Time and space Complexity

Time complexity

Amount of time taken by an algorithm to run as a function of length of input

CPU time consumed by an algorithm
or
how much CPU works on that algorithm as a function of input

e.g
```
cin>>N;
for(int i=0; i<N; i++)
{
  ____
  ____
}
```
⇒ taking input from user N
loop is run by CPU till N times
So if we increase value of N
no of operations performed will be increased by CPU

Time taken or no of operation = f(N)
i.e $T \propto N$ ∴ T.C : O(N)

Amount of time
↳ actual Time (x)
↳ CPU operation

Need for Time & space complexity

- Resources are limited
- Measure algorithm to make efficient program

Space Complexity

Amount of space taken by the algorithm to run as a function of length of input
so as we increase length of input how much more space algorithm will take

e.g ①
```
int a=1;  // variable
int b[5]; // array
```
} amount of space = 24 bytes (x)
since T.C is a f(input)
S.C is also a f(input)

if there is no change in the value of a and b by increasing the no. of input ∴ S.C = O(1)
so it won't be considered in space complexity constant space

eg 2

```
int n; cin>>n;
int *b = new int[n];
// print array b
for (int i=0; i<n; i++)
{
  cout << b[i];
}
```

So here if we increase value of n size of array increases

n = 2; b[2]

n = 2000; b[2000]

∴ S.C = O(n)

Unit to Represent complexity

1. Big O : upper bound → algo's upper bound → maximum amount of time taken by an algorithm
2. Theta θ : Average case →
3. Omega Ω : Lower bound → algo's lower bound → minimum amount of time taken by an algorithm

for eg

Linear search;

1	2	3	4	5	6

To find 1 it is in first position ∴ O(1) — Best case ⇒ minimum time to find the element so it represents the lower bound

6 it is last position in array ∴ O(n) — Worst case ⇒ maximum time to find the element

So Big O is better representation as it represents worst case complexity as it will help to understand complexity of algorithm and how it perform in its worst case

~~In worst case~~

Big O: Complexities

1. constant time : O(1) doesn't depend upon value of N or user input
2. Linear time : O(n) directly proportional to value of N or user input

```
for (i=0; i<N; i++)
{
  cond^n --;
}
```

3. Logarithmic time : O(log N)
4. Quadratic time : O(N²) ∝ Square of value of N eg

Worst complexity i → N, j → N

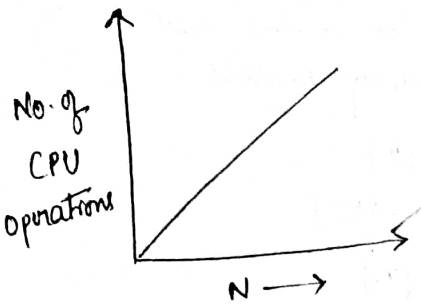
Nested for loop 1<i<N, 1<j<N

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        --
    }
}
```

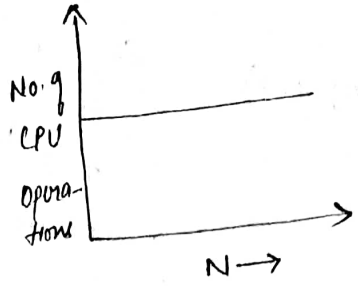
5. Cubic time : O(N³) → eg 3 nested for loops one inside another

Worst complexity i → N, j → N, k → N

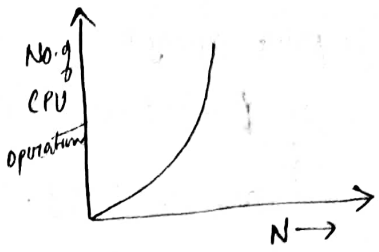
O(n)



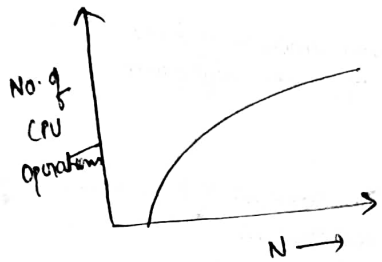
O(1)



O(n²)



O(logn)



To find complexity

① $f(n) = 2n^2 + 3n \Rightarrow O(2n^2)$

we always check the upper bound so here upper bound is n^2 $\Rightarrow O(n^2)$, we can ignore the constants

② $f(n) = 4n^4 + 3n^3 \Rightarrow O(n^4)$

③ $f(n) = n^2 + \log n \Rightarrow O(n^2)$

④ $f(n) = 200 \Rightarrow O(1)$

⑤ $f(n) = N/4 \Rightarrow O(N/4) = O(N)$

Increasing order of complexity

$O(1), O(\log N), O(\sqrt{N}), O(N), O(n\log n), O(n^2), O(n^3), O(2^n), O(n!), O(n^n)$

$O(1)$ → Least complex

$O(n^n)$ → Most complex

$O(\log_2(n)) \rightarrow$ It is of base 2

sorted array

1	2	3	4	5	6	7

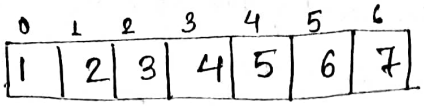
① Linear search → $O(n)$

② Binary search → $O(\log n)$

Binary search :-

prerequisite condition - sorted array

Here we start search from middle of an array by dividing array into 2 equal halves

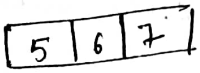


To find 6, step 1 $\frac{0+6}{2} = 3$

step 2 Data [3] = 4 , $4 < 6$

so search in right half

step 3 $\frac{4+6}{2} = 5$, Data [5] = 6



$6 = 6$

∴ element found

So at starting search space is → N

then → N/2

→ N/4

⋮

→ 1

K → no of times to divide the array i.e no of operations

$\left(\frac{N}{2^K}\right) = 1$

$\Rightarrow N = 2^K$

applying log on both sides

$\log_2 N = \log_2 2^K$ [$\because \log_a a = 1$]

$K = \log_2 N$

Examples

①
```
for (int i=0; i<n; i++)
{
  cout << "Hi \n";
}
```
→ O(N)

```
for (int i=0; i<m; i++)
{
  cout << "Hi2 \n";
}
```
→ O(M)

Note
If there are no nested complexities then complexities get added

Total complexity

∴ O(N) + O(M)

= O(N+M)

```
for (int i=0; i<n; i++){
    for (int i=0; i<n; i++){
        cout << "Hi1 \n";
    }
}
```
→ $O(n^2)$

```
for (int i=0; i<n; i++){
    cout << "Hi2 \n";
}
```
→ O(n)

Total complexity

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

but we always take upper bound

∴ T.C = $O(n^2)$

③
```
for (int i=0; i<n; i++) {        ⟶ (N)
    for (int j=n; j>i; j--) {    ⟶ at i=0
        cout << "Hi\n";              no. of operation → N
    }                                upper bound of inner loop → (N)
}                                    it is nested loop
                                     ∴ Complexity = O(N²)
```

examples of space complexity

① `int a = 5;` → O(1)

② `arr[5];` → O(1)

③ `int *a = new int[N]` → O(N)

④ `int *b = new int[N²]` → O(N²)