* Time complexity:

The amount of time taken by an algorithm to run a function depending upon length of input.

for ex:

```
for (int i=0; i<n; i++){
    //code. cout<<i;
}
```

The CPU will run the for loop for n no. of times to print i. ∴ The time complexity of above code will be O(n): means to worst case the CPU will run this loop for n no. of times.

⇒ The above code if we write in this way.

```
for (int i=0; i<n; i++){
    int k=n;
    while(k>=0){
        cout<<i+1;
}
```

The T.C of this will be $O(n^2)$

This is not an optmized code we as use of while loop is useless. we always try to write optimized code.

* why to study Time and space complexity:

1. To write an optimal algorithm for any problem which leads to less utilization of CPU.
2. To compare two algo i.e algoA & algoB which is more efficient to use.
3. As Resources are limited.
4. Interviewer always ask for T.C

Algo A	Algo B	
use CPU High Processing	use CPU less Processing.	∴ AlgoB is faster than AlgoA as it use less CPU usage.

* Space complexity: The amount of space taken by an algorithm to run a function depending upon the length of inputs.

ex:
```
int a = 5 // variable
int b[5] // array.
```
→ 4byte, 20byte = 24byte

if we create a for loop. i.e
```
for(i=0; i<n; i++){
    cout<<i;
}
```
→ so there would be no change in S.C as inside for loop running for n no. of times doesn't create extra space to. or effect the space for variable & array so the space complexity of above code in O(1): constant space.

*Note: Until unless any variable or array etc don't change with the inputs i.e length of inputs the space complexity would remain constant i.e O(1)

* Example of change in space complexity:

consider an array with size n

∴ int n;
cin >> n;
int *b = new int [n];

ex: n=2;
→ b[2] → array of 2 size.

n= 1000
b[1000] array of 1000 size.

```
for(int i=0; i<n; i++){
    cout<<b[i]
}
}
```

∴ The T.C will be O(n) and S.C will be O(n)
↓
As the space varies with change in the value of n i.e inputs of n.

Unit to Represent complexity:

1. Big O: Upper Bound.
   ↓
   The maximum amount of time taken by an algorithm to run a function. i.e Hi the worst case the algorithm will be run.

2. Omega Ω: Lower Bound
   ↓
   The minimum amount of time taken by an algorithm to run a function to get an resulted requirement.

3. Average case: Theta Θ:
   The average amount of time taken by an algorithm to run a function

Let understand it with a example:

Consider an array:

1	2	3	4	5	6

suppose we need to find 1 we will use linear search approach as 1 is present on 0th index. so the algorithm will run for 1 time. so the minimum amount time taken will be Ω(1)
↓
lower bound

Now if we need to find 6 or to L.S we have to look for every element i.e from 1 to 6 i.e 0th-5th index of an array. so max amount i.e worse to worse the algo can take time to run is O(n)
↓
upper bound.
(to check every iteration)

we always look for worst case i.e upper bound.

* Big O : Complexities

1. Constant time : $O(1)$ → int a=5; no change in time
2. Linear Time : $O(n)$ → for single for loop running n no. of times
3. Logarithmic time : $O(\log n)$ → If length of input i.e n divides after every execution i.e /2
4. Quadratic Time : $O(n^2)$ → nested loop for(<n) {
5. Cubic Time : $O(n^3)$

for (<n) {
for (<n) {
}
}
$O(n^2)$

also nested loop

```
for (<n) {
    for (<n) {
        for (<n) {
        }
    }
}
```

→ 3 for nested loops

$O(n^3)$

* Consider

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

It is not nested loop for nested loop multiply else add the complexity

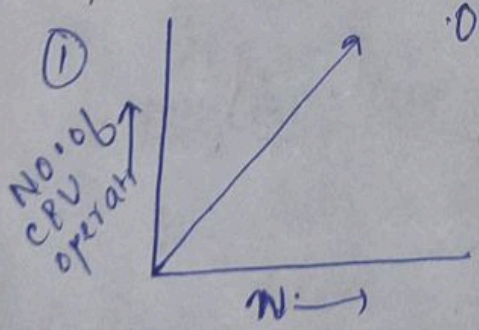
$O(n) + O(n) = O(2n)$

$O(n)$ + $O(n)$ → here 2 is constant

∴ $O(2n)$ can be considered as $O(n)$ T.C

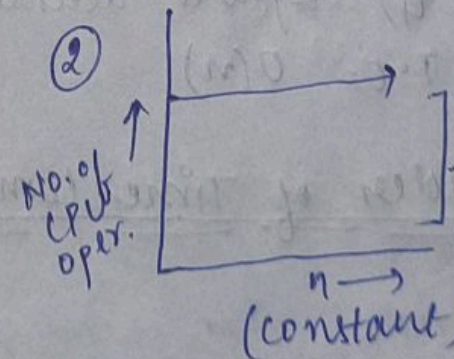
* Graphs :

① $O(n)$



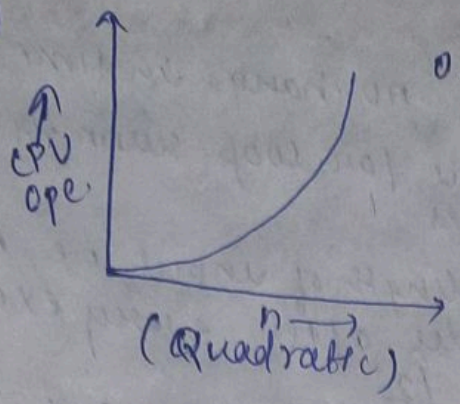
(linear)

② $O(1)$



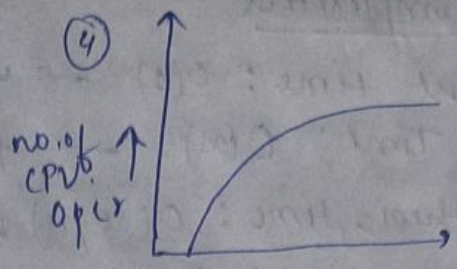
(constant)

③



(Quadratic)

④



(Lograthmic)

* Consider some examples:

① $f(n) = 2n^2 + 3n$

It will always look for upper Bound.

$O(2n^2) + O(3n)$

Ignore 2 & 3

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

as n^2 will be max or worse time the algo will run.

→ T.C is $O(n^2)$

② $4n^4 + 3n^3 = O(4n^4) + O(3n^3)$

Ignore 4 & 3

$O(n^4) + O(n^3)$

upper Bound is n^4

T.C $O(n^4)$

③ $n^2 + \log n = O(n^2) + O(\log n)$

= upper Bound is n^2

T.C = $O(n^2)$

④ $200 = O(200) = O(1)$ → constant T.C

T.C = $O(1)$

⑤ $f(\frac{n}{4}) = O(\frac{n}{4})$ ignore deride by 4

T.C = $O(n)$

* Increasing order of Time complexity:

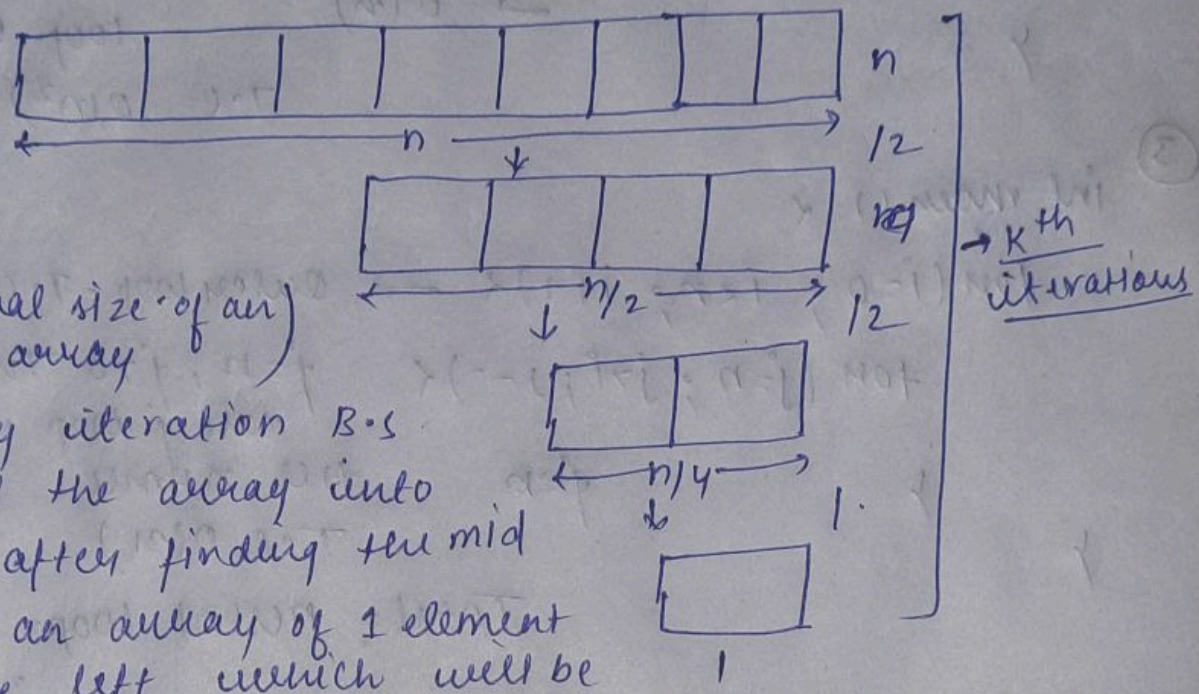
$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(2^n)$

$O(n!) > O(n^n) > O(2^n)$

↓ least complex

↓ Most complex.

① $O(\log n)$: consider an sorted array.



n (original size of an array)

for every iteration B.S divided the array into half after finding the mid

at last an array of 1 element will be left which will be required one in B.S

$\therefore \frac{n}{2^k} = 1$ K = Kth iteration i.e K times dive

$n = 2^k$

$\log n = K \quad \therefore O(K) = O(\log n)$ = T.C for B.S.

* Examples:

① int main() {

```
for(i=0; i<n; i++){
}
```

→ T.C of this loop = O(n)

```
for(i=0; i<m; i++){
}
```

→ T.C of this loop = O(m)

As it is not nested loop

$O(n) + O(m) = O(n+m)$

(2) int main() {

for(i=0; i<n; i++){ ⟶ O(n) = T.C for outer loop

for (i=0; i<n; i++){ ⟶ O(n) = T.C for inner loop.

} nested loop = O(n) * O(n)
= O(n²)

}

for (i=0; i<n; i++){ ⟶ O(n)

}

T.C = O(n²) + O(n) ≈ loop upper Bound

T.C = O(n²)

(3)

int main() {

for(i=0; i<n; i++){ ⟶ Outer loop = T.C = O(n)

for (j=n; j>i; j--){ j=n, j>0 The worse case j loop will run till n time

}

}

T.C = O(n)

Total = nested loop = O(n) * O(n)
= O(n²)

* Calculate space complexity:

(1) int a=5; No effect with change in length of input

S.C = O(1)

(2) array[5] = S.C = O(1) → No effect with size.

(3) int *a = new int[N]

change in space with (↑) in length

SC = O(n)

(4) int *b = new int[n²]

SC = O(n²)