- Time & Space complexity. (Book → cormen)

Order Complexity Analysis.

Amount of space or time taken up by an algorithm/code as function of input size.
NOT the actual time taken.

Big O Notation.
↳ upper bound.

Note :- we always try to find worst case complexity.

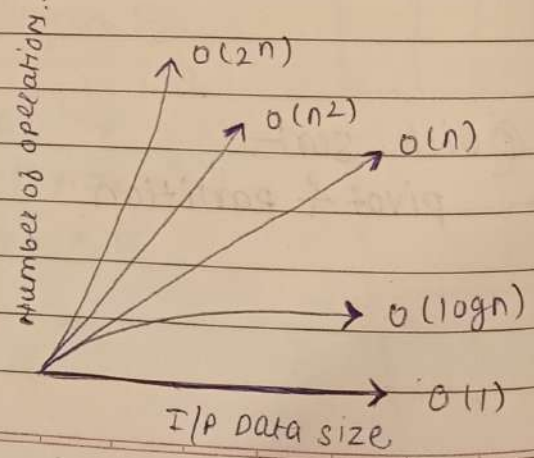
$$f(n) = O(g(n))$$

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

Big Omega Notation (optional)
↓
(Lower Bound) (Best case)

Big Theta (Θ) → (average case)
~~O(n²)~~

- common complexities :-



Page No.
Date

- Space Complexity
  Input space + auxiliary space.

- Loops
(i) simple Loop

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

Time complexity ⇒ O(n).

(2) Nested Loop 1.

```
for (int i=0 ; i<n; i++) {
    for (int j=i+1; j<n; j++) {
        // some const. work.
        // is done in this loop
    }
}
```

0 to n → $\frac{n(n+1)}{2}$

0 to n-1 → $\frac{n(n-1)}{2} = \frac{n^2-n}{2}$

$O\left(\frac{n^2-n}{2}\right) = O(n^2)$

(2) ↳ constant ignore & consider Largest value.

③ Nested Loop 2.

```
for (int i=0; i<n ; i++)
    for(int j=0; j<i; j++) {
        // some const work.
    }
}.
```

$i=0$ to $n-1$ → n times.

$j=0$ to $i-1$ →

$i=0$	0 times.		
$i=1$	$1\times$	$j=0$ to 0	k times.
$i=2$	$2\times$	$j=0$ to 1	$k=2k$
$i=3$	$3\times$	$j=0$ to 2	$k=3k$.
⋮			
$i=n-1$	$(n-1)\times$	$j=0$ to $n-1$	$k=(n-1)k$

$\therefore k+2k+3k \cdots (n-1)k$

$k(1+2\cdots(n-1))$

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

$O(n^2)$

④
```
for (int i=0; i<n ; i=i+k) {
    for (int j=i+1 ; j<=k ; j++) {
    }
}.
```

assume $n=50$ $k=5$

Outer Loop ⇒ $\frac{n}{k} = \frac{50}{5} = 10$

inner Loop ⇒ $\frac{n}{k} \times k = n$

$O(n/k \times k) = O(n)$

∴ Time complexity : O(n) not $O(n^2)$

Sorting

Bubble Sort (Worst & Best case)

```
public static void modifiedbubblesort (int arr[]){
    for (int turn=0; turn<arr.length-1; turn++){
        boolean swapped = false;
        for (int j=0; j<arr.length-1-turn; j++){
            if (arr[j] > arr[j+1]) {
                int temp = arr[j]
                arr[j] = arr[j+1]
                arr[j+1] = temp
                swapped = true;
            }.
        }
        if (swapped == = false) {
            break;
        }
    }
}
```

Best case :- O(n)

worst case :- $O(n^2)$.

② Binary search.

$$\frac{n}{2^k} \rightarrow \frac{n}{2^1}, \frac{n}{2^2}, \frac{n}{2^3} \cdots \frac{n}{2^k} = 1$$

$$\therefore \frac{n}{2^k} = 1 \Rightarrow n = 2^k$$

$$\log_2 n = k.$$

$$\therefore O(\log n) \text{ TC}$$

Recursive Algorithms

① Total work done = (no of calls * work in each call)

② Recurrence Equation

Space Complexity = (max depth * memory in each call)

① Factorial.

```
public static int fact (int n){
    if (n==0){
        return 1;
    }
    return n * fact(n-1);
}
```

Total no. of calls :- n.

∴ Time complexity = O(n).

∴ Space complexity = max depth * each call.
= (n) * (k)
↳ const
= O(n)

② Sum of n.

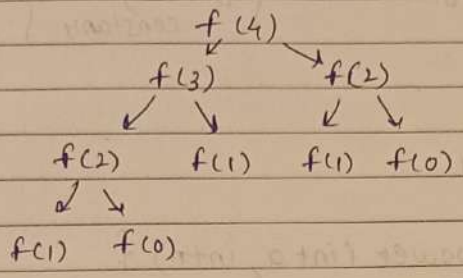
```
Static int sum (int n){
    if (n==0){
        return 0
    }
    return n + sum (n-1);
}
```

T.C = O(n)
S.C = O(n)

- fibonacci (recursion)

```
Public class fibonacci {
    Static int fib (int n) {
        if (n==0 || n==1){
            return n;
        }
        return fib(n-1) * fib(n-2);
    }
```

∴ Time complexity



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

SC = depth * ram

= n * 1k

= O(n).

merge sort.

$T(n) = T(n/2) + T(n/2) + nk$ ⟶ nk

$T(n) = 2T(n/2) + nk$ ⟶ nk

$2T(n/2) = 4T(n/2) + nk(2)$ ⟶ nk

$4T(n/4) = 8T(n/8) + 4(n/4)k$ ⟶ nk

$T(n/8) = 2T(n/16) + n/8\,k$ ⟶ nk

⋮

$T(1) = O(1) + nk$.

$n \rightarrow \frac{n}{2^0}$

$n/2 \rightarrow n/2^1$

$n/4 \rightarrow n/2^2$

$n/8 \rightarrow n/2^3$.

$\frac{n}{2^x} = 1$

$n = 2^x$

$\boxed{\log_2 n = x}$

$i = \frac{n}{2^x}$

$\therefore T(n) = O(1) + (\log n)(n+1)$

$= n \log(n) \longrightarrow$ { ignoring the constant }

Recursion

power function :

```
public static int power (int a, int n) {
    if (n==0){
        return 1;
    }
    return a * power (a, n-1);
}
```

$\therefore f(a,n)\ a^n$

$a\, f(a, n-1)\ a^{n-1}$

$a f(a, n-2)\ a^{n-2}$

$a * f(a, n-3)\ a^{n-3}$.

$1 * f(a, 0)\ a^0$

Worddone = no.of calls * times in each call

$= n * k$

$= n$

$\therefore TC = O(n)$

Space Compx $= n * O(1)$

(n: Total Levels; O(1): each level.)

$= O(n)$