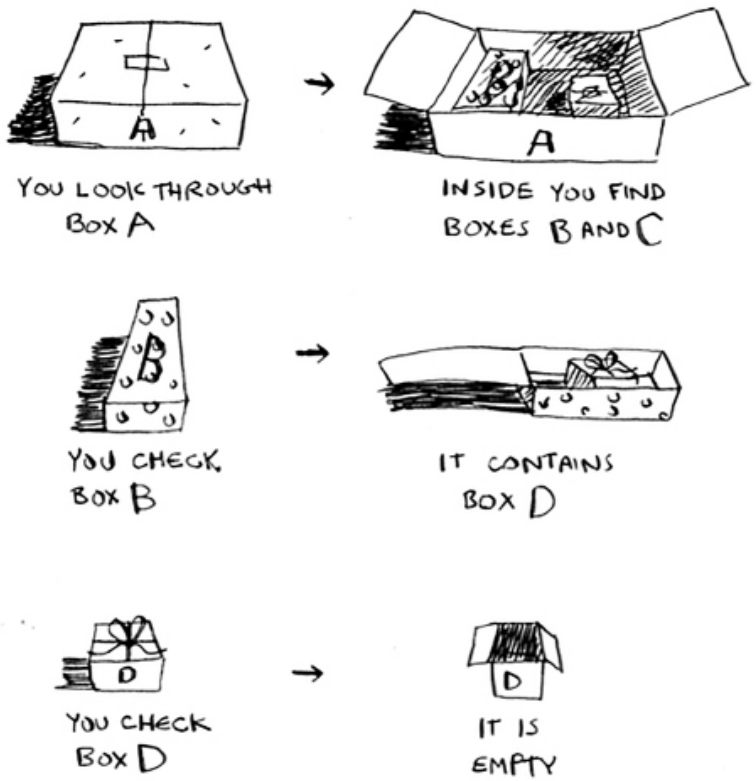
This way, you make a pile of boxes to search through, so you always know what boxes you still need to search.



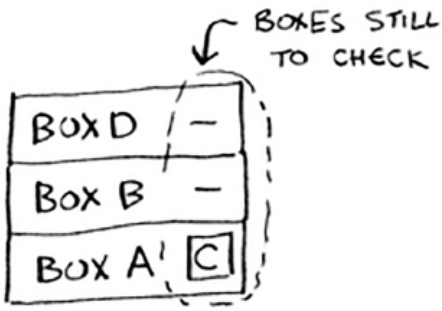
But in the recursive approach, there's no pile.



If there's no pile, how does your algorithm know what boxes you still have to look through? Here's an example.



At this point, the call stack looks like this.



The "pile of boxes" is saved on the stack! This is a stack of half-completed function calls, each with its own half-complete list of boxes to look through. Using the stack is convenient because you don't have to keep track of a pile

of boxes yourself—the stack does it for you.

Using the stack is convenient, but there's a cost: saving all that info can take up a lot of memory. Each of those function calls takes up some memory, and when your stack is too tall, that means your computer is saving information for many function calls. At that point, you have two options:

You can rewrite your code to use a loop instead.

You can use something called *tail recursion*. That's an advanced recursion topic that is out of the scope of this book. It's also only supported by some languages, not all.

Exercise

1. 3.2 Suppose you accidentally write a recursive function that runs forever. As you saw, your computer allocates memory on the stack for each function call. What happens to the stack when your recursive function runs forever?

Recap

- Recursion is when a function calls itself.
- Every recursive function has two cases: the base case and the recursive case.
- A stack has two operations: push and pop.
- All function calls go onto the call stack.
- The call stack can get very large, which takes up a lot of memory.

Quicksort

4

In this chapter

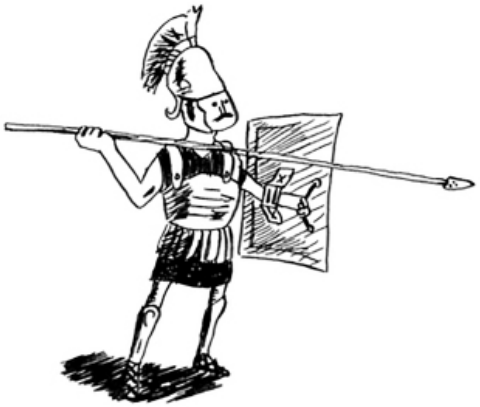
- You learn about divide-and-conquer. Sometimes you'll come across a problem that can't be solved by any algorithm you've learned. When a good algorithmist comes across such a problem, they don't just give up. They have a toolbox full of techniques they use on the problem, trying to come up with a solution. Divide-and-conquer is the first general technique you learn.
- You learn about quicksort, an elegant sorting algorithm that's often used in practice. Quicksort uses divide-and-conquer.

You learned all about recursion in the last chapter. This chapter focuses on using your new skill to solve problems. We'll explore *divide and conquer* (D&C), a well-known recursive technique for solving problems.

This chapter really gets into the meat of algorithms. After all, an algorithm isn't very useful if it can only solve one type of problem. Instead, D&C gives you a new way to think about solving problems. D&C is another tool in your toolbox. When you get a new problem, you don't have to be stumped. Instead, you can ask, "Can I solve this if I use divide and conquer?"

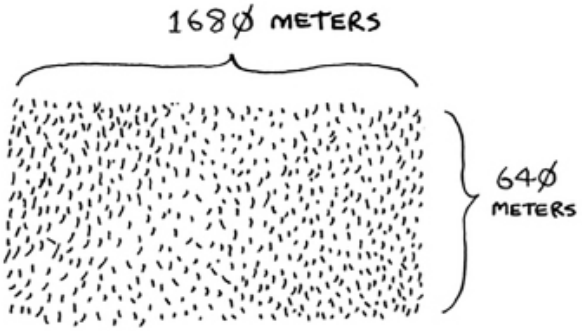
At the end of the chapter, you'll learn your first major D&C algorithm: *quicksort*. Quicksort is a sorting algorithm, and a much faster one than selection sort (which you learned in chapter 2). It's a good example of elegant code.

Divide & conquer



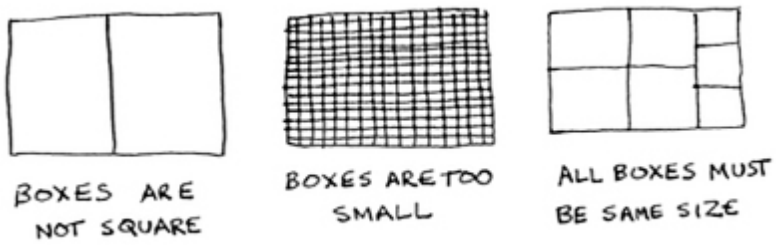
D&C can take some time to grasp. So, we'll do three examples. First I'll show you a visual example. Then I'll do a code example that is less pretty but maybe easier. Finally, we'll go through quicksort, a sorting algorithm that uses D&C.

Suppose you're a farmer with a plot of land.



You want to divide this farm evenly into *square* plots. You

want the plots to be as big as possible. So none of these will work.

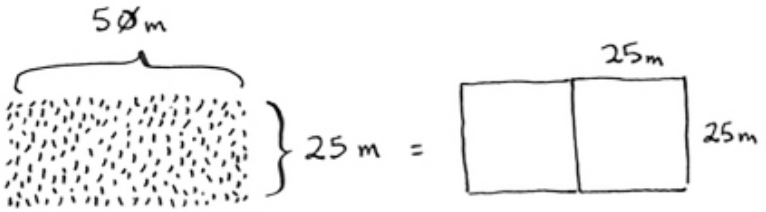


How do you figure out the largest square size you can use for a plot of land? Use the D&C strategy! D&C algorithms are recursive algorithms. To solve a problem using D&C, there are two steps:

1. Figure out the base case. This should be the simplest possible case.
2. Divide or decrease your problem until it becomes the base case.

Let's use D&C to find the solution to this problem. What is the largest square size you can use?

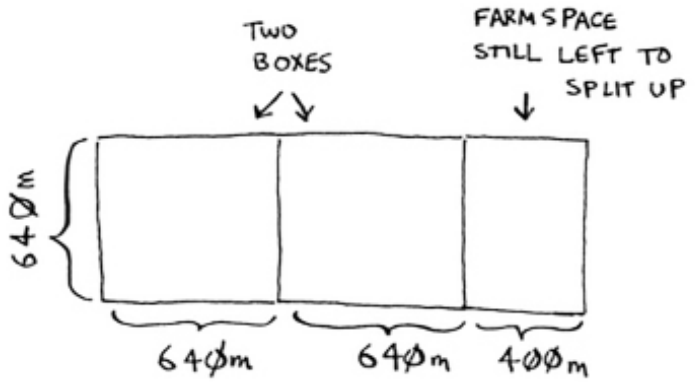
First, figure out the base case. The easiest case would be if one side was a multiple of the other side.



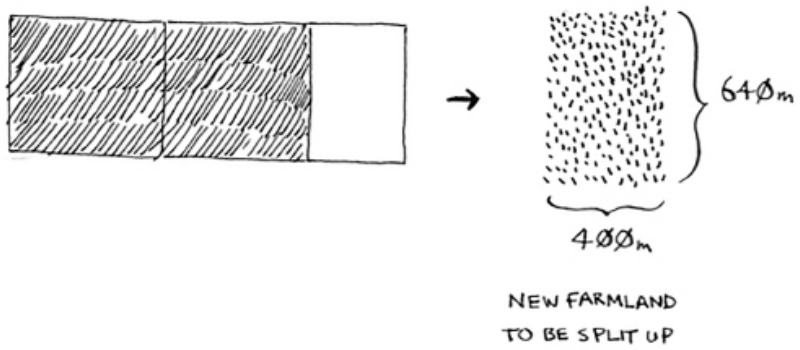
Suppose one side is 25 meters (m) and the other side is 50 m. Then the largest box you can use is 25 m × 25 m. You need two of those boxes to divide up the land.

Now you need to figure out the recursive case. This is where D&C comes in. According to D&C, with every recursive call, you have to reduce your problem. How do

you reduce the problem here? Let's start by marking out the biggest boxes you can use.



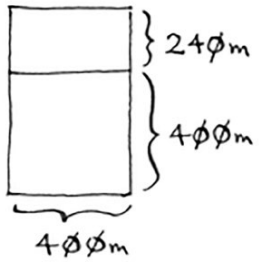
You can fit two 640 × 640 boxes in there, and there's some land still left to be divided. Now here comes the "Aha!" moment. There's a farm segment left to divide. *Why don't you apply the same algorithm to this segment?*



So you started out with a 1680 × 640 farm that needed to be split up. But now you need to split up a smaller segment, 640 × 400. If you *find the biggest box that will work for this size, that will be the biggest box that will work for the entire farm*. You just reduced the problem from a 1680 × 640 farm to a 640 × 400 farm!

Euclid's algorithm

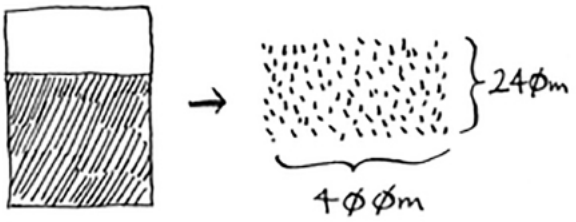
"If you find the biggest box that will work for this size, that will be the biggest box that will work for the entire farm." If it's not obvious to you why this statement is true, don't worry. It isn't obvious. Unfortunately, the proof for why it works is a little too long to include in this book, so you'll just have to believe me that it works. If you want to understand the proof, look up Euclid's algorithm for finding the Greatest Common Denominator. The Khan academy has a good explanation here: https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm.



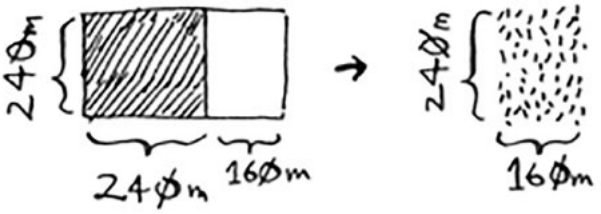
1680 and 640

Let's apply the same algorithm again. Starting with a 640 × 400m farm, the biggest box you can create is 400 × 400 m.

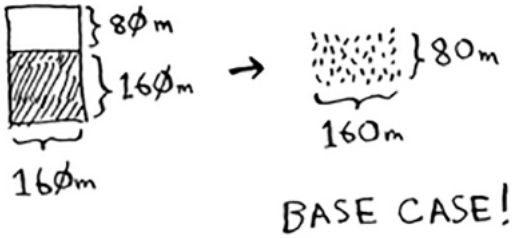
And that leaves you with a smaller segment, 400 × 240 m.



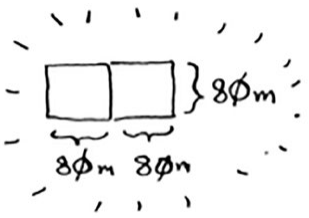
And you can draw a box on that to get an even smaller segment, 240 × 160 m.



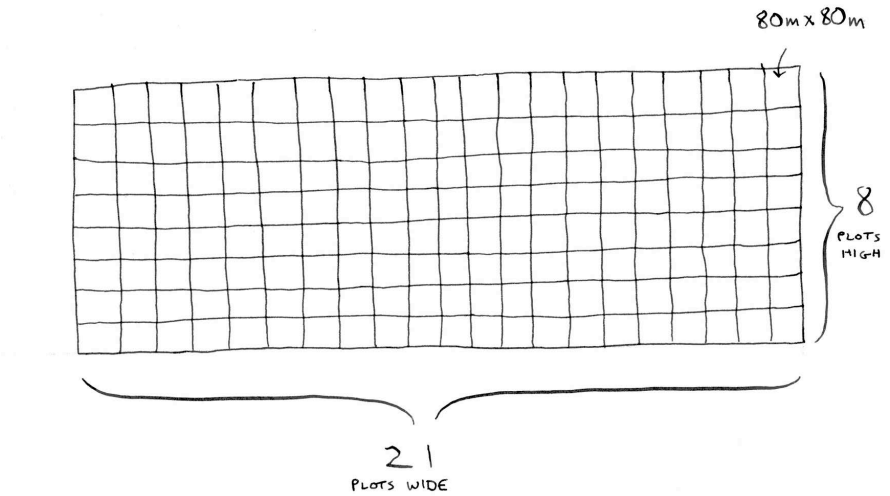
And then you draw a box on that to get an even *smaller* segment.



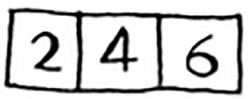
Hey, you're at the base case: 80 is a factor of 160. If you split up this segment using boxes, you don't have anything left over!



So, for the original farm, the biggest plot size you can use is 80 × 80 m.



To recap, here's how D&C works:



1. Figure out a simple case as the base case.
2. Figure out how to reduce your problem and get to the base case.

D&C isn't a simple algorithm that you can apply to a problem. Instead, it's a way to think about a problem. Let's do one more example.

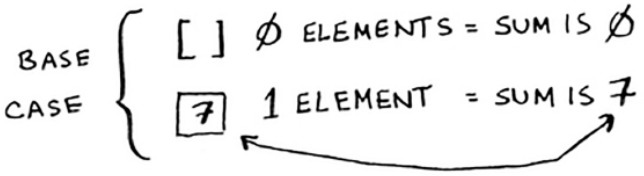
You're given an array of numbers.

You have to add up all the numbers and return the total. It's pretty easy to do this with a loop:

```
def sum(arr):
  total = 0
  for x in arr:
    total += x
  return total

print(sum([1, 2, 3, 4]))
```

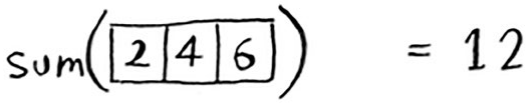
But how would you do this with a recursive function?

1. Step 1: Figure out the base case. What's the simplest array you could get? Think about the simplest case, and then read on. If you get an array with 0 or 1 element, that's pretty easy to sum up.

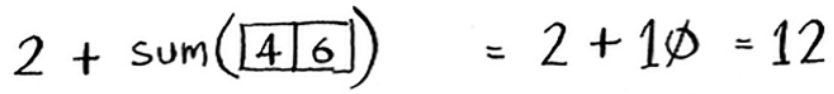


   So that will be the base case.

2. Step 2: You need to move closer to an empty array with every recursive call. How do you reduce your problem size? Here's one way.

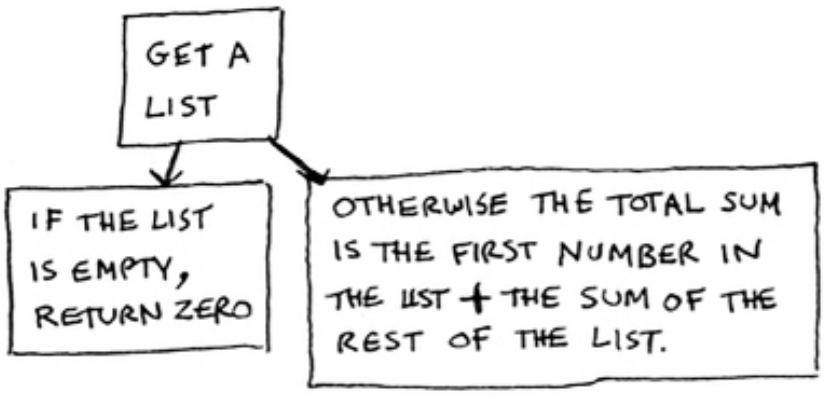


It's the same as this.

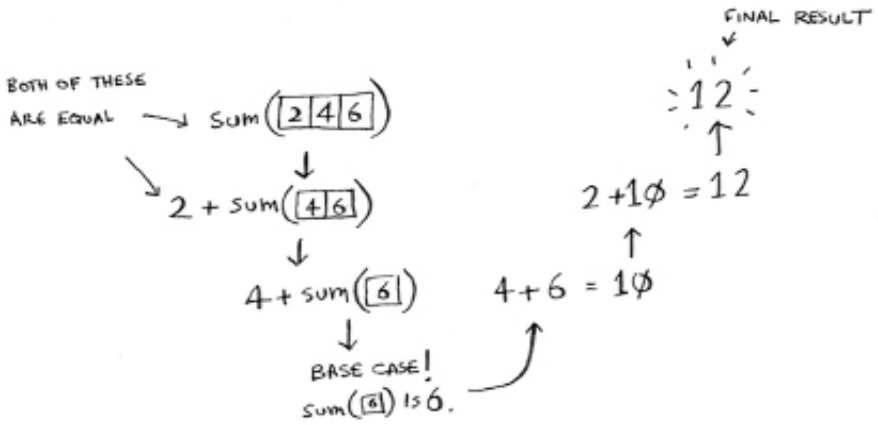


In either case, the result is 12. But in the second version, you're passing a smaller array into the `sum` function. That is, *you decreased the size of your problem!*

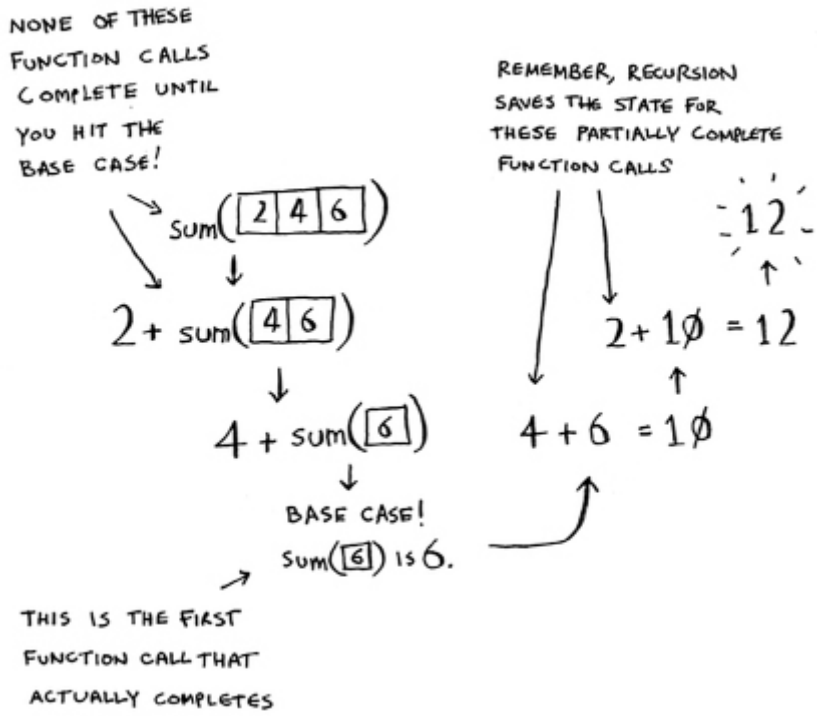
Your `sum` function could work like this.



Here it is in action.



Remember, recursion keeps track of the state.



TIP When you're writing a recursive function involving an array, the base case is often an empty array or an array with one element. If you're stuck, try that first.

Sneak peak at functional programming

"Why would I do this recursively if I can do it easily with a loop?" you may be thinking. Well, this is a sneak peek into functional programming! Functional programming languages like Haskell don't have loops, so you have to use recursion to write functions like this. If you have a good understanding of recursion, functional languages will be easier to learn. For example, here's how you'd write a `sum` function in Haskell:

```
sum [] = 0      #A
sum (x:xs) = x + (sum xs)    #B
```

#A Base case

#B Recursive case

Notice that it looks like you have two definitions for the function. The first definition is run when you hit the base case. The second definition runs at the recursive case. You can also write this function in Haskell using an if statement:

```
sum arr = if arr == []
        then 0
        else (head arr) + (sum (tail arr))
```

But the first definition is easier to read. Because Haskell makes heavy use of recursion, it includes all kinds of niceties like this to make recursion easy. If you like recursion, or you're interested in learning a new language, check out Haskell.

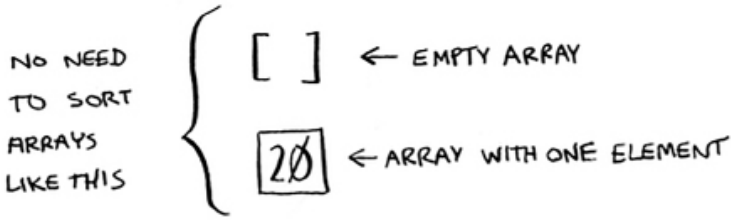
Exercises

1. 4.1 Write out the code for the earlier `sum` function.
2. 4.2 Write a recursive function to count the number of items in a list.
3. 4.3 Write a recursive function to find the maximum number in a list.
4. 4.4 Remember binary search from chapter 1? It's a divide-and-conquer algorithm, too. Can you come up with the base case and recursive case for binary search?

Quicksort

Quicksort is a sorting algorithm. It's much faster than selection sort and is frequently used in real life. Quicksort also uses divide and conquer.

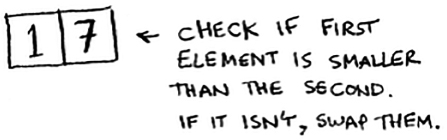
Let's use quicksort to sort an array. What's the simplest array that a sorting algorithm can handle (remember my tip from the previous section)? Well, some arrays don't need to be sorted at all.



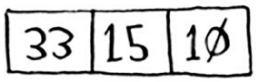
Empty arrays and arrays with just one element will be the base case. You can just return those arrays as is—there's nothing to sort:

```
def quicksort(array):
  if len(array) < 2:
    return array
```

Let's look at bigger arrays. An array with two elements is pretty easy to sort, too.



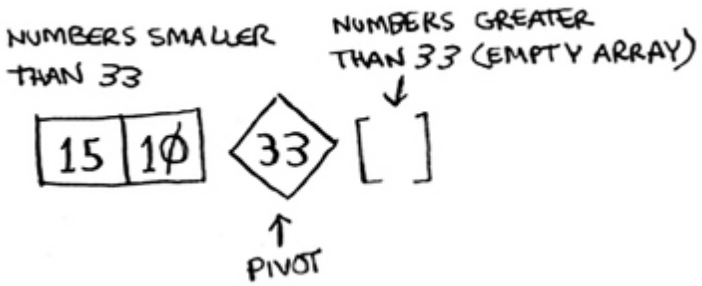
What about an array of three elements?



Remember, you're using D&C. So you want to break down this array until you're at the base case. Here's how quicksort works. First, pick an element from the array. This element is called the *pivot*.

We'll talk about how to pick a good pivot later. For now, let's say the first item in the array is the pivot.

Now find the elements smaller than the pivot and the elements larger than the pivot.



This is called *partitioning*. Now you have

- A sub-array of all the numbers less than the pivot
- The pivot
- A sub-array of all the numbers greater than the pivot

The two sub-arrays aren't sorted. They're just partitioned. But if they *were* sorted, then sorting the whole array would be pretty easy.

If the sub-arrays are sorted, then you can combine the whole thing like this—`left array + pivot + right array`—and you get a sorted array. In this case, it's `[10, 15] + [33] + [] = [10, 15, 33]`, which is a sorted array.

How do you sort the sub-arrays? Well, the quicksort base case already knows how to sort empty arrays (the right sub-array), and it can recursively sort arrays of two elements (the left sub-array). So if you call quicksort on the two sub-arrays and then combine the results, you get a sorted array!

```
quicksort([15, 10]) + [33] + quicksort([])
> [10, 15, 33]         #A
```

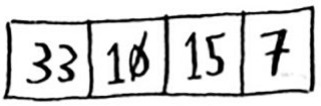
#A A sorted array

This will work with any pivot. Suppose you choose 15 as the pivot instead.

Both sub-arrays have only one element, and you know how to sort those. So now you know how to sort an array of three elements. Here are the steps:

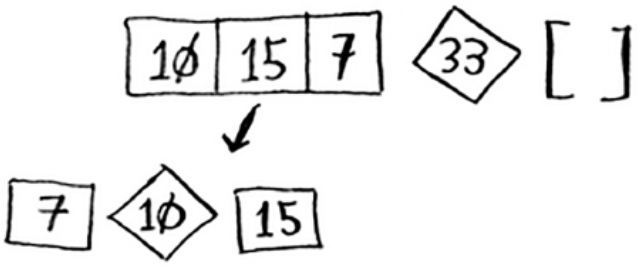
1. Pick a pivot.
2. Partition the array into two sub-arrays: elements less than the pivot and elements greater than the pivot.
3. Call quicksort recursively on the two sub-arrays.

What about an array of four elements?

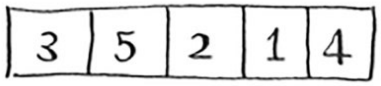


Suppose you choose 33 as the pivot again.

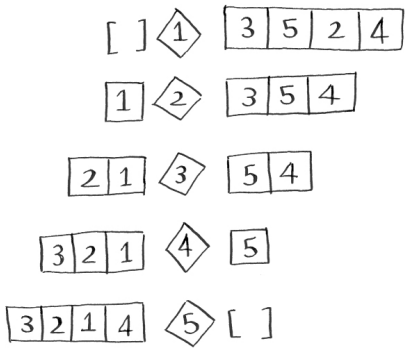
The array on the left has three elements. You already know how to sort an array of three elements: call quicksort on it recursively.



So you can sort an array of four elements. And if you can sort an array of four elements, you can sort an array of five elements. Why is that? Suppose you have this array of five elements.

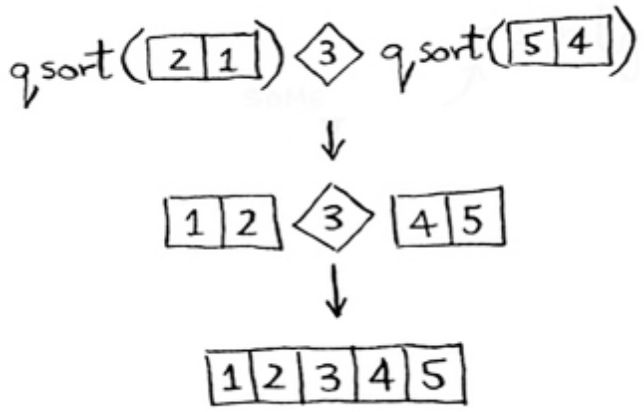


Here are all the ways you can partition this array, depending on what pivot you choose.

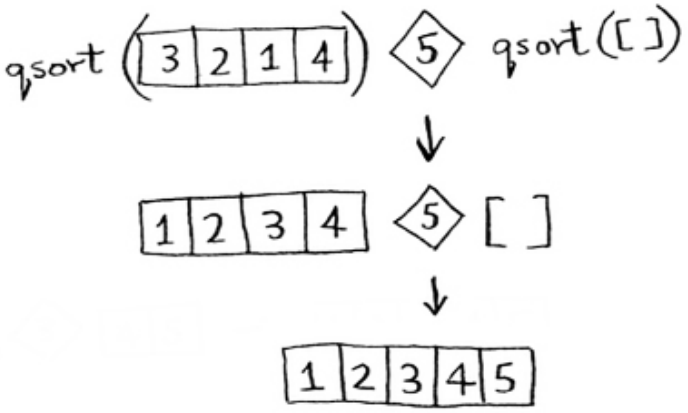


Notice that all of these sub-arrays have somewhere between 0 and 4 elements. And you already know how to sort an array of 0 to 4 elements using quicksort! So no matter what pivot you pick, you can call quicksort recursively on the two sub-arrays.

For example, suppose you pick 3 as the pivot. You call quicksort on the sub-arrays.



The sub-arrays get sorted, and then you combine the whole thing to get a sorted array. This works even if you choose 5 as the pivot.



This works with any element as the pivot. So you can sort an array of five elements. Using the same logic, you can sort an array of six elements, and so on.

Inductive proofs

You just got a sneak peak into *inductive proofs!* Inductive proofs are one way to prove that your algorithm works. Each inductive proof has two steps: the base case and the inductive case. Sound familiar? For example, suppose I want to prove that I can climb to the top of a ladder. In the inductive case, if my legs are on a rung, I can put my legs on the next rung. So if I'm on rung 2, I can climb to rung 3. That's the inductive case. For the base case, I'll say that my legs are on rung 1. Therefore, I can climb the entire ladder, going up one rung at a time.

You use similar reasoning for quicksort. In the base case, I showed that the algorithm works for the base case: arrays of size 0 and 1. In the inductive case, I showed that if quicksort works for an array of size 1, it will work for an array of size 2. And if it works for arrays of size 2, it will work for arrays of size 3, and so on. Then I can say that quicksort will work for all arrays of any size. I won't go deeper into inductive proofs here, but they're fun and go hand-in-hand with D&C.

Here's the code for quicksort:

```
def quicksort(array):
  if len(array) < 2:
    return array          #A
  else:
    pivot = array[0]        #B
    less = [i for i in array[1:] if i <= pivot]    #C

    greater = [i for i in array[1:] if i > pivot]   #D
    return quicksort(less) + [pivot] + quicksort(greater)

print(quicksort([10, 5, 2, 3]))
```

#A Base case: arrays with 0 or 1 element are already "sorted."

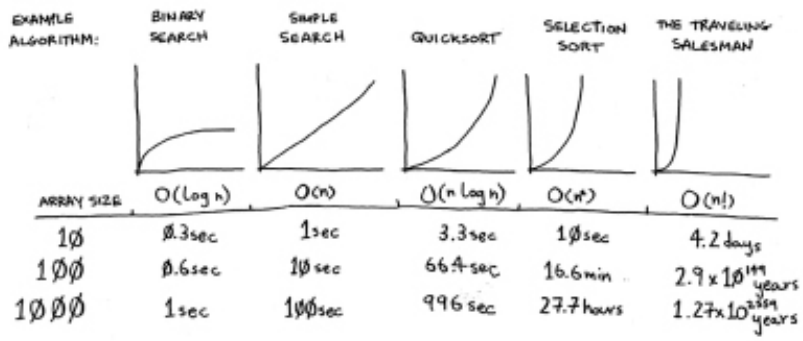
#B Recursive case

#C Sub-array of all the elements less than the pivot

#D Sub-array of all the elements greater than the pivot

Big O notation revisited

Quicksort is unique because its speed depends on the pivot you choose. Before I talk about quicksort, let's look at the most common Big O run times again.



Estimates based on a slow computer that performs 10 operations per second

The example times in this chart are estimates if you perform 10 operations per second. These graphs aren't precise—they're just there to give you a sense of how different these run times are. In reality, your computer can do way more than 10 operations per second.

Each run time also has an example algorithm attached. Check out selection sort, which you learned in chapter 2. It's O(n^2). That's a pretty slow algorithm.

There's another sorting algorithm called *merge sort*, which is O(n log n). Much faster! Quicksort is a tricky case. In the worst case, quicksort takes O(n^2) time.

It's as slow as selection sort! But that's the worst case. In the average case, quicksort takes O(n log n) time. So you might be wondering:

What do *worst case* and *average case* mean here?

If quicksort is O(n log n) on average, but merge sort is O(n log n) always, why not use merge sort? Isn't it faster?

Merge sort vs. quicksort

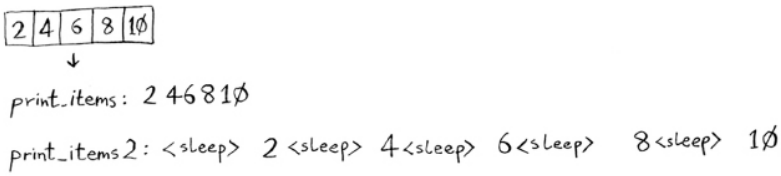
Suppose you have this simple function to print every item in a list:

```
def print_items(myList):
  for item in myList:
    print(item)
```

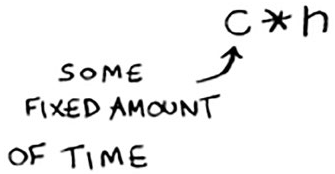
This function goes through every item in the list and prints it out. Because it loops over the whole list once, this function runs in O(n) time. Now, suppose you change this function so it sleeps for 1 second before it prints out an item:

```
from time import sleep
def print_items2(myList):
  for item in myList:
    sleep(1)
    print(item)
```

Before it prints out an item, it will pause for 1 second. Suppose you print a list of five items using both functions.

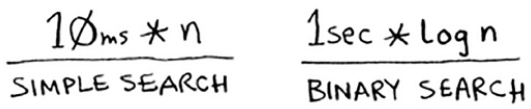


Both functions loop through the list once, so they're both O(*n*) time. Which one do you think will be faster in practice? I think `print_items` will be much faster because it doesn't pause for 1 second before printing an item. So even though both functions are the same speed in Big O notation, `print_items` is faster in practice. When you write Big O notation like O(*n*), it really means this.

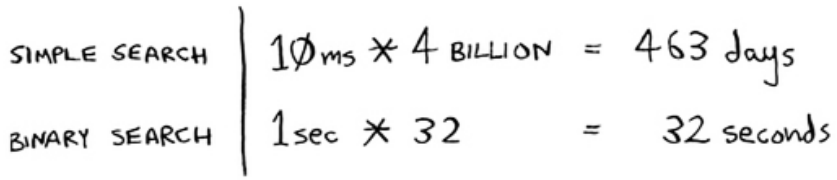


`c` is some fixed amount of time that your algorithm takes. It's called the *constant*. For example, it might be `10 milliseconds * n` for `print_items` versus `1 second * n` for `print_items2`.

You usually ignore that constant, because if two algorithms have different Big O times, the constant doesn't matter. Take binary search and simple search, for example. Suppose both algorithms had these constants.



You might say, "Wow! Simple search has a constant of 10 milliseconds, but binary search has a constant of 1 second. Simple search is way faster!" Now suppose you're searching a list of 4 billion elements. Here are the times.



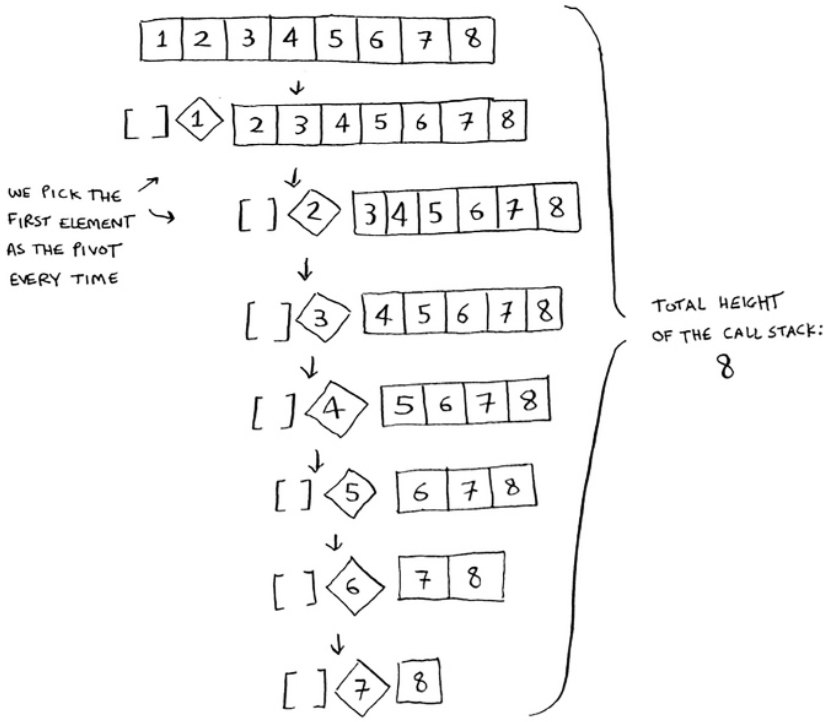
As you can see, binary search is still way faster. That constant didn't make a difference at all.

But sometimes the constant *can* make a difference. Quicksort versus merge sort is one example. Often, the way Quicksort and merge sort are implemented, if they're both O(*n* log *n*) time, quicksort is faster. And quicksort is faster in practice because it hits the average case way more often than the worst case.

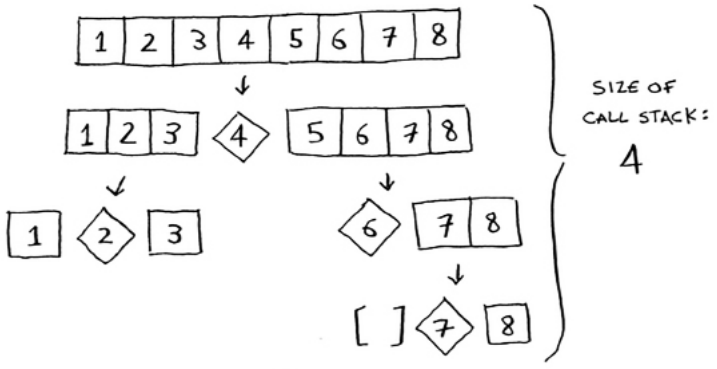
So now you're wondering: what's the average case versus the worst case?

Average case vs. worst case

The performance of quicksort heavily depends on the pivot you choose. Suppose you always choose the first element as the pivot. And you call quicksort with an array that is *already sorted*. Quicksort doesn't check to see whether the input array is already sorted. So it will still try to sort it.



Notice how you're not splitting the array into two halves. Instead, one of the sub-arrays is always empty. So the call stack is really long. Now instead, suppose you always picked the middle element as the pivot. Look at the call stack now.

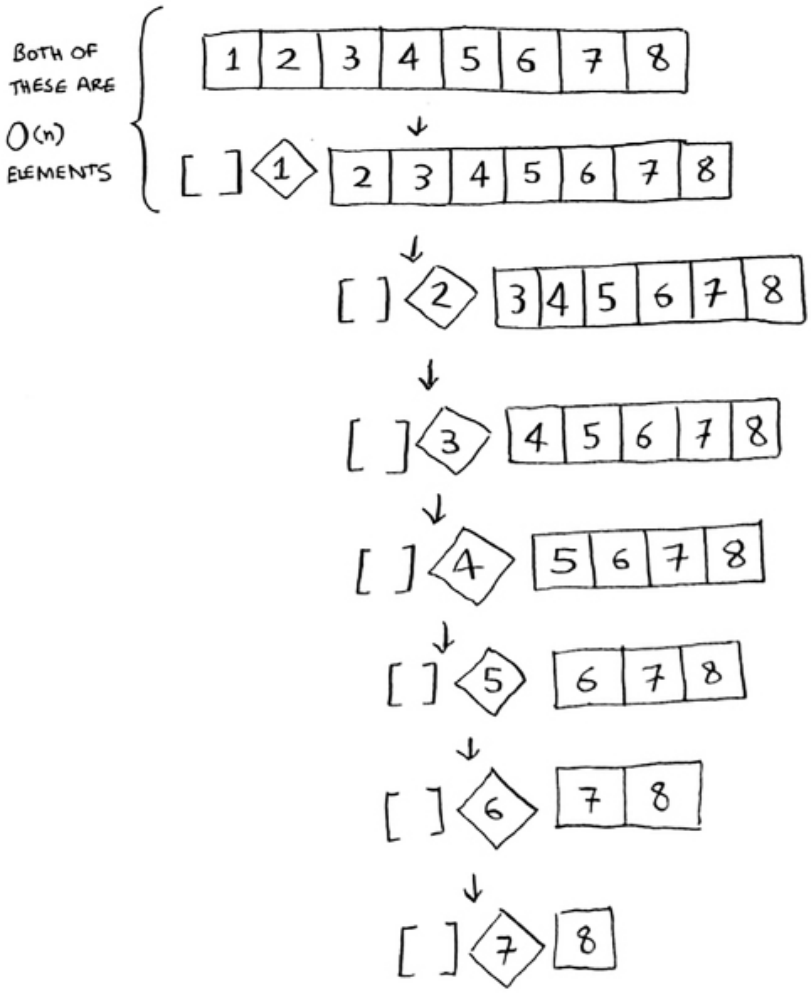


It's so short! Because you divide the array in half every time,

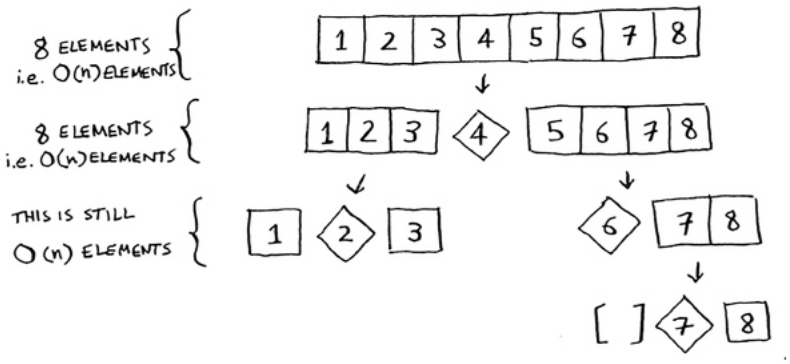
you don’t need to make as many recursive calls. You hit the base case sooner, and the call stack is much shorter.

The first example you saw is the worst-case scenario, and the second example is the best-case scenario. In the worst case, the stack size is O(*n*). In the best case, the stack size is O(log *n*). You can get the best case consistently, as long as you always choose a random element as the pivot. Read on to find out why.

Now look at the first level in the stack. You pick one element as the pivot, and the rest of the elements are divided into sub-arrays. You touch all eight elements in the array. So this first operation takes O(*n*) time. You touched all eight elements on this level of the call stack. But actually, you touch O(*n*) elements on every level of the call stack.



Even if you partition the array differently, you're still touching O(*n*) elements every time.



In this example, there are O(log *n*) levels (the technical way to say that is, "The height of the call stack is O(log *n*)"). And each level takes O(*n*) time. The entire algorithm will take O(*n*) * O(log *n*) = O(*n* log *n*) time. This is the best-case scenario.

In the worst case, there are O(*n*) levels, so the algorithm will take O(*n*) * O(*n*) = O(n^2) time.

Well, guess what? I'm here to tell you that the best case is also the average case. *If you always choose a random element in the array as the pivot*, quicksort will complete in O(*n* log *n*) time on average. Quicksort is one of the fastest sorting algorithms out there, and it's a very good example of D&C.

Exercises

How long would each of these operations take in Big O notation?

1. 4.5 Printing the value of each element in an array.
2. 4.6 Doubling the value of each element in an array.
3. 4.7 Doubling the value of just the first element in an array.
4. 4.8 Creating a multiplication table with all the elements in the array. So if your array is [2, 3, 7, 8, 10], you first multiply every element by 2, then multiply every

element by 3, then by 7, and so on.

	2	3	7	8	10
2	4	6	14	16	20
3	6	9	21	24	30
7	14	21	49	56	70
8	16	24	56	64	80
10	20	30	70	80	100

Recap

D&C works by breaking a problem down into smaller and smaller pieces. If you're using D&C on a list, the base case is probably an empty array or an array with one element.

If you're implementing quicksort, choose a random element as the pivot. The average runtime of quicksort is O(*n* log *n*)!

The constant in a function can matter sometimes. Given two algorithms with the same big-O running time, one can be consistently faster than the other. That's why quicksort is faster than merge sort.

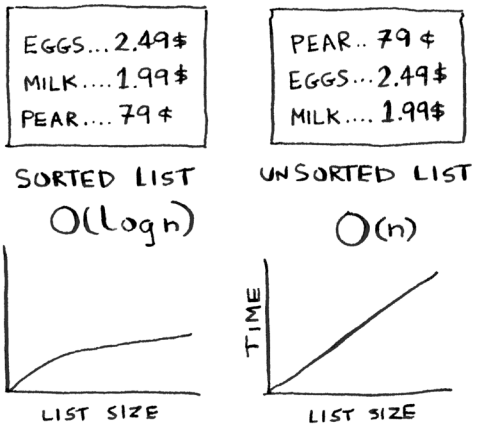
The constant almost never matters for simple search versus binary search, because O(log *n*) is so much faster than O(*n*) when your list gets big.

5 Hash tables

In this chapter

- You learn about hash tables, one of the most useful data structures. Hash tables have many uses; this chapter covers the common use cases.
- You learn about the internals of hash tables: implementation, collisions, and hash functions. This will help you understand how to analyze a hash table's performance.

Suppose you work at a grocery store. When a customer buys produce, you have to look up the price in a book. If the book is unalphabetized, it can take you a long time to look through every single line for *apple*. You'd be doing simple search from chapter 1, where you have to look at every line. Do you remember how long that would take? O(*n*) time. If the book is alphabetized, you could run binary search to find the price of an apple. That would only take O(log *n*) time.



As a reminder, there's a big difference between O(*n*) and O(log *n*) time! Suppose you could look through 10 lines of the book per second. Here's how long simple search and binary search would take you.

# OF ITEMS IN THE BOOK	O(n)	O(log n)
100	10 sec	1 sec ← YOU NEED TO CHECK $\log_2 100 = 7$ LINES
1000	1.66 min	1 sec ← NEED TO CHECK $\log_2 1000 = 10$ LINES
10000	16.6 min	2 sec ← $\log_2 10000 = 14$ LINES = 2 sec

You already know that binary search is darn fast. But as a cashier, looking things up in a book is a pain, even if the

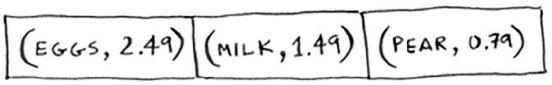
book is sorted. You can feel the customer steaming up as you search for items in the book. What you really need is a buddy who has all the names and prices memorized. Then you don't need to look up anything: you ask her, and she tells you the answer instantly.

Your buddy Maggie can give you the price in O(1) time for any item, no matter how big the book is. She's even faster than binary search.

# OF ITEMS IN THE BOOK	SIMPLE SEARCH O(n)	BINARY SEARCH O(log n)	MAGGIE O(1)
100	10 sec	1 sec	INSTANT
1000	1.6 min	1 sec	INSTANT
10000	16.6 min	2 sec	INSTANT

What a wonderful person! How do you get a "Maggie"?

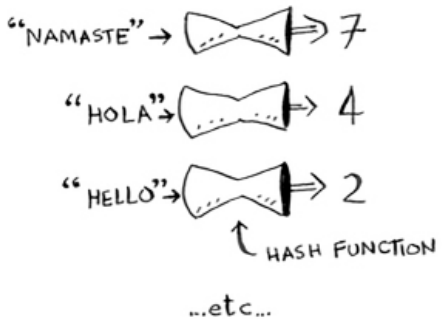
Let's put on our data structure hats. You know two data structures so far: arrays and lists (I won't talk about stacks because you can't really "search" for something in a stack). You could implement this book as an array.



Each item in the array is really two items: one is the name of a kind of produce, and the other is the price. If you sort this array by name, you can run binary search on it to find the price of an item. So you can find items in O(log *n*) time. But you want to find items in O(1) time. That is, you want to make a "Maggie." That's where hash functions come in.

Hash functions

A hash function is a function where you put in a string[1] and you get back a number.



In technical terminology, we'd say that a hash function "maps strings to numbers." You might think there's no discernable pattern to what number you get out when you put a string in. But there are some requirements for a hash function:

- It needs to be consistent. For example, suppose you put in "apple" and get back "4". Every time you put in

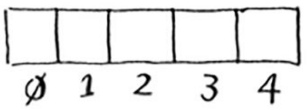
[1] *String* here means any kind of data—a sequence of bytes.

"apple", you should get "4" back. Without this, your hash table won't work.

- It should map different words to different numbers. For example, a hash function is no good if it always returns "1" for any word you put in. In the best case, every different word should map to a different number.

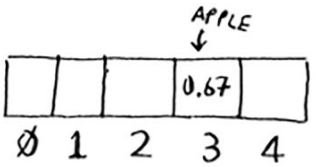
So a hash function maps strings to numbers. What is that good for? Well, you can use it to make your "Maggie"!

Start with an empty array:



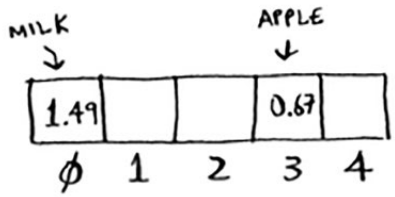
You'll store all of your prices in this array. Let's add the price of an apple. Feed "apple" into the hash function.

The hash function outputs "3". So let's store the price of an apple at index 3 in the array.

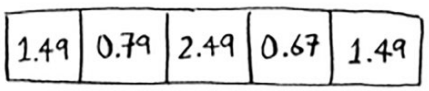


Let's add milk. Feed "milk" into the hash function.

The hash function says "0". Let's store the price of milk at index 0.

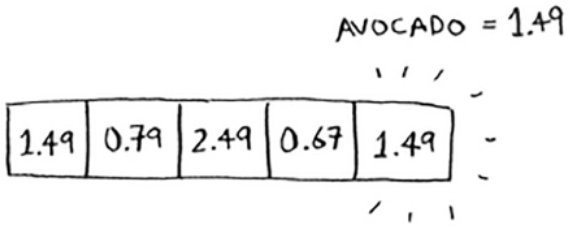


Keep going, and eventually the whole array will be full of prices.



Now you ask, "Hey, what's the price of an avocado?" You don't need to search for it in the array. Just feed "avocado" into the hash function.

It tells you that the price is stored at index 4. And sure enough, there it is.



The hash function tells you exactly where the price is stored, so you don't have to search at all! This works because

- The hash function consistently maps a name to the same index. Every time you put in "avocado", you'll get the same number back. So you can use it the first time to find where to store the price of an avocado, and then you can use it to find where you stored that price.

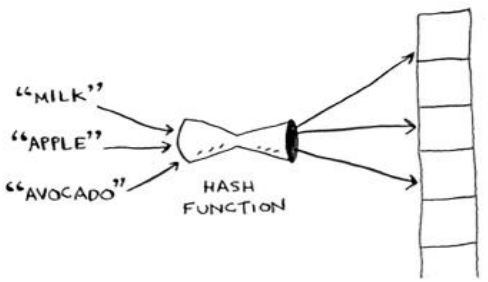
- The hash function maps different strings to different indexes. “Avocado” maps to index 4. “Milk” maps to index 0. Everything maps to a different slot in the array where you can store its price.
- The hash function knows how big your array is and only returns valid indexes. So if your array is 5 items, the hash function doesn’t return 100 … that wouldn’t be a valid index in the array.

You just built a “Maggie”! Put a hash function and an array together, and you get a data structure called a *hash table*. A hash table is the first data structure you’ll learn that has some extra logic behind it. Arrays and lists map straight to memory, but hash tables are smarter. They use a hash function to intelligently figure out where to store elements.

Hash tables are probably the most useful complex data structure you’ll learn. They’re also known as hash maps, maps, dictionaries, and associative arrays. And hash tables are fast! Remember our discussion of arrays and linked lists back in chapter 2? You can get an item from an array instantly. And hash tables use an array to store the data, so they’re equally fast.

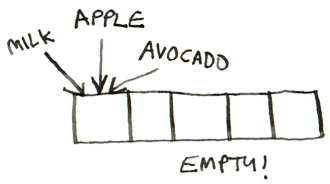
What's the catch?

The hash function we just saw is what the chaps call a *perfect hash function*. It deftly maps each grocery item to its very own slot in the array:



Look at them all snug in their own slots. In reality, you probably won't get a perfect 1-to-1 mapping like this. Your

items will need to share a room. Some grocery items will map to the same slot, while other slots will go empty:



There's a section on collisions coming up that discusses this. For now, just know that while hash tables are very useful, they rarely map items to slots so perfectly.

By the way, this kind of 1-to-1 mapping is called an *injective function*. Use that to impress your friends!

You'll probably never have to implement hash tables yourself. Any good language will have an implementation for hash tables. Python has hash tables; they're called *dictionaries*. You can make a new hash table using

```
>>> book = {}
```


AN EMPTY HASH TABLE

```
book is a new hash table. Let's add some prices to book:
>>> book["apple"] = 0.67     #A
>>> book["milk"] = 1.49       #B
>>> book["avocado"] = 1.49
>>> print(book)
{'avocado': 1.49, 'apple': 0.67, 'milk': 1.49}
```

#A An apple costs 67 cents.

#B Milk costs $1.49.

A HASH TABLE OF PRODUCE PRICES

Pretty easy! Now let's ask for the price of an avocado:

```
>>> print(book["avocado"])
1.49          #A
```

#A The price of an avocado

A hash table has keys and values. In the `book` hash, the names of produce are the keys, and their prices are the values. A hash table maps keys to values.

In the next section, you'll see some examples where hash tables are really useful.

Exercises

It's important for hash functions to consistently return the same output for the same input. If they don't, you won't be able to find your item after you put it in the hash table!

Which of these hash functions are consistent?

1. 5.1 f(x) = 1 #A

   #A Returns "1" for all input

2. 5.2 f(x) = random.random() #B

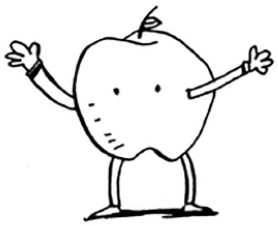
   #B Returns a random number every time

3. 5.3 f(x) = next_empty_slot() #C

   #C Returns the index of the next empty slot in the hash table

4. 5.4 f(x) = len(x) #D

#D Uses the length of the string as the index



Use cases

Hash tables are used everywhere. This section will show you a few use cases.

Using hash tables for lookups

Your phone has a handy phonebook built in.

Each name has a phone number associated with it.

BADE MAMA → 581 660 9820
ALEX MANNING → 484 234 4680
JANE MARIN → 415 567 3579

Suppose you want to build a phone book like this. You're mapping people's names to phone numbers. Your phone book needs to have this functionality:

- Add a person's name and the phone number associated with that person.
- Enter a person's name, and get the phone number associated with that name.

This is a perfect use case for hash tables! Hash tables are great when you want to

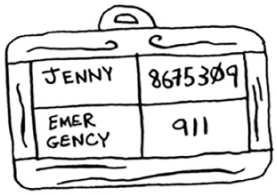
- Create a mapping from one thing to another thing
- Look something up

Building a phone book is pretty easy. First, make a new hash table:

```
>>> phone_book = {}
```

Let's add the phone numbers of some people into this phone book:

```
>>> phone_book["jenny"] = "8675309"
>>> phone_book["emergency"] = "911"
```



A HASH TABLE AS A PHONE BOOK

That's all there is to it! Now, suppose you want to find Jenny's phone number. Just pass the key in to the hash:

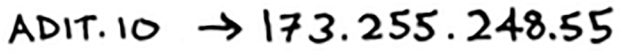
```
>>> print(phone_book["jenny"])
8675309      #A
```

#A Jenny's phone number

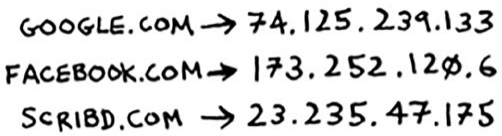
Imagine if you had to do this using an array instead. How would you do it? Hash tables make it easy to model a relationship from one item to another.

Hash tables are used for lookups on a much larger scale. For example, suppose you go to a website like

http://adit.io. Your computer has to translate adit.io to an IP address.

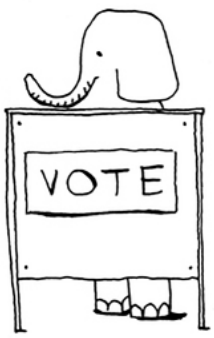


For any website you go to, the address has to be translated to an IP address.



Wow, mapping a web address to an IP address? Sounds like a perfect use case for hash tables! This process is called *DNS resolution*. Hash tables are one way to provide this functionality. Your computer has a *DNS cache* which stores this mapping for websites you have recently visited, and a good way to build a DNS cache is to use a hash table.

Preventing duplicate entries



Suppose you're running a voting booth. Naturally, every person can vote just once. How do you make sure they haven't voted before? When someone comes in to vote, you ask for their full name. Then you check it against the list of people who have voted.

If their name is on the list, this person has already voted—kick them out! Otherwise, you add their name to the list and let them vote. Now suppose a lot of people have come in to vote, and the list of people who have voted is really long.

Each time someone new comes in to vote, you have to scan this giant list to see if they've already voted. But there's a better way: use a hash!

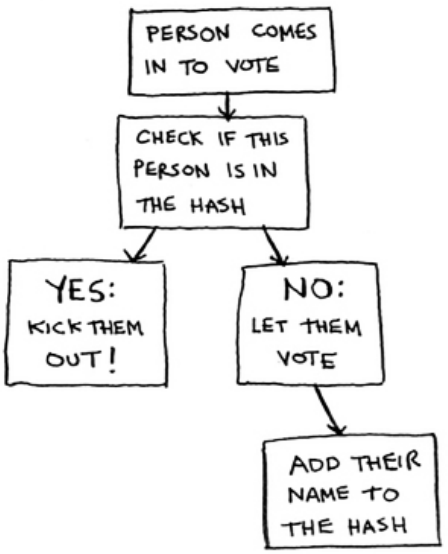
First, make a hash to keep track of the people who have voted:

```
>>> voted = {}
```

When someone new comes in to vote, check if they're already in the hash:

```
>>> value = voted.get("tom")
```

The `get` function returns the value if "tom" is in the hash table. Otherwise, it returns `None`. You can use this to check if someone has already voted!



Here's the code:

```
voted = {}

def check_voter(name):
  if name in voted:
    print("kick them out!")
  else:
    voted[name] = True
    print("let them vote!")
```

Let's test it a few times:

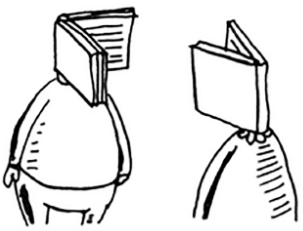
```
>>> check_voter("tom")
let them vote!
>>> check_voter("mike")
let them vote!
>>> check_voter("mike")
kick them out!
```

The first time Tom goes in, this will print, "let them vote!" Then Mike goes in, and it prints, "let them vote!" Then Mike tries to go a second time, and it prints, "kick them out!"

Remember, if you were storing these names in a list of people who have voted, this function would eventually become really slow, because it would have to run a simple

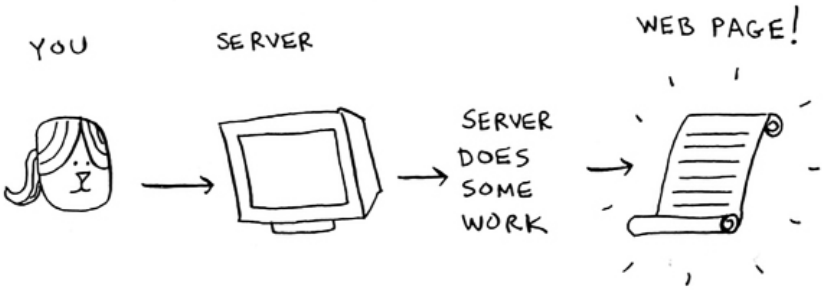
search over the entire list. But you're storing their names in a hash table instead, and a hash table instantly tells you whether this person's name is in the hash table or not. Checking for duplicates is very fast with a hash table.

Using hash tables as a cache



One final use case: caching. If you work on a website, you may have heard of caching before as a good thing to do. Here's the idea. Suppose you visit facebook.com:

1. You make a request to Facebook's server.
2. The server thinks for a second and comes up with the web page to send to you.
3. You get a web page.

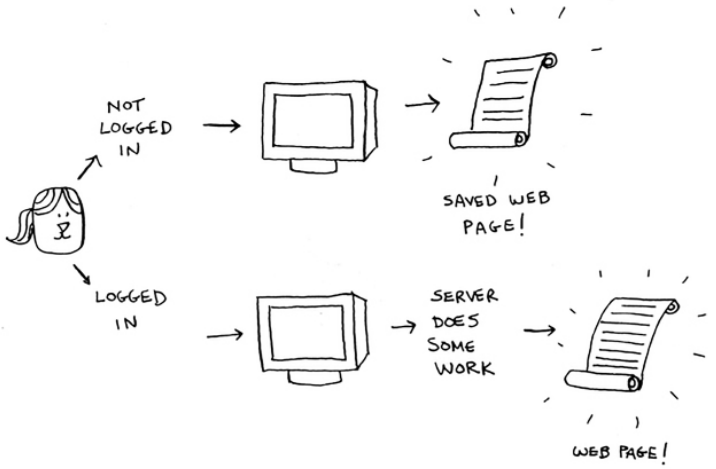


For example, on Facebook, the server may be collecting all of your friends' activity to show you. It takes a couple of seconds to collect all that activity and shows it to you. That couple of seconds can feel like a long time as a user. You might think, "Why is Facebook being so slow?" On the other hand, Facebook's servers have to serve millions of

people, and that couple of seconds adds up for them. Facebook's servers are really working hard to serve all of those websites. Is there a way to have Facebook's servers do less work?

Suppose you have a niece who keeps asking you about planets. "How far is Mars from Earth?" "How far is the Moon?" "How far is Jupiter?" Each time, you have to do a Google search and give her an answer. It takes a couple of minutes. Now, suppose she always asked, "How far is the Moon?" Pretty soon, you'd memorize that the Moon is 238,900 miles away. You wouldn't have to look it up on Google … you'd just remember and answer. This is how caching works: websites remember the data instead of recalculating it.

If you're logged in to Facebook, all the content you see is tailored just for you. Each time you go to facebook.com, its servers have to think about what content you're interested in. But if you're not logged in to Facebook, you see the login page. Everyone sees the same login page. Facebook is asked the same thing over and over: "Give me the home page when I'm logged out." So it stops making the server do work to figure out what the home page looks like. Instead, it memorizes what the home page looks like and sends it to you.



This is called *caching*. It has two advantages:

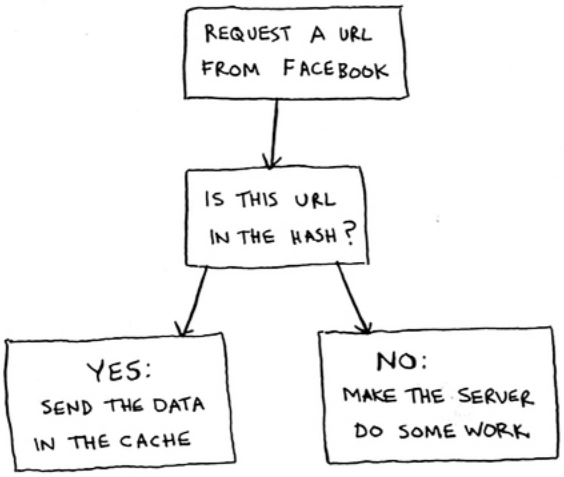
You get the web page a lot faster, just like when you memorized the distance from Earth to the Moon. The next time your niece asks you, you won't have to Google it. You can answer instantly.

Facebook has to do less work.

Caching is a common way to make things faster. All big websites use caching. And that data is cached in a hash!

Facebook isn't just caching the home page. It's also caching the About page, the Contact page, the Terms and Conditions page, and a lot more. So it needs a mapping from page URL to page data.

When you visit a page on Facebook, it first checks whether the page is stored in the hash.



Here it is in pseudocode:

```
cache = {}

def get_page(url):
  if cache.get(url):
    return cache[url]      #A
  else:
    data = get_data_from_server(url)
    cache[url] = data          #B
    return data
```

#A Returns cached data

#B Saves this data in your cache first

Here, you make the server do work only if the URL isn't in the cache. Before you return the data, though, you save it in the cache. The next time someone requests this URL, you can send the data from the cache instead of making the server do the work.

Recap

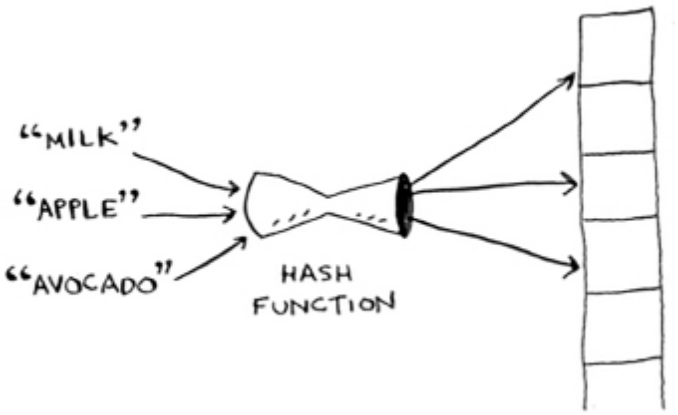
To recap, hashes are good for

- Modeling relationships from one thing to another thing
- Filtering out duplicates
- Caching/memoizing data instead of making your server do work

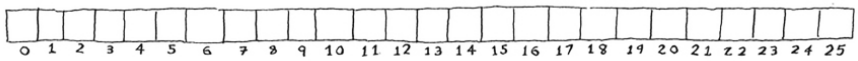
Collisions

Like I said earlier, most languages have hash tables. You don't need to know how to write your own. So, I won't talk about the internals of hash tables too much. But you still care about performance! To understand the performance of hash tables, you first need to understand what collisions are. The next two sections cover collisions and performance.

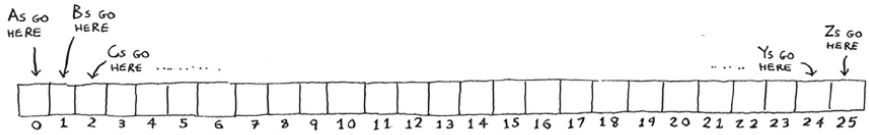
First, I've been telling you a white lie. I told you that a hash function always maps different keys to different slots in the array.



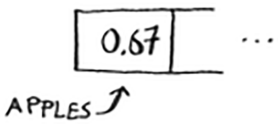
In reality, it's almost impossible to write a hash function that does this. Let's take a simple example. Suppose your array contains 26 slots.



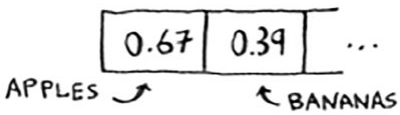
And your hash function is really simple: it assigns a spot in the array alphabetically.



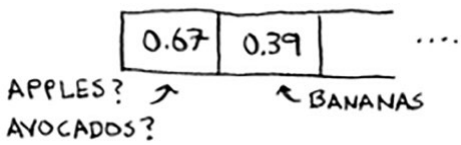
Maybe you can already see the problem. You want to put the price of apples in your hash. You get assigned the first slot.



Then you want to put the price of bananas in the hash. You get assigned the second slot.

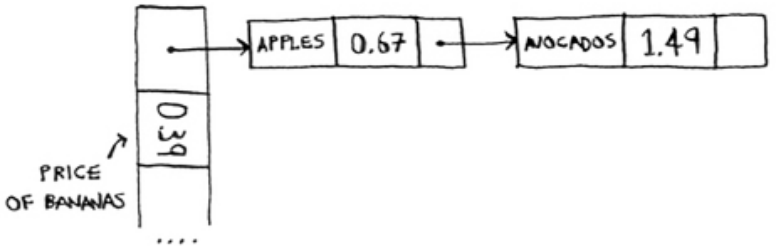


Everything is going so well! But now you want to put the price of avocados in your hash. You get assigned the first slot again.

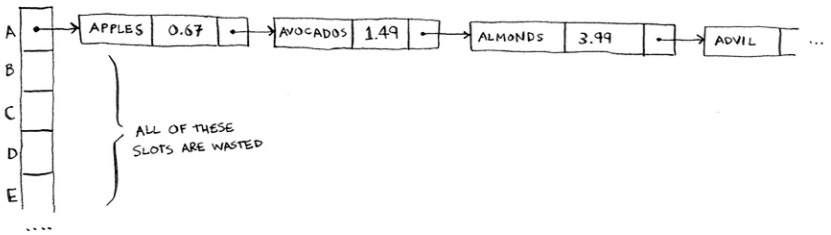


Oh no! Apples have that slot already! What to do? This is called a *collision*: two keys have been assigned the same slot. This is a problem. If you store the price of avocados at that slot, you'll overwrite the price of apples. Then the next

time someone asks for the price of apples, they will get the price of avocados instead! Collisions are bad, and you need to work around them. There are many different ways to deal with collisions. The simplest one is this: if multiple keys map to the same slot, start a linked list at that slot.



In this example, both "apple" and "avocado" map to the same slot. So you start a linked list at that slot. If you need to know the price of bananas, it's still quick. If you need to know the price of apples, it's a little slower. You have to search through this linked list to find "apple". If the linked list is small, no big deal—you have to search through three or four elements. But suppose you work at a grocery store where you only sell produce that starts with the letter *A*.



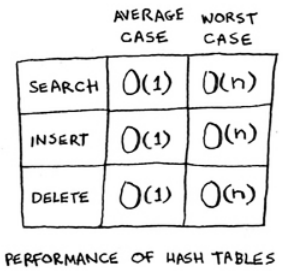
Hey, wait a minute! The entire hash table is totally empty except for one slot. And that slot has a giant linked list! Every single element in this hash table is in the linked list. That's as bad as putting everything in a linked list to begin with. It's going to slow down your hash table.

There are two lessons here:

- *Your hash function is really important.* Your hash function mapped all the keys to a single slot. Ideally, your hash function would map keys evenly all over the hash.
- If those linked lists get long, it slows down your hash table a lot. But they won't get long if you *use a good hash function*!

Hash functions are important. A good hash function will give you very few collisions. So how do you pick a good hash function? That's coming up in the next section!

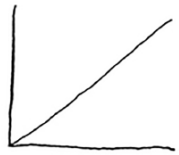
Performance



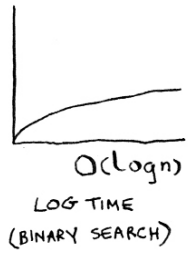
PERFORMANCE OF HASH TABLES

You started this chapter at the grocery store. You wanted to build something that would give you the prices for produce *instantly*. Well, hash tables are really fast.

In the average case, hash tables take O(1) for everything. O(1) is called *constant time*. You haven't seen constant time before. It doesn't mean instant. It means the time taken will stay the same, regardless of how big the hash table is. For example, you know that simple search takes linear time.



Binary search is faster—it takes log time:



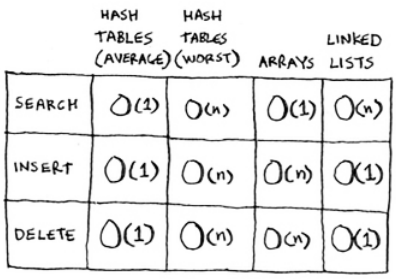
Looking something up in a hash table takes constant time.

See how it's a flat line? That means it doesn't matter whether your hash table has 1 element or 1 billion elements—getting something out of a hash table will take the same amount of time. Actually, you've seen constant time before. Getting an item out of an array takes constant time. It doesn't matter how big your array is; it takes the same amount of time to get an element. In the average case, hash tables are really fast.

In the worst case, a hash table takes O(*n*)—linear time—for everything, which is really slow. Let's compare hash tables to arrays and lists.

Side note: other ways to resolve collisions

here's a different way to handle collisions, called cuckoo hashing. With cuckoo hashing, you handle the collision during insertion, not during look up. So in the worst case, look ups will still be constant time.



	HASH TABLES (AVERAGE)	HASH TABLES (WORST)	ARRAYS	LINKED LISTS
SEARCH	O(1)	O(n)	O(1)	O(n)
INSERT	O(1)	O(n)	O(n)	O(1)
DELETE	O(1)	O(n)	O(n)	O(1)

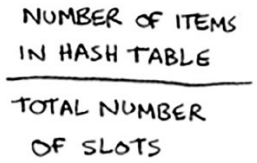
Look at the average case for hash tables. Hash tables are as fast as arrays at searching (getting a value at an index). And they're as fast as linked lists at inserts and deletes. It's the best of both worlds! But in the worst case, hash tables are slow at all of those. So it's important that you don't hit worst-case performance with hash tables. And to do that, you need to avoid collisions. To avoid collisions, you need

- A low load factor
- A good hash function

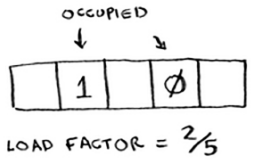
> **NOTE** Before you start this next section, know that this isn't required reading. I'm going to talk about how to implement a hash table, but you'll never have to do that yourself. Whatever programming language you use will have an implementation of hash tables built in. You can use the built-in hash table and assume it will have good performance. The next section gives you a peek under the hood.

Load factor

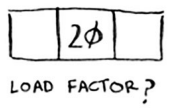
The load factor of a hash table is easy to calculate.



Hash tables use an array for storage, so you count the number of occupied slots in an array. For example, this hash table has a load factor of 2/5, or 0.4.

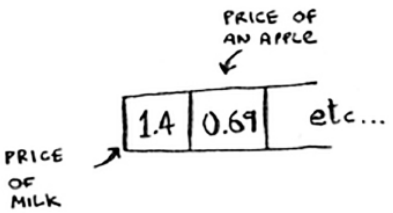


What's the load factor of this hash table?



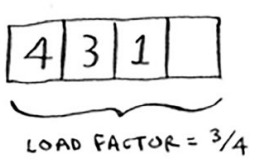
If you said 1/3, you're right. Load factor measures how many empty slots remain in your hash table.

Suppose you need to store the price of 100 produce items in your hash table, and your hash table has 100 slots. In the best case, each item will get its own slot.

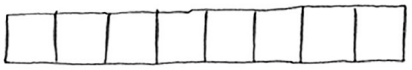


This hash table has a load factor of 1. What if your hash table has only 50 slots? Then it has a load factor of 2.

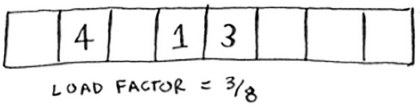
There's no way each item will get its own slot, because there aren't enough slots! Having a load factor greater than 1 means you have more items than slots in your array. Once the load factor starts to grow, you need to add more slots to your hash table. This is called *resizing*. For example, suppose you have this hash table that is getting pretty full.



You need to resize this hash table. First you create a new array that's bigger. The rule of thumb is to make an array that is twice the size.



Now you need to re-insert all of those items into this new hash table using the `hash` function:

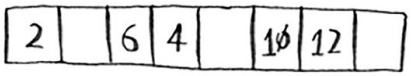


This new table has a load factor of 3/8. Much better! With a lower load factor, you'll have fewer collisions, and your table will perform better. A good rule of thumb is, resize when your load factor is greater than 0.7.

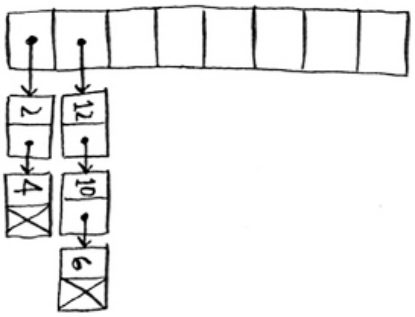
You might be thinking, "This resizing business takes a lot of time!" And you're right. Resizing is expensive, and you don't want to resize too often. But averaged out, hash tables take O(1) even with resizing.

A good hash function

A good hash function distributes values in the array evenly.



A bad hash function groups values together and produces a lot of collisions.



What is a good hash function? That's something you'll never have to worry about—folks with big beards sit in dark rooms and worry about that. If you're really curious, look up CityHash. That's what Google's Abseil library uses. Abseil is an open source C++ library based on internal Google code. It provides all kinds of general-purpose C++ functions. Abseil is a building block for Google's code, so if it uses CityHash, you can be sure that CityHash is pretty good. You could use that as your hash function.

Exercises

It's important for hash functions to have a good distribution. They should map items as broadly as possible. The worst case is a hash function that maps all items to the same slot in the hash table.

Suppose you have these four hash functions that work with strings:

1. a. Return "1" for all input.
2. b. Use the length of the string as the index.
3. c. Use the first character of the string as the index. So, all strings starting with *a* are hashed together, and so on.
4. d. Map every letter to a prime number: a = 2, b = 3, c = 5, d = 7, e = 11, and so on. For a string, the hash function is the sum of all the characters modulo the size of the hash. For example, if your hash size is 10, and the string is "bag", the index is 3 + 2 + 17 % 10 = 22 % 10 = 2.

For each of these examples, which hash functions would provide a good distribution? Assume a hash table size of 10 slots.

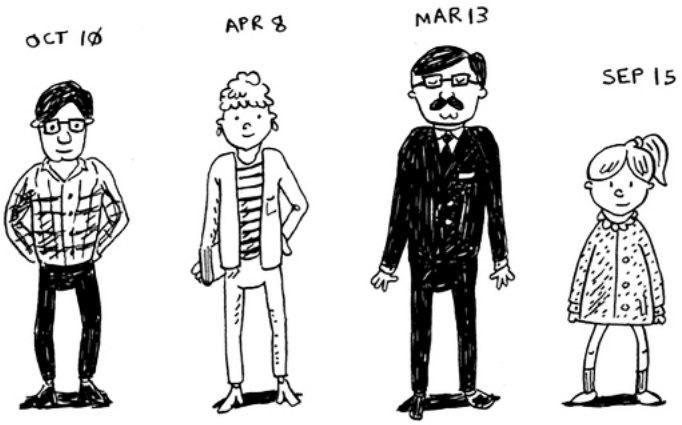
5. 5.5 A phonebook where the keys are names and values are phone numbers. The names are as follows: Esther, Ben, Bob, and Dan.
6. 5.6 A mapping from battery size to power. The sizes are A, AA, AAA, and AAAA.
7. 5.7 A mapping from book titles to authors. The titles are *Maus*, *Fun Home*, and *Watchmen*.

Recap

You'll almost never have to implement a hash table yourself. The programming language you use should provide an implementation for you. You can use Python's hash tables and assume that you'll get the average case performance: constant time.

Hash tables are a powerful data structure because they're so fast and they let you model data in a different way. You might soon find that you're using them all the time:

- You can make a hash table by combining a hash function with an array.
- Collisions are bad. You need a hash function that minimizes collisions.
- Hash tables have really fast search, insert, and delete.
- Hash tables are good for modeling relationships from one item to another item.
- Once your load factor is greater than 0.7, it's time to resize your hash table.
- Hash tables are used for caching data (for example, with a web server).
- Hash tables are great for catching duplicates.



Breadth-first search

6

In this chapter

- You learn how to model a network using a new, abstract data structure: graphs
- You learn breadth-first search, an algorithm you can run on graphs to answer questions like, "What's the shortest path to go to X?"
- You learn about directed versus undirected graphs.
- You learn topological sort, a different kind of sorting algorithm that exposes dependencies between nodes.

This chapter introduces graphs. First, I'll talk about what graphs are (they don't involve an X or Y axis). Then I'll show you your first graph algorithm. It's called *breadth-first search* (BFS).

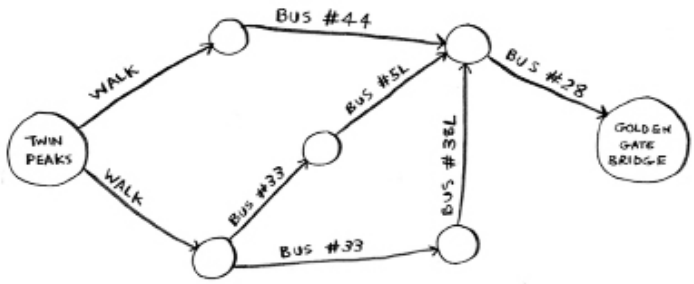
Breadth-first search allows you to find the shortest distance between two things. But shortest distance can mean a lot of things! You can use breadth-first search to

- Write a spell checker (fewest edits from your misspelling to a real word—for example, READED -> READER is one edit)
- Find the doctor closest to you in your network
- Build a search engine crawler

Graph algorithms are some of the most useful algorithms I know. Make sure you read the next few chapters carefully—these are algorithms you'll be able to apply again and again.

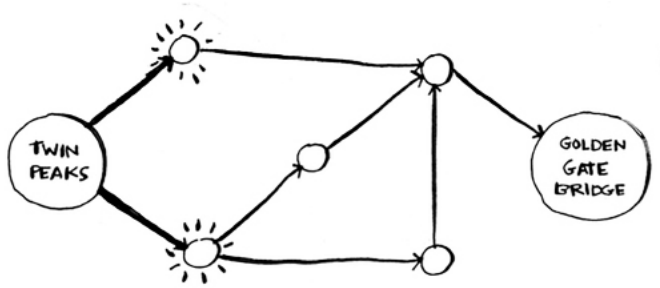
Introduction to graphs

Suppose you're in San Francisco, and you want to go from Twin Peaks to the Golden Gate Bridge. You want to get there by bus, with the minimum number of transfers. Here are your options.

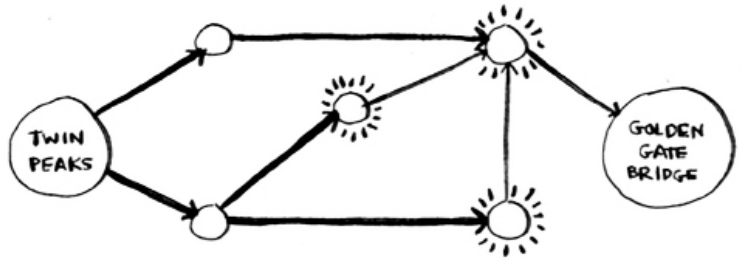


What's your algorithm to find the path with the fewest steps?

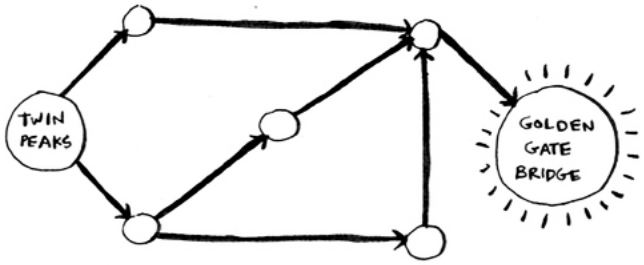
Well, can you get there in one step? Here are all the places you can get to in one step.



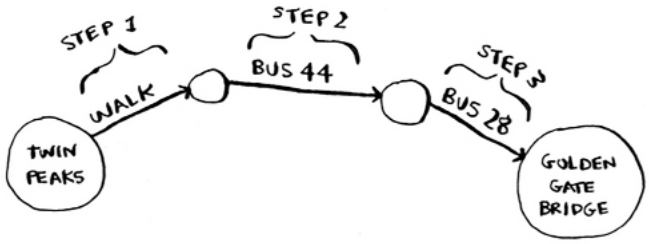
The bridge isn't highlighted; you can't get there in one step. Can you get there in two steps?



Again, the bridge isn't there, so you can't get to the bridge in two steps. What about three steps?



Aha! Now the Golden Gate Bridge shows up. So it takes three steps to get from Twin Peaks to the bridge using this route.



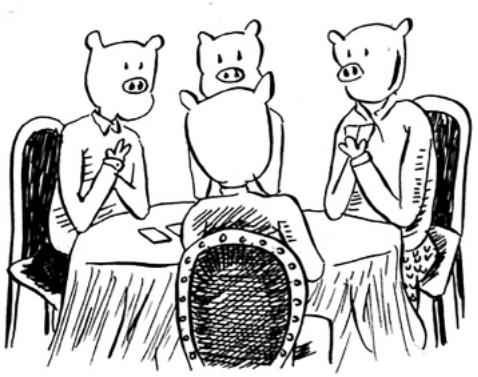
There are other routes that will get you to the bridge too, but they're longer (four steps). The algorithm found that the shortest route to the bridge is three steps long. This type of problem is called a *shortest-path problem*. You're always trying to find the shortest something. It could be the shortest route to your friend's house. Or maybe you're browsing the web. Without you knowing it, the network is looking for the shortest path between your computer and a website's server. The algorithm to solve a shortest-path problem is called *breadth-first search*.

To figure out how to get from Twin Peaks to the Golden Gate Bridge, there are two steps:

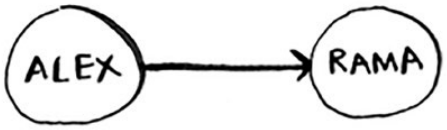
1. Model the problem as a graph.
2. Solve the problem using breadth-first search.

Next I'll cover what graphs are. Then I'll go into breadth-first search in more detail.

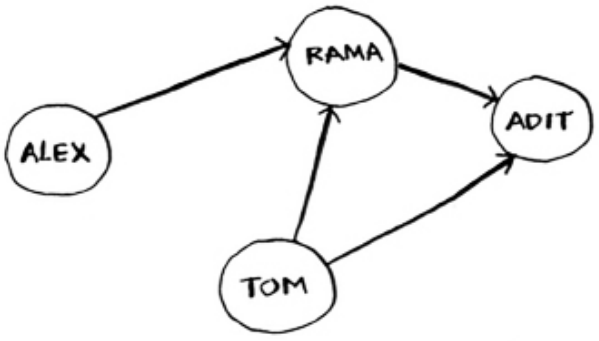
What is a graph?



A graph models a set of connections. For example, suppose you and your friends are playing poker, and you want to model who owes whom money. Here's how you could say, "Alex owes Rama money."



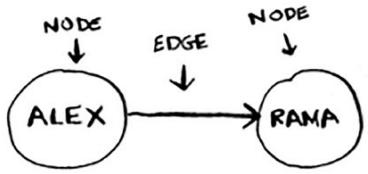
The full graph could look something like this.



Graph of people who owe other people poker money

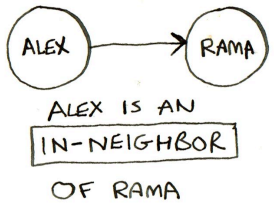
Alex owes Rama money, Tom owes Adit money, and so on.

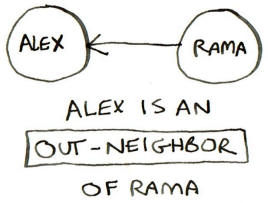
Each graph is made up of *nodes* and *edges*.



That's all there is to it! Graphs are made up of nodes and edges. A node can be directly connected to many other nodes. Those nodes are called its in-*neighbors* or *out-neighbors*.

Since Alex is pointing to Rama, Alex is Rama's *in-neighbor* (and Rama is Alex's *out-neighbor*). This terminology can be confusing, so here's a diagram to help:





In this graph, Adit isn't Alex's in-neighbor or out-neighbor, because they aren't directly connected. But Adit is Rama's and Tom's out-neighbor.

Graphs are a way to model how different things are connected to one another. Now let's see breadth-first search in action.

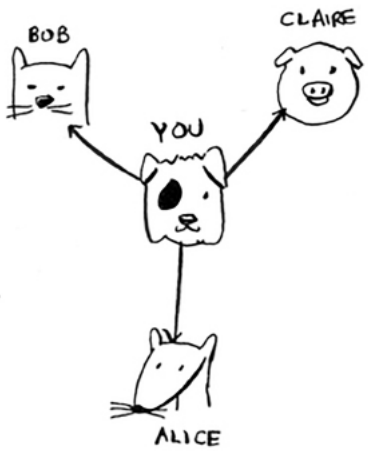
Breadth-first search

We looked at a search algorithm in chapter 1: binary search. Breadth-first search is a different kind of search algorithm: one that runs on graphs. It can help answer two types of questions:

- Question type 1: Is there a path from node A to node B?
- Question type 2: What is the shortest path from node A to node B?

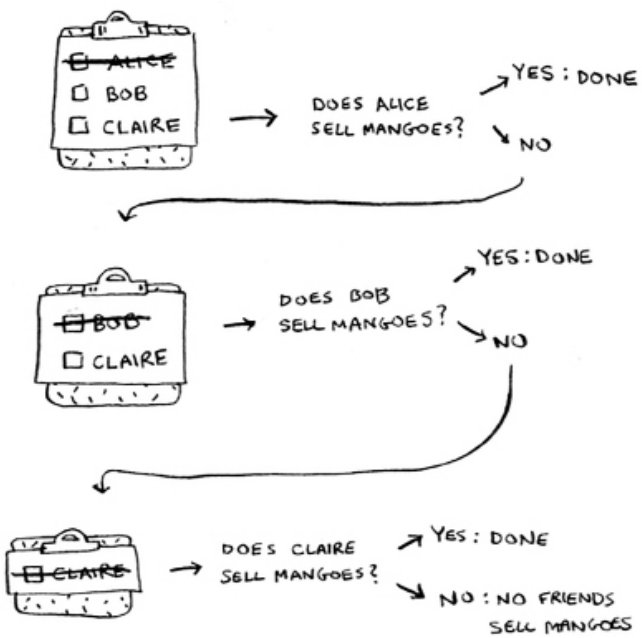
You already saw breadth-first search once, when you calculated the shortest route from Twin Peaks to the Golden Gate Bridge. That was a question of type 2: "What is the shortest path?" Now let's look at the algorithm in more detail. You'll ask a question of type 1: "Is there a path?"

Suppose you're the proud owner of a mango farm. You're looking for a mango seller who can sell your mangoes. Are you connected to a mango seller on Facebook? Well, you can search through your friends.

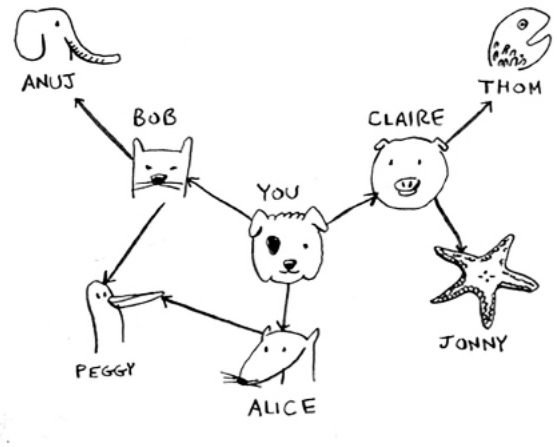


This search is pretty straightforward. First, make a list of friends to search.

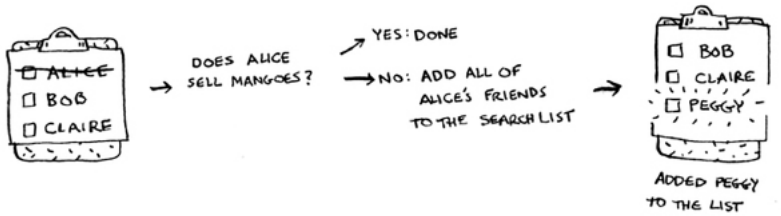
Now, go to each person in the list and check whether that person sells mangoes.



Suppose none of your friends are mango sellers. Now you have to search through your friends' friends.



Each time you search for someone from the list, add all of their friends to the list.



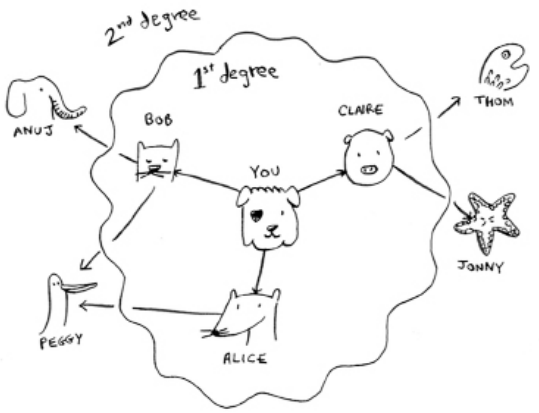
This way, you not only search your friends, but you search their friends, too. Remember, the goal is to find one mango seller in your network. So if Alice isn't a mango seller, you add her friends to the list, too. That means you'll eventually search her friends—and then their friends, and so on. With this algorithm, you'll search your entire network until you come across a mango seller. This algorithm is breadth-first search.

Finding the shortest path

As a recap, these are the two questions that breadth-first search can answer for you:

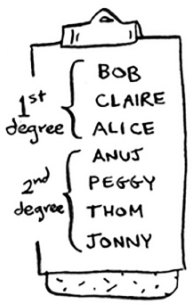
- Question type 1: Is there a path from node A to node B? (Is there a mango seller in your network?)
- Question type 2: What is the shortest path from node A to node B? (Who is the closest mango seller?)

You saw how to answer question 1; now let's try to answer question 2. Can you find the closest mango seller? For example, your friends are first-degree connections, and their friends are second-degree connections.



You'd prefer a first-degree connection to a second-degree connection, and a second-degree connection to a third-degree connection, and so on. So you shouldn't search any second-degree connections before you make sure you don't have a first-degree connection who is a mango seller. Well, breadth-first search already does this! The way breadth-first search works, the search radiates out from the starting point. So you'll check first-degree connections before second-degree connections. Pop quiz: who will be checked first, Claire or Anuj? Answer: Claire is a first-degree connection, and Anuj is a second-degree connection. So Claire will be checked before Anuj.

Another way to see this is, first-degree connections are added to the search list before second-degree connections.



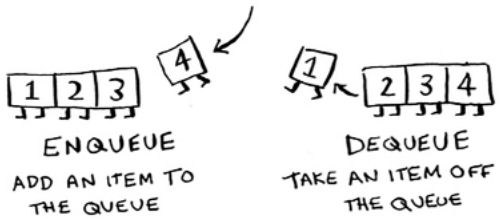
You just go down the list and check people to see whether

each one is a mango seller. The first-degree connections will be searched before the second-degree connections, so you'll find the mango seller closest to you. Breadth-first search not only finds a path from A to B, it also finds the shortest path.

Notice that this only works if you search people in the same order in which they're added. That is, if Claire was added to the list before Anuj, Claire needs to be searched before Anuj. What happens if you search Anuj before Claire, and they're both mango sellers? Well, Anuj is a second-degree contact, and Claire is a first-degree contact. You end up with a mango seller who isn't the closest to you in your network. So you need to search people in the order that they're added. There's a data structure for this: it's called *a queue*.

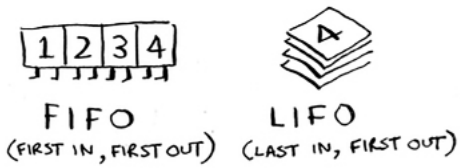
Queues

A queue works exactly like it does in real life. Suppose you and your friend are queueing up at the bus stop. If you're before him in the queue, you get on the bus first. A queue works the same way. Queues are similar to stacks. You can't access random elements in the queue. Instead, there are only two operations, *enqueue* and *dequeue*.



If you enqueue two items to the list, the first item you added will be dequeued before the second item. You can use this for your search list! People who are added to the list first will be dequeued and searched first.

The queue is called a *FIFO* data structure: First In, First Out. In contrast, a stack is a *LIFO* data structure: Last In, First Out.

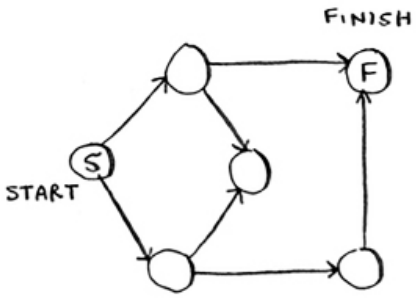


Now that you know how a queue works, let's implement breadth-first search!

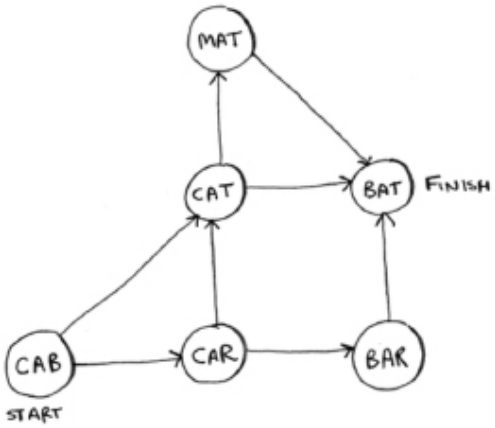
Exercises

Run the breadth-first search algorithm on each of these graphs to find the solution.

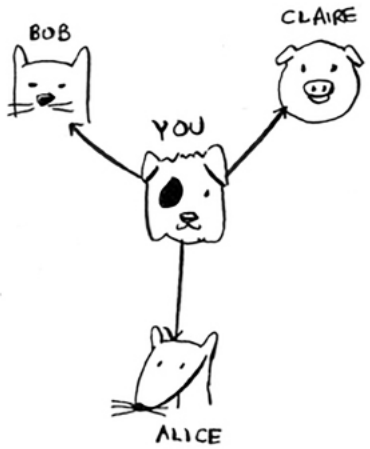
1. 6.1 Find the length of the shortest path from start to finish.



2. 6.2 Find the length of the shortest path from “cab” to “bat”.



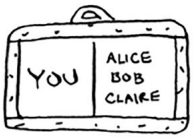
Implementing the graph



First, you need to implement the graph in code. A graph consists of several nodes.

And each node is connected to other nodes. How do you express a relationship like "you -> bob"? Luckily, you know a data structure that lets you express relationships: *a hash table*!

Remember, a hash table allows you to map a key to a value. In this case, you want to map a node to all of its out-neighbors.

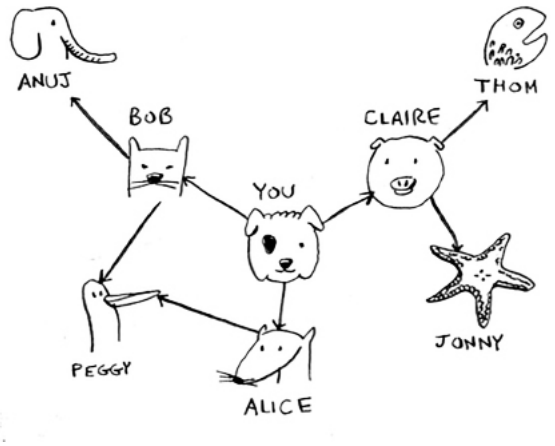


Here's how you'd write it in Python:

```
graph = {}
graph[“you”] = [“alice”, “bob”, “claire”]
```

Notice that "you" is mapped to an array. So `graph["you"]` will give you an array of all the out-neighbors of "you". Remember that the out-neighbors are the nodes that the "you" node points to.

A graph is just a bunch of nodes and edges, so this is all you need to have a graph in Python. What about a bigger graph, like this one?



Here it is as Python code:

```
graph = {}
graph[“you”] = [“alice”, “bob”, “claire”]
graph[“bob”] = [“anuj”, “peggy”]
graph[“alice”] = [“peggy”]
graph[“claire”] = [“thom”, “jonny”]
graph[“anuj”] = []
graph[“peggy”] = []
graph[“thom”] = []
graph[“jonny”] = []
```

Pop quiz: does it matter what order you add the key/value pairs in? Does it matter if you write

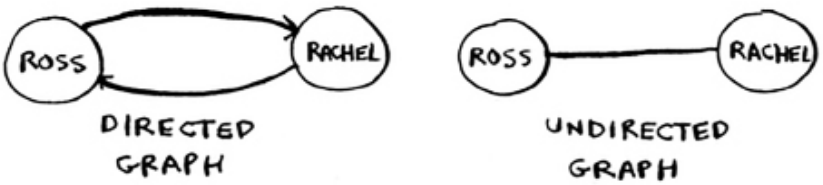
```
graph[“claire”] = [“thom”, “jonny”]
graph[“anuj”] = []
```

instead of

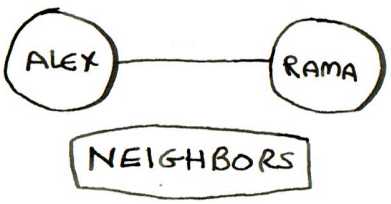
```
graph[“anuj”] = []
graph[“claire”] = [“thom”, “jonny”]
```

Think back to the previous chapter. Answer: It doesn’t matter. Hash tables have no ordering, so it doesn’t matter what order you add key/value pairs in.

Anuj, Peggy, Thom, and Jonny don’t have any out-neighbors. They have in-neighbors, since they have arrows pointing to them – but no arrows from them to someone else. This is called a *directed graph*: the relationship is only one way. An undirected graph doesn’t have any arrows. For example, both of these graphs are equal.

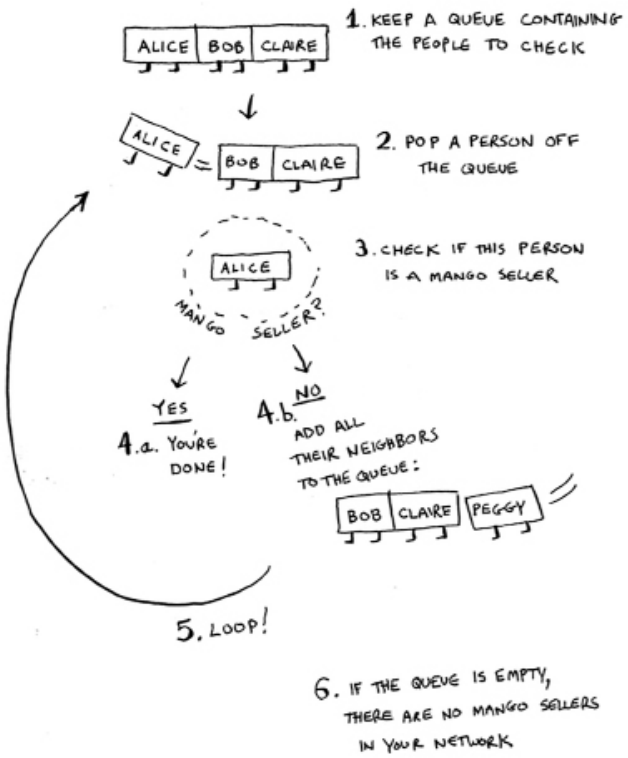


If you have an undirected graph, you can forget the terms in-neighbor and out-neighbor, and use the simpler term *neighbor*.



Implementing the algorithm

To recap, here's how the implementation will work.



NOTE When updating queues, I use the terms *enqueue* and *dequeue*. You'll also encounter the terms *push* and *pop*. *Push* is almost always the same thing as *enqueue*, and *pop* is almost always the same thing as *dequeue*.

Make a queue to start. In Python, you use the double-ended queue (`deque`) function for this:

```
from collections import deque
search_queue = deque()       #A
search_queue += graph["you"]   #B
```

#A Creates a new queue

#B Adds all of your out-neighbors to the search queue

Remember, `graph["you"]` will give you a list of all your out-neighbors, like `["alice", "bob", "claire"]`. Those all get added to the search queue.

Let's see the rest:

```
while search_queue:     #A
    person = search_queue.popleft()  #B
    if person_is_seller(person):         #C
        print(person + " is a mango seller!") #D
        return True
    else:
        search_queue += graph[person]           #E
return False                        #F
```

#A While the queue isn't empty …

#B … grabs the first person off the queue

#C Checks whether the person is a mango seller

#D Yes, they're a mango seller.

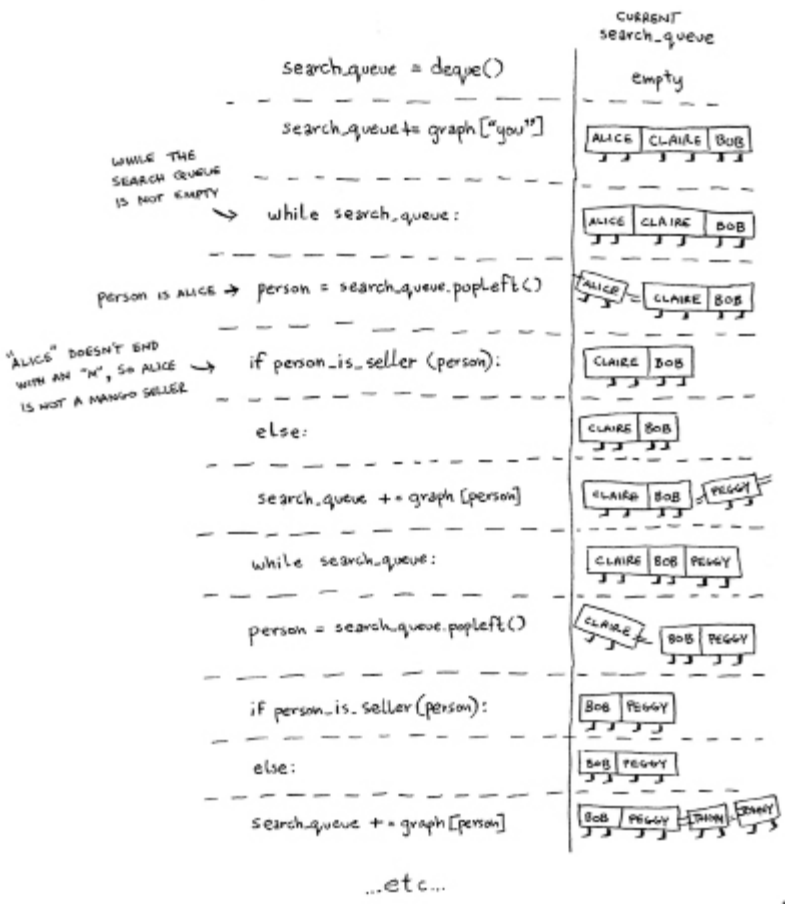
#E No, they aren't. Add all of this person's friends to the search queue.

#F If you reached here, no one in the queue was a mango seller.

One final thing: you still need a `person_is_seller` function to tell you when someone is a mango seller. Here's one:

```
def person_is_seller(name):
    return name[-1] == 'm'
```

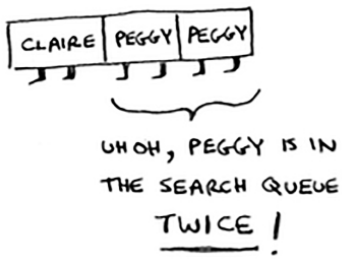
This function checks whether the person's name ends with the letter *m*. If it does, they're a mango seller. Kind of a silly way to do it, but it'll do for this example. Now let's see the breadth-first search in action.



And so on. The algorithm will keep going until either

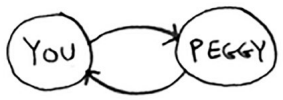
- A mango seller is found, or
- The queue becomes empty, in which case there is no mango seller.

Alice and Bob share a friend: Peggy. So Peggy will be added to the queue twice: once when you add Alice's friends, and again when you add Bob's friends. You'll end up with two Peggys in the search queue.



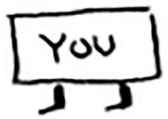
But you only need to check Peggy once to see whether she's a mango seller. If you check her twice, you're doing unnecessary, extra work. So once you search a person, you should mark that person as searched and not search them again.

If you don't do this, you could also end up in an infinite loop. Suppose the mango seller graph looked like this.



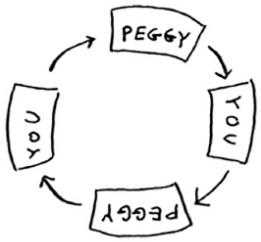
To start, the search queue contains all of your out-neighbors.

Now you check Peggy. She isn't a mango seller, so you add all of her out-neighbors to the search queue.



Next, you check yourself. You're not a mango seller, so you add all of your out-neighbors to the search queue.

And so on. This will be an infinite loop, because the search queue will keep going from you to Peggy.



Before checking a person, it's important to make sure they haven't been checked already. To do that, you'll keep a list of people you've already checked.

Here's the final code for breadth-first search, taking that into account:

```
def search(name):
    search_queue = deque()
    search_queue += graph[name]
    searched = set()      #A
    while search_queue:
        person = search_queue.popleft()
        if not person in searched:          #B
            if person_is_seller(person):
                print(person + " is a mango seller!")
                return True
            else:
                search_queue += graph[person]
                searched.add(person)           #C
    return False

search("you")
```

#A This **set** is how you keep track of which people you've searched before.

#B Only search this person if you haven't already searched them.

#C Marks this person as searched

Try running this code yourself. Maybe try changing the `person_is_seller` function to something more meaningful, and see if it prints what you expect.

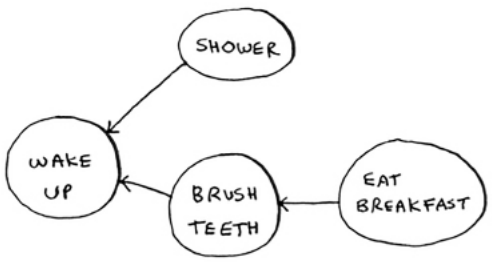
Running time

If you search your entire network for a mango seller, that means you'll follow each edge (remember, an edge is the arrow or connection from one person to another). So the running time is at least O(number of edges).

You also keep a queue of every person to search. Adding one person to the queue takes constant time: O(1). Doing this for every person will take O(number of people) total. Breadth-first search takes O(number of people + number of edges), and it's more commonly written as O(V+E) (V for number of vertices, E for number of edges).

Exercise

Here's a small graph of my morning routine.



It tells you that I can't eat breakfast until I've brushed my teeth. So "eat breakfast" *depends on* "brush teeth".

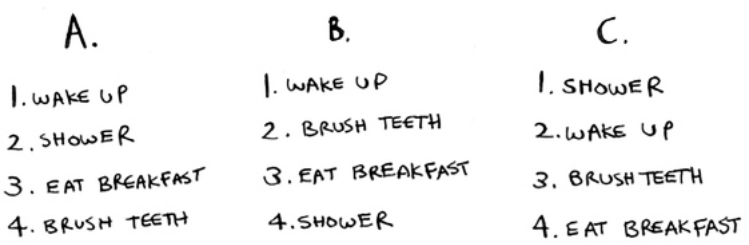
On the other hand, showering doesn't depend on brushing

my teeth, because I can shower before I brush my teeth. From this graph, you can make a list of the order in which I need to do my morning routine:

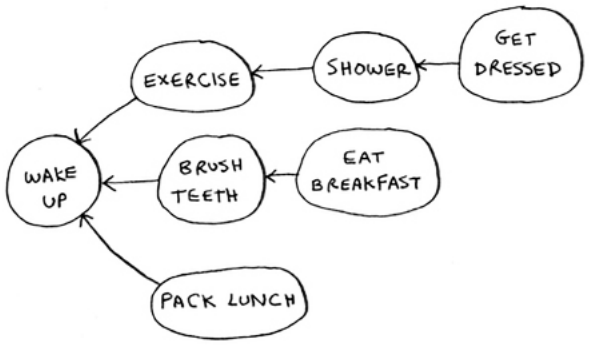
1. Wake up.
2. Shower.
3. Brush teeth.
4. Eat breakfast.

Note that "shower" can be moved around, so this list is also valid:

1. Wake up.
2. Brush teeth.
3. Shower.
4. Eat breakfast.
5. 6.3 For these three lists, mark whether each one is valid or invalid.



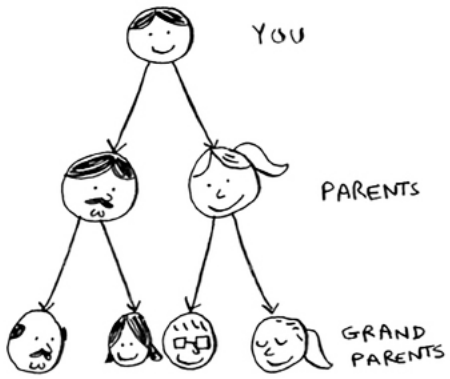
6. 6.4 Here's a larger graph. Make a valid list for this graph.



You could say that this list is sorted, in a way. If task A depends on task B, task A shows up later in the list. This is called a *topological sort*, and it's a way to make an ordered list out of a graph. Suppose you're planning a wedding and have a large graph full of tasks to do—and you're not sure where to start. You could *topologically sort* the graph

and get a list of tasks to do, in order.

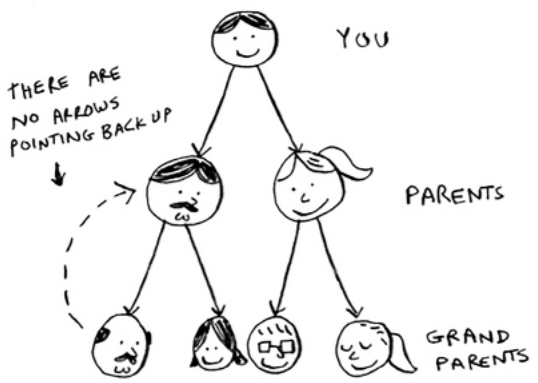
Suppose you have a family tree.



This is a graph, because you have nodes (the people) and edges.

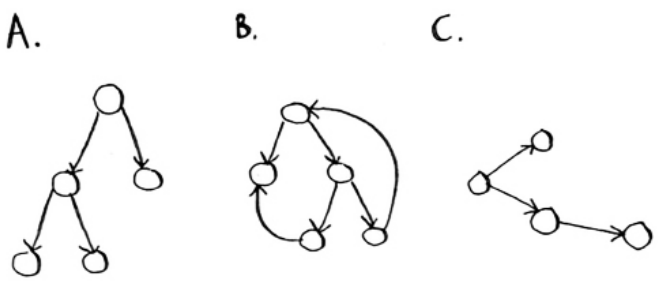
The edges point to the nodes' parents. But all the edges go down—it wouldn't make sense for a family tree to have an edge pointing back up! That would be meaningless—your

dad can't be your grandfather's dad!



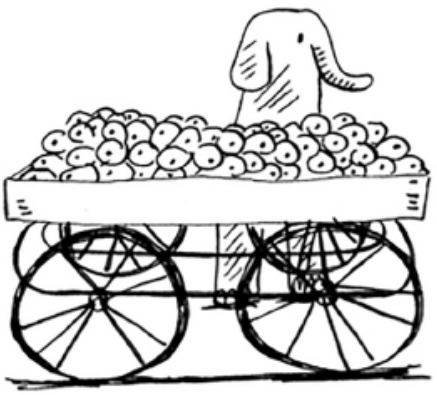
This is called a *tree*. A tree is a special type of graph, where no edges ever point back.

7. 6.5 Which of the following graphs are also trees?



Recap

- Breadth-first search tells you if there's a path from A to B.
- If there's a path, breadth-first search will find the shortest path.
- If you have a problem like "find the shortest X," try modeling your problem as a graph, and use breadth-first search to solve.
- A directed graph has arrows, and the relationship follows the direction of the arrow (rama -> adit means "rama owes adit money").
- Undirected graphs don't have arrows, and the relationship goes both ways (ross - rachel means "ross dated rachel and rachel dated ross").
- Queues are FIFO (First In, First Out).
- Stacks are LIFO (Last In, First Out).
- You need to check people in the order they were added to the search list, so the search list needs to be a queue. Otherwise, you won't get the shortest path.
- Once you check someone, make sure you don't check them again. Otherwise, you might end up in an infinite loop.



Trees 7

In this chapter

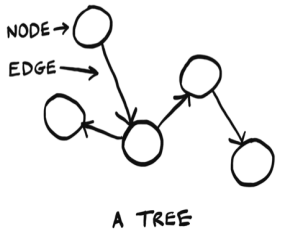
- You learn what a tree is, and the difference between trees and graphs
- You get comfortable with running an algorithm over a tree
- You learn depth-first search, and see the difference between depth-first search and breadth-first search
- You learn Huffman coding, a compression algorithm that makes use of trees

What do compression algorithms and database storage have in common? There is often a tree underneath doing all the hard work. Trees are a subset of graphs. They are worth covering separately as there are many specialized types of trees. For example, Huffman coding, which is a compression algorithm you will learn in this chapter, uses binary trees. Most databases on the other hand, use a balanced tree like a B-tree, which you will learn about in the next chapter. There are so many types of trees out there. These two chapters will give you the vocabulary and concepts you need to understand them.

Your first tree

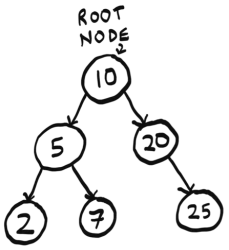
Trees are a type of graph. We will have a more thorough definition later. First let's learn some terminology and look at an example.

Just like graphs, trees are made of nodes and edges:

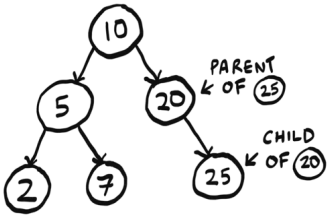


A TREE

In this book, we will work with rooted trees. Rooted trees have one node that leads to all the other nodes:

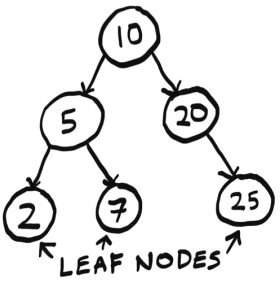


We will work exclusively with rooted trees, so when I say "tree" in this chapter, I mean a rooted tree. Nodes can have children, and child nodes can have a parent:



In a tree, nodes have at most one parent. The only node with no parents is the root. Nodes with no children are

called leaf nodes:

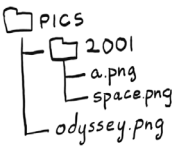


If you understand root, leaf, parent, child, you are ready to read on!

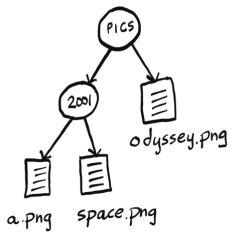
File Directories

Since a tree is a type of a graph, we can run a graph algorithm on it. In chapter 6, we had met breadth-first search, an algorithm for finding the shortest path in a graph. We are going to use breadth-first search on a tree. If you are not comfortable with breadth-first search, check out chapter 6.

A file directory is a tree that all of us interact with every day. Suppose I have this file directory:



I want to print the name of every file in the pics directory, including all its subdirectories. Here there is only one subdirectory, 2001. We can use breadth-first search to do this! First, let me show you what this file directory looks like as a tree:



Since this file directory is a tree, we can run a graph algorithm on it. We had earlier used breadth-first search as a search algorithm. But search isn't the only thing it's good for. Breadth-first search is a traversal algorithm. That means it is an algorithm that visits every node in a tree, aka traverses or walks the tree. That's exactly what we need! We need an algorithm that will go to every file in this tree and print out its name. We will use breadth-first search to list all the files in a directory. The algorithm will also go into subdirectories, find files in there, and print out their names. My logic will be:

1. Visit every node in the tree.
2. If this node is a file, print out its name.
3. If the node is a folder, add it to a queue of folders to search for files.Perhaps for readers new to this terminology, it might be good to stay with either folder or directory/sub-directory consistently.Perhaps for readers new to this terminology, it might be good to stay with either folder or directory/sub-directory consistently.

The code is below. It is very similar to the mango seller code from chapter 6:

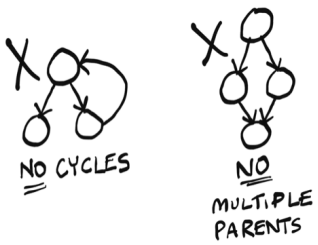
```
from os import listdir
from os.path import isfile, join
from collections import deque

def printnames(start_dir):
    # we use a queue to keep track of
    # folders to search
    search_queue = deque()
    search_queue.append(start_dir)
    # while the queue is not empty,
    # pop off a folder to look through
    while search_queue:
        dir = search_queue.popleft()
        # loop through every file and folder in this folder
        for file in sorted(listdir(dir)):
            fullpath = join(dir, file)
            if isfile(fullpath):
                # if it is a file, print out the name
                print(file)
            else:
                # if it is a folder,
                # add it to the queue of folders to search
                search_queue.append(fullpath)

printnames("pics")
```

Here we use a queue like we did in the mango seller example. In the queue we keep track of what folders we still need to search. Of course, in that example we stopped once we found a mango seller, but here we go through the whole tree.

There's one other important difference from the mango seller code. Can you spot it?

Remember how in the mango seller example, we had to keep track of whether we had already searched a person:

```
...
        # Only search this person if you haven't already searched
them.
        if person not in searched:
            if person_is_seller(person):
...
```



We don't have to do that here! Trees don't have cycles, and each node only has one parent. There’s no way we would accidentally search the same folder more than once, or end up in an infinite loop, so there's no need to keep track of which folders we have already searched. There simply isn't a way to revisit a folder.

This property of trees has made our code simpler. That's an important takeaway from this chapter: trees don't have cycles.

We don't have to do that here! Trees don't have cycles, and each node only has one parent. There’s no way we would accidentally search the same folder more than once, or end up in an infinite loop, so there's no need to keep track of which folders we have already searched. There simply isn't a way to revisit a folder.

This property of trees has made our code simpler. That's an important takeaway from this chapter: trees don't have cycles.

A note on symbolic links

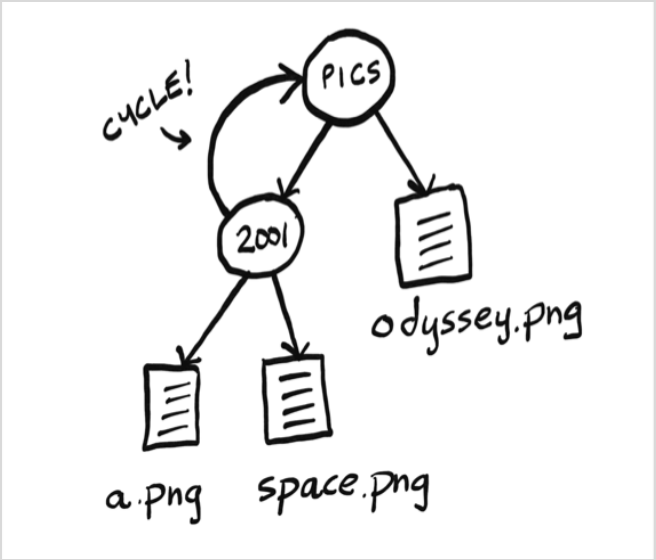
You may know what symbolic links are. If you don't, symbolic links are a way to introduce a cycle in a file directory. Here's how I could make a symbolic link on macOS or Linux:

```
ln -s pics/ pics/2001/pics
```

Or on Windows:

```
mklink /d pics/ pics/2001/pics
```

If I did that, the tree would look like this:



Now our file directory isn't a tree anymore! To keep things simple for this example, we are going to ignore symbolic links. If we did have a symbolic link, Python is smart enough to avoid an infinite loop. Here is the error it throws:

```
OSError: [Errno 62] Too many levels of symbolic links: 'pics/2001/pics'
```

A Space Odyssey: depth-first search

Let's traverse our file directory again, doing it recursively this time:

```
from os import listdir
from os.path import isfile, join

def printnames(dir):
    # loop through every file and folder in the current folder
    for file in sorted(listdir(dir)):
        fullpath = join(dir, file)
        if isfile(fullpath):
            # if it is a file, print out the name
            print(file)
        else:
            # if it is a folder, call this function recursively on
it
            # to look for files and folders
            printnames(fullpath)

printnames("pics")
```

Notice that now we are not using a queue. Instead when we come across a folder, we immediately look inside for more files and folders.

Now we have two ways of listing the file names. But here's the surprising part: *both solutions will print the file names in different orders!*

One prints the names out like this:

```
a.png
space.png
odyssey.png
```

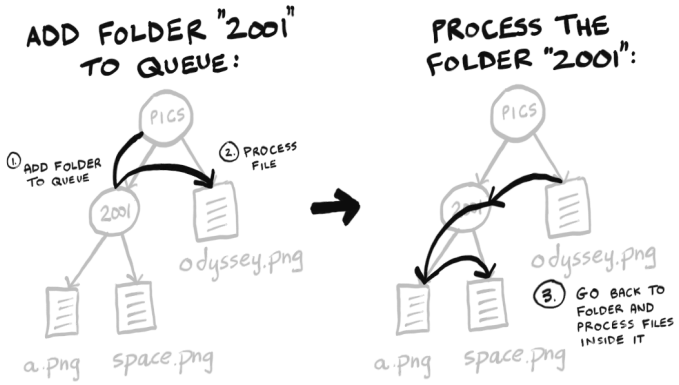
The other prints this:

```
odyssey.png
a.png
space.png
```

Can you figure out which solution prints which order, and why? Try it yourself before moving on.

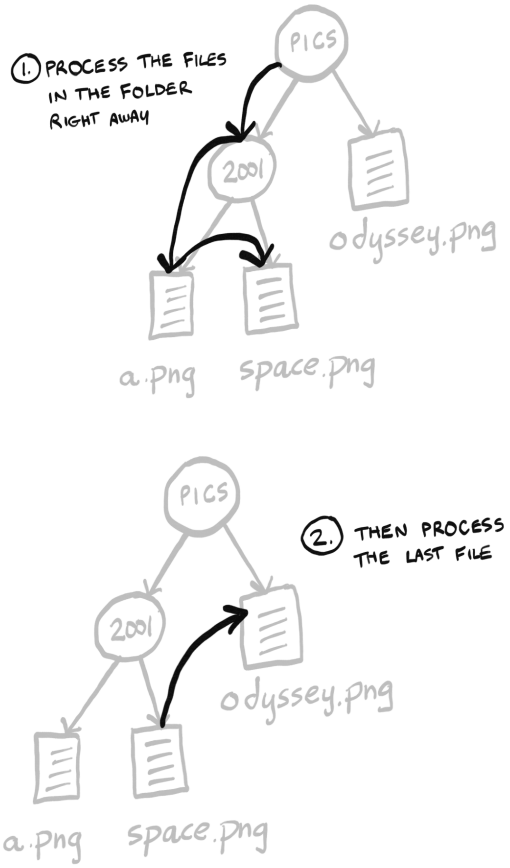
The first solution uses breadth-first search. When it finds a

folder, that folder is added to the queue, to be checked later. So the algorithm goes to the 2001/ folder, does not go into it but adds it to the queue to be looked at later, prints all the file names in the pics/ folder, then goes back to the 2001/ folder and prints the file names in there:



You can see the algorithm visits the 2001 folder first, but doesn't look inside. That folder is just added to the queue, and breadth-first search moves on to odyssey.png.

The second solution uses an algorithm called depth-first search. Depth-first search is also a graph and tree traversal algorithm. When it finds a folder, it looks inside immediately instead of adding it to a queue:



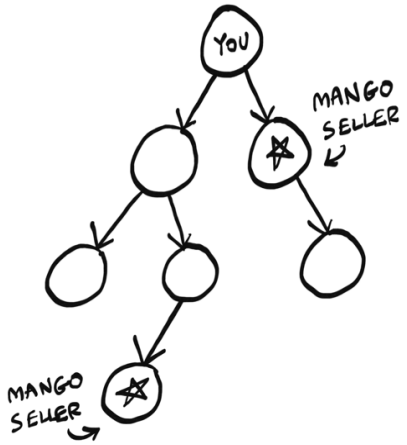
The second solution is the one that prints out:

```
a.png
space.png
odyssey.png
```

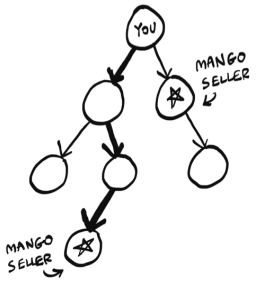
Breadth-first search and depth-first search are closely related, and often where one is mentioned, the other will be also. Both algorithms printed out all the file names, so they both work for this example. But there is a big difference. Depth-first search cannot be used for finding the shortest path!

In the mango seller example, we could not have used depth-first search. We rely on the fact that we are checking all our first-degree friends before the second-degree

friends, and so on. That's what breadth-first search does. But depth-first search will go as deep as possible right away. It may find you a mango seller three degrees away, when you have a closer contact! Suppose this is your social network:

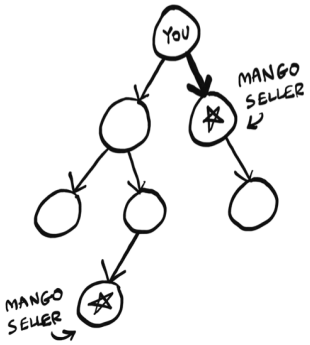


Let's say we process nodes in order from left to right. Depth-first search will get to the leftmost child node and go deep:



Because the depth-first search went deep on the left node, it failed to realize that the right node is a mango seller that is much closer.

Breadth-first search will correctly find the closest mango seller:



So while both algorithms worked for listing files, only breadth-first search works for finding the shortest path. Depth-first search has other uses. It can be used to find the topological sort, a concept we saw briefly in chapter 6.

A better definition of trees

Now that you have seen an example, it's time for a better definition of a tree. A tree is a *connected, acyclic graph*:

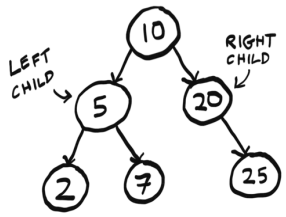
As I had said earlier, we are working exclusively with rooted trees, so our trees all have a root as well. And we are working exclusively with connected graphs. So the most important thing to remember is: *trees cannot have cycles*.

Now we have seen a tree in action, let’s zoom in on one specific type of tree.

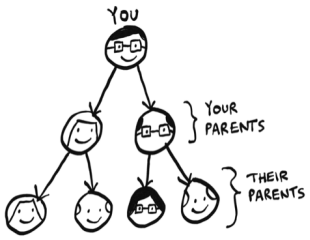
Binary trees

Computer science is full of different types of trees. Binary trees are a very common type of tree. For the rest of this chapter, and most of the next, we will work with binary trees.

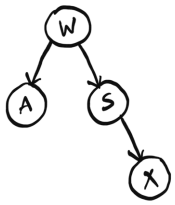
A binary tree is a special type of tree where nodes can have at most two children (hence the name “binary”, meaning “two”). These are traditionally called left child and right child:



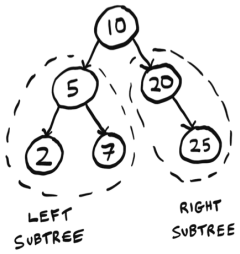
An ancestry tree is an example of a binary tree, since everyone has two biological parents:



In that example there's a clear connection between nodes — they are all family. But the data can be totally arbitrary:



The important thing is you never have more than two children. Sometimes you will also see people referring to the left subtree or right subtree:



Binary trees show up everywhere in computer science. We are going to spend the rest of this chapter looking at an example that uses a binary tree.

Huffman coding

Huffman Coding is a neat example of using binary trees. It's also the foundation for text compression algorithms. We won't describe the algorithm, but will spend time focusing on how it works, and how it makes clever use of trees.

First, a little background. To know how compression works, we need to know how much space a text file takes. Suppose we have a text file with just one word: "tilt". How much space does that use? You can use the `stat` command (available on Unix). First, save the word in a file called `test.txt`. Then, using `stat`:

```
$ cat test.txt
tilt
$ stat -f%z test.txt
4
```

So that file takes up four bytes: one per character.

This makes sense. Assuming we are using ISO-8859-1 (see sidebar for what this means), each letter takes up exactly one byte. For example, the letter a is ISO-8859-1 code 97, which I can write in binary as: 01100001. That is eight bits. A bit is a digit that can be either zero or one. And there are eight of them. Eight bits is one byte. So the letter a is represented using one byte. ISO-8859-1 code goes from 00000000, which represents the null character, all the way to 11111111, which represents ÿ ("Latin small letter y with diaeresis"). There are 256 possible combinations of zeros and ones with eight bits, so the ISO-8859-1 code allows for 256 possible letters.

Character encoding

As this example will show you, there are many different ways to encode characters. That is, the letter a could be written in binary in many different ways.

It started with ASCII. In the 1960s, ASCII was created. ASCII is a 7-bit encoding. Unfortunately, ASCII did not include a lot of characters. ASCII does not include any characters with umlauts (ü or ö for example), or common currencies like the British pound or Japanese yen.

So ISO-8859-1 was created. ISO-8859-1 is an 8-bit encoding, so it doubles the number of characters that ASCII provided. We went from 128 characters to 256 characters. But this was still not enough, and countries began making their own encodings. For example, Japan has several encodings for Japanese, since ISO-8859-1 and ASCII were focused on European languages. The whole situation was a mess until Unicode was introduced.

Unicode is an encoding standard. It aims to provide characters for any language. Unicode has 149,186 characters as of 2022 — quite a jump from 256! More than 1000 of these are emojis.

Unicode is the standard, but you need to use an encoding that follows the standard. The most popular encoding today is UTF-8. UTF-8 is variable length character encoding, which means characters can be anywhere from one to four bytes (8 to 32 bits).

You don't need to worry too much about this. I've kept the example simple intentionally by using ISO-8859-1, which is 8 bits, which is a nice consistent quantity of bits to work with.

Just remember these takeaways:

- Compression algorithms try to reduce the number of bits needed to store each character.
- If you need to pick an encoding for a project, UTF-8 is a good default choice.

Let's decode some binary to ISO-8859-1 together: 011100100110000101100100. You can Google an ISO-8859-1 table or binary to ISO-8859-1 converter to make this easier.

First, we know that each letter is eight bits, so I am going to divide this into chunks of eight, to make it easier to read:

```
01110010 01100001 01100100
```

Great, now we see that there are three letters. Looking them up in an ISO-8859-1 table, I see they spell out `rad`: `01110010` is `r`, and so on. This is how your text editor takes the binary data in a text file and displays it as ISO-8859-1. You can view the binary information by using `xxd`. This utility is available on Unix. Here is how `tilt` looks in binary:

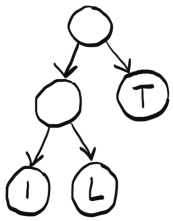
```
$ xxd -b test.txt
00000000: 01110100 01101001 01101100 01110100
tilt
```

Here is where the compression comes in. For the word “tilt”, we don't need 256 possible letters; we just need three. So we don't need eight bits, we only need two. We could come up with our own two bit code, just for these three letters:

```
t = 00
i = 01
l = 10
```

Here is how we could write tilt using our new code: 00011000. I can make this easier to read by adding spaces again: 00 01 10 00. If you compare it to the mapping above, you'll see this spells out tilt.

This is what Huffman Coding does — it looks at the characters being used and tries to use less than eight bits. That is how it compresses the data. Huffman Coding generates a tree:

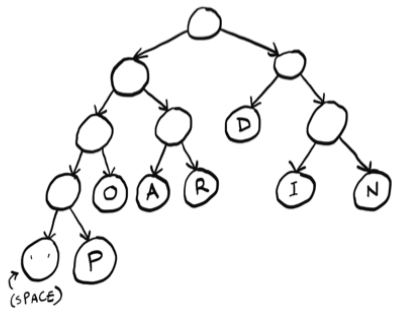


You can use this tree to find the code for each letter. Starting at the root node, find a path down to the letter L. Whenever you choose a left branch, append a zero to your code. When you choose a right branch, append one. When you get to a letter, stop progressing down the tree. So the code for the letter L is 01. Here are the three codes given by the tree:

```
i = 00
l = 01
t = 1
```

Notice that the letter T has a code of just one digit. Unlike ISO-8859-1, **in Huffman Coding, the codes don't all have to be the same length**. This is important — let's see another example to understand why.

We want to now compress the phrase “paranoid android”. Here is the tree generated by the Huffman Coding algorithm:



Check yourself: what is the code for the letter P? Try it yourself before reading on. It is 0001. What about the letter

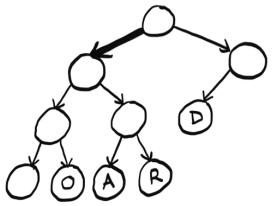
D? It is 10.

In this case, there are actually three different possible lengths! Suppose we try to decode some binary data: 01101010. We see the problem right away: we can't chunk this up the way we did with ISO-8859-1! While all ISO-8859-1 codes were eight digits, here the code could be 2, 3, or 4 digits. **Since the code length varies, we can't use chunking anymore.**

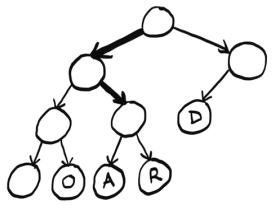
Instead, we need to look one digit at a time, as if we are looking at a tape.

Here's how to do it: first number is zero, so go left (I'm only showing part of the tree here):

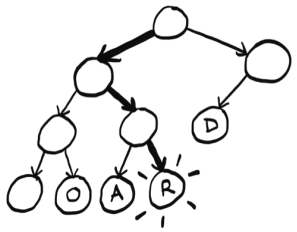
In this case, there are actually three different possible lengths! Suppose we try to decode some binary data: 01101010. We see the problem right away: we can't chunk this up the way we did with ISO-8859-1! While all ISO-8859-1 codes were eight digits, here the code could be 2, 3, or 4 digits. **Since the code length varies, we can't use chunking anymore.**



Then we get a one, so go right:



Then another one, so right again:



Aha! We found a letter. This is the binary data we have left: 01010. We can start over at the root node and find the other letters. Try decoding the rest yourself and then read on. Did you get the word? It was rad. This is a big difference between Huffman Coding and ISO-8859-1. The codes can vary, so the decoding needs to be done differently.

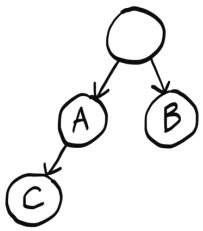
It is more work to do it this way instead of chunking. But there is one big benefit. Notice that the letters that show up more often, have shorter codes. D appears three times so its code is just two digits, versus I which appears twice, and P which appears only once. Instead of assigning four bits to everything, we can compress frequently used letters even more. You can see how, in a longer piece of text, this would be a big savings!

Now that we understand at a high level how Huffman Coding works, let's see what properties of trees Huffman is taking advantage of here.

First of all, could there be overlap between codes? Take this code for example:

```
a = 0
b = 1
c = 00
```

Now if you see the binary 001, is that aab or cb? c and a share part of their code, so it's unclear. Here is what the tree for this code would look like:

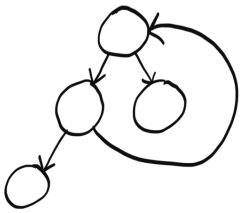


We pass A on the way to C, which causes the problem.

That’s not a problem with Huffman Coding, because letters only show up at leaf nodes. And there’s a unique path from the root to each leaf node — that's one of the properties of trees. So we can guarantee overlap is not a problem.

This also guarantees there is only one code for each letter. Having multiple paths to each letter would mean there are multiple codes assigned to each letter, which would be unnecessary.

When we read the code one digit at our time, we are assuming we will eventually end up at a letter. If this was a graph with a cycle, we couldn’t make that assumption. We could get stuck in the cycle and end up in an infinite loop:



But since this is a tree, we know there are no cycles, and so we are guaranteed to end up at some letter.

We are using a rooted tree. Rooted trees have a root node, which is important because we need to know where to start! Graphs do not necessarily have a root node.

Finally, the type of tree used here is called a binary tree. Binary trees can have at most two children — the left child and the right child. This makes sense because binary only has two digits. If there was a third child, it would be unclear what digit it is supposed to represent.

This chapter introduced you to trees. In the next chapter, we will see some different types of trees and what they are used for.

Recap

- Trees are a type of graph, but trees don't have cycles.
- Depth-first search is another graph traversal algorithm. It can't be used to find shortest paths.
- A binary tree is a special type of tree where nodes can have at most two children.
- There are many different types of character encodings. Unicode is the international standard, and UTF-8 is the most common Unicode encoding.