Time complexity & space complexity

Not the actual time but amount of time taken as function of input size (n)

- this time differs machine-to-machine
- depends of operations

For Example:-

Linear search in array

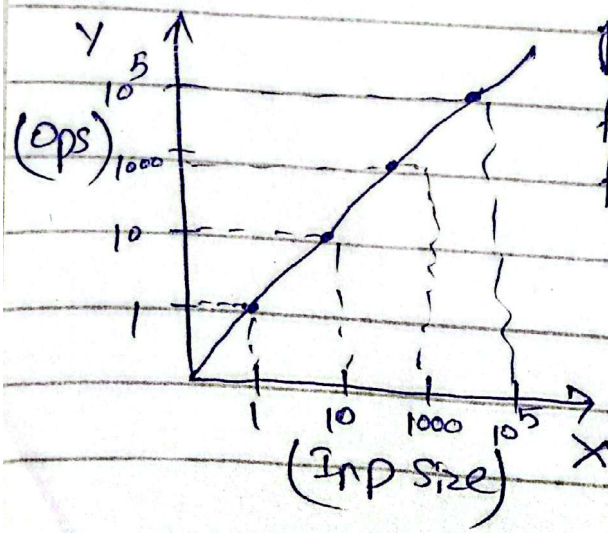

```
for(i=0; i<n; i++){
  if(target == A[i]){
    return i}
return -1
}
```

$n=1$	$op=1$
$n=2$	$op=2$
$n=3$	$op=3$
$n=4$	$op=4$
$n=1000$	$op=1000$
$n=10^5$	$op=10^5$

As the input size increases operations also increases

So, the time complexity

Graph Plotting



$y = x$

$f(x) = x$

$f(n) = n$

$O(n)$ — Linear Time complexity

Big O

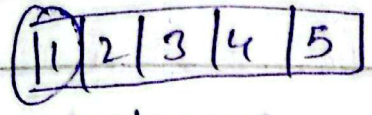
Time Complexity ↓ gives the behaviour not actual time.

Time complexity is used to prioritize the worst case scenarios.

→ Big O Notation
↳ Symbol

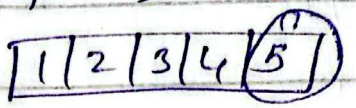
O(n) → Function of time complexity
↳ gives the worst case (upper bound)

for i=1



best case

for i=5



worst case.

→ O(n) operations

Calculating the Time complexity for complex functions

→ $f(n) = 4n^2 + 3n + 5$

$n^2 + n + 1$

n^2

$O(n^2)$

① Ignore constants
② Largest from

→ $10 + 5n^2 + \sqrt{n}$

$n^2 + n^{1/2}$

$O(n^2)$

Other Notations

Worst case	Avg. case	Best case
O	θ (theta)	Ω (omega)
Upper Bound		Lower Bound

Space complexity

Amount of space taken by an algorithm as function of input size (n).
Not the actual space.

Code → Input (vec, arr, string) — we perform functions like L.S
→ auxilary space → Extra space

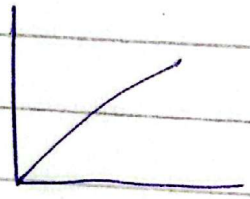
Now suppose you have array and want to replicate the values of this array into the inner memory

| 1 | 2 | 3 | 4 | 5 | — (n) arr input

| 1 | 4 | 9 | 16 | 25 | — (n) extra space will also increase

Graph will also be the same

Notation
$O(n)$



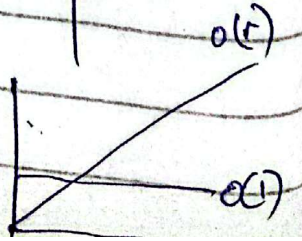
Suppose in this case
inp arr[]
Sum — will remain constant

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

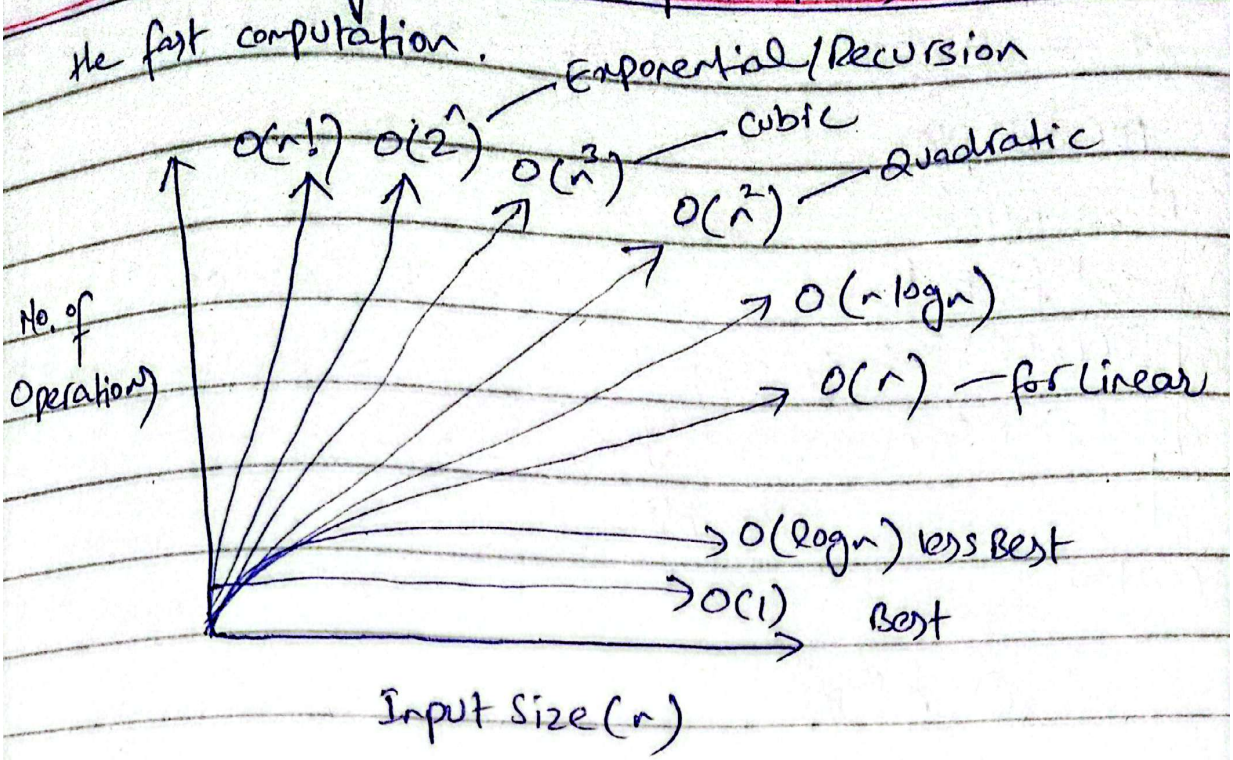
size of array n	auxilary	
$n=10$	constant	$O(1)$
$n=100$		
$n=10^5$		
$n=10^6$		

$O(1) \rightarrow O(1)$

In graph ←



Time complexity is more important as we need the fast computation.



Common Time Complexities

① — O(1) → Constant Time Complexity

Sum of Numbers upto n

```
int n
cin >> n
int ans = n * (n+1) / 2
```

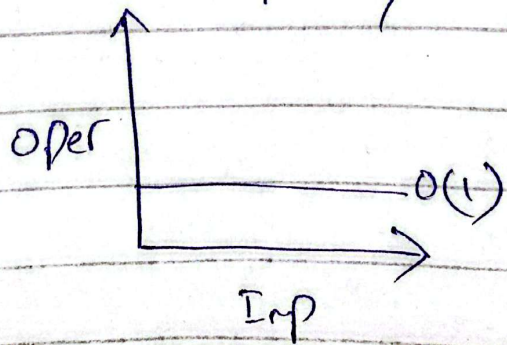
} will remain constant for any type of value

So

O(1) OR

getting the first element of array.

getting the largest element of array.



(2) — O(n) Linear Time Complexity

For example

N factorial

```
int fact=1
for(int i=1; i<=n; i++){
    fact *= i;
}
```

constant | variable

→ Kadane's Algo

→ Moore's Voting method

$O(n * k)$

$O(n)$

(3) — $O(n^2)$ & $O(n^3)$

Used in Bubble, Insertion, Selection sortings

Bubble Sort

```
for(int i=0; i<n-1; i++){          ~(variable)
    for(j=0; j<n-i-1; j++){        ~(variable)
        if(arr[j] > arr[j+1]){
            swap(arr[j], arr[j+1]);
        }
    }
}
```

constant

i=0 j=0,1,2

1 j=0,1

2 j=0

$3*k + 2*k + 1*k$

$(n-1)k + (n-2)k + (n-3)k + 2k + k$

(no. of operations)

$k[(n-1)+(n-2)+(n-3)] -$

$k * \left[\frac{n*(n-1)}{2}\right]$

$\frac{kn^2}{2} - \frac{k}{2} \approx O(n^2)$

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

$O(n^2)$ for pattern as well.

④ $O(n^3)$

```
for( ){
  for( ){
    for( ){
    }}}
```

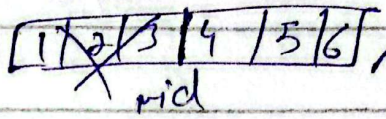
Examples

Possible 'Sub-array'

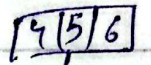
⑤ $O(\log n)$

Binary search → BST

sorted for=5



$n/2$



$\frac{n}{4}$

$n/8$

$\vdots$

(1)

$\frac{n}{2^2}$

Time Complexity

```
int s = 0, e = n-1
while(s <= e){
  int mid = s + (e-s)/2;
  if(arr[mid] < target){
    s = mid+1 }
  else if(arr[mid] > target){
    e = mid-1 }
  else{
    return mid;
    }
  }
```

⑥ $O(n \log n)$

Sortings (merge, Quick sort (Aug), Greedy Algos)

As we know

low in graph ← $O(\log n)$ ← better — $O(n)$ ($\times n$)

low in graph ← $O(n \log n)$ ← better — $O(n^2)$

⑦ $O(2^n)$ / Exponential

Recursion (Brute force)

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

⑧ $O(n!)$ not so common (worst)

- n-queen
- knights
- String's permutation

Practical Example

Prime Number

```
for(i=2; i*i<=n; i++){
    if(n%i==0){
        cout<<"Non Prime";
        break;
    }
}
```

$O(\sqrt{n})$ Better

$i=2$ $i^2 \le n$

$i^2 = n$ (worst)

$i = \sqrt{n}$ (worst) ✓ well

$i=2$ to $\sqrt{n}$

$i=0$ to $\sqrt{n}$ — will be same

① $O(\sqrt{n})$ OR $O(n)$

② $O(\sqrt{n})$ OR $O(\log n)$

Selection Sort

```
for(int i=0; i<n-1; i++){
    int minInd=i;
    for(int j=i+1; j<n; j++){
        if(arr[j]<arr[minInd]){
            minInd=j;
        }
    }
    swap(arr[i], arr[minInd]);
}
```

outer loop: n; inner loop: n (for worst case); K; $O(n^2)$

Recursion

Time Complexity Methods

```
int factorial(int n){
    if(n==0){
        return 1;
    }
    return n * factorial(n-1);
}
```

① Recurrence Relation

② TC = total No. of calls * work done in each call

↓ Most preffered

Recursion Tree is Used

$O(n * k)$ — constant work being done in each call itself involves calls as well

$O(n)$

n=4

4 × (f(4) ↖ 6)

n → f(3) * 3 — value calls n → n-1

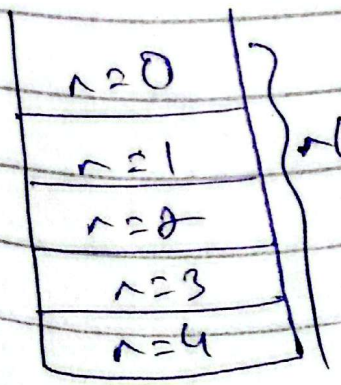
n * f(2) — n ⇒ no calls

f(1) * 1

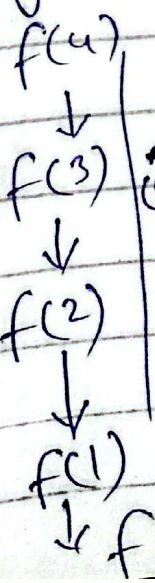
f(0)

Space Complexity

In recursion, stacks fills itself with recursive variables again and again



Rec call Stack



S.C = Depth of Call Stack * memory in each call

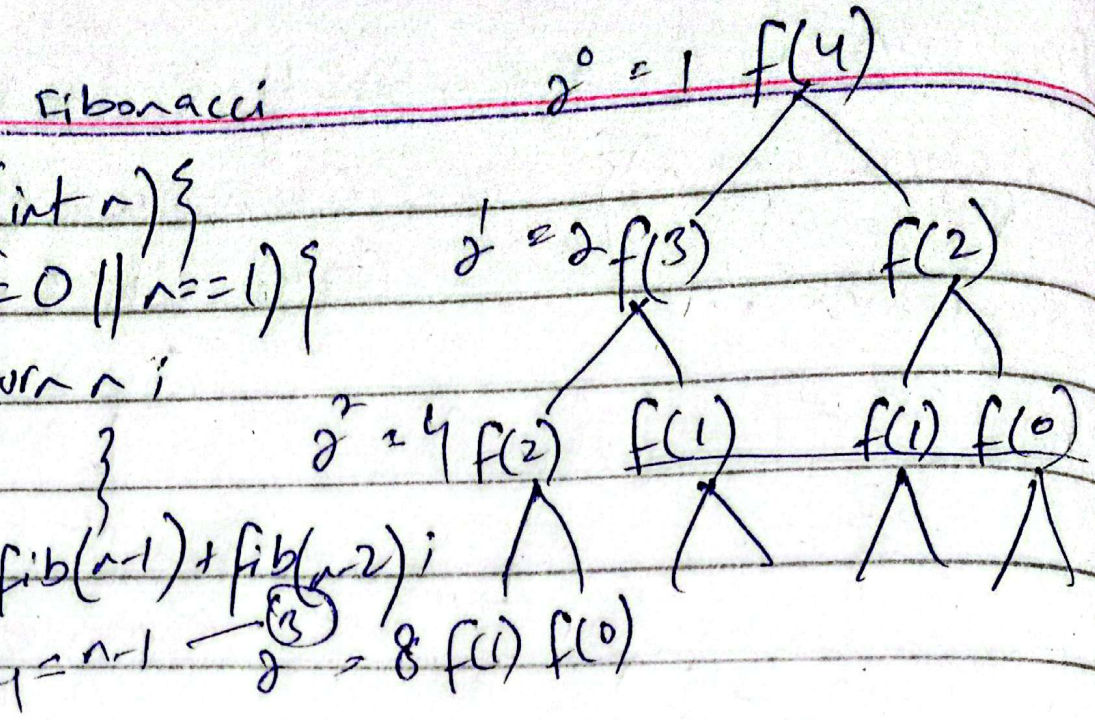
S.C = Height of call stack * Memory in ~~each~~ call

S.C = n * 1

S.C = O(n)

Recursion Fibonacci

```
int fib(int n){
 if(n==0||n==1){
   return n;
 }
return fib(n-1)+fib(n-2);
}
```



T.C = total calls * w.D in each call

$[2^0 + 2^1 + 2^2 + 2^3 + 2^{n-1}]$

$2^0 * \left(\frac{2^n - 1}{2 - 1}\right) \rightarrow \frac{2^n - 1}{1}$

$\times$

2^{n-1}

$(2^{n-1}) * K$

$O(2^n)$

$O(2^n)$

→ if the branches are not balanced then the

T.C $O((1.618)^n)$

Golden Ratio

Space Complexity =

Depth of RT * Memory

n * 1

S.C = O(n)

Merge Sort

first of all T.C & S.C of merge function is O(n) and O(n)

then in merge sort, it will be:

T.C → O(logn * O(n)) | O(n * logn)

S.C → O(logn + n) | S.C → O(n)

Practical Usuage

Code = 1s → 10^8 operations

if exceeded gives error

if it comes → TLE (Time Limit Error)

we have to optimize

$n > 10^8$	$O(\log n)$ $O(1)$
$n \leq 10^8$	$O(n)$
$n \leq 10^6$	$O(n \log n)$ — Sorting
$n \leq 10^4$	$O(n^2)$
$n \leq 500$	$O(n^3)$
$n \leq 25$	$O(2^n)$ — Recursion ↓ Brute Force
$n \leq 12$	$O(n!)$