Time & Space Complexity

Time Complexity can be explained as a relation between input size and time taken by a computer to execute it.

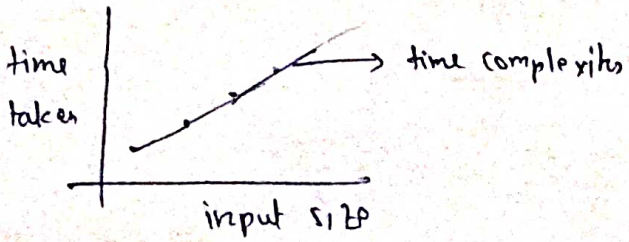
* Amout of Space or Time taken up by an algorithm/code as function of input size. is called complexity.

* Relationship btwn space & input size : Space Complexity
Relationship btwn time & input size : Time complexity
it isn't about actual execution time.

How to derive Time Complexity:

1. Experimental Analysis:
For the code that we have written, we use inbuilt classes to find the execution time.
Then we experiment with input size to find different execution times.
We plot all the values and find a function as time Complexity
→ here we actually run code multiple times

2. Therefical Analysis



$y = ax + b$
ignore constants
$t = ax + b$
$t = O(n)$
$TC = O(n)$ for linear Search

* Always analyse for the worst case.

Big O Notation

Big O notation denotes the upper bound of the upper limit of the time complexity function.

It means, the programme won't exeeced time represented in bigO notation in any case.

NOTE:- We always try to find worst case complexity

How to write

1. consider that we are given function F(n) were t is a function of n.

$$F(n) = an^2 + bn + c$$

Step1:- Ignore all constants

$$= n^2 + n + 1$$

Step2:- largest term

$$= n^2$$

$$\boxed{TC = O(n^2)}$$

There are 5-6 common patterns in TC for which we derive theoretically

Ex(1)

$$t(n) = an^3 + b\log n + c$$

$$n^3 + \log n + 1$$

n^3 largest

So $\boxed{O(n^3) - TC}$

Mathametically

$$f(n) = O(g(n))$$

$$\boxed{\lim_{n \to \infty} \frac{|F(n)|}{g(n)} < \infty}$$

$f(n) = n^2 + 2n + 2$

$g(n) = n^2$

time Complexity $O(n^2)$

$$\lim_{n \to \infty} \frac{n^2 + 2n + 2}{n^2} = \frac{n^2}{n^2} + \frac{2n}{n^2} + \frac{2}{n^2}$$

$$= \lim_{n \to \infty} 1 + \frac{2}{n} + \frac{2}{n^2}$$

$$= 1 + 0 + 0$$

$$LHS = 1$$

$$1 < \infty$$

Big Omega Notation

Represents lower bound or the best case of time complexity function.

This means code can't run in the time less than specified in the omega notation

——————— UB $O(n)$

——————— LB $\Omega(n)$

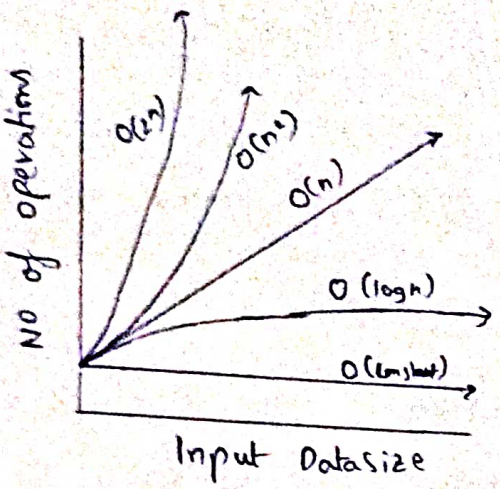
Big Theta (Θ) Notation

Average bound

In cases where LB = UB, we say it is Average Bound

Ex $\Omega(n^2)$ & $O(n^2)$ we represent it as $\Theta(n^2)$

Common Complexities



note:

exponential time complexity is good only for very small inputs.

It is the worst time complexity & is not advised to ~~using~~ use then interviews or coding Contests

Most of the recursion are in exponential time Complexity

Space Complexity

memory/space is of two types.

1. heap → has objects
2. Stack → has Function calls

The relationship btwn input size and amount of space taken to store it is called space complexity

A programme has two types of spaces

1. Input space :- Space occupied by permenant variable
2. Auxilary space :- Space occupied by temporary variables (We optimise this one)

Priority of Complexities

Time > Space

Some common Cases

1- Loops

K instructions are executed n times in a for loop

So it will be $O(n+k)$

as k is a constant, it is ignored

$= O(n)$

→ Linear time - space relation

2. Nested loops

Case 1 `i=0 to n; j=0 to n`

Time Complexity = outer loop (worst TC) × inner loop (worst TC)

$n \times n$

Nested loop TC = $O(n^2)$

Case 2

outer loop i:0 to n

inner loop j:0 to i

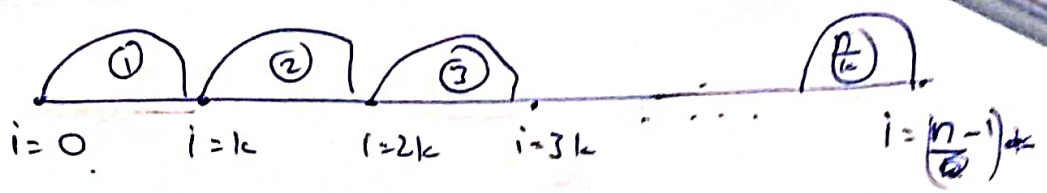
$= O(n^2)$ only

Case 3 : Find time Complexity for given code

Jumping for loop

```
// some k<n
For (int i=0; i<n; i=i+k){
    For (int j=j+1; j<=k; j++) {
        // some constant work is done in
        // the loop
    }
}
```

Outer loop



* no of jumps or iterations done by outer loop is $\frac{n}{k}$

Inner loop:

If we consider for the a single outer loop iteration the inner loop runs k times at maximum

$k, k-1, k-2 \ldots$

so time complexity = $\frac{n}{k}$ (worst outer loop) * (worst inner loop)

$= \left(\frac{n}{k}\right) \times k$

$= n$

$$\boxed{TC = O(n)}$$

Time Complexity of Sorting:

Bubble Sort:

Consider we are sorting an array to ascending order.

1. The worst case occurs when given array is in descending order

outer loop runs n times

inner loop runs n-1 times in worst case

So time complexity = $n(n-1) = (n^2 - n)k$.

$= O(n^2)$

Even for the best case both loops will run but swapping isn't done

So it is $O(n^2)$ only

Best case TC for optimized Bubble Sort

Here we introduce a boolean variable in the begenning of the code, which turns to true if the task is done.

So again we use an if condition to check the wether the value turns to true or not. If it turns is false then we assign break.

So here outer loop runs 1 time and the inner loop runs n times so,

time complexity will be O(n).

Code:

```
public static void modifiedBS (int arr[]){
    for (int i = 0; i<arr.length-1; i++) {
        boolean swapped = false;
        for (int j=0; j < arr.length-1-i; j++ ){
            if (arr[j] > arr[j+1]) {
                //swapping is done
                // Boolean variable is updated
                swapped = true;

            }
        }
        if (swapped == false ){
            break;
        }
    }
}
```

Time Complexity for Binary Search

Condition for worst case:

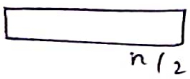
The element is not present in given array

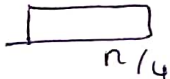
Condition For best case:

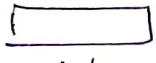
The element is present in middle index of given array.

Worst case:

1st step iteration 

2nd iteration 

3rd iteration 

4th iteration 

kth iteration $= \frac{n}{2^{k-1}}$ [size]

$$\frac{n}{2^{k-1}} = 1$$

$$n = 2^{k-1}$$

by ignoring Constant

$$n = 2^k$$

$$\boxed{k = \log_2(n)}$$

Time Complexity of worst case is $O(\log_2(n))$

Best Case

O(1). only once the loop is run