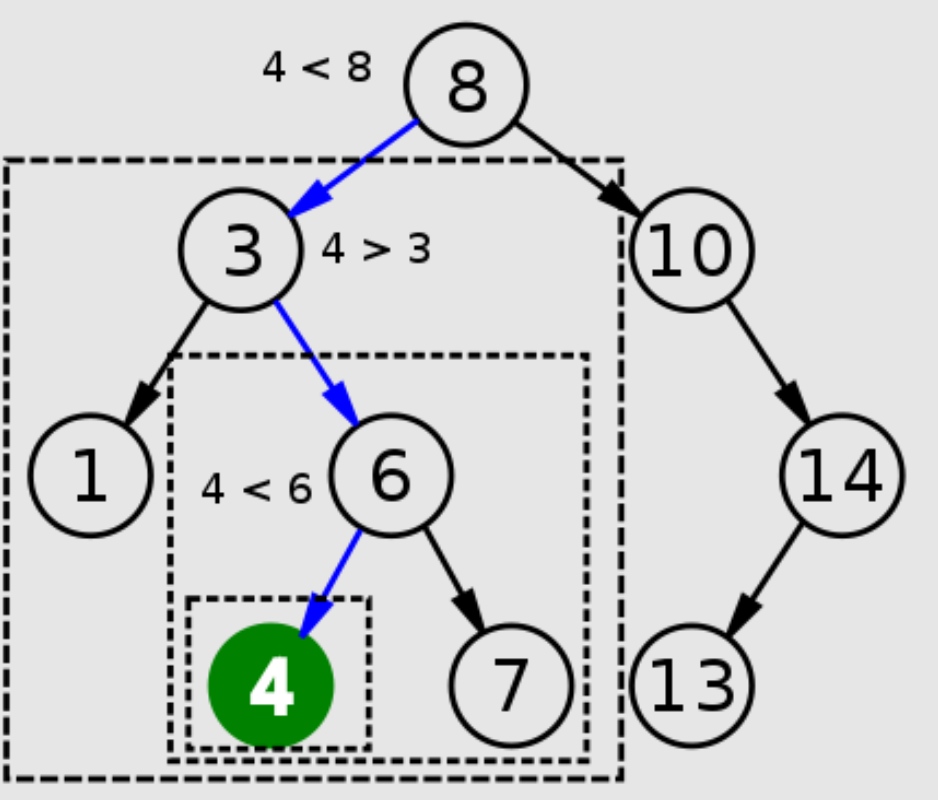
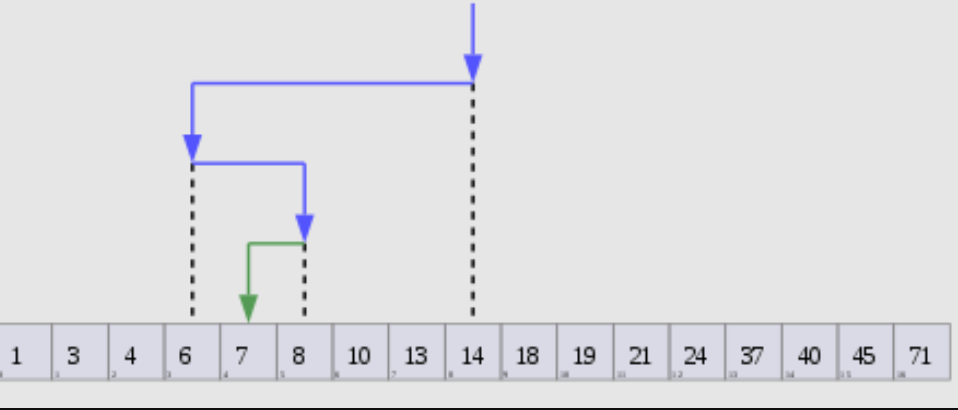
BINARY SEARCH

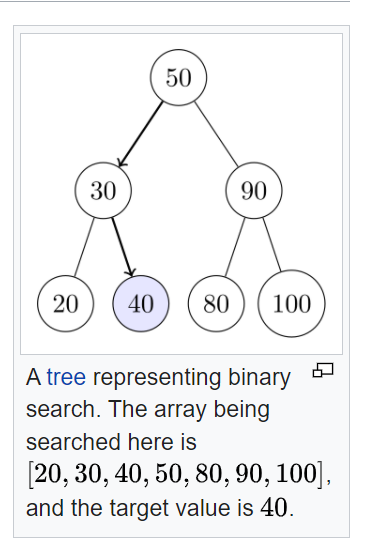
In computer science, binary search, also known as ha­­­lf-interval search, logarithmic search­, or binary chop is a search algorithm that finds the position of a target value within a sorted array Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

Binary search is an algorithm; its input is a sorted list of elements If an element you’re looking for is in that list.





Visualization of the binary search algorithm where 7 is the target value



Suppose you are looking for a **word/contact** start with M from a dictionary/Contacts.

More likely to start at a page in the middle. Binary search returns the position

Where it’s located. Otherwise, the binary search returns null.

Guess a number between 1-100. Guessing 1,2,3,4,5 is a stupid idea. The better way to search is to start with

50. if you say 50 then it's too low, but you just eliminated half the numbers! Now you know that

1–50 is all too low. Next guess: 75. Too high, but again you cut down half the remaining numbers! With

In 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 a binary search. You just learned your First algorithm! Here’s how many numbers you can eliminate every time.

**100>50>25>13>7>4>2>1** only 7 step instead of 99 . This is c Called 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. He dictionary has

240,000 words. In the worst case, how many steps do you think each

Search will take?

Simple search:Simple search could take 240,000 steps if the word you’re looking for is

The very last one in the book.

Binary search:With each step of binary search, you cut the number of words in half until you’re let with only one word.

Binary search will take 18 steps—a big difference! In general, for any

List of n, binary search will take log2 n steps to run in the worst case,

Whereas simple search will take n steps.

When you search for an element using

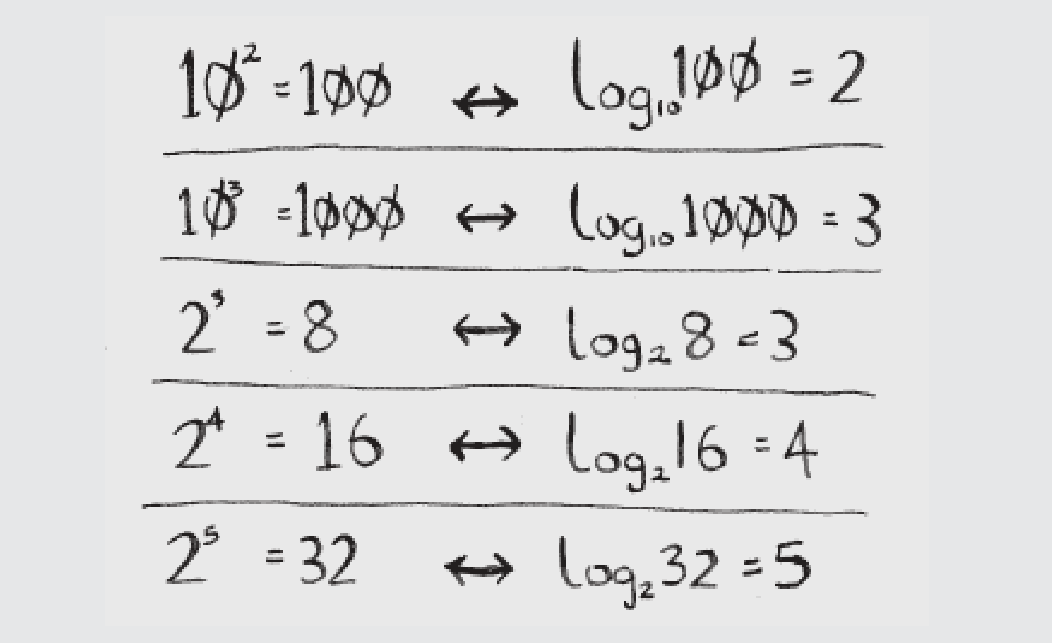
**Simple search:** in the worst case you might have to look at every single

Element. So for a list of 8 numbers, you’d have to check 8 numbers at most.

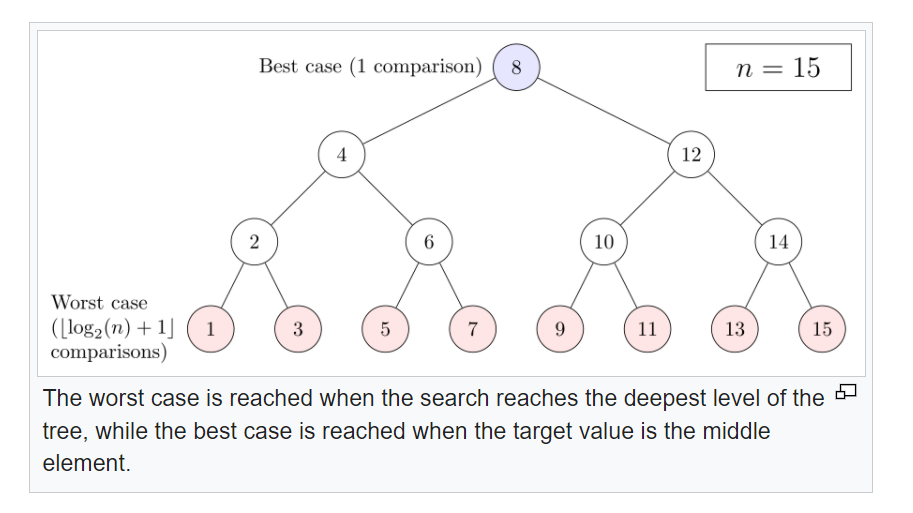
For **binary search:** you have to check log n elements in the worst case. For

A list of 8 elements, log 8 == 3, because 23 == 8. So for a list of 8 numbers,

You would have to check 3 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.



**Logarithm:** For a number x , the power to which a given base number must be raised in order to obtain x . Written \log\_base x . For example, \log\_{10} 1000 = 3 because 10^3 = 1000 and \log\_2 16 = 4 because 2^4 = 16 .3 is the logarithm of 100 and 4 is the logarithm of 1000 . base number must be power up.

logarithm, the exponent or power to which a base must be raised to yield a given number. Expressed mathematically, x is the logarithm of n to the base b if bx = n, in which case one writes x = logb n(3 = log2 8). For example, 23 = 8; therefore, 3 is the logarithm of 8 to base 2, or 3 = log2 8. In the same fashion, since 102 = 100, then 2 = log10 100. Logarithms of the latter sort (that is, logarithms with base 10) are called common, or Briggsian, logarithms and are written simply log n.

log2 16 = 4, since 24 = 2 × 2 × 2 × 2 = 16. Here 4 is the logarithm of 16

Binary search only works when your list is in sorted order or Array. For example,

the names in a phone book are sorted in alphabetical order, so you can

use **a binary search** to look for a name.

**EXERCISES**

**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?

**Ans**: 7 times. Log2 128 = 7

**1.2** Suppose you double the size of the list. What’s the maximum

number of steps now?

**Ans**: 8 times. Log2 256 = 8

**Linear Time:** If it 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. his is

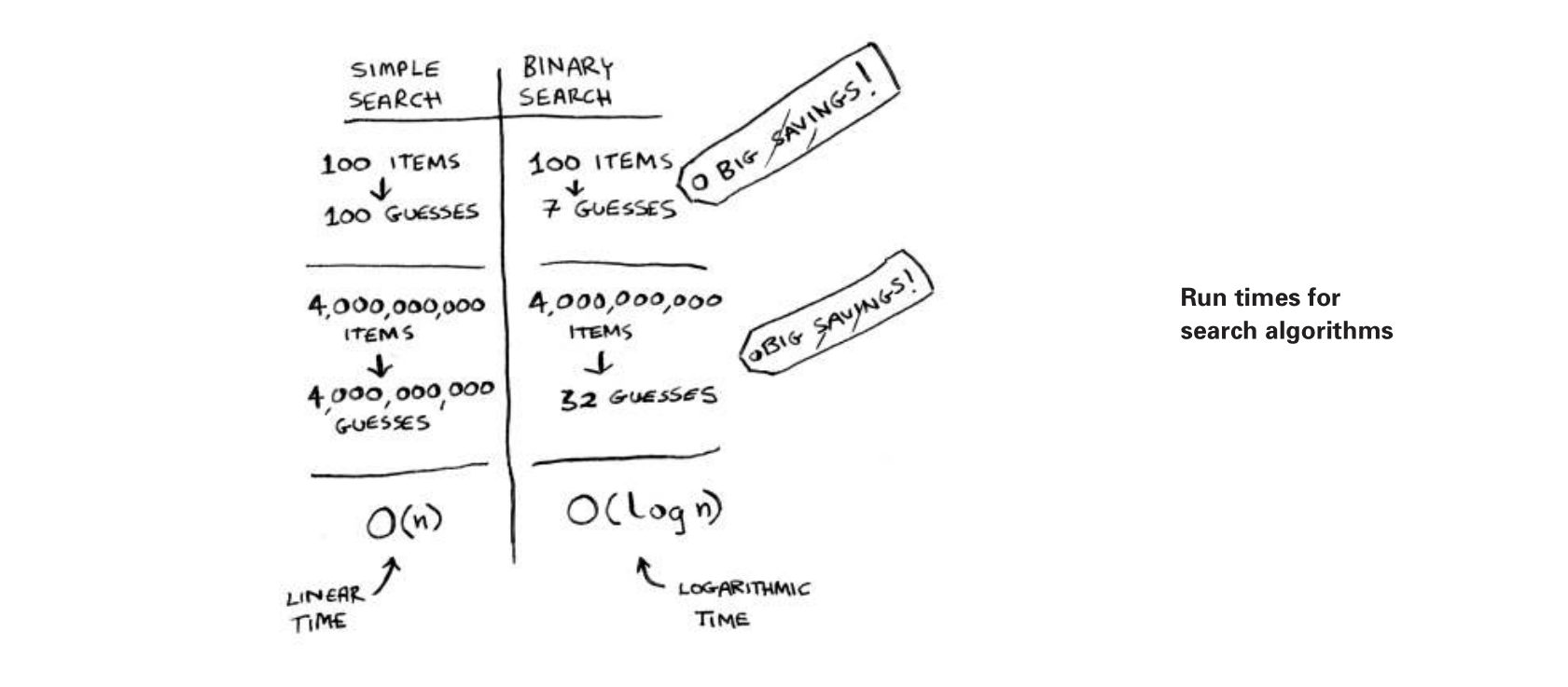
called **linear time O(n)**.

**logarithmic 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? The binary search runs in **logarithmic time(O(**log n**)** (or log time, as the natives call it).

Here’s a table summarizing our findings today.



**Big O notation**

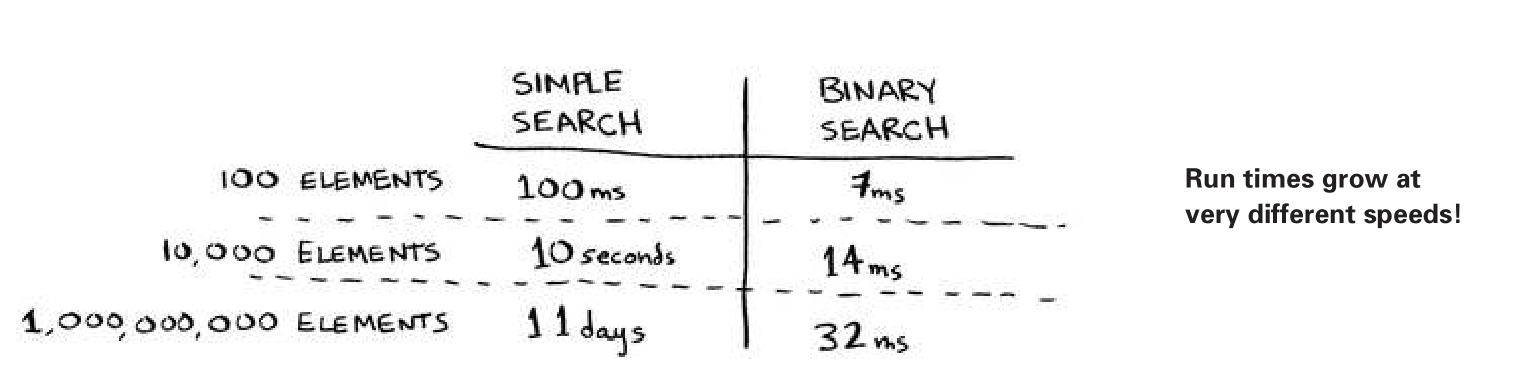
Big O notation is a special notation that tells you how fast an algorithm is.

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.



he runs time for a simple search with 1 billion items will be 1 billion ms, which is 11 days! he

problem is, the run times for binary search and simple search don’t

grow at the same rate.

That is, as the number of items increases, the 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. hat’s why it’s not enough

to know how long an algorithm takes to run—you need to know how

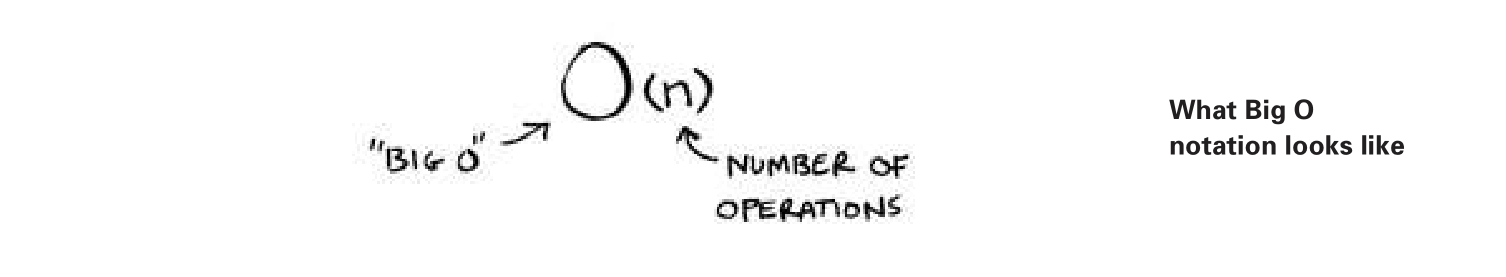
the running time increases as the list size increases. That’s where Big O

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. he run time in **Big O notation is O(n)[Linear Time].** Where are the seconds? here 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

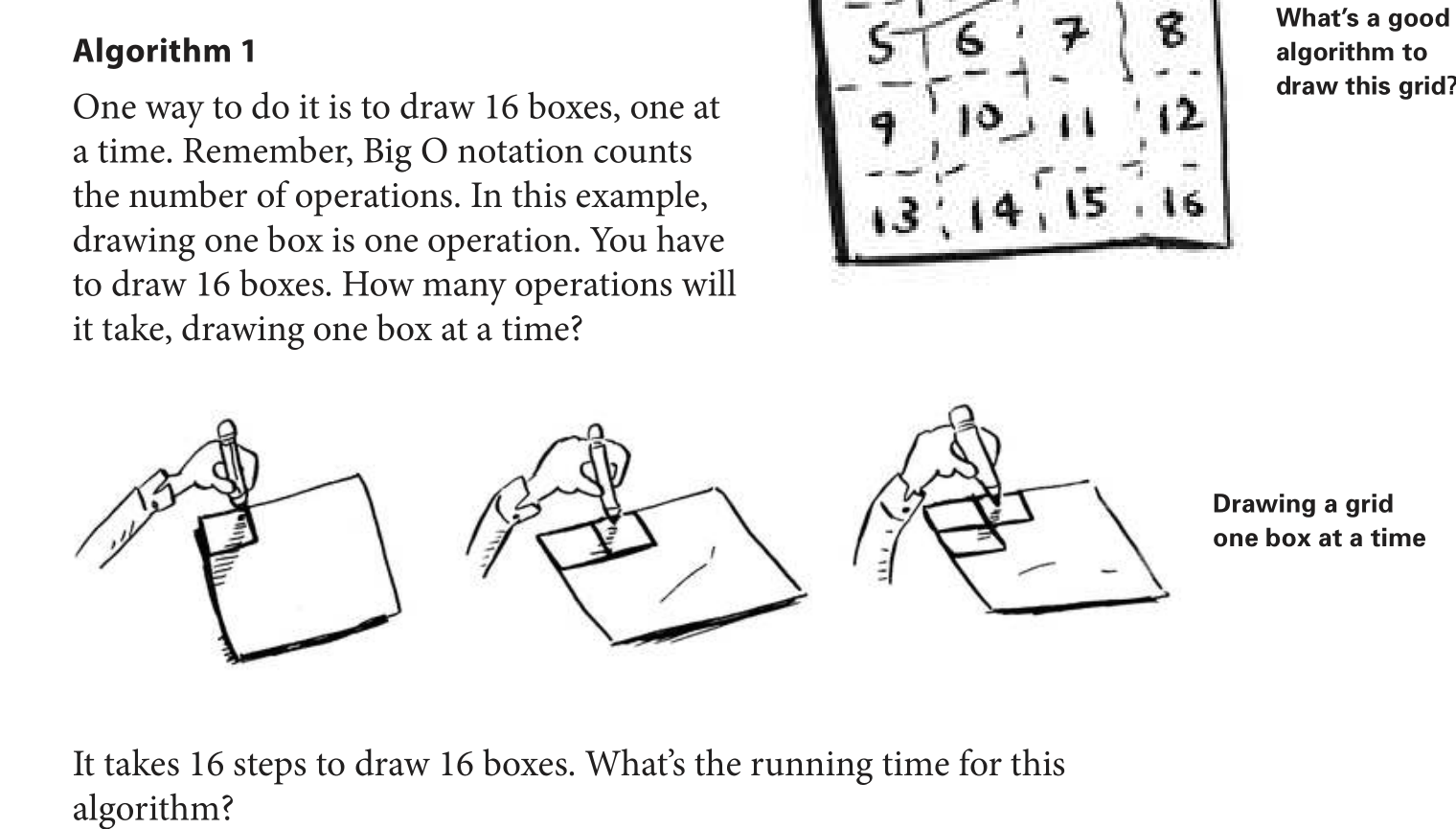
a list of size n. What’s the running time in Big O notation? It’s **logarithmic time** **O(log n).** In general, Big O notation is written as follows.

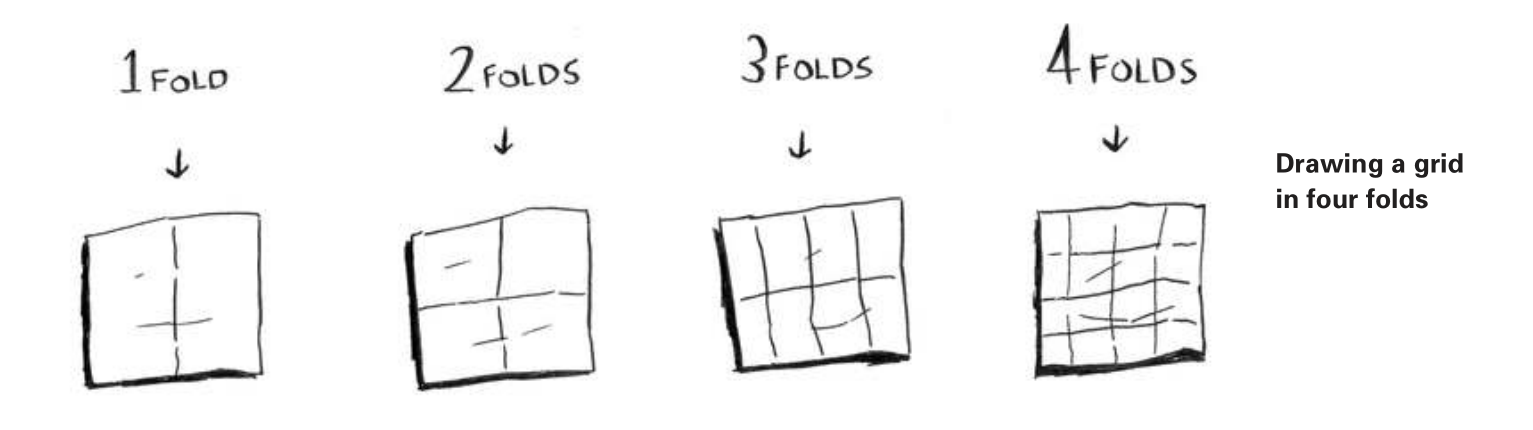


This tells you the **number of operations** an algorithm will make. It’s

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!).





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. his guy is

the irst entry in your phone book. So you didn’t have to look at every

entry—you found it on the irst try. Did this algorithm take O(n) time?

Or did it take O(1) time because you found the person on the irst try?

Simple search still takes O(n) time. In this case, you found what you

were looking for instantly. hat’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.

hat’s O(n) time. It’s a reassurance—you know that simple search will

never be slower than O(n) time.

**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(n2).** 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.

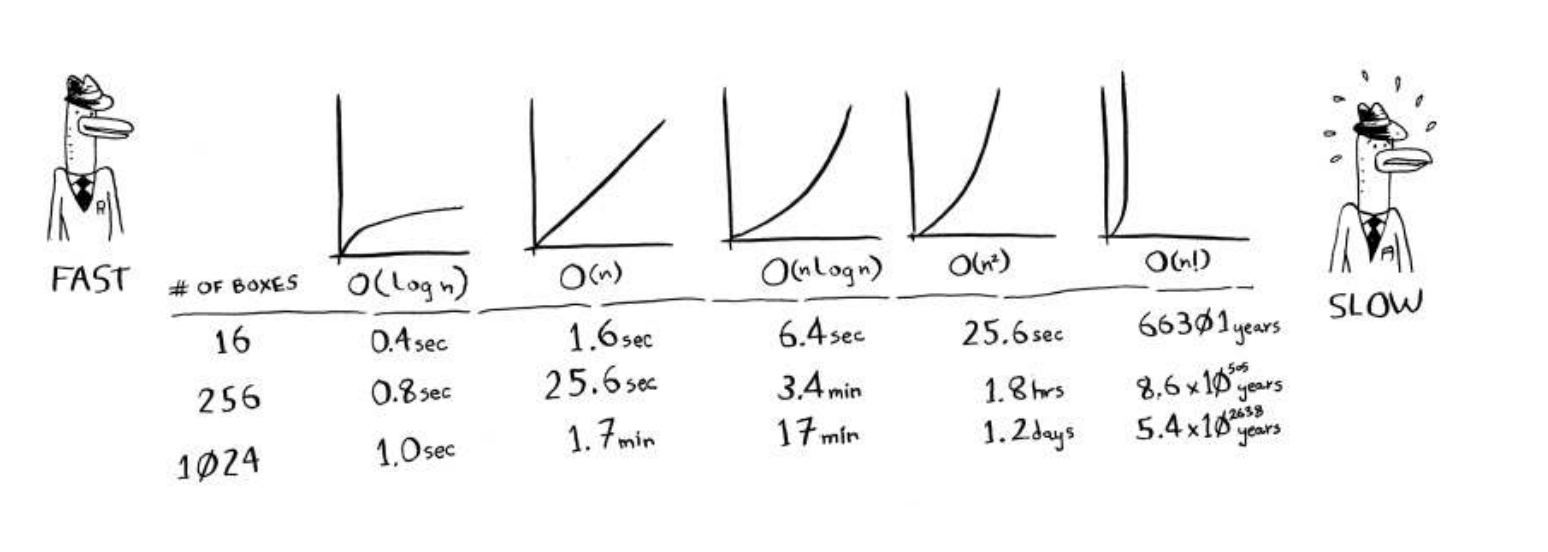
The second algorithm is slower: it takes O(n) time. It will take 16

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:



**The main takeaways are as follows:**

• Algorithm speed isn’t measured in seconds, but in the growth of the

**number of operations.O(n).** Here n Is a 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(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.

**1.3** You have a name, and you want to ind the person’s phone number

in the phone book.

Ans: O(log n) Binary

**1.4** You have a phone number, and you want to ind the person’s name

in the phone book. (Hint: You’ll have to search through the whole book!)

Ans: O(n)

**1.5** You want to read the numbers of every person in the phone book.

Ans: O(n)

**1.6** You want to read the numbers of just the As. (his is a tricky one!

It involves concepts that are covered more in chapter 4. Read the

answer—you may be surprised!)

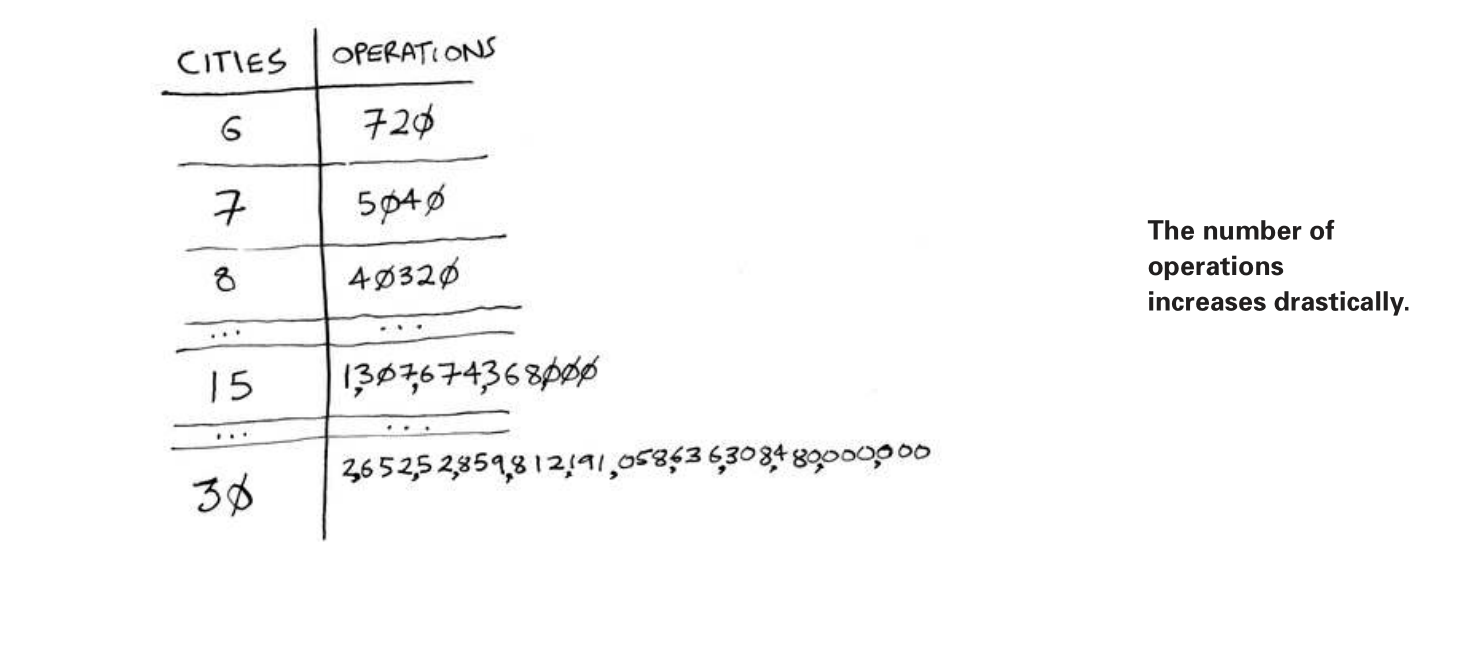
Ans: O(n). You may think, “I’m only doing this for 1 out of 26 characters, so the run time should be O(n/26).” A simple rule to remember is, ignore numbers that are added, subtracted, multiplied, or divided. None of these are correct Big O run times: O(n + 26), O(n - 26), O(n \* 26), O(n / 26). They’re all the same as O(n)! Why? If you’re 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).

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.



**Recap**

• Binary search is a lot faster than simple search.

• O(log 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 the growth of an algorithm.

• Algorithm times are written in Big O notation.