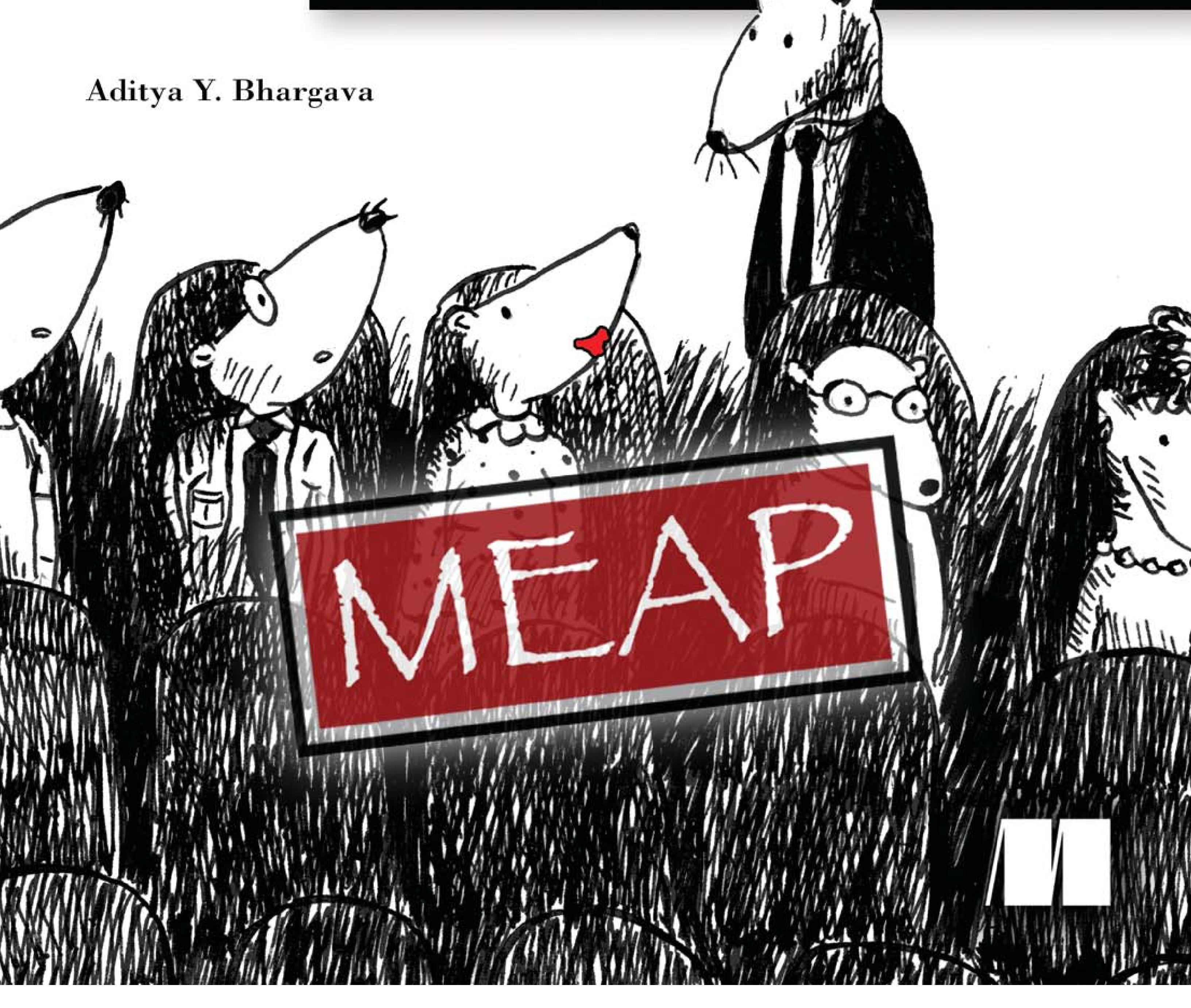
grokking algorithms

An *illustrated* guide for programmers and other curious people

Aditya Y. Bhargava



MEAP Edition
Manning Early Access Program
Grokking Algorithms
Version 1

Welcome

Thank you for purchasing the MEAP for *Grokking Algorithms*. I'm excited to see the book reach this stage and look forward to its continued development and eventual release. This is a beginner-level book that covers a set of basic, core algorithms. It is designed to be the first book you would read if you wanted to learn algorithms.

I'm releasing the first three chapters to start. Chapter 1 covers what an algorithm is with a simple search algorithm. Then it explains how we talk about algorithm performance. Performance consideration is a common thread you will see throughout this book. Chapter 2 explains two basic data structures—arrays and lists—before moving on a sorting algorithm.

I've striven to make the material easy to follow. Every chapter introduces a new concept and then immediately shows how to use it.

Looking ahead, I'll be covering higher-level data structures that are built on arrays and lists. I'll also talk about graph algorithms and dynamic programming. These are two general techniques for solving programming problems. By the end of this book, you will have expanded the types of problems you can solve with code.

I expect to have updates at least once a month, in the form of a new chapter or an update to an existing chapter.

I have read technical books that are boring, and I have read technical books that take big leaps of logic. This book aims to avoid both. As you're reading, I hope you will let me know about these issues in the Author Online forum.

— Aditya Y. Bhargava

brief contents

1 Introduction To Algorithms

This chapter covers:

- Your first search algorithm (binary search)
- How to talk about the running time of an algorithm (big-O notation)

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

- Binary search (chapter 1) shows how an algorithm can speed up your code. In one example, the number of steps needed goes from 4 billion to 32!
- A GPS device uses graph algorithms (chapters 4 and 5) to calculate the shortest route to your destination for you.
- You can use dynamic programming (chapter 7) to write an AI that plays checkers.

In each case, I'll talk about 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 talk about what other types of problems could be solved by the same algorithm.

What you will learn on performance

The good news is, there is probably an implementation of every algorithm in this book 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 tradeoffs. In this book, you will learn to compare tradeoffs between different algorithms: Should I use mergesort or quicksort? Should I use an array or a list? Just using a different data structure can make a big difference.

What you will learn on solving problems

You will 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 that follows the user around using graph algorithms.
- You'll learn how data compression works using huffman coding.
- Some problems aren't solvable! The part on 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 will know some of the most widely applicable algorithms. You can then use your new knowledge to learn about more specific algorithms for AI, databases etc. Or you can take on bigger challenges at work.

What you will learn in this chapter

This chapter gives you a foundation for the rest of the book. You will learn:

- A simple algorithm (binary search)
- How to talk about the running time of an algorithm (big-O notation)
- A common technique for designing algorithms (recursion).

What you need to know

You will need to know basic algebra before starting this book. In particular, take this function: `f(x) = x * 2`. What is `f(5)`? If you answered `10`, you are set.

Additionally, this chapter (and this book) will go easier if you are 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, pick Python. In my experience it is the easiest one to pick up. If you know another language like Ruby, you'll be just fine.

1.1 Binary Search

Figure 1.1

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

Or suppose you are searching for a word in a dictionary, and it starts with "O". Again, you're going to start near the middle.

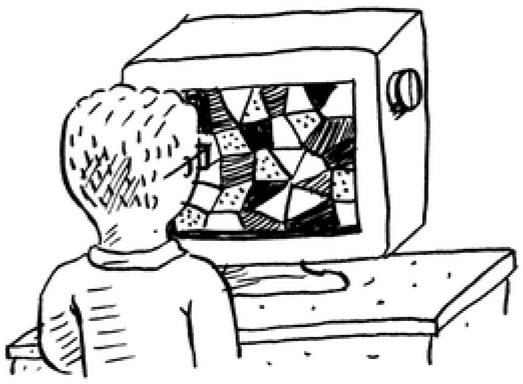


Figure 1.2

Now suppose you log on to facebook. When you log on, facebook has to verify that you have an account on their site. So they need to search for your username in their database. Suppose your username is "karlmageddon". Facebook could start from the As and start searching for your name…but it makes more sense for them to start somewhere in the middle.

This is a search problem. And in all these cases, we're using the same algorithm to solve the problem: binary search.

Binary search is an algorithm where the input is a list of elements, and an element that you're looking for in the list. If the element is in the list, binary search returns the position it is at. Otherwise it returns null.

For example:

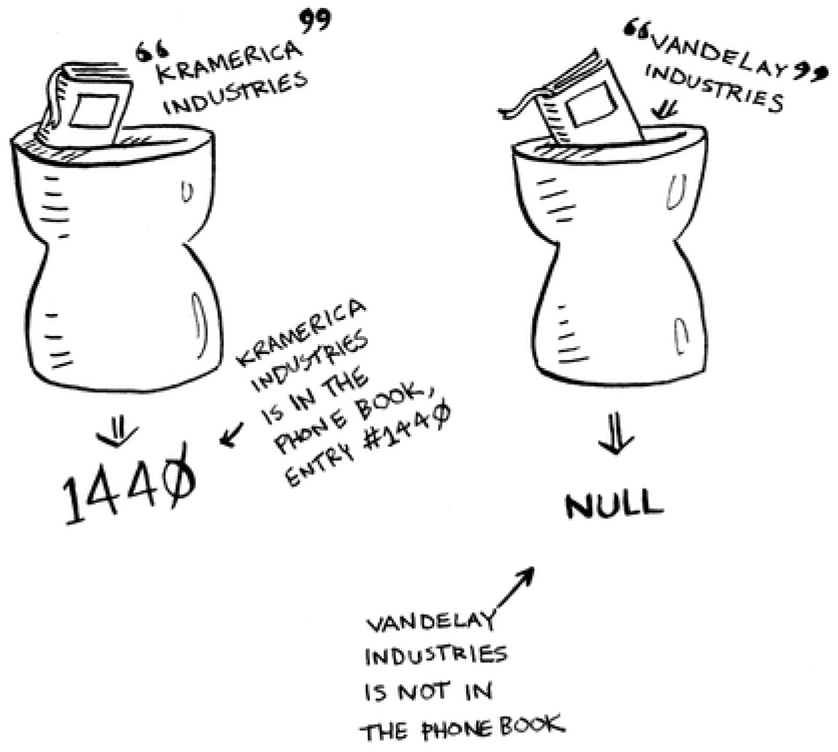

"KRAMERICA INDUSTRIES"
1440
KRAMERICA INDUSTRIES IS IN THE PHONE BOOK, ENTRY #1440
"VANDELAY INDUSTRIES"
NULL
VANDELAY INDUSTRIES IS NOT IN THE PHONE BOOK

Figure 1.3 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:

Figure 1.4

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

Now suppose you start guessing like this: `1, 2, 3, 4 ....` Here's how it would go:

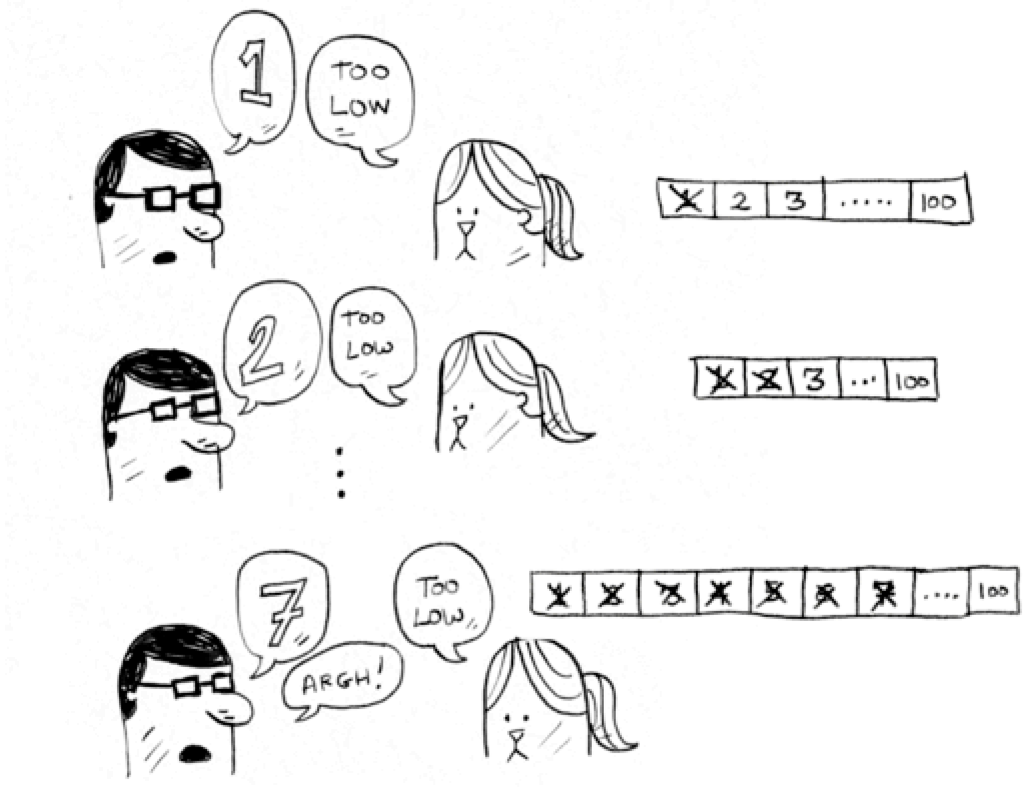


Figure 1.5 A bad approach to number guessing

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

1.1.1 A better way to search

Here's a better technique. Start with 50:

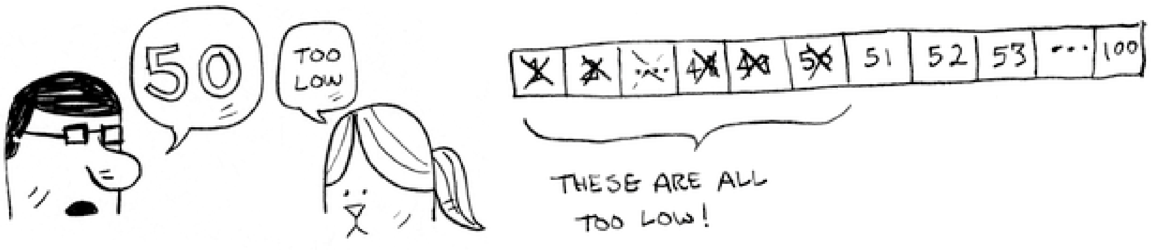


Figure 1.6

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

Figure 1.7

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

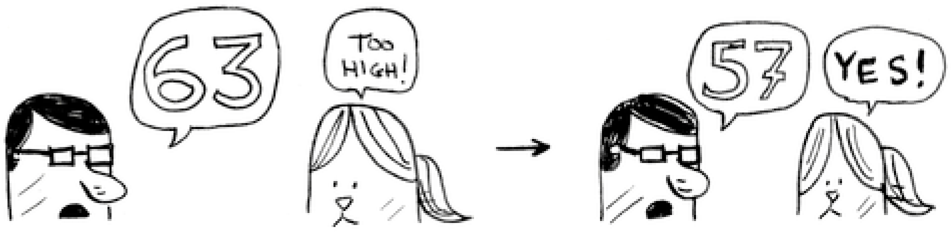


Figure 1.8

This is binary search. You just learned your first algorithm! Here’s now many numbers we can eliminate every time:

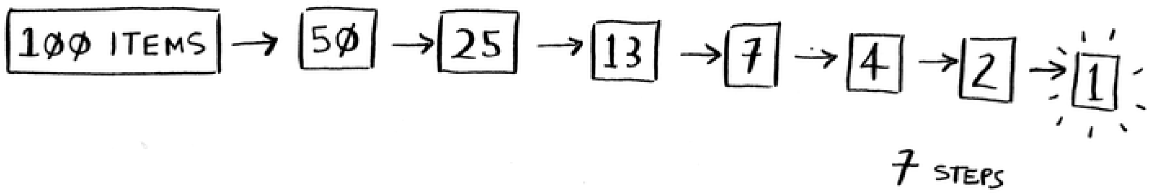


Figure 1.9 Eliminate half the numbers every time with binary search

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

EXERCISE

Here’s another example: suppose I’m 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?

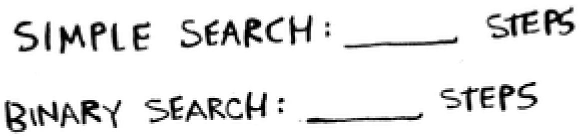


Figure 1.10

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

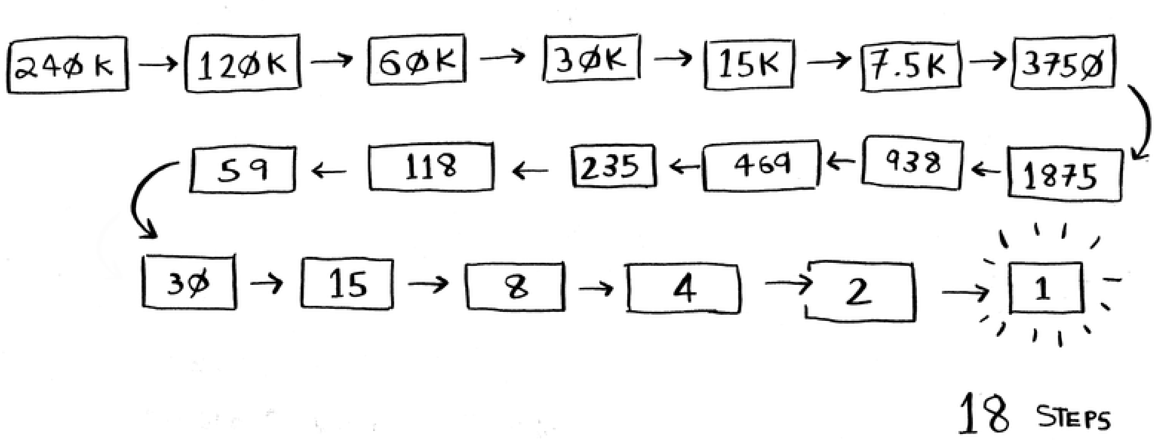


Figure 1.11

So binary search will take 18 steps…big difference! In general, for any list of size `n`, binary search will take `log2 n` steps to run in the worst case, while simple search will take `n` steps.

A SIDEBAR ON 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"? and the answer is 2: `10 x 10`. So $\log_{10} 100 = 2$. Logs are the flip of exponentials:

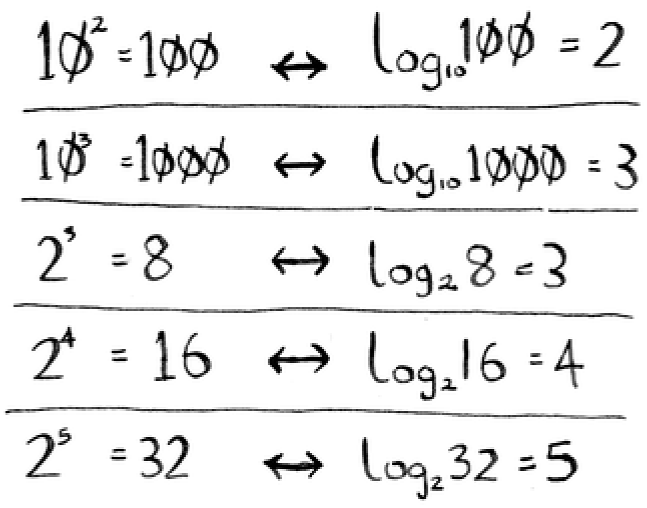

$10^2 = 100 \leftrightarrow \log_{10} 100 = 2$
$10^3 = 1000 \leftrightarrow \log_{10} 1000 = 3$
$2^3 = 8 \leftrightarrow \log_2 8 = 3$
$2^4 = 16 \leftrightarrow \log_2 16 = 4$
$2^5 = 32 \leftrightarrow \log_2 32 = 5$

Figure 1.12 Logs are the flip of exponentials

NOTE When talking about running time in `big-O`, we use `lg` to mean $\log_2$. The running time for binary search is O(lg n). So, for a list of 8 elements, `lg 8 == 3`, because `2^3^ == 8`. For a list of 256 elements, `lg 256 = 8`, because `2^8^ == 256`.

1.1.2 Code listing

Finally, lets see how to write this function in Python. The `binary_search` function will take a list and an item. If the item is in the list, the function returns its position. We keep track of what part of the list we have to search through. At the start, this is the whole list:

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

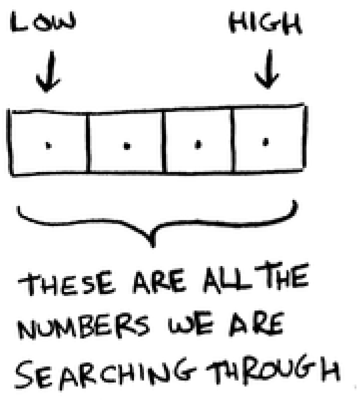


Figure 1.13

Each time, we check the middle element:

```
mid = (low + high) / 2
guess = list[mid]
```

❶ **mid is rounded down if (low + high) is not an even number**

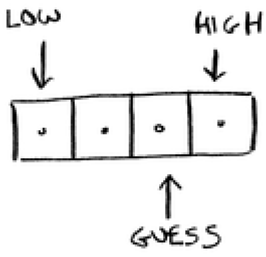


Figure 1.14

If the guess is too high, we update `high` accordingly:

```
if guess >= item:
  high = mid - 1
```

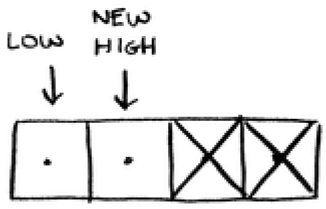


Figure 1.15

And if the guess is too low, we update low. Here's the full code:

```
def binary_search(list, item):
  low = 0 ❶
  high = len(list) - 1

  while low <= high: ❷
    mid = (low + high) / 2 ❸
    guess = list[mid]
    if guess == item: ❹
      return mid
    if guess >= item: ❺
      high = mid - 1
    else: ❻
      low = mid + 1

  return None ❼

my_list = [1, 3, 5, 7, 9] ❽

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

❶ low and high keep track of what part of the list we'll search in
❷ while we haven't narrowed it down to one element
❸ check the middle element
❹ found the item
❺ our guess was too high
❻ our guess was too low
❼ item doesn't exist
❽ Lets test it out!

1.1.3 Running time

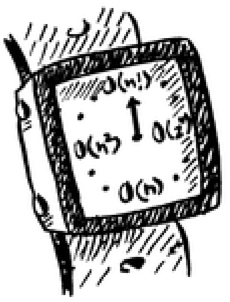


Figure 1.16

Any time we talk about an algorithm, I'll follow it up with a discussion on 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 we save by using it? Well, our first approach was to check each number one by one. If this is a list of 100 numbers, it takes us 100 guesses. If it's a list of 4 billion numbers, it takes us 4 billion guesses. So our number of guesses is the same as the size of the list. This is called *linear time*.

Binary search is different. If our list is 100 items long, it takes 7 guesses. If out list is 4 billion items, it takes 32 guesses. Powerful, eh? Binary search runs in *logarithmic time* (or *log time* as the natives call it). Here's a table summarizing our findings today:

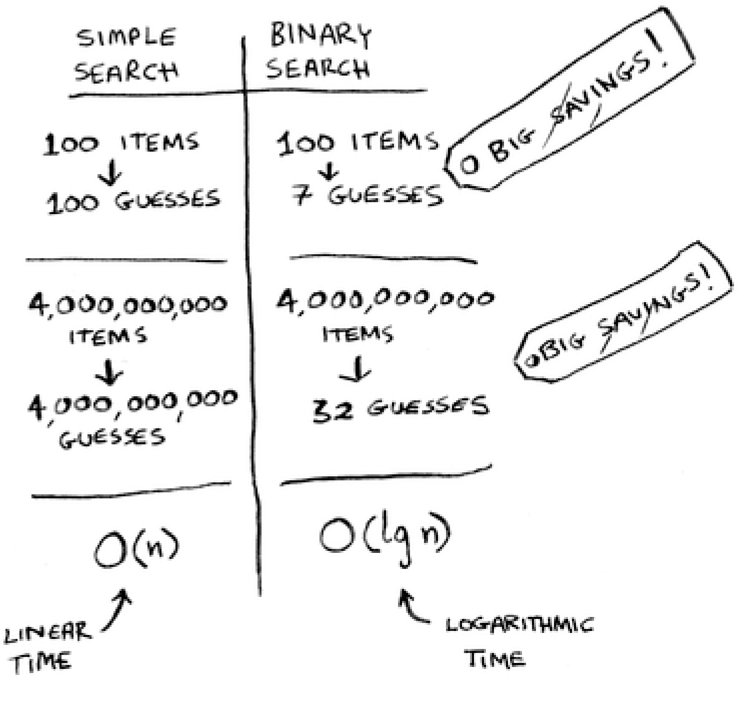

SIMPLE SEARCH
BINARY SEARCH
100 ITEMS
100 GUESSES
100 ITEMS
7 GUESSES
BIG SAVINGS!
4,000,000,000 ITEMS
4,000,000,000 GUESSES
4,000,000,000 ITEMS
32 GUESSES
BIG SAVINGS!
O(n)
O(lg n)
LINEAR TIME
LOGARITHMIC TIME

Figure 1.17 Runtimes for search algorithms

I know we haven’t talked about linear time and log time yet…it’s coming up in the next section!

1.2 Big-O-notation

Big-O notation is special notation that tells you how fast an algorithm is. But who cares? Well, it turns out that you will use other people’s algorithms very often…and at that time, 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 big-O notation.

1.2.1 Algorithm speed is not measured in seconds

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

Figure 1.18

Bob is trying to decide between simple search and binary search. The algorithm needs to be both fast and correct. On the one hand, binary search is faster. And Bob’s only has *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…less chance of bugs being introduced! And you *really* don’t want bugs in your code to land a rocket. To be extra careful, Bob decides to time both algorithms with a list of 100 elements:

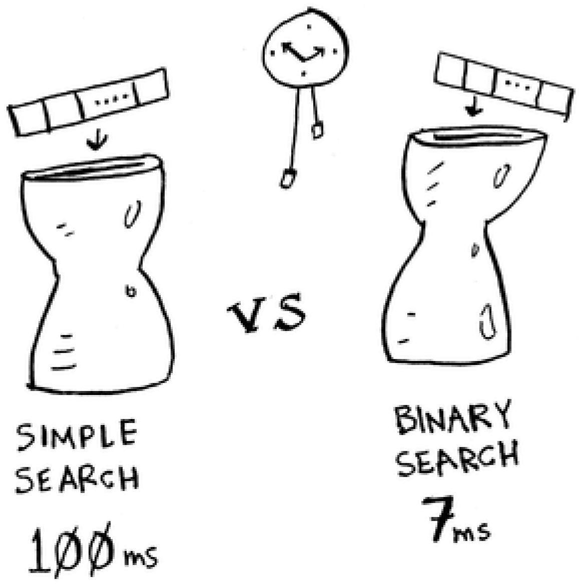
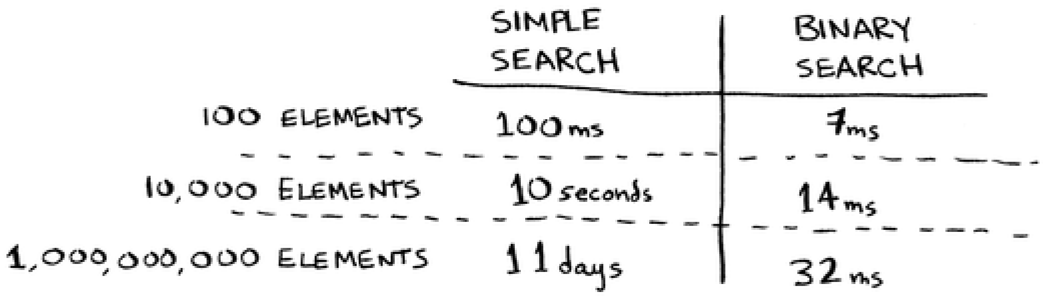


Figure 1.19 Running time for simple search vs binary search, on a list of 100 elements

Looks like simple search took 100ms to run, and binary search took 7ms to run. So binary search is about 15 times faster. But realistically, the list will have more like a billion elements in it. So Bob runs binary search with a billion elements, and it takes 32ms. "32ms! And since binary search is about 15 times faster, simple search will take 32x15 = 480ms", reasons Bob. "Way under my threshold of 10 seconds". Bob decides to go with simple search.

Turns out, Bob is wrong. Dead wrong. 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	100ms	7ms
10,000 ELEMENTS	10 seconds	14ms
1,000,000,000 ELEMENTS	11 days	32ms

Figure 1.20 Running times grow at very different speeds!

Simple search will take 11 days to run…way over Bob’s threshold of 10 seconds!

That’s why it’s not enough to know how long an algorithm takes to run…you need to know how the runtime grows. That’s where big-O notation comes in.

1.2.2 Big-O notation

Figure 1.21

For a list of size `n`, simple search will run in `n` operations. Here's how you write that in big-O notation: `O(n)`. That's it. There's no mention of seconds, or minutes, or any other time measurement. It just tells you that for a list of size `n`, simple search will take `n` operations. And actually, big-O is an upper bound, so it tells you that the algorithm will perform *at most* n operations.

Here's another example. There's an algorithm called `insertion sort` that takes O(n2) time. That means, for a list of size `n`, insertion sort will perform `n^2^` operations. See the pattern? For every algorithm, it's running time is written as:

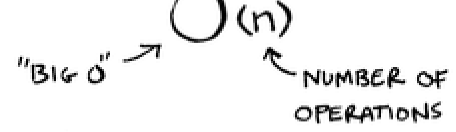


Figure 1.22 What Big-O notation looks like

Finally, the runtime for binary search: O(lg n). This means that for a list of size `n`, binary search takes `lg n` operations.

Now lets look at some examples of big-O notation.

1.2.3 The five most common big-O runtimes

Here’s a practical example you can follow along at home, with a few pieces of paper and a pencil. Suppose you have to draw a grid of 16 boxes on a piece of paper:

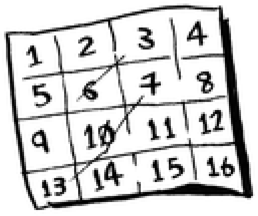


Figure 1.23

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:

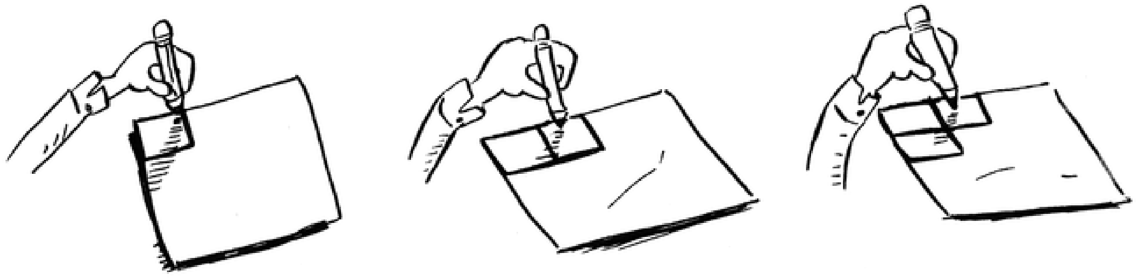


Figure 1.24 Drawing a grid one box at a time

It takes 16 steps to draw 16 boxes, so this is O(n).

ALGORITHM #2

There’s a faster way to do it: fold the paper up:

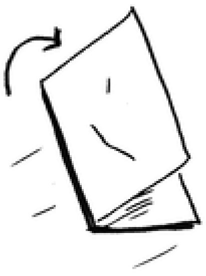


Figure 1.25

Then fold it again, and again, and again:

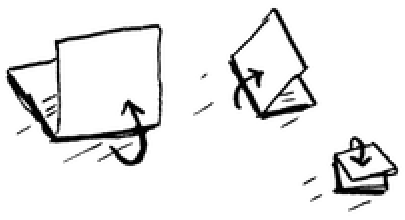


Figure 1.26

Unfold it after 4 folds, and you will have a beautiful grid! Every fold doubles the number of boxes you have:

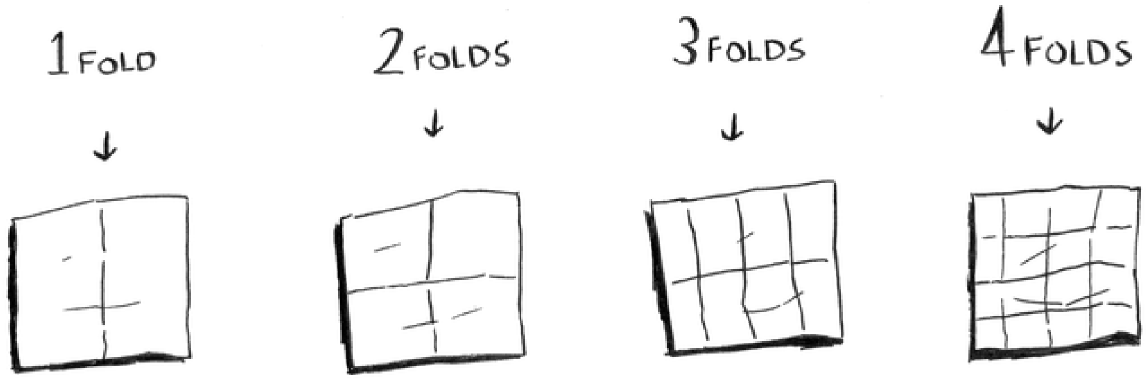


Figure 1.27 Drawing a grid in four folds

We can "draw" twice as many boxes with every fold…so this algorithm takes O(lg n) time.

ALGORITHM #3

Now suppose you decide that each box in the grid will be it's own piece of paper, with it's own 16 square grid on it:

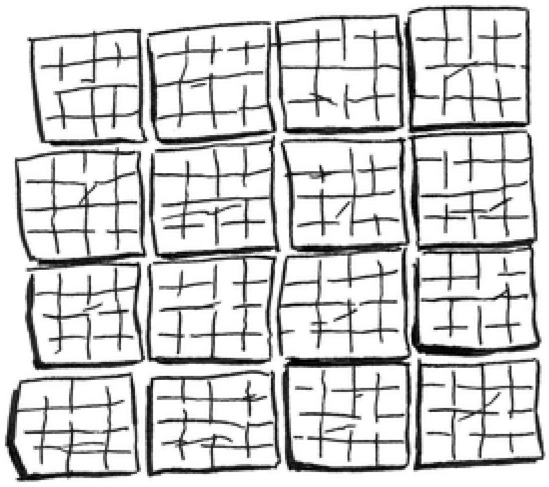


Figure 1.28

You can fold up the piece of paper to make the grid. Making the grid takes O(lg

n) time, and we have to do it for `n` papers, so this algorithm takes `n * lg n` or `O(n lg n)` time.

ALGORITHM #4

Again, suppose that each box on the grid is it's own piece of paper with it's own grid. This time you decide to draw the grid yourself, drawing one box at a time. Each grid takes `O(n)` time to draw, and you have to draw `n` grids, So this will take `O(n^2^)` time.

ALGORITHM #5

TODO need to verify that this example is correct, i.e. this is representing O(n!) time.

Here's a completely different algorithm. Give each box a number, from 1 - 16:

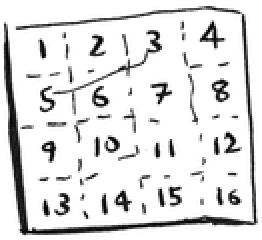


Figure 1.29

Now for each box, starting with box #1, run this algorithm:

1. Roll a 16-sided die. If the number that comes up is the box that you are on, draw that box in. So at the start, you keep rolling until you get a `1`. Then you draw that box in and move on to the next box.
2. If the number that comes up is not the box you are on, start over.

So, suppose you start by rolling a `1`. You can draw that box in. Next you roll a `12`. Well, you had to roll a `2`, and you didn't, so start over and try to roll a `1` again.

Figure 1.30

We'll count each die roll as a move…then this algorithm takes O(n!) time. Suppose you could roll a die 10 times a second. It would still take you over 60,000 years to finish drawing the grid this way!

Now lets imagine that instead, you have to draw a grid of 256 squares:

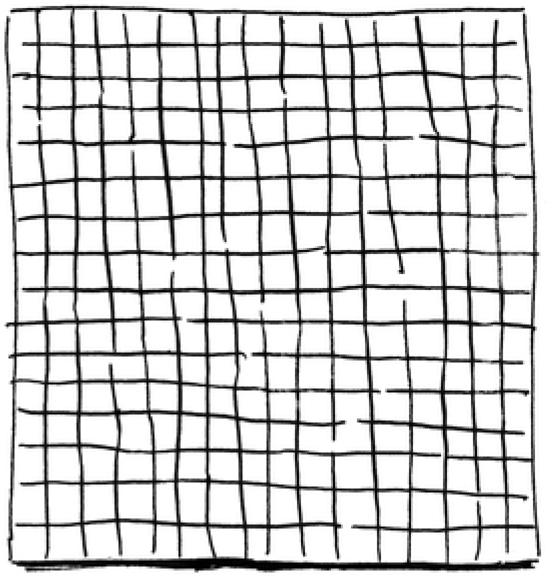
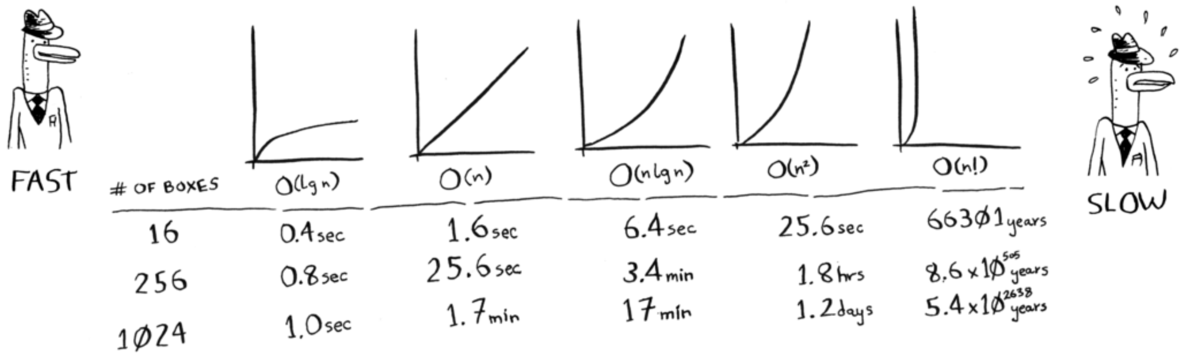


Figure 1.31

- You can do this with Algorithm #1: you'll have to draw 256 squares.
- It's easy with Algorithm #2: with four more folds, you will have 256 squares.
- It's hard but doable with Algorithm #3: you will have to fold up 256 individual pieces of paper.
- It's really hard with Algorithm #4: You'll have to draw 256 squares on 256 pieces of paper, for a total of 65536 squares! That will take a couple of days.
- It's impossible with Algorithm #5: the universe will collapse before you are able to finish drawing the grid.

I hope this gives you some idea that all these algorithms grow at very different speeds. `O(n!)` is not only worse than `O(n)`, it gets much much worse as you have bigger and bigger lists. Suppose we can do 10 operations a second. Here's how long it would take us to make a grid, based on the algorithm we use:



# OF BOXES	O(lg n)	O(n)	O(n lg n)	O(n²)	O(n!)
16	0.4 sec	1.6 sec	6.4 sec	25.6 sec	66301 years
256	0.8 sec	25.6 sec	3.4 min	1.8 hrs	8.6×10^{505} years
1024	1.0 sec	1.7 min	17 min	1.2 days	5.4×10^{2638} years

Figure 1.32

1.2.4 The Traveling Salesman

You might have read through 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 it's growth is appalling and some very smart people think it can't be improved. This problem is called the "traveling salesman problem". You have a salesman:

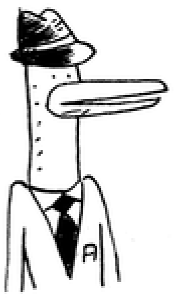


Figure 1.33

And the salesman has to go to five cities:

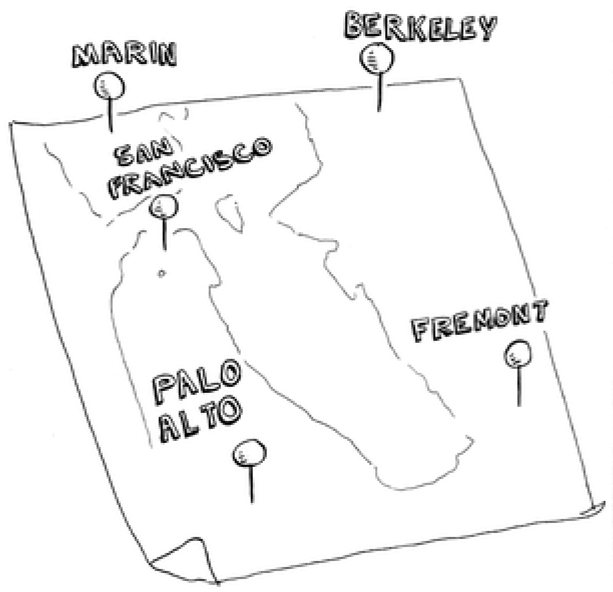


Figure 1.34

This salesman, who 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 he could travel the cities in:

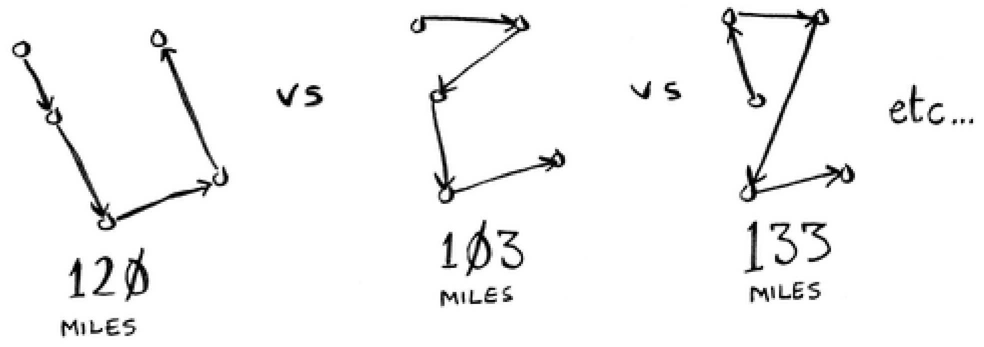
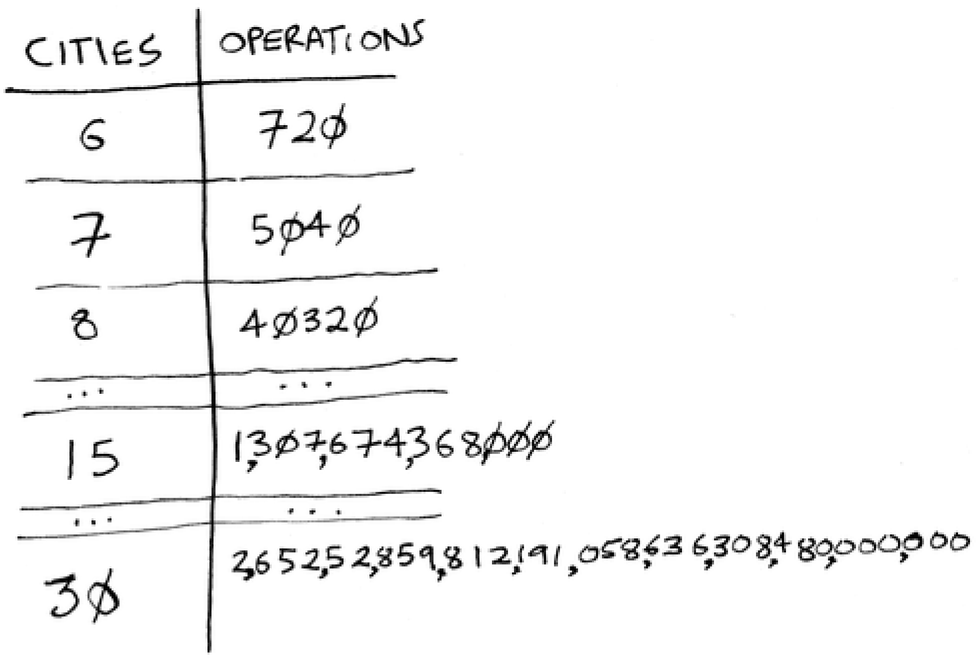


Figure 1.35 As a salesman, there are lots of ways to hit five cities

And adds up the total distance. Then he 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 5040 operations!



CITIES	OPERATIONS
6	720
7	5040
8	40320
...	...
15	1,307,674,368000
...	...
30	265,252,859,812,191,058,636,308,480,000,000

Figure 1.36 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 are dealing with 100+ cities, it's impossible to calculate the answer in time…the sun will collapse before you get the answer.

So this is a terrible algorithm! We should use a different one, right? But we can't. The traveling salesman is one of the unsolved problems in computer science.

There is no fast algorithm for it, and smart people think that it's *impossible* to have a smart algorithm for this problem. The best we can do is come up with an approximate a solution – see the chapter on NP-complete problems for more.

1.2.5 A caveat and recap

Suppose you have an algorithm that processes a list of size `n` in `2 * n` operations. What's the runtime for this algorithm in big-O notation? It's not O(2n) … it's O(n). This is a tricky point that trips a lot of people up. When you read a running time like O(n), you can assume that the real running time is `c1*n + c2`, where c1 and c2 are constants that will stay the same, irrespective of the size of the list.

For example, suppose Alice, Bob and Claire all write their own simple search. Alice's simple search takes `2*n + 10` operations, Bob's takes `3*n + 12` operations, and Claire's takes `10*n + 0` operations. Even though Alice, Bob and Claire all have different values for `c1` and `c2`, those numbers stay constant: they don't change when the size of the list changes. So we don't count them in the runtime for big-O. So all of these runtimes are still `O(n)`.

TO RECAP:

- algorithm times are measured in terms of *growth* of an algorithm
- algorithm times are written in big-O notation
- big-O notation establishes an upper bound. If an algorithm's runtime is O(n), it won't take longer than `c1*n + c2` operations.