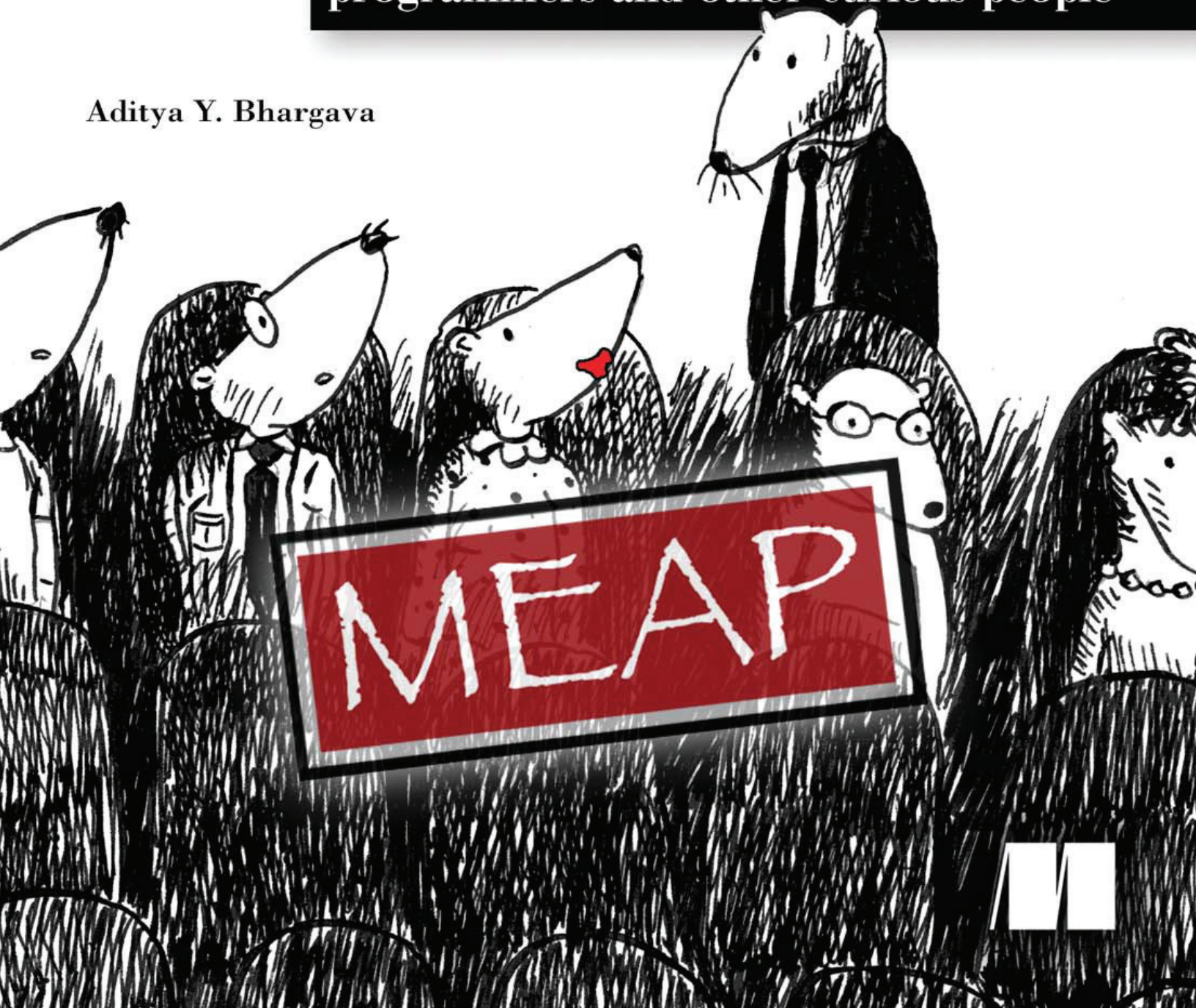


grokking

algorithms

*An illustrated guide for
programmers and other curious people*

Aditya Y. Bhargava





**MEAP Edition
Manning Early Access Program
Grokking Algorithms
Version 12**

Copyright 2015 Manning Publications

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

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. Chapter 3 explains recursion, a very common technique for writing algorithms.

Together, chapters 1 - 3 create a foundation that I'll build on in the rest of the book. 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*
- 2 Selection Sort*
- 3 Recursion*
- 4 Quicksort*
- 5 Hash Tables*
- 6 Breadth-first Search*
- 7 Dijkstras Algorithm*
- 8 Greedy Algorithms*
- 9 Dynamic Programming*
- 10 K-Nearest Neighbors*
- 11 Where to go next*

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).
- You're introduced to a common technique for designing algorithms (recursion).

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 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 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 10) 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

Selection 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 particular, take this function: $f(x) = x \times 2$. 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 is great for beginners. If you know another language, like Ruby, you'll be fine.

1.1 Binary Search



Figure 1.1

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.

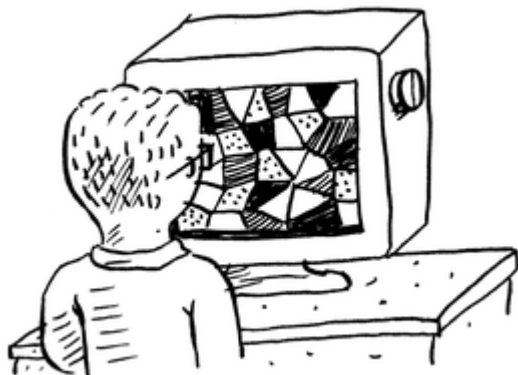


Figure 1.2

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 why it needs to be sorted later). 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:

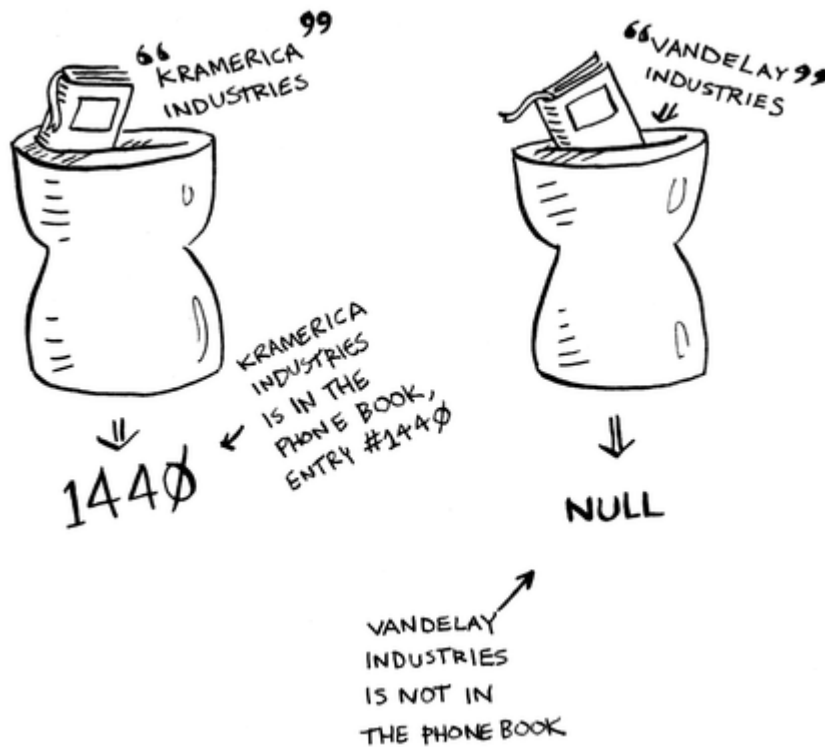


Figure 1.3 Looking for companies in a phone book with binary search

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

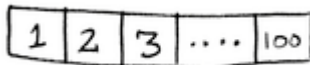


Figure 1.4

You have to try to guess my number in the fewest tries possible. With every guess, I'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.

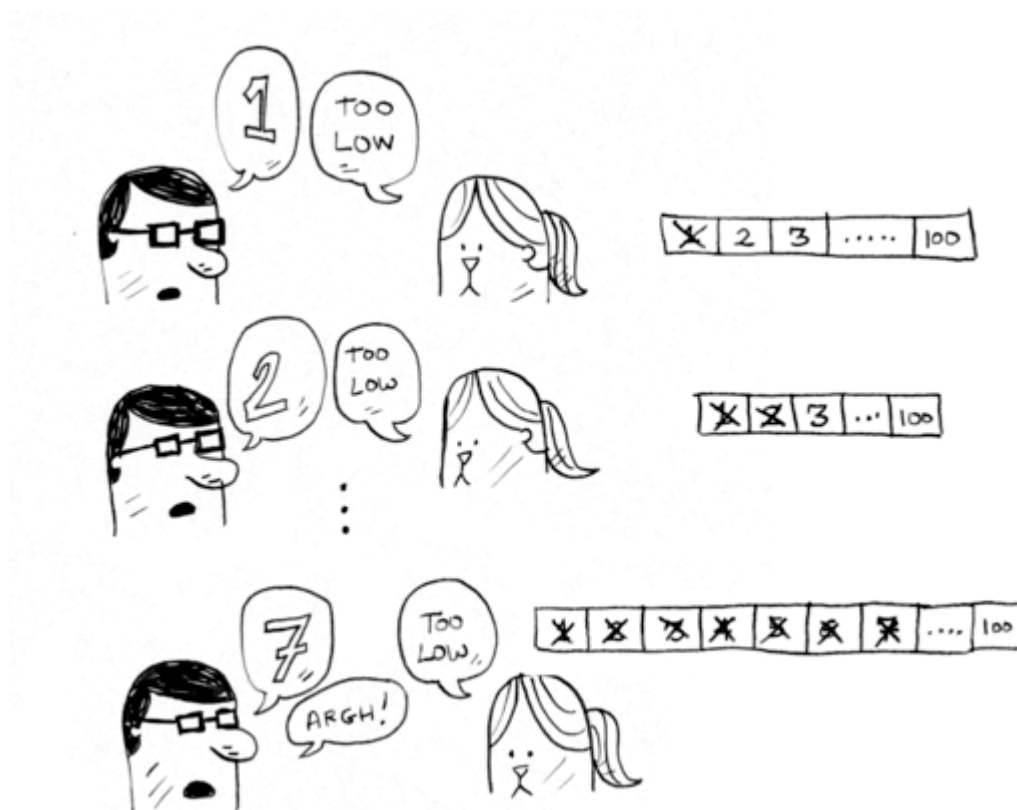


Figure 1.5 A bad approach to number guessing

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

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 remaining 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 you can eliminate every time.

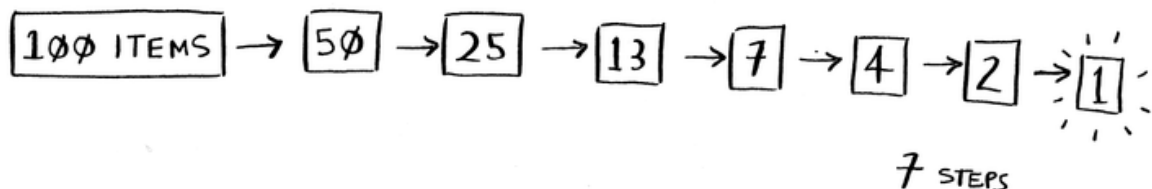


Figure 1.9 Eliminate half the numbers every time with binary search

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

EXERCISE

Here's another example. 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: _____ STEPS

BINARY SEARCH: _____ STEPS

Figure 1.10

Did you get it right? 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.

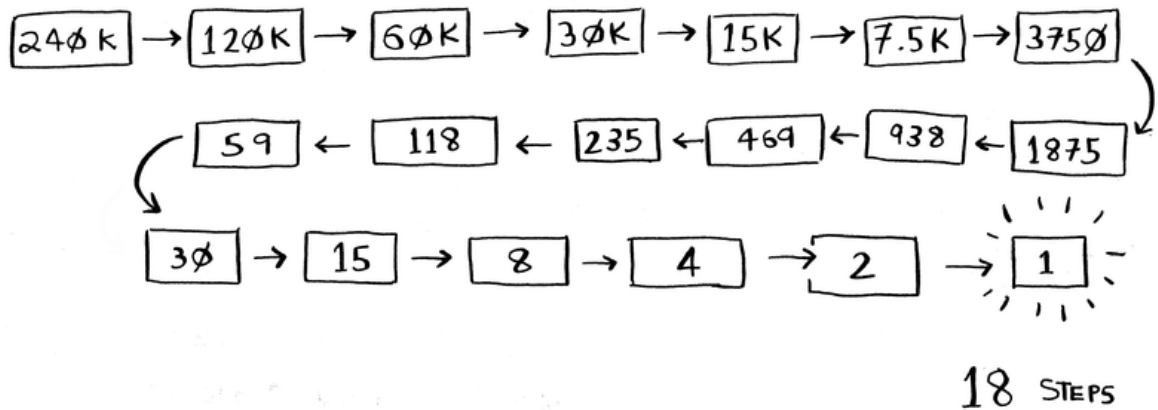


Figure 1.11

So binary search will take 18 steps – a big difference! In general, for any list of size n , binary search will take $\log_2 n$ steps to run in the worst case, whereas 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?" The answer is 2: 10×10 . So $\log_{10} 100 = 2$. Logs are the flip of exponentials.

$$\begin{array}{l}
 10^2 = 100 \leftrightarrow \log_{10} 100 = 2 \\
 \hline
 10^3 = 1000 \leftrightarrow \log_{10} 1000 = 3 \\
 \hline
 2^3 = 8 \leftrightarrow \log_2 8 = 3 \\
 \hline
 2^4 = 16 \leftrightarrow \log_2 16 = 4 \\
 \hline
 2^5 = 32 \leftrightarrow \log_2 32 = 5
 \end{array}$$

Figure 1.12 Logs are the flip 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 would have to check: 8 numbers. 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 3 numbers at most. For a list of 1024 elements, $\log 1024 = 10$, because $2^{10} = 1024$. So for a list of 1024 numbers, you would have to check 10 numbers at most.

NOTE we will 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](https://www.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?

1.1.2 Binary search in code

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 are 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 zero. So the first bucket is at position #0, the second is #1, the third is #2 and so on.

The `binary_search` function will take a sorted array and an item. If the item is in

the array, the function will return its position. You'll keep track of what part of the array you have to search through. At the beginning, this is the whole array:

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

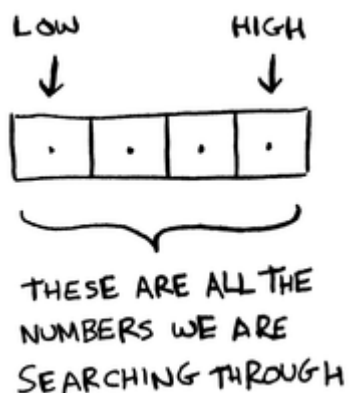


Figure 1.13

Each time, you check the middle element:

```
mid = (low + high) / 2 ①
guess = list[mid]
```

- ① mid is rounded down by Python automatically if $(low + high)$ isn't an even number

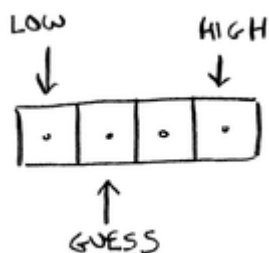


Figure 1.14

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

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

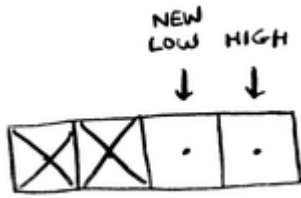


Figure 1.15

And if the guess is too high, you update high.

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
```

- ❶ low and high keep track of which part of the list you'll search in.
- ❷ While you haven't narrowed it down to one element ...
- ❸ ... check the middle element
- ❹ Found the item.
- ❺ The guess was too high.
- ❻ The guess was too low.
- ❼ Item doesn't exist

Lets test it out!

```
my_list = [1, 3, 5, 7, 9]

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

- ❶ remember, lists start at zero. The second slot has index 1
- ❷ 'None' means nil in Python. We use to indicate that the item wasn't found.

EXERCISE

Suppose you have a sorted list of 128 names, and you are searching through it using binary search. What is the maximum number of steps it would take?

Answer: 7.

- Suppose you double the size of the list. What is the maximum number of steps now?

Answer: 8.

1.1.3 Running time



Figure 1.16

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 100 guesses. If it's a list of 4 billion numbers, it takes 4 billion guesses. So the 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 7 guesses. If the 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:

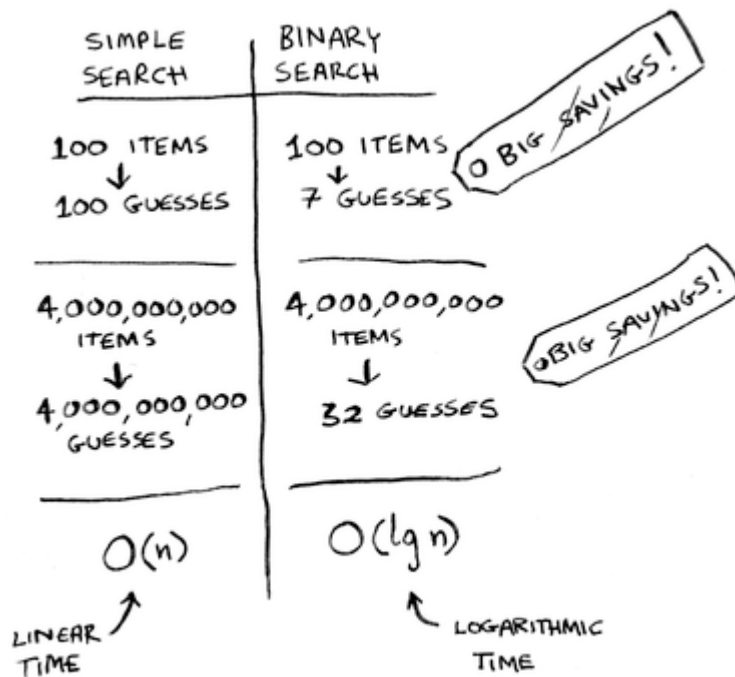


Figure 1.17 Run times for search algorithms

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. 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.

1.2.1 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.



Figure 1.18

This is an example of how the runtime 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.

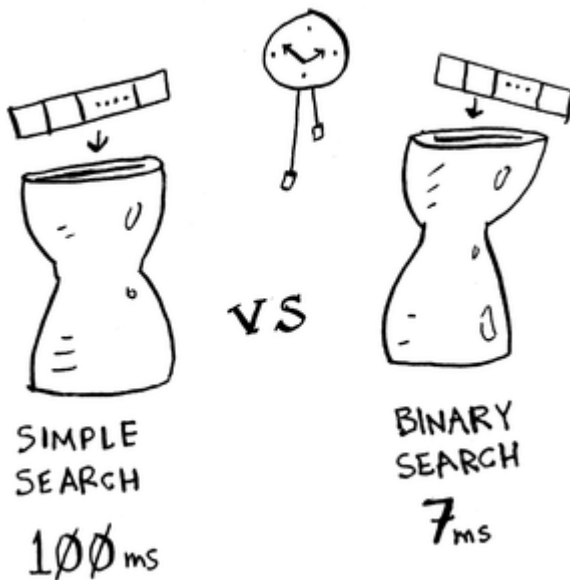


Figure 1.19 Running time for simple search vs. binary search, with a list of 100 elements

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.

Bob runs binary search with 1 billion elements, and it takes 30 ms ($\log_2 1,000,000,000$ is roughly 30 because $2^{30} = \sim 1$ billion). "32 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 $30 \times 15 = 450$ 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 milliseconds, 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	100ms	7ms
10,000 ELEMENTS	10 seconds	14ms
1,000,000,000 ELEMENTS	11 days	32ms

Figure 1.20 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.



Figure 1.21

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.

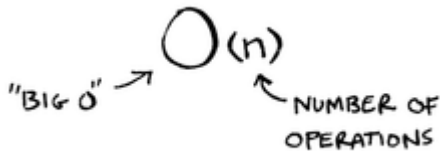


Figure 1.22 What Big O notation looks like

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.

1.2.2 Drawing squares

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.

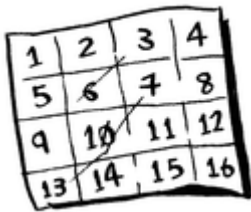


Figure 1.23

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. Remember, big-O notation counts the number of *operations*. In this example, drawing one box is one operation. We have to draw sixteen boxes. How many operations will it take, drawing boxes one at a time?

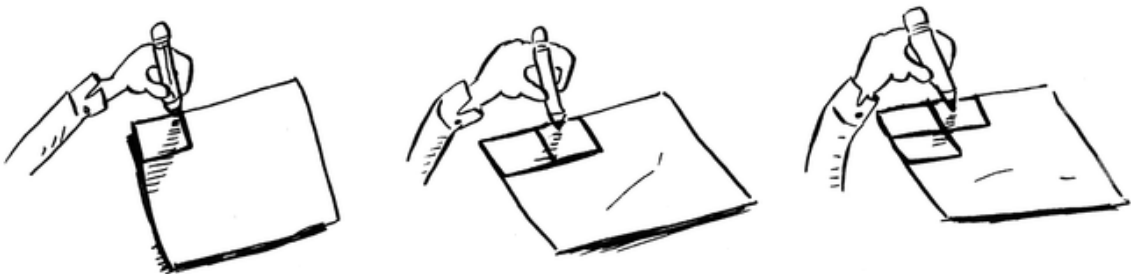


Figure 1.24 Drawing a grid one box at a time

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.



Figure 1.25

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

Fold the paper again, and again, and again.

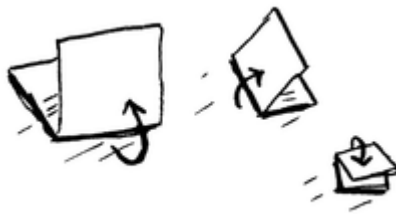


Figure 1.26

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

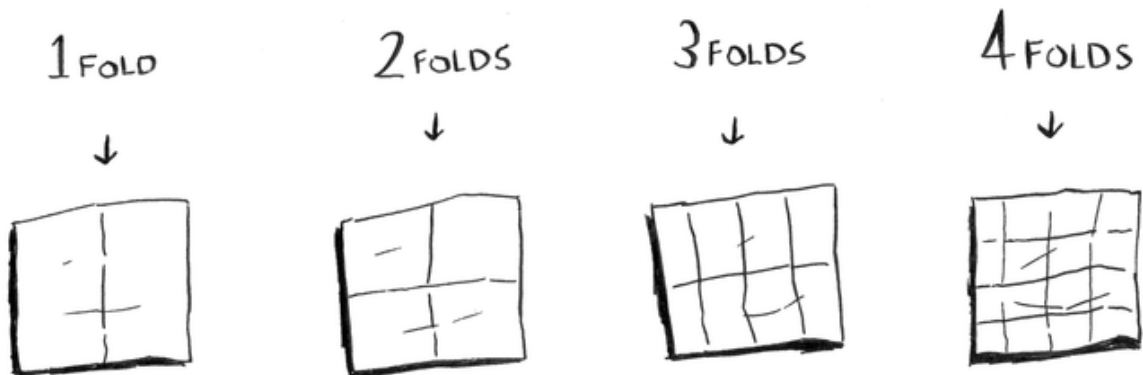


Figure 1.27 Drawing a grid in four folds

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.

1.2.3 Big O establishes a worst-case runtime

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 Big O notation is about the *worst* case scenario. 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 worst-case runtime, it is also important to look at the *average-case* runtime. Worst case vs average case is discussed in chapter 4.

1.2.4 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!).

SIDEBAR

What does " $n!$ " mean?

If you have never seen this notation before, don't worry. This is read as " n factorial". Here's a simple example: $3! = 1 * 2 * 3 = 6$. It is a shorthand notation for multiplying every positive integer up to that number. Here are some more examples:

- $5! = 1 * 2 * 3 * 4 * 5 = 120$
- $10! = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10 = 3,628,800$

As you can see, the numbers get big really fast. What do you think $20!$ is? Take a guess, then Google it and see the result.

Suppose you are drawing a grid of 16 boxes again, and you can choose from five different algorithms to draw this grid. If you use the first algorithm, it will take you $O(\log n)$ time to draw the grid. You can do 10 operations a second. So 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 this grid. What if you have to draw 1024 boxes? It will take you $\log 1024 = 10$ operations, or one second to draw a grid of 1024 boxes. These numbers are using the first algorithm.

The second algorithm is slower: it takes $O(n)$ time. So it will take 16 operations to draw 16 boxes. It will take 1024 operations to draw 1024 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:

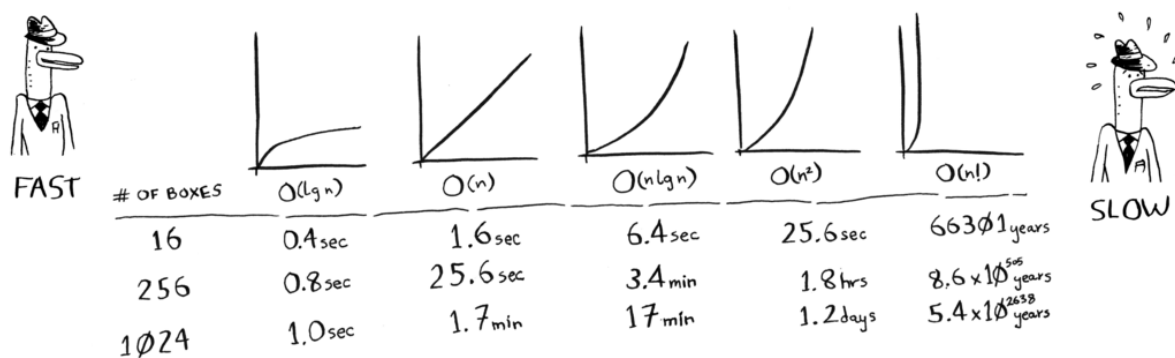


Figure 1.28

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 runtime 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 growth of the number of operations.
- Instead, 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.
- $O(\lg n)$ is faster than $O(n)$, but it gets a lot faster as the list of items you're searching grows.

EXERCISE

Give the run time for each of these questions in terms of Big-O.

- You have a name and you want to find their phone number in the phone book. A: $O(\log n)$.

- You have a phone number and you want to find their name in the phone book (hint: you will have to search through the whole book!). A: $O(n)$.
- You want to read out the numbers of every person in the phone book. A: $O(n)$.
- You want to read out the numbers of just the "A"s (This is a tricky one! It talks about concepts that are covered more in chapter 4. Read the answer to this one – you might be surprised!). A: $O(n)$. Did you get it right? This is a tricky one. You might think "I am only doing this for one out of twenty six characters, so the run time should be $O(n/26)$ ". A simple rule to remember is, you ignore numbers that are added, subtracted, multiplied, or divided. So none of these are correct big-O runtimes: $O(n + 26)$, $O(n - 26)$, $O(n * 26)$, $O(n/26)$. Because they are all the same as $O(n)$. Why? I'll come back to this in chapter 4, but if you are curious, flip to "Big-O Notation Revisited" in chapter 4 and read up on "constants" in big-O notation (a constant is just a number – "26" was the constant in this question).

1.2.5 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.



Figure 1.29

The salesperson has to go to five cities.

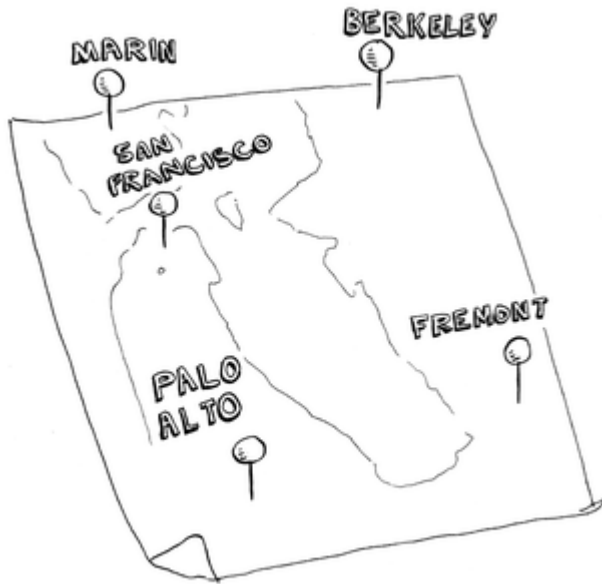


Figure 1.30

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.

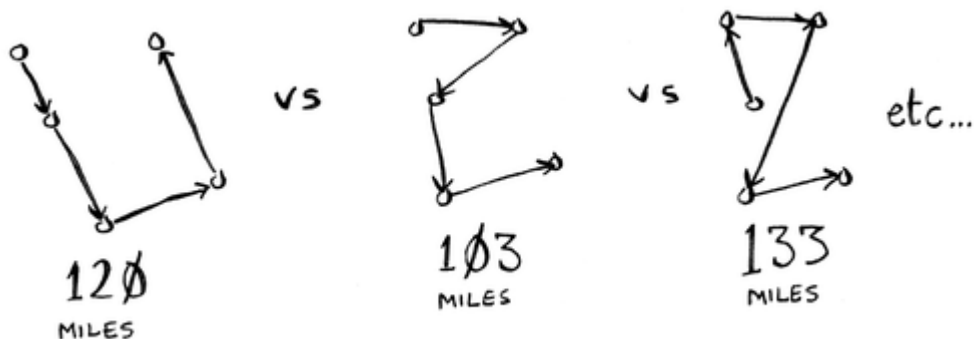


Figure 1.31

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!

CITIES	OPERATIONS
6	720
7	5040
8	40320
...	...
15	1,307,674,368,000
...	...
30	2,652,528,598,121,910,586,363,084,800,000,000

Figure 1.32 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! The salesperson 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 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 9 for more.

One final note: if you are an advanced reader check out Binary Search Trees! There's a brief description of them in the last chapter.

RECAP

- Binary search is a lot faster than simple search.
- $O(\lg n)$ is faster than $O(n)$, but it gets a lot faster once the list of items you're searching through grows.
- 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.