Algorithm Analysis and Time Complexity

Algorithms Analysis

If you run the same program on a computer, cellphone, or even a smartwatch, will it take same time or different time?

Wouldn’t it be great if we can compare algorithms regardless of the hardware where we run them?

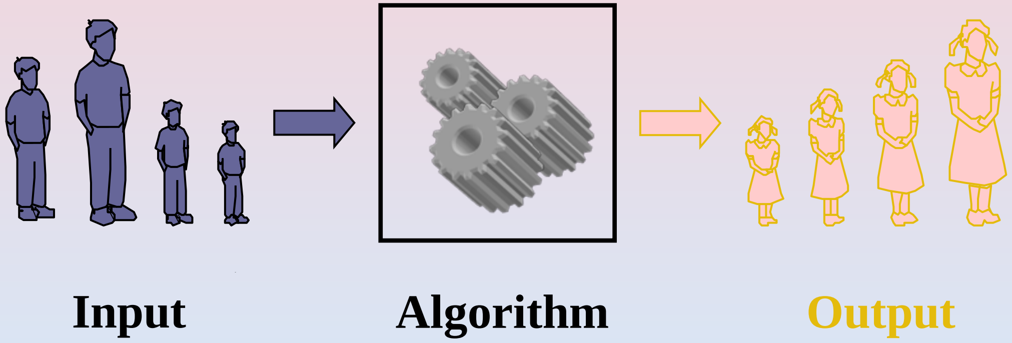
That’s what **time complexity** is for!

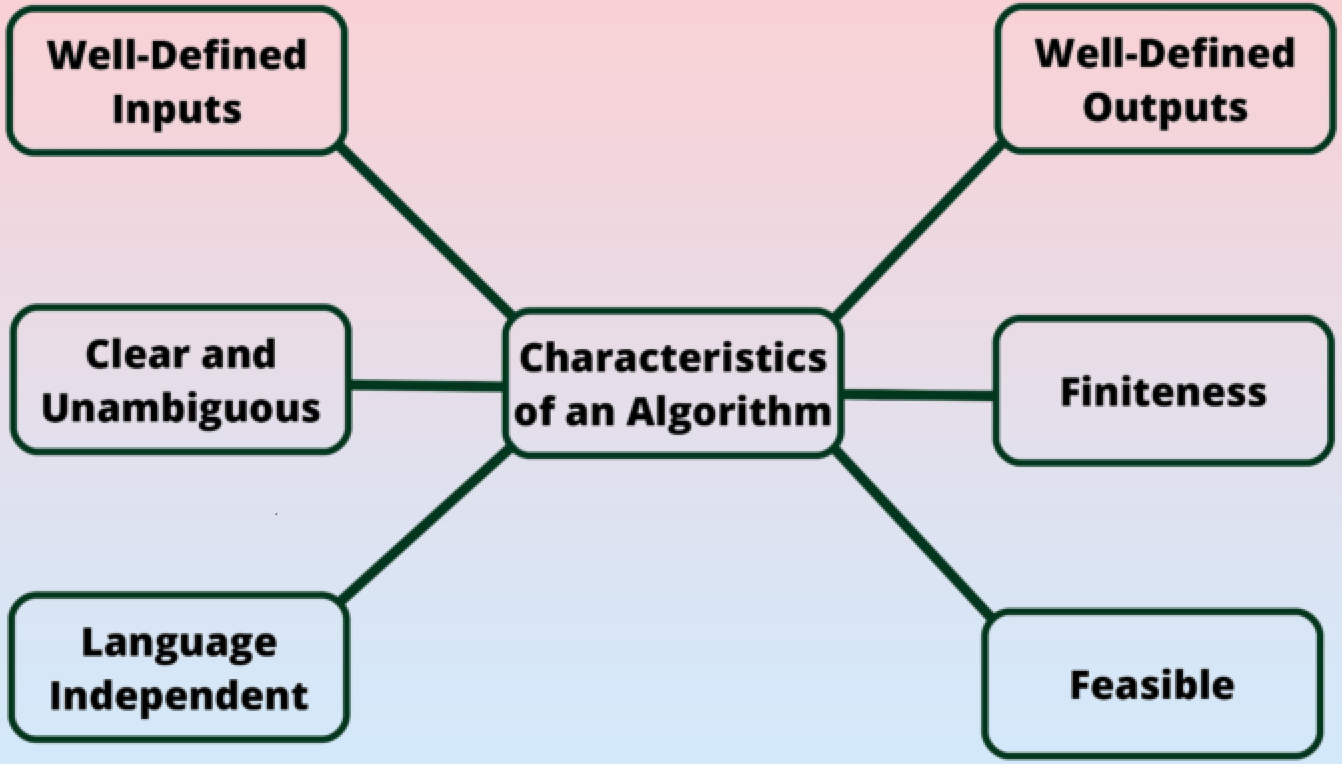
But, why stop with the running time?

We could also compare the memory "used" by different algorithms, and we call that **space complexity**.

What is an Algorithm?

- An Algorithm is a set of step by step instructions to be followed to solve a particular problem.
- **Characteristics of an Algorithm**





Write an algorithm to find the maximum value from N numbers

Algorithm to find the maximum value from N numbers

Step 1: Read N.

Step 2 : Let Counter = 1.

Step 3: Read a Number.

Step 4 : Maximum = Number.

Step 5: Read next Number.

Step 6 : Counter = Counter + 1.

Step 7 : If (Number > Maximum) then Maximum = Number.

Step 8 : If (Counter < = N) then go to step 5.

Step 9: Print Maximum.

Step 10: End.

Comparing Algorithms

- Not all algorithms are created equal.
- There are “good” and “bad” algorithms.
- The good ones are fast; the bad ones are slow.
- Slow algorithms cost more money to run.
- Inefficient algorithms could make some calculations impossible in our lifespan!
- Let’s say you want to compute the shortest path from Bombay to Surathkal.
- Slow algorithms can take hours or crash before finishing.
- On the other hand, a "good" algorithm might compute in a few seconds.
- Usually, algorithms time grows as the size of the input increases.
- For instance, calculating the shortest distance from your hostel room to NITK beach will take less time than other destination thousands of miles away.

Relationship between algorithm input size and time taken to complete

Input size →	**10**	**100**	**10k**	**100k**	**1M**
Finding if a number is odd	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
Sorting array with merge sort	< 1 sec.	< 1 sec.	< 1 sec.	few sec.	20 sec.
Sorting array with Selection Sort	< 1 sec.	< 1 sec.	2 minutes	3 hours	12 days
Finding all subsets	< 1 sec.	40,170 trillion years	> centillion years	∞	∞
Finding string permutations	4 sec.	> vigintillion years	> centillion years	∞	∞

Relationship between algorithm input size and time taken to complete

- As you can see in the table, most algorithms on the table are affected by the input size.
- But not all and not at the same rate.
- Finding out if a number is odd will take the same if it is 1 or 1 million.
- We say then that the growth rate is constant.
- Others grow very fast.
- Finding all the permutations on a string of length 10 takes a few seconds, while if the string has a size of 100, it won't even finish!

Calculating Time Complexity

- In computer science, time complexity describes the number of operations a program will execute given the size of the input n.

```
int   findMaximum ( int array[ ], int n)
{  int   maximum = array[0];
       for ( int  i=1; i<n; i++)
              if ( maximum < array[i])
                  maximum = array[i];
       return( maximum);
 }
```

1 Operation

1 Loop n-1 times

1 Operation

1 Operation

1 Operation

2(n-1) + 2

Assuming that each line of code is an operation, we get **2(n-1) + 2**

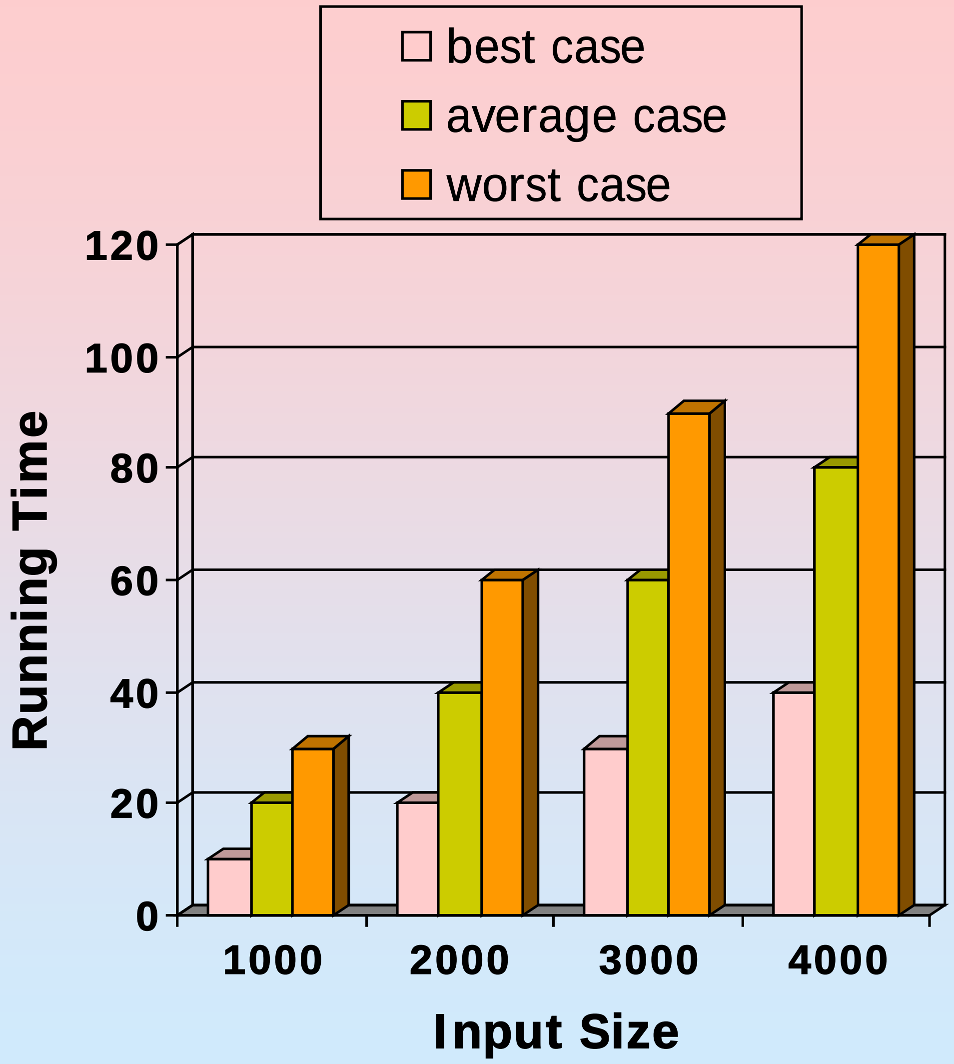
Calculating Time Complexity

- **For input size n=5, 2(n-1) + 2 = 10 operations.**
- **For input size n=8, 2(n-1) + 2 = 16 operations.**
- **This is not for every case. Line 5 executed only if line 4 condition is TRUE.**
- **So, we need the big picture and get rid of smaller terms to compare algorithms easily.**

- Asymptotic analysis describes the behavior of functions as their inputs approach to infinity.

Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average-case running time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



All algorithms have three scenarios:

- **Best-case scenario**: the most favorable input arrangement where the program will take the least amount of operations to complete.

  E.g., a sorted array is beneficial for some sorting algorithms.

- **Average-case scenario**: this is the most common case.

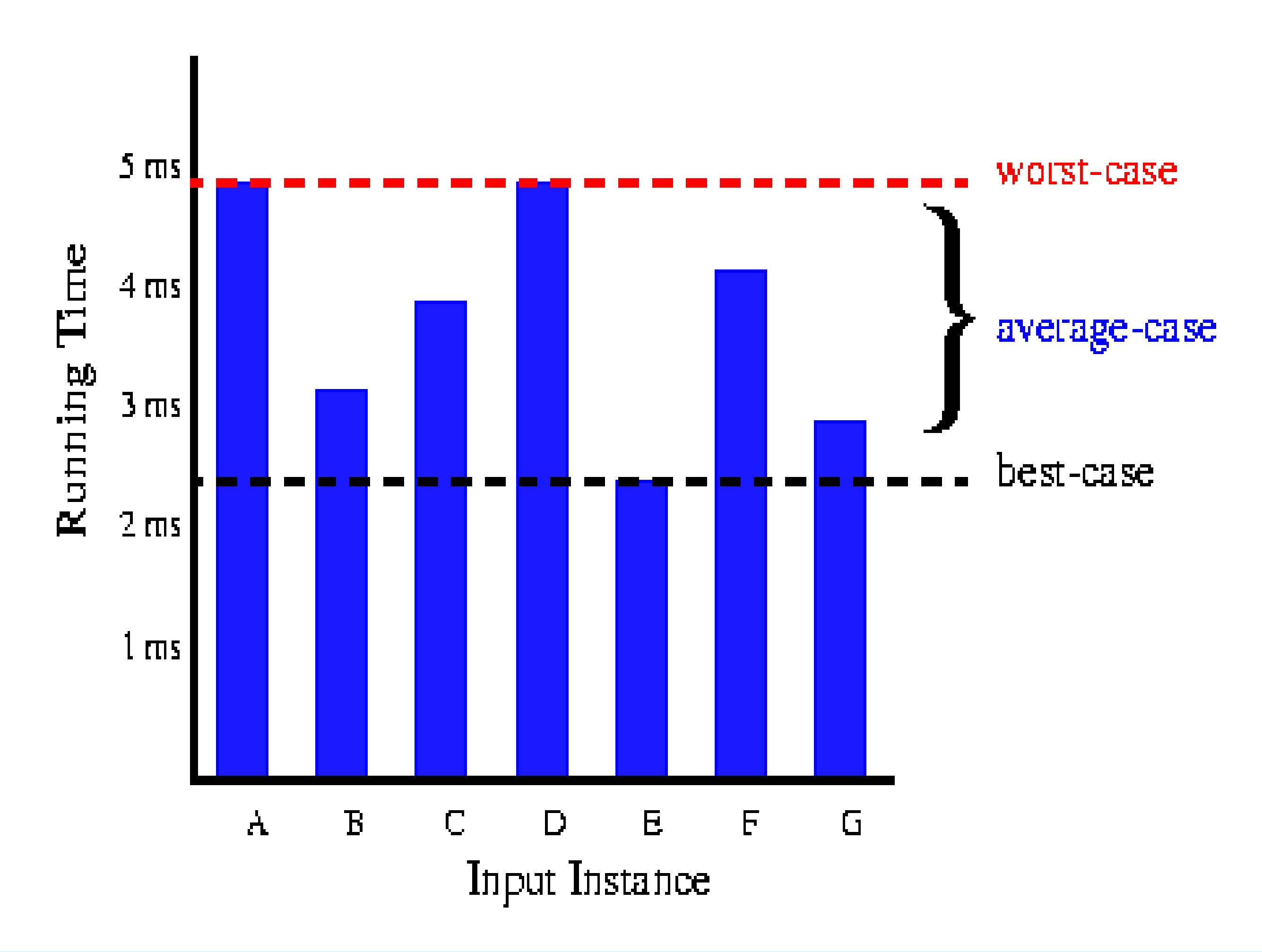
  E.g., array items in random order for a sorting algorithm.

- **Worst-case scenario**: the inputs are arranged in such a way that causes the program to take the longest to complete.

  E.g., array items in reversed order for some sorting algorithm will take the longest to run.

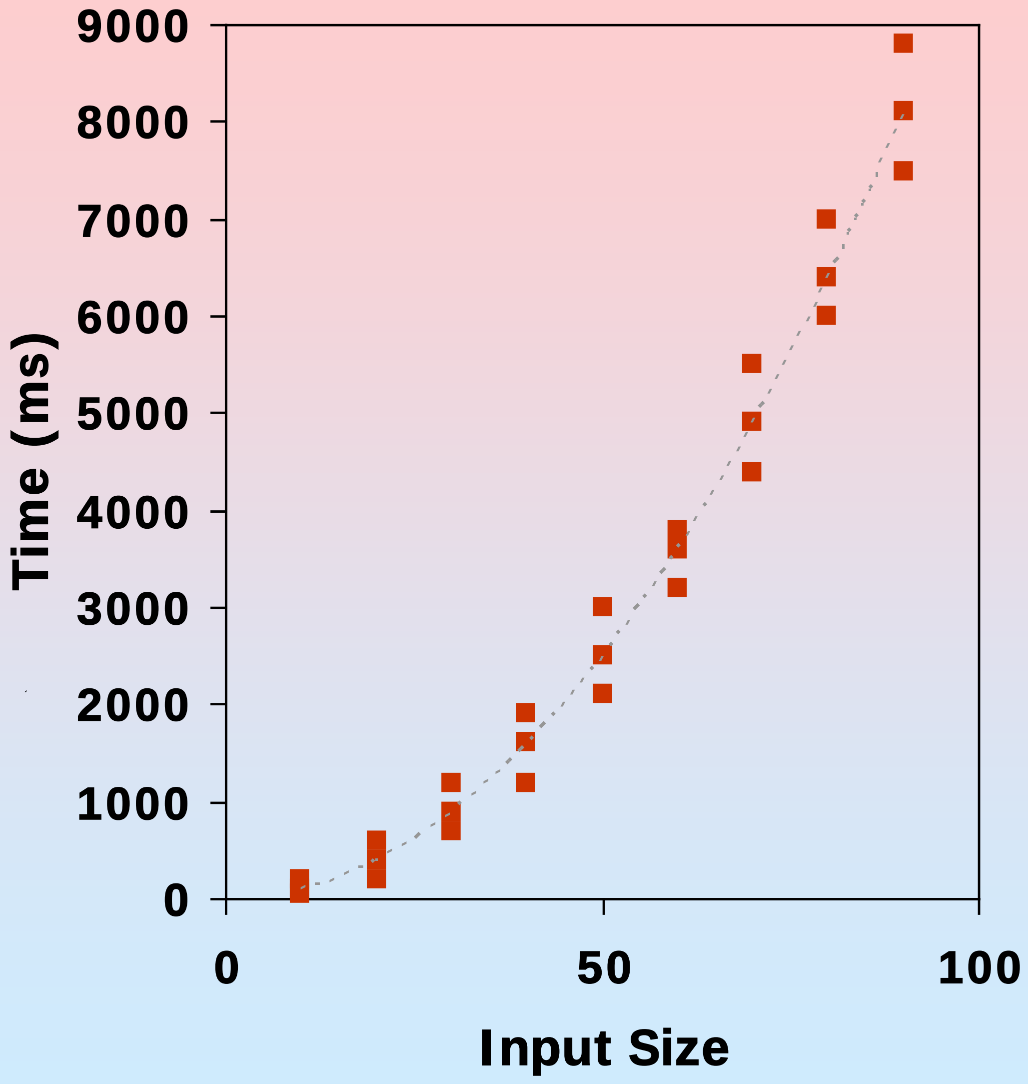
Average Case vs. Worst Case

- The **average case running time is harder** to analyze because you need to know the probability distribution of the input.
- In certain apps (air traffic control, weapon systems,etc.), knowing **the worst case time is important.**



Experimental Approach

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a wall clock to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult and often time-consuming
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
  - Restrictions

Theoretical Analysis

- **Uses a high-level description of the algorithm instead of an implementation**

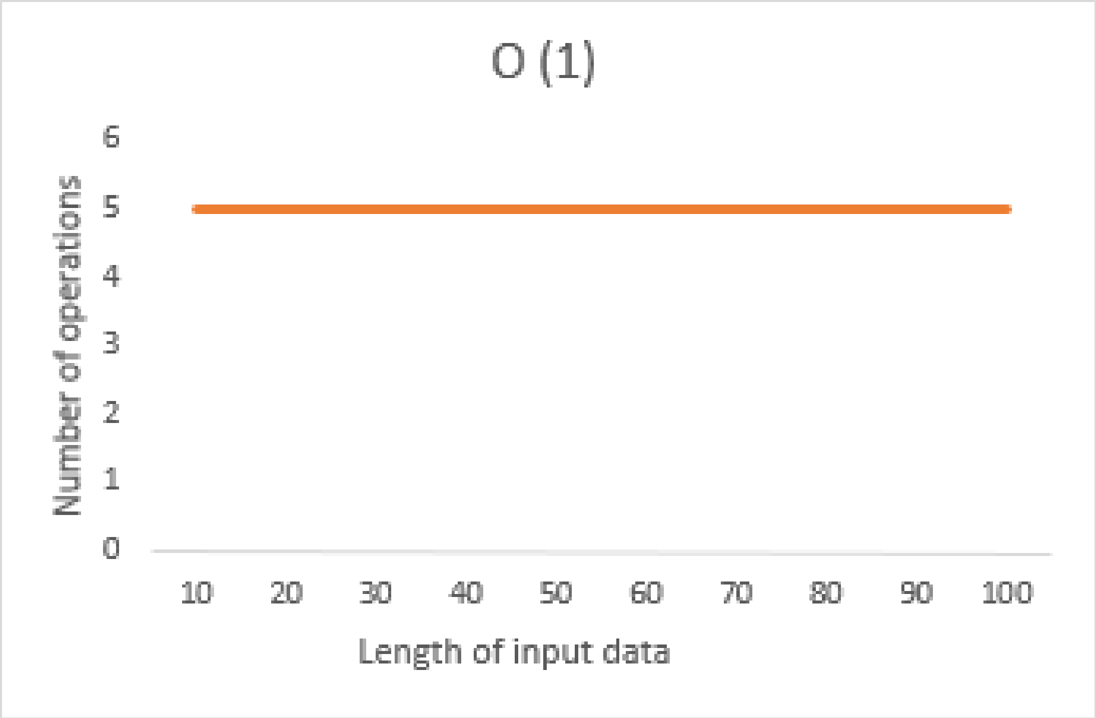
- **Characterizes running time as a function of the input size, *n*.**

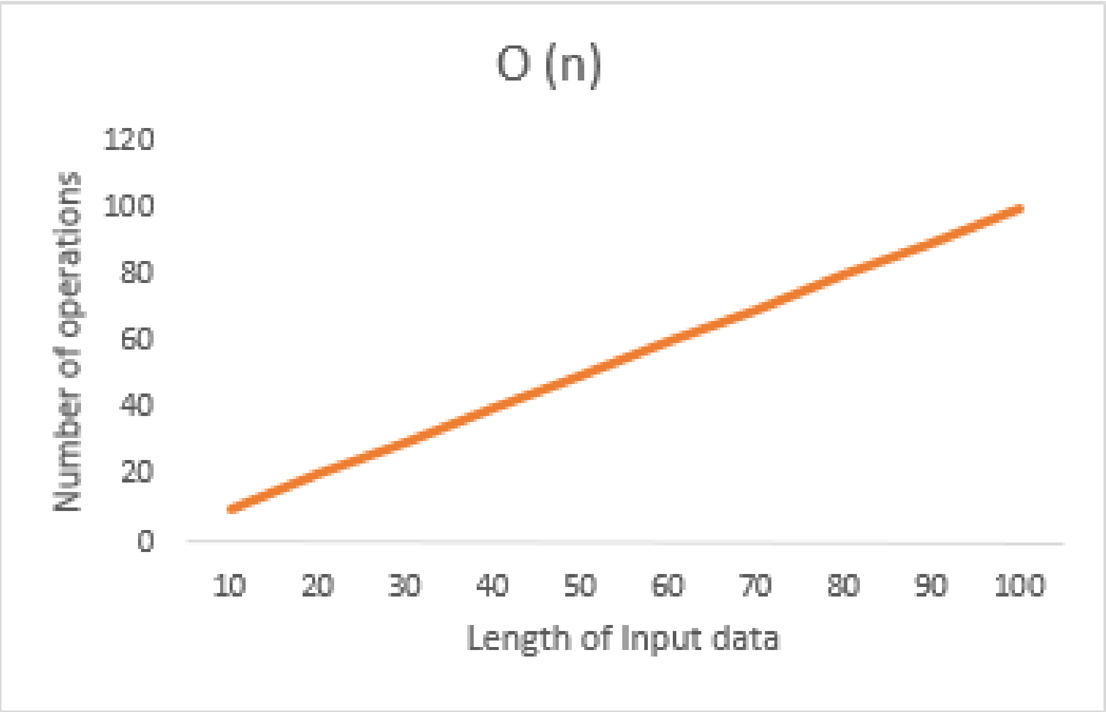
- **Takes into account all possible inputs**

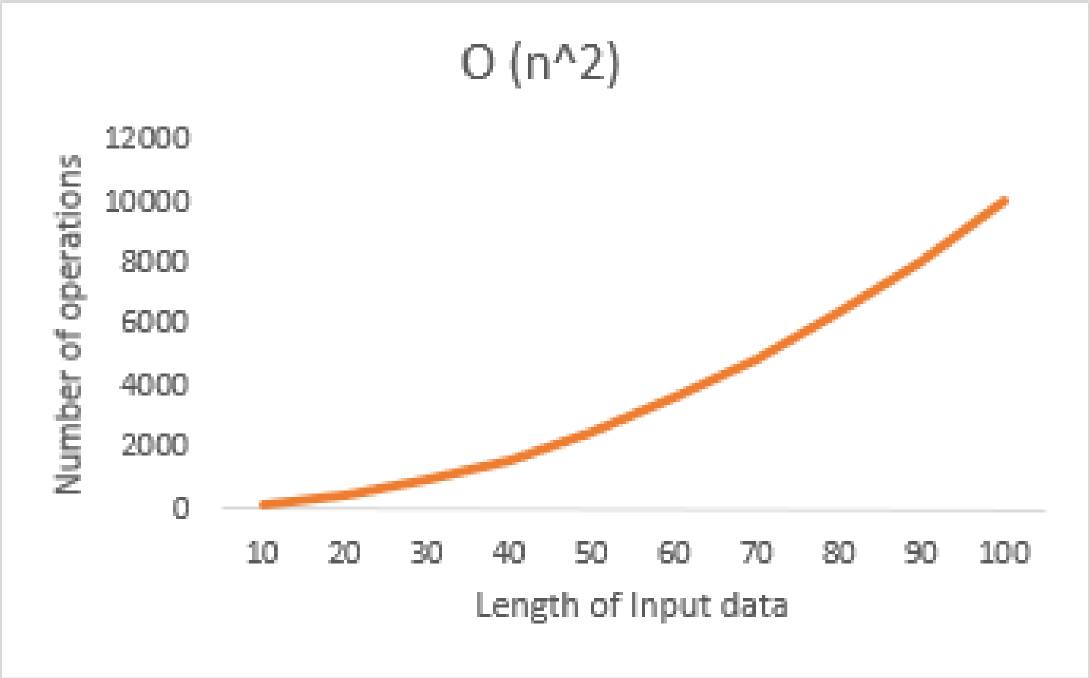
- **Allows us to evaluate the speed of an algorithm *independent of* the hardware/software environment**

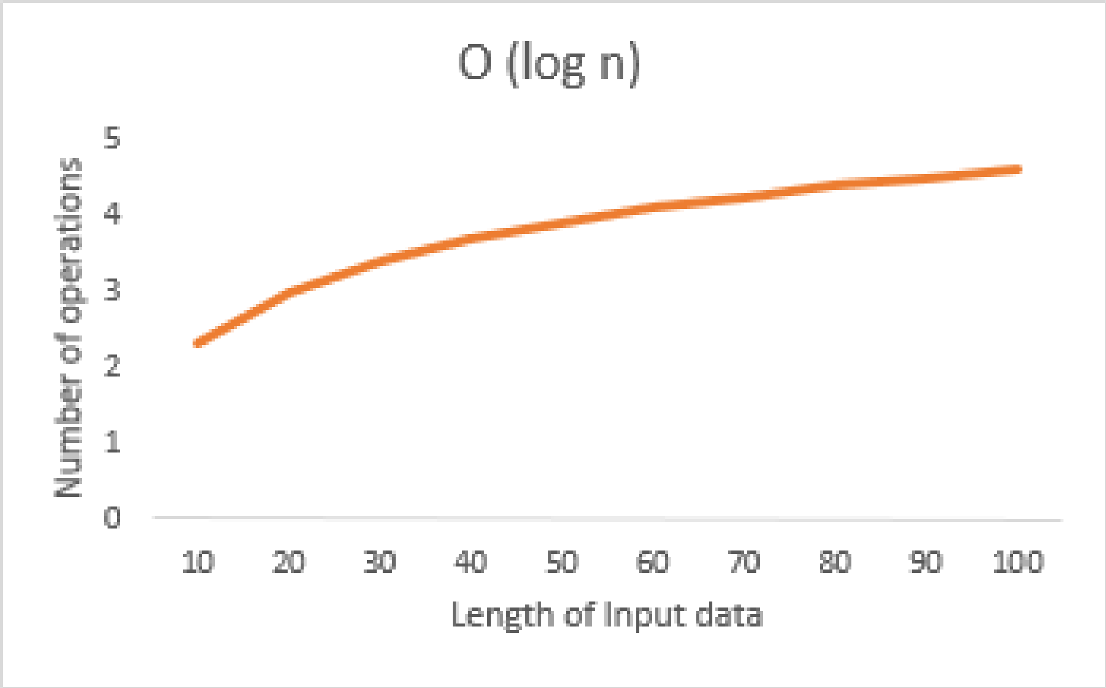
Simplifying Complexity with Asymptotic Analysis

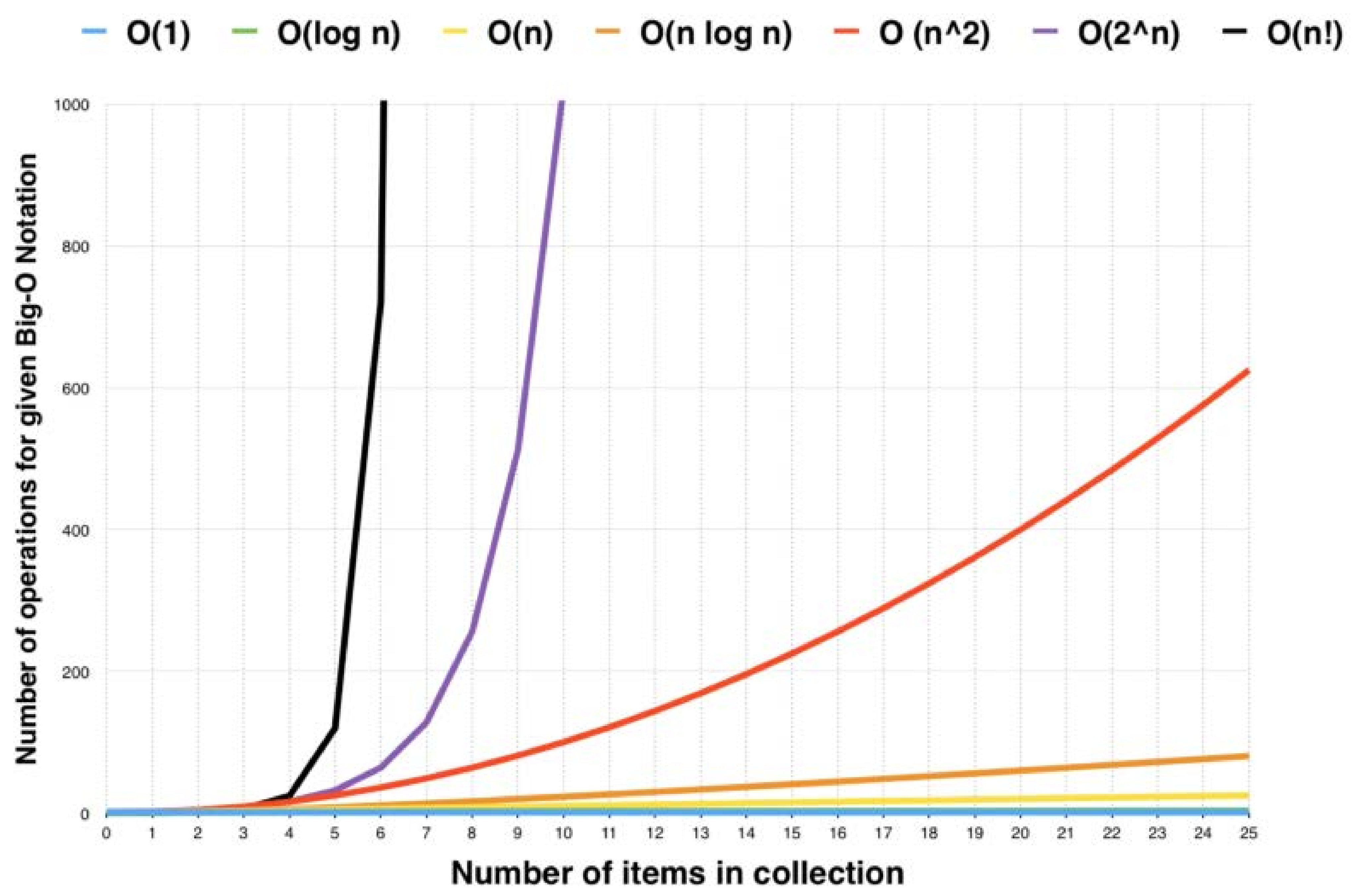
- **What is Big O Notation?**
- **Big O, where O refers to the order of a function in the worst-case scenario.**
- **Big O = Big Order (rate of growth) of a function.**
- **If you have a program that has a runtime of: $7n^3 + 3n^2 + 5$**
- **You can express it in Big O notation as $O(n^3)$. The other terms ($3n^2 + 5$) will become less significant as the input grows bigger.**
- **Big O notation only cares about the "biggest" terms in the time/space complexity.**


O (1)
Number of operations
6
5
4
3
2
1
0
10
20
30
40
50
60
70
80
90
100
Length of input data

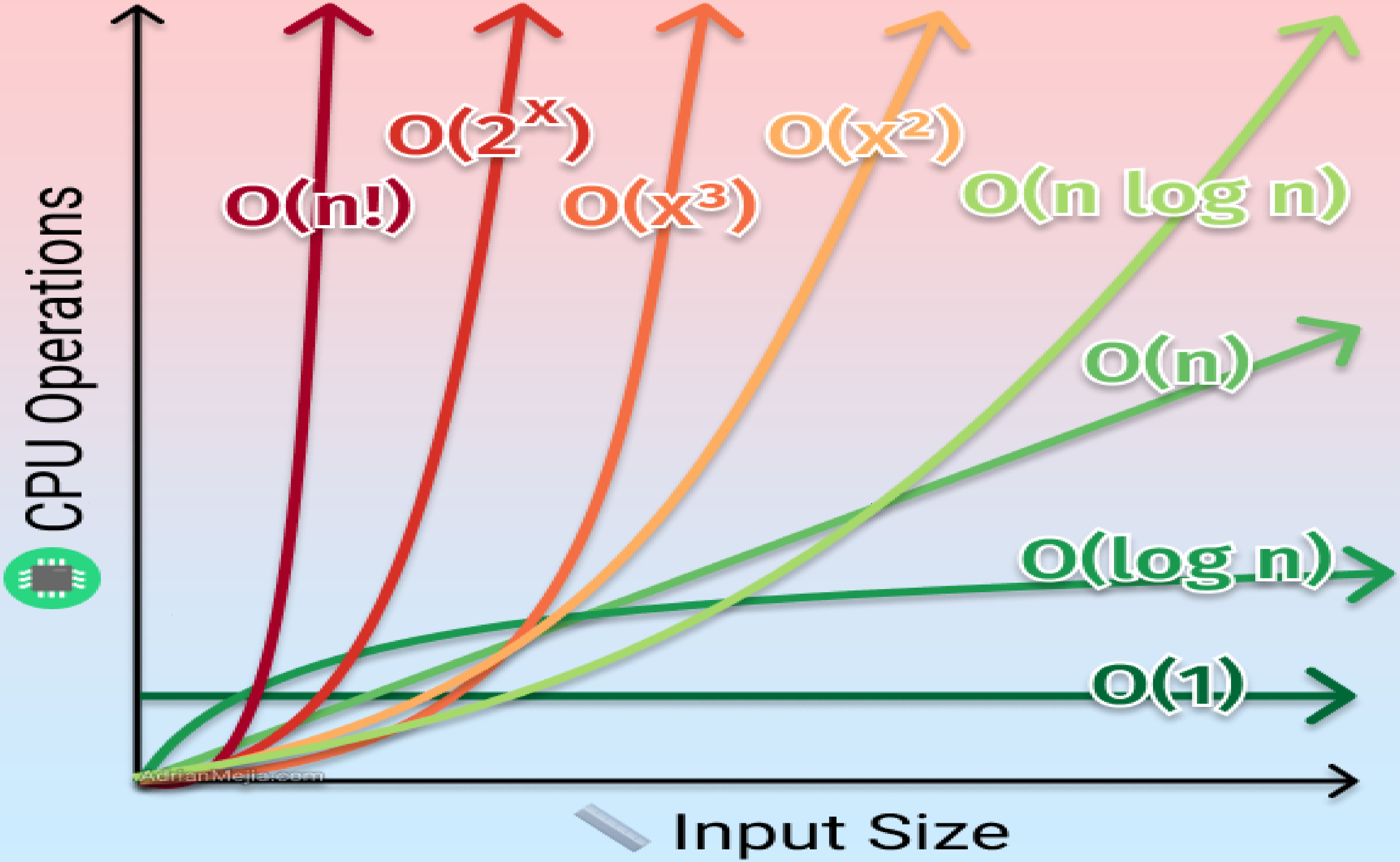

O (n)
Number of operations
120
100
80
60
40
20
0
10
20
30
40
50
60
70
80
90
100
Length of Input data


O (n^2)
Number of operations
12000
10000
8000
6000
4000
2000
0
10
20
30
40
50
60
70
80
90
100
Length of Input data


O (log n)
Number of operations
5
4
3
2
1
0
10
20
30
40
50
60
70
80
90
100
Length of Input data


O(1)
O(log n)
O(n)
O(n log n)
O (n^2)
O(2^n)
O(n!)
Number of operations for given Big-O Notation
1000
800
600
400
200
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Number of items in collection

Time Complexity



How long an algorithm takes to run based on their time complexity and input size

Input Size	O(1)	O(n)	O(n log n)	$O(n^2)$	$O(2^n)$	O(n!)
1	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
10	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	4 seconds
10k	< 1 sec.	< 1 sec.	< 1 sec.	2 minutes	∞	∞
100k	< 1 sec.	< 1 sec.	1 second	3 hours	∞	∞
1M	< 1 sec.	1 second	20 seconds	12 days	∞	∞

Note

This is just an illustration since, in different hardware, the times will be distinct. These times are under the assumption of running on 1 GHz CPU, and it can execute on average one instruction in 1 nanosecond (usually takes more time). Also, keep in mind that each line might be translated into dozens of CPU instructions depending on the programming language.

Space Complexity

- Space complexity is similar to time complexity.
- Instead of the count of operations executed, it will account for the amount of memory used additionally to the input.
- For calculating the space complexity, we keep track of the “variables” and memory used.
- In the findMaximum example, we create a variable called maximum, which only holds one value at a time. So, the space complexity is 1.
- On other algorithms, If we have to use an auxiliary array that holds the same number of elements as the input, then the space complexity would be n.

Pseudocode

- High-level description of an algorithm
- More structured than english prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find the max element of an array

Algorithm *arrayMax*(A, n)
  Input array A of n integers
  Output maximum element of A

  $currentMax \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $A[i] > currentMax$ **then**
      $currentMax \leftarrow A[i]$
  return $currentMax$

Pseudocode Details

- Control flow
  - if … then … [else …]
  - while … do …
  - repeat … until …
  - for … do …
  - Indentation replaces braces
- Method declaration

  Algorithm *method (arg [, arg…])*

  Input …

  Output …

- Method call

  var.method (arg [, arg…])

- Return value

  return *expression*

- Expressions

$\leftarrow$ Assignment (like $=$ in C, C++)

$=$ Equality testing (like $==$ in C, C++)

n^2 Superscripts and other mathematical formatting allowed