1) Asymptotic Notations :-

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

There are mainly three of asymptotic notations :-
- Big-O notation
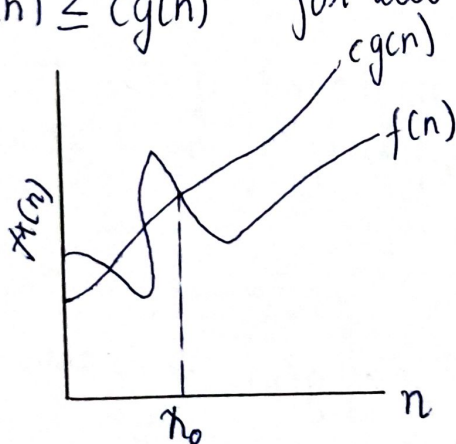- Omega notation
- Theta notation

(i) Big O Notation :-
- gives upper bound of the running time of an algorithm
- gives worst-case complexity of an algorithm
  Given two functions $f(n)$ & $g(n)$

$$f(n) = O(g(n))$$
iff
$$f(n) \leq cg(n) \quad \text{for all } n \geq n_0 \quad c > 0$$

Example In bubble sort,
When the input array is in reverse condition, the algorithm takes the maximum time (n^2) to sort the elements i.e the worst case.

(ii) Omega Notation (Ω):-
- represents the lower bound of the running time of an algorithm
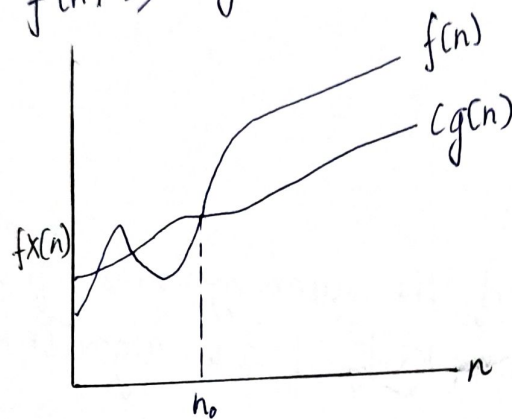- It provides the best case complexity of an algorithm

Given two functions
$$f(n) \text{ and } g(n)$$
$$f(n) = \Omega(g(n))$$
iff
$$f(n) \geq c\, g(n) \quad \text{for all } n \geq n_0, \; c > 0$$

Example In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear ie best case.

(iii) Theta Notation (θ):-
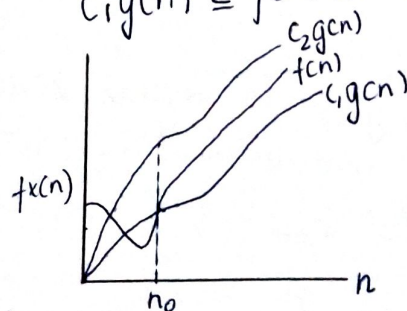- represents the upper & lower bound of the running time of an al
- provides the average-case complexity of an algorithm

Given two fn
$$f(n) \text{ & } g(n)$$
$$f(n) = \theta(g(n))$$
iff
$$c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \text{for all } n \geq n_0, \; c_1, c_2 > 0$$

Example:- In bubble sort, when the input array is neither sorted nor in reverse order, then it takes average-time.

```
fonl( in=1 to n)
       { i = i*2 ;}
```

$i = 1, 2, 4, 8, 16, 32 \cdots 2^k$

$n = a r^{k-1}$

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

$n = 2^{k-1}$

$n = \dfrac{2^k}{2}$

$2n = 2^k$

$\log(2n) = \log(2^k)$

$\log(2n) = k \log_2 2$

$R = \log(2n)$

$\Rightarrow$ Complexity $= O(\log n)$

) $T(n) = \{3T(n-1)$ if $n > 0$, otherwise $1\}$

$T(n) = 3T(n-1)$

$T(n-1) = 3T(n-2) \quad (\overset{\text{putting}}{h = n-1})$

$T(n) = 3(3(T(n-2)))$

$\qquad = 3^3 T(n-3)$

$\qquad \vdots$

$\qquad = 3^n (T(n-n))$

$\qquad = 3^n T(0)$

$\qquad = 3^n \qquad [\because T(0) = 1]$

$\Rightarrow$ complexity $= O(3^n)$

4   $T(n) = \{2T(n-1)-1$ if $n>0$, otherwise $1\}$

$$T(n) = 2T(n-1) - 1 \quad \text{—①}$$

putting $n = n-1$

$$T(n-1) = 2T(n-2) - 1$$

$$T(n) + 1 = 2(2T(n-2)\} - 1) \quad (\text{from ①})$$

$$T(n) = 2^2 T(n-2) - 2 - 1$$

putting $n = n-1$

$$T(n-1) = 2^2 T(n-3) - 2 - 1$$

$$T(n)+1 = 2^3 T(n-3) - 2^2 - 2 \quad (\text{from ①})$$

$$T(n) = 2^3 T(n-3) - 2^2 - 2 - 1$$

$\vdots$

$$T(1) = 2T(1-1) - 1$$
$$= 2T(0) - 1$$
$$T(1) = 2 - 1$$
$$T(1) = 1 \quad \text{—②}$$

$$T(n) = 2^k T(n-k) - 2^{k-1} - 2^{k-2} - 2^{k-3} \cdots - 2^2 - 2^1 - 2^0$$

$T(1) = 1$ from ②

$n - k = 1$

$$T(n) = 2^{n-1} T(1) - [2^0 + 2^1 + 2^2 + \cdots + 2^{n-3} + 2^{n-2}]$$

$$= 2^{n-1} \times 1 - [2^{n-1} - 1]$$

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

$$T(n) = 1$$

$\Rightarrow$ Complexity $= O(1)$

```
..1 , s=1;
while (s <= n) {
    i++; s=s+i;          S
        printf("#"),      3
    }                     8
                         10      } k times
                         15
                         21
                          ⋮
                          n
```

$S_k = 3+6+10+15+21+ \cdots T_k$

$S_{R-1} = 3+6+10+15+ \cdots T_{R-1}$

$S_R - S_{R-1} = 3+3+4+5+6+\cdots + T_k - T_{k-1}$

$T_k = 3+ [3+4+5+6+\cdots + (k-1) \text{ times}]$

$\quad = 3 + \frac{(k-1)}{2} (6 + (k-2)\cdot 1)$

$\quad = 3 + \frac{(R-2)(R+4)}{2}$

(T_k) R^{th} term

for last iteration k^{th} term is n

$T_R = n$

$\frac{(R-2)(R+4)}{2} + 3 = n$

For time complexity removing lower order terms

$R = \sqrt{n}$

$\Rightarrow$ Complexity $= O(\sqrt{n})$

6)
```
void function(int n) {
    int i, count = 0;
    for (i=1; i*i <= n; i++)
        count ++;
}
```

i	times
1	$\sqrt{1}$
2	$\sqrt{2}$
3	$\sqrt{3}$
4	$\sqrt{4}$
$\vdots$	$\vdots$
n	$\sqrt{n}$

$\Rightarrow$ complexity $= O(\sqrt{n})$

7)
```
void function (int n) {
    int i, j, k, count = 0;
    for (i= n/2; i <= n; i++)
        for (j =1; j <= n; j =j*2)
            for (k=1; k <= n; k = k*2)
                count ++ }
```

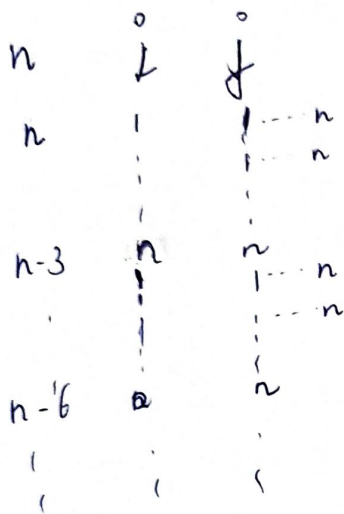| i	j	k
n	1	1
n-1	2	2
$\vdots$	4	$\vdots$
n/2	$\vdots$	$\vdots$

$\frac{n}{2}$ times | $\log n$ times | $\log n$ times $= O\left(\frac{n}{2}(\log n)(\log n)\right)$

$\Rightarrow$ complexity $= O(n \log n)^2)$

```
function ( int n) {
    if (n == 1) return;
        for (i = 1 to n) {
            for (j = 1 to n) {
                print("**");
            }
        }
        function (n -3);
}
```
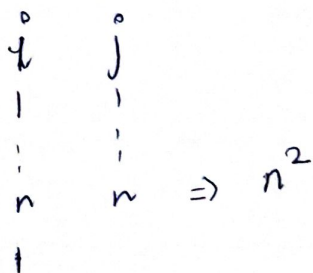
$$n \quad i \quad j$$
$$n \quad 1 \quad \begin{matrix} n \\ n \end{matrix}$$
$$n-3 \quad \begin{matrix} n \\ \\ \end{matrix} \quad \begin{matrix} n \\ n \end{matrix}$$
$$n-6 \quad \quad n$$

$$= \quad n + n-3 + n-6 \cdots + 7+ 4 + 1$$

or $\quad 1 + 4 + 7 + \cdots + n-3 + n$

$$n = 1 + 3(k-1) \quad (a + (n-1)d)$$

$$k = \frac{n+2}{3}$$

$\Rightarrow$ Time complexity of recursion = $O(n)$

$$i \quad j$$
$$1 \quad 1$$
$$\vdots \quad \vdots$$
$$n \quad n \Rightarrow n^2$$

∴ Total time complexity $= O(n \cdot n^2)$
$$= O(n^3)$$

a) void function (int n) {
   for (i=1 to n) {
      for (j=1; j<=n; j=j+1)
         {printf ("* ")}
   }
}

3

i j
1   1
             ⋮
             n
2   1
             3
             ⋮
             n
3   ⋮
⋮   1
n n = n^2

→ Time complexity = $O(n^2)$