Chapter 1 Notes

A set of instructions that accomplish (or helps accomplish) a task.

↳ Definition of an algorithm

Performance

Understand the trade-offs and the benefits of each algorithm

- Also remember that using a different data structure does create a difference in performance.

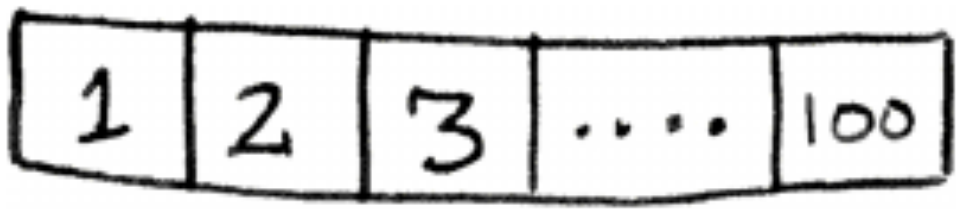
Binary Search

An algorithm for searching for the location (or the existence) of an element from a well-organized data structure (E.g arrays)

If the binary search cannot find the element. It returns null.

Example of Binary Search

Find a number from 1 to 100,



With every guess, you are told whether the number you guessed is too high or too low.

↳ If you were to guess a number from 0 to 100, it would be incredibly inefficient to guess one by one you are eliminating one number out of many.

↳ This idea of eliminating elements one-by-one is called Simple Search.

↳ incredibly inefficient

Where Binary Search comes in

↳ Instead of starting with 0, start halfway.

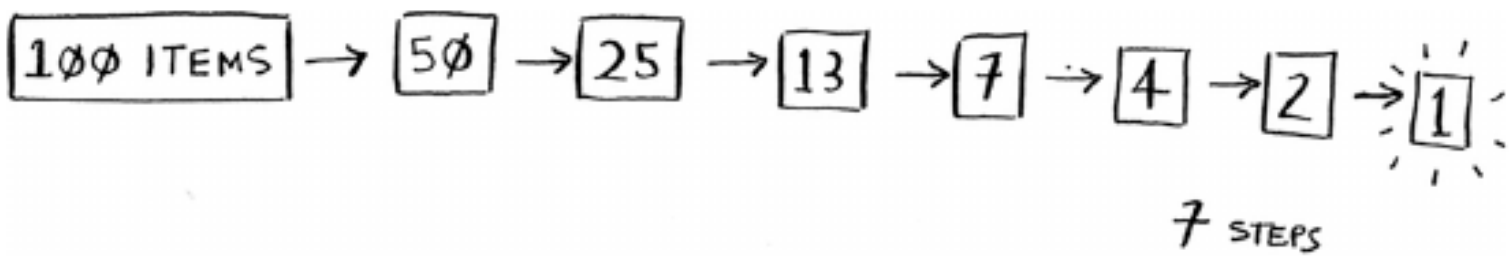
↳ Based on the response, you can remove all elements lesser or greater than the halfway guess

Result: (~~<60~~)

↳ Removed half of all numbers.

If you were to guess the remaining half elements, you can remove another 50% of available elements.

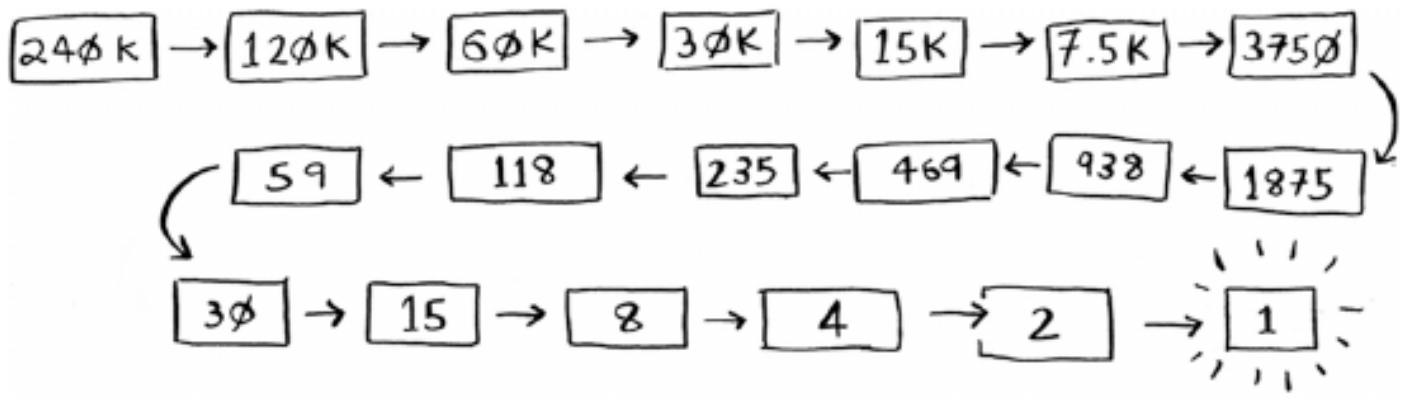
By choosing the half of the amount of available elements left, you lessen the amount of steps needed to find the element.



Comparison between Binary Search and Simple Search.

To see the comparison, imagine finding an element out of 240,000 elements

↳ Although it would take Binary Search more attempts to find the element, in comparison to Simple Search, it would be far more preferable than selecting 240,000 elements one-by-one



SIMPLE SEARCH: 240,000 STEPS

BINARY SEARCH: 18 STEPS

↳ In general, for any list of n, binary search will take $\log_2 n$ steps to run in it's worst case

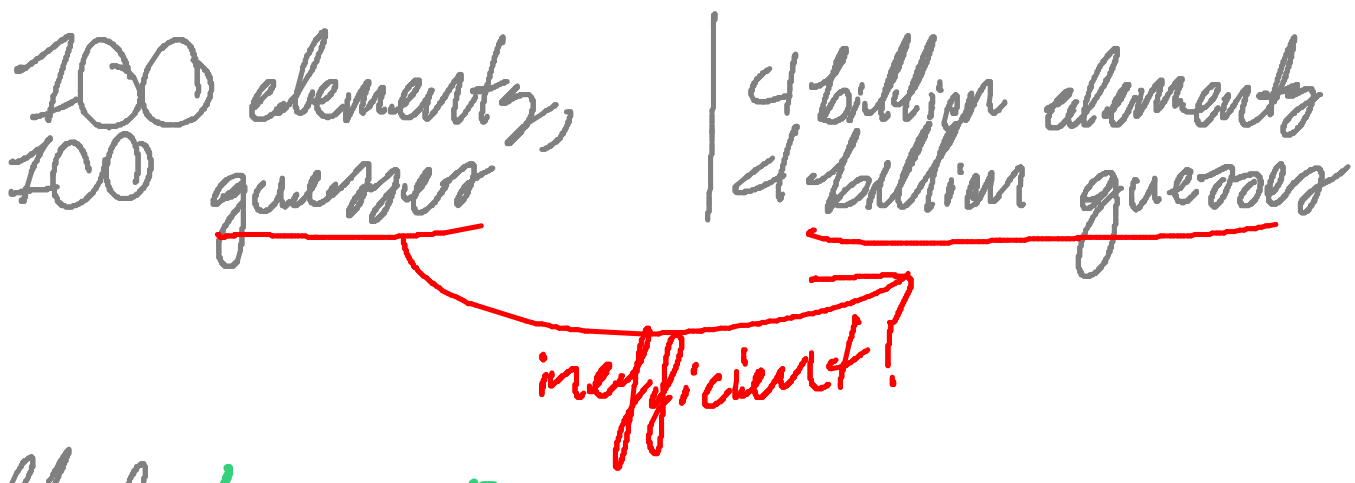
for Simple Search, it'll only take n steps to run.

Running Time

↳ Generally when you're talking about algorithms you typically also mention the algorithms performance, or Running Time. Usually the target would be either time or space.

↳ Linear Time

Find a number out of 100 elements, guess what, it would take 100 attempts, well if the amount explodes to 4 billion, the amount of attempts will also explode to that same amount.



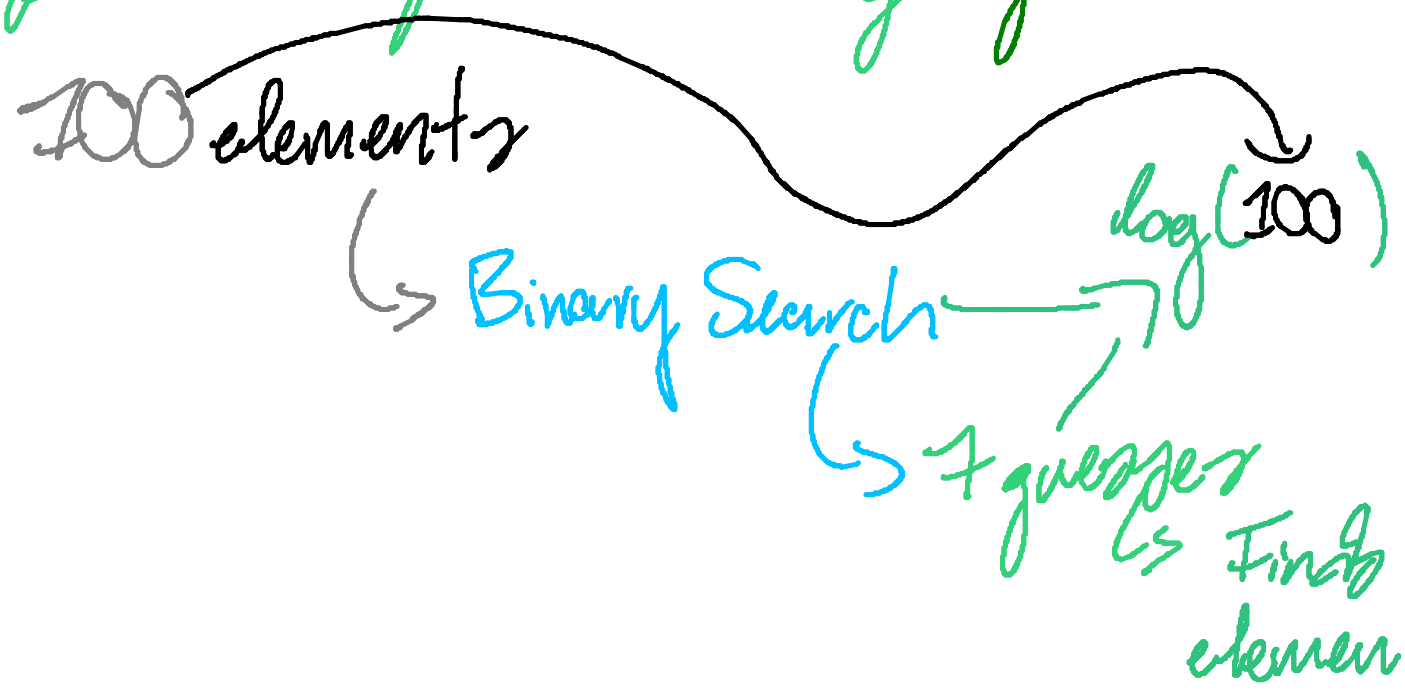
That is called Linear Time.

↳ under Simple Search

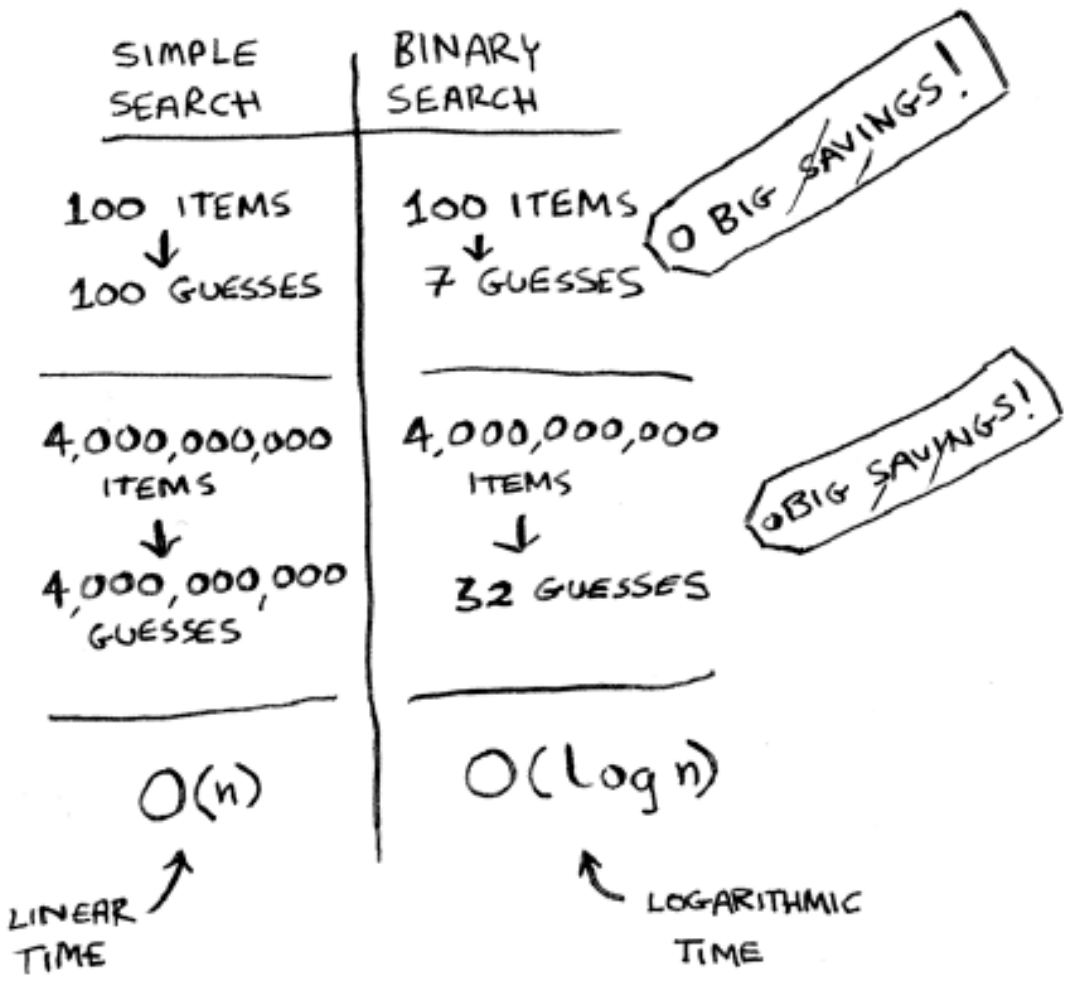
Logarithmic Time — or "log time"

↳ Under Binary Search

↳ Takes a list of n elements, (e.g 100) and passes through the list by log n.



Linear vs Log Time



Big O Notation

↳ Tells you how fast an algorithm is.

Why know Big O?

↳ Usually it's to see different algorithms' running times for compare & contrast OR visualise a worst-case scenario runtime.

Algorithm running times grow at different rates

↳ When you compare Simple Search with Binary Search by parsing through one of the two algorithms and assuming that since one algorithm is "n times" faster or slower than the other and multiply the either or result with the multiple, then that would also be assuming one thing, that both algorithms run at the same rate. Which is wrong.

The problem is that the run times for binary search and simple search *don't grow at the same rate.*

O(100)=100

O(log2(100))

↳7

	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

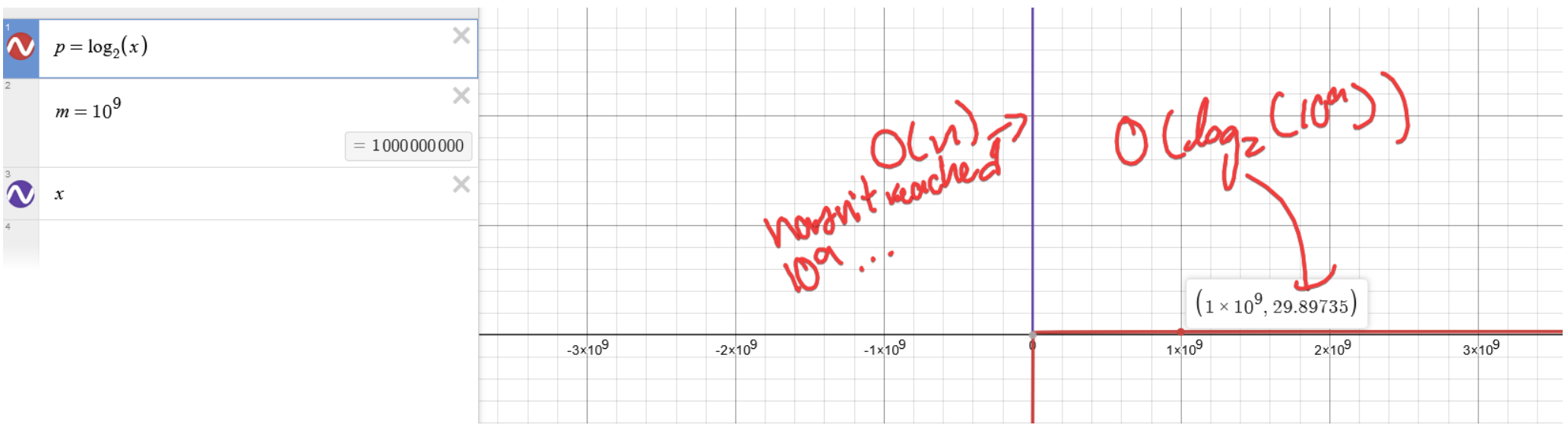
O(10^9)

O(log2(10^9))

What was the problem?

↳ the problem is that you did not compare both algorithms side-by-side with a table nor a graph.

Here's the result of that



	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

The lesson is:

↳ Make sure you know the running times of any algorithm over time

↳ More elements,
More Running Time

The Purpose of Big O Notation

Big O Notation tells you how fast an algorithm is.

↳ There are NO seconds in Big O, only run times are the way to measure & compare algorithms.

↳ Big O notation lets you compare the # of operations.
- A.K.A tells you how fast an algorithm grows.

How Big O is written

↳ If Binary Search needs $\log_2(n)$ operations to check a list of size n, then the running time in Big O would be:

$$O(\log_2 n)$$

↑ "big 'O'"

↑ number of operations

Remember, Big O establishes the worst-case scenario

↳ Just because you found the element you wanted on the very first try, using Simple Search doesn't mean it took O(1) time, Big O is there to time on finding the element when you have to parse through the entire list of elements, meaning than no matter what, Simple Search takes O(n) time.

Here's a table to give some perspective:

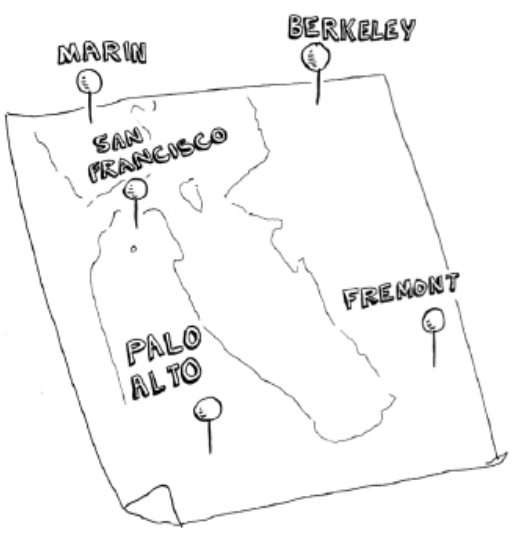
# OF BOXES	O(log n)	O(n)	O(n log n)	O(n²)	O(n!)
16	0.4 sec	1.6 sec	6.4 sec	25.6 sec	6630 1 years
256	0.8 sec	25.6 sec	3.4 min	1.8 hrs	2.7×10^{498} years
		1.7 min	17 min	1.2 days	1.7×10^{2631} years

The Travelling Salesperson Problem

↳ Demonstrates an algorithm in $O(n!)$, the worst running time for an algorithm.

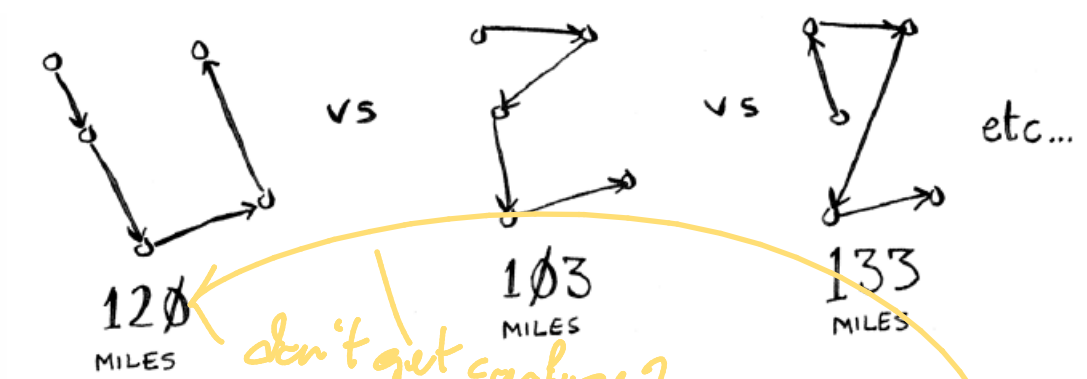
(n! = factorial)

A salesperson has to go to 5 cities...



The main goal for the salesperson is that they have to get through all 5 cities while traveling the minimum distance.

There's only one way to figure it out, look at every possible order in which he could travel.



don't get confused w/ this

With 5 cities, there are 120 possibilities, for more cities, it will take an exponential amount of possibilities

O(n!) →

CITIES	OPERATIONS
6	720
7	5040
8	40320
...	...
15	1,307,674,368,000
...	...
30	265,252,859,812,191,058,636,308,480,000,000

The number of operations increases drastically.

Terrible Algorithm!

It would take O(n!) or factorial time to get through every possible path!

The worst part is that the salesperson can't choose another algorithm for this problem. This is one of the unsolved algorithms in Computer Science. ← thought to be impossible to solve.