Time Complexity

Aymane Ouahbi
Khalid Samaoui

ANRT
الوكالة الوطنية لتقنين المواصلات
Agence Nationale de Réglementation des Télécommunications

INPT
المعهد الوطني للبريد والمواصلات
Institut National des Postes et Télécommunications

WHAT DO WE MEAN BY TIME COMPLEXITY ?

In a straightforward manner, time complexity refers to the total amount of time required by an algorithm to complete its execution. The lesser the time complexity, the faster the execution.

It's a way of measuring the algorithm's efficiency independently of the processing power .

Which will have a faster execution time, a slow algorithm on a fast computer, or an optimized algorithm on a much slower computer ?

Computer A (Faster)

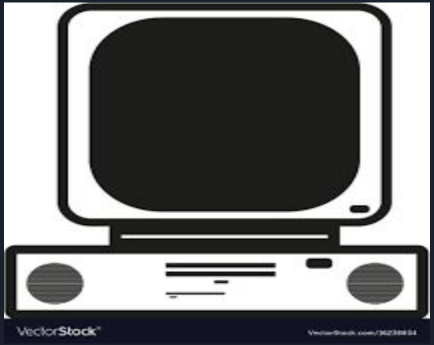
Insertion Sort

Does $K1*n^2$ instructions to sort n items

10 billion instructions per second

Suppose $K1 = 2$

Computer B (Slower)



Merge Sort

Does $K2*n*\log(n)$ instructions to sort n items

10 million instructions per second

Suppose $K2 = 50$

Task: Sort an array of 10 million numbers

so,

the differences due to time complexity can be much more significant than differences due to hardware and software.

HOW DO WE MEASURE TIME COMPLEXITY ?

We use a mathematical notation called Big-O notation;

Mathematical Definition:

Let n -> f(n) and n -> g(n) be functions defined over the natural numbers.

Then we say that f = O(g) if and only if f(n)/g(n) is bounded when n approaches infinity. In other words, **f = O(g) if and only if there exists a constant A, such that for all n, f(n)/g(n) <= A.**

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? 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.

Big O generally suppose the context of the Worst Case Scenario for the algorithm.

Common Big O run times

1. O (1) : Constant Time:

```
read(x)     // O(1)
a = 10;     // O(1)
a = 1.000.000.000.000.000.000 // O(1)
```

This means that whatever is the input size, the running time is always constant. This applies to basic operations (arithmetic, comparisons), as well as some built-in functions.

2. O (n) : Linear Time:

For an input of size n, the algorithm performs n operations.

Example: Linear Search

```python
x = 4
numbers = [22, 7, 16, 1, 3, 120, 4, 11]
for i in range(len(numbers)):
  if numbers[i] == x:
    print(x, "is found in the list.")
    break;
```

3. O (logn): Logarithmic Time:

We see this complexity when an algorithm divides the problem/input to smaller subproblems with the same size.

Algorithms having this complexity are much faster than linear time algorithms.

Examples: Binary Search, Binary conversion Algorithm

4. O(n log n) : Linearithmic Time

This running time is often found in “divide & conquer algorithms” which divide the problem into subproblems recursively and then merge them in n time.

Example: Merge Sort algorithm.

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

5. $O(n^2)$: Quadratic Time

This occurs when having a loop inside of loop:

Example: Bubble Sort, Insertion Sort, Selection Sort

6. $O(n^3)$: Cubic Time

When having a triple loop.

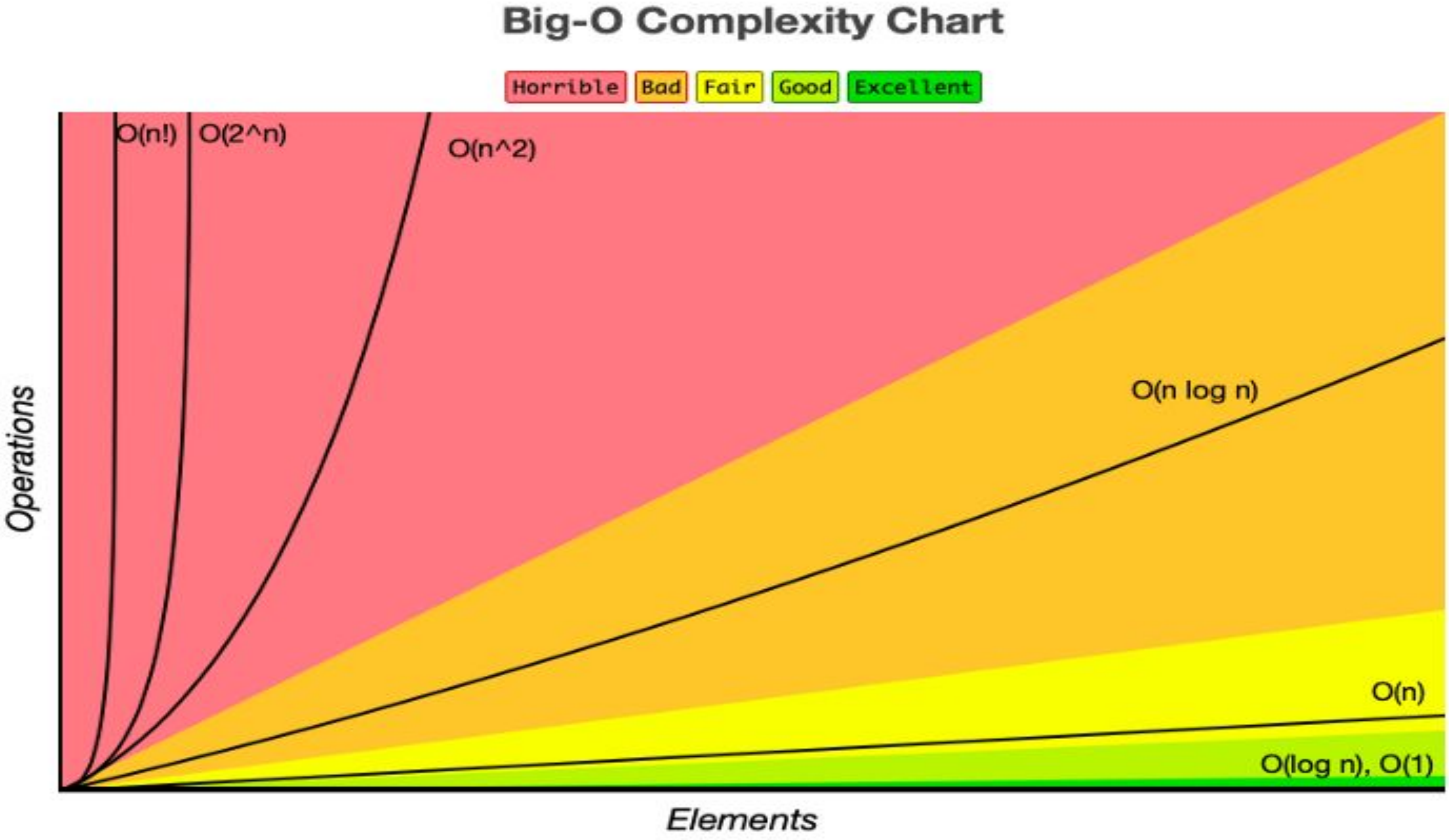
Can you come up with an example ?

7. O (2^n) : Exponential Time:

It is very slow as input get larger, if n = 50, T(n) would be 1 125 899 906 842 624. Brute Force algorithms has this running time.

8. O (n!) : Factorial Time:

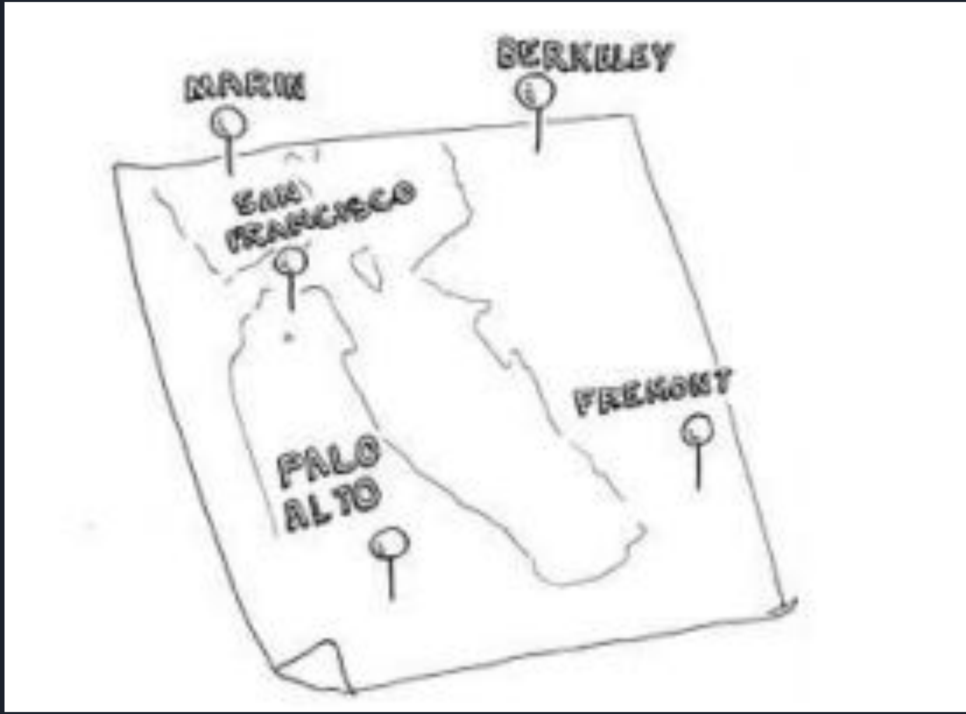
It’s the slowest of them all.

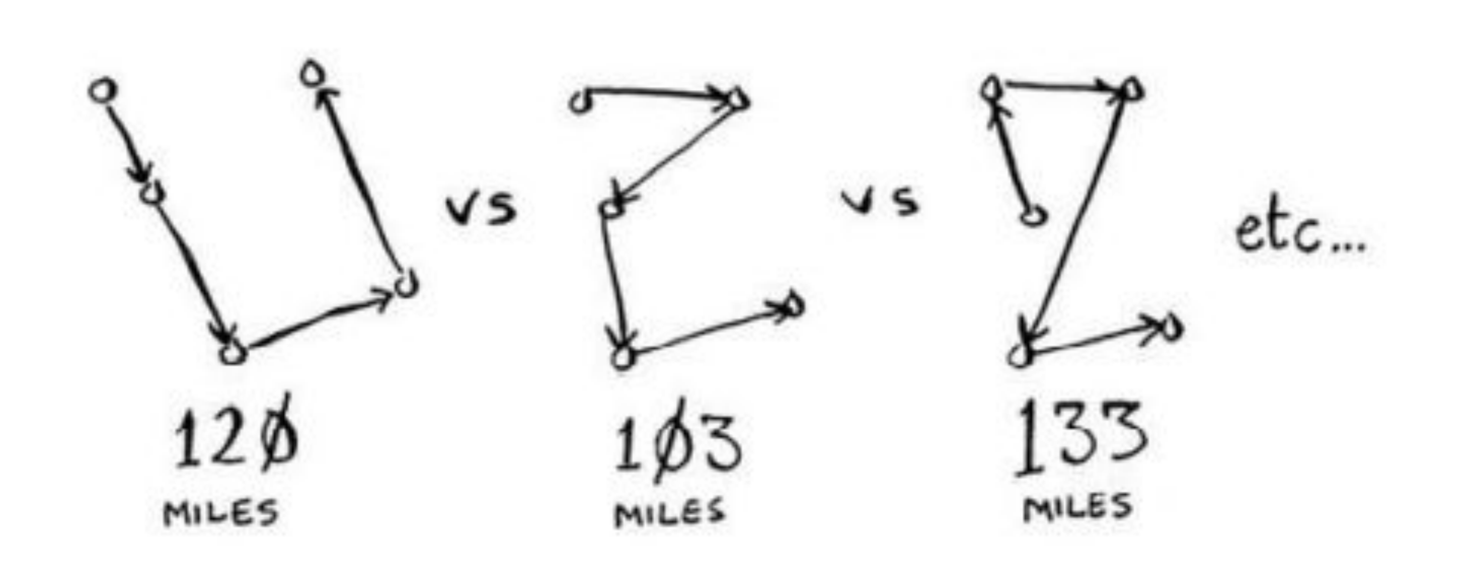

Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

An Example of O (n!) Solution: Traveling Salesman Problem

The salesman has to go to five cities:





vs
vs
etc...
120
MILES
103
MILES
133
MILES

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	265,252,859,812,191,058,636,308,480,000,000

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.

Problem: Maximum Subarray Sum

Given an array of n numbers, calculate the maximum subarray sum, i.e., the largest possible sum of a sequence of consecutive values in the array

−1	2	4	−3	5	2	−5	2

Quick Recap

- ➔ O(log n) is faster than O(n), but it gets a lot faster as the list of items you're searching grows.
- ➔ Algorithms speed isn't measured in seconds.
- ➔ Algorithms time are measured in terms of growth of an algorithm (Big O notation).
- ➔ O (n!) is the slowest known complexity.

Take Home Problem

2.3-7 ★

Describe a $\Theta(n \lg n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.