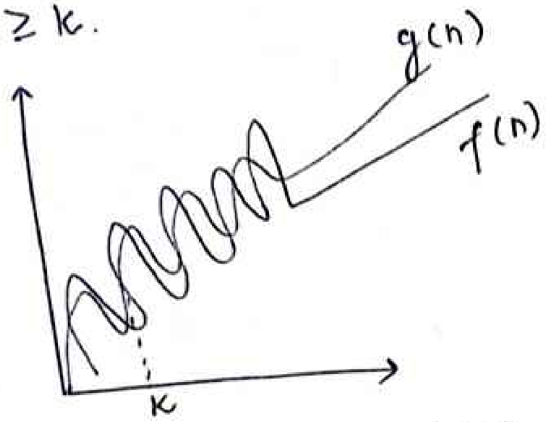
Tutorial sheet-1

Sol 1:- Asymptotic Notation :-

→ These notations are used to tell the complexity of an algorithm when the input is very large.
→ It describes the algorithm effeciency and performance in a meaningful way. It describes the behaviour of time or space complexity for large instance characteristics.

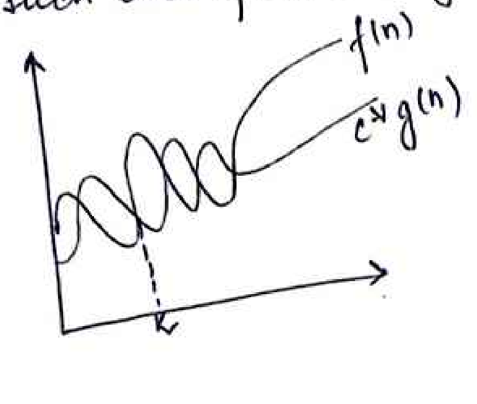
- The asymptotic notation of an algorithm is classified into 5 types :-

(i) Big Oh notation (O) :- (Asymptotic upper Bound) The function $f(n) = O(g(n))$, if and only if there exist a +ve constant C and k such that $f(n) \leq C * g(n)$ for all n $n \geq k$.



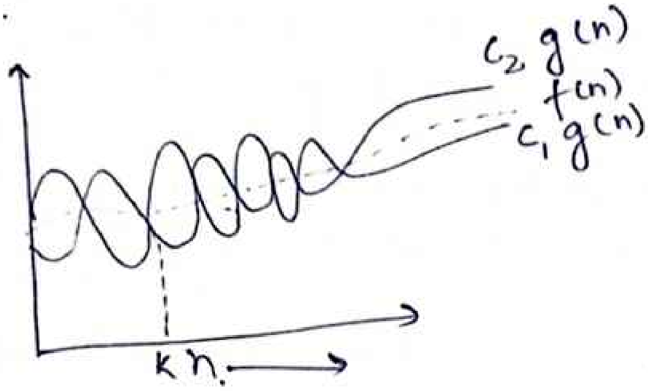
$f(n) = O(g(n))$
iff
$f(n) \leq C \cdot g(n)$
$\forall\ n \geq n_0$,
Some constant $C > 0$.

(ii) Big omega notation (Ω) :(Asymptotic lower bound)
The function $f(n) = \Omega(g(n))$, iff there exists a +ve constant C and k such that $f(n) \geq C * g(n)$ for all n, $n \geq k$.



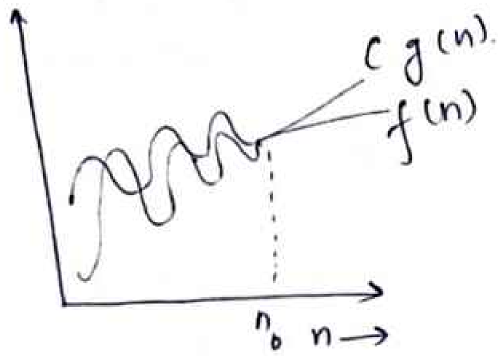
$f(n) = \Omega\, g(n)$ iff
$f(n) \geq C \cdot g(n)$
$\forall\ n \geq n_0$ & some const $C > 0$.

(iii) Big theta notation (Θ): (Asymptotic tight bound) The function $f(n) = \Theta(g(n))$, iff there exists a +ve constant c_1, c_2 & k such that $c_1 * g(n) < f(n) < c_2 * g(n)$ for all n, $n \geq k$.



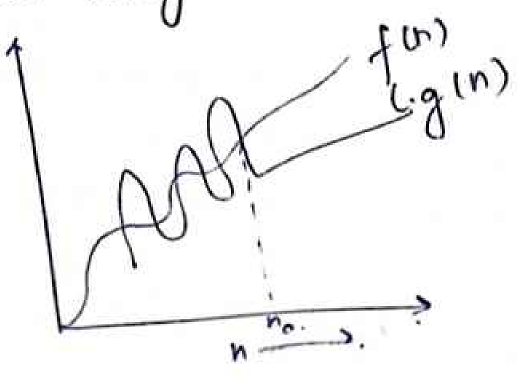
$f(n) = \Theta(g(n))$
iff
$c_1 g(n) \leq f(n) \leq c_2 \cdot g(n)$
$\forall n \geq \max(n_1, n_2)$

(iv) Small-oh (o) :- o gives us upper bound.
$f(n) = o(g(n))$.



$f(n) < c\, g(n)$
$\forall n > n_0$ & $\forall c > 0$
$n = o(n^2)$
$n < 1 n^2$
$2 n^2$
$0.5 n^2$.
$n < 0.001 n^2$ n_0.

(v) Small-omega (ω):



lower bound
$f(n) = \omega\, g(n)$
$f(n) > c \cdot g(n)$
$\forall n > n_0$ & $\forall c > 0$.
$n^2 = \omega(n)$.

Sol2:- for (i = 1 to n)

```
{   i = i*2;
}
```

Time complexity for a loop means no. of times loop has run.

→ For the above loop, the loop will run for the following values of i :-

i	1	2	4	8	16	32	...	2^k
value	2^0	2^1	2^2	2^3	2^4	2^5	...	n

$i = 1, 2, 4, 8, 16, 32, \ldots, 2^k$ this means k times

i.e. $2^k = n$

$k \log_2 2 = \log_2 n$

$k = \log n$ $\{ \log_2 2 = 1$.

$\boxed{\therefore T.C = O(\log n)}$

Sol 3:- $T(n) = \begin{cases} 3T(n-1), & n > 0 \\ 1 \end{cases}$

By forward substitution,

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

$T(0) = 1$

$T(1) = 3T(1-1)$
$= 3T(0)$
$= 3$.

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

$T(3) = 3T(3-1)$
$= 3T(2)$
$= 3 * 3^2 = 3^3$.

$\vdots$

$T(n) = 3^n$.

$\therefore \boxed{T.C = O(3^n)}$

Sol4:- $T(n) = \begin{cases} 2T(n-1) - 1 & , n > 0 \\ 1 \end{cases}$

By forward substitution,

$T(0) = 1$

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

$= (2 - 1)$

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

$= 2T(1) - 1$

$= 2(2-1) - 1$

$= 2^2 - 2^1 - 1$

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

$= 2T(2) - 1$

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

$= 2^3 - 2^2 - 2^1 - 1$

$\vdots$

$2^n - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots - 2^2 - 2^1 - 2^0$

$\Rightarrow 2^n - (2^n - 1)$

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

$\therefore \boxed{T.C = 1}$

Sol5:-
```
int-1, s=1;
while (s<=n)
{
    i++;
    s=s+i;
    printf("#");
}
```

$s_i = s_{i-1} + i$

The value of 'i' increases by one for each ite[ration]. The value contained in 's' at the i^{th} iteration is the sum of the first 'i' +ve integers. If k is the total no. of iterations taken by any program then while loop terminates if : $1 + 2 + 3 + \dots + k$

$$= [k(k+1)/2] > n$$

so, $k = O(\sqrt{n})$

$\therefore \boxed{T.C = O(\sqrt{n})}$

<u>Sol6</u>:- void function (int n)

```
{
    int i, count = 0;
    for (i = 1; i <= n; i++)      O(n)
        count++;
}
```

Time complexity :- $O(n)$.

<u>Sol7</u>:- 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)          O(log n)
            for (k = 1; k <= n; k = k * 2)  O(log n)
                count++;
}
```

$T.C = \log n * \log n = O(n\log^2 n)$

$\boxed{T.C = O(n \log^2 n)}$

Sol8:- function (int n)

```
function (int n)
{
    if (n==1)
        return;
    for (i=1 to n)              O(n) times
    {
        for (j=1 to n)          O(n) times
        {   printf("*");
        }
    }
    function(n-3);
}
```

Time complexity :- $O(n^2)$ ans.

Sol9:-

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

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

$\boxed{T.C = O(n^2)}$

Sol10:- n^k is $O(c^n)$ ans.

$n^k = O(c^n)$