Tutorial - 01

PAGE No.
DATE

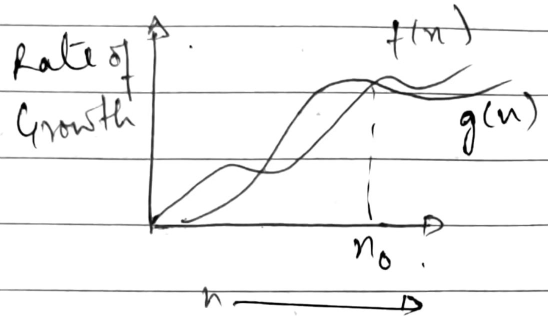
Asymptotic Notation:-

→ These notations are used to tell complexity of an algo when i/p is very large.

→ It describes the algo efficiency & performance in a meaningful way. It describes the behaviour of time or space complexity for large instance characteristics.

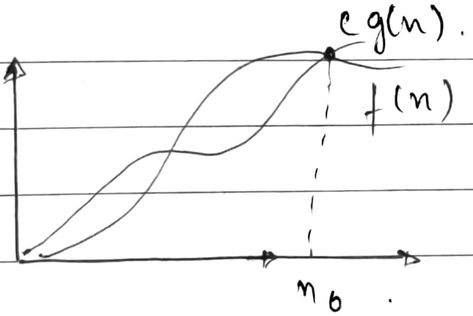
1. Big oh (O) : $f(n) = O(g(n))$ if $c(g(n)) \geq f(n)$ $\forall\, n \geq n_0$ for some constant $c > 0$.

2. Big Omega (Ω) : $f(n) = \Omega(g(n))$ if $f(n) \geq c(g(n)) \geq 0$ $\forall\, n \geq n_0$ & some constant $c > 0$



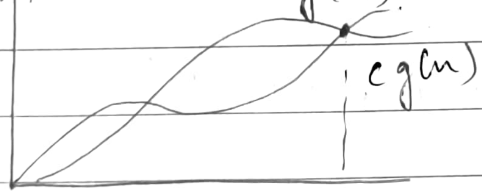
* g(n) - tight lower bound.
* f(n) - tight upper bound.

3. Small o : $f(n) = o\, g(n)$ if $c(g(n)) > f(n)$ $\forall\, n > n_0$ $\forall\, c > 0$



f(n) = upper bound.

4. Small Omega (ω): $f(n) = \omega\, g(n)$ if, $c\,g(n) < f(n)$ $\forall\, n > n_0$ & $\forall\, c > 0$.



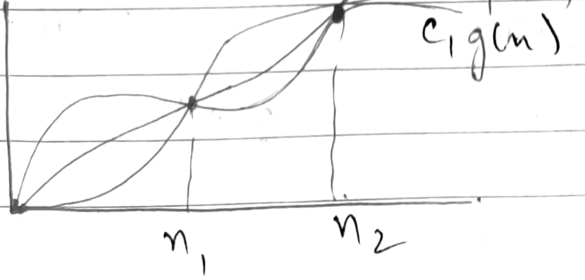
$f(n) = \Omega\, g(n)$

5. Theta Notation (Θ) : $f(n) = \Theta\, g(n)$ iff

$f(n) = O(g(n))$ $\boxed{c_1\, g(n) \leq f(n) \leq c_2\,(g(n))}$

$\forall\, n \geq \max(n_1, n_2)$.

& some const. c_1 & $c_2 > 0$.



2. for (i = 1 to n)

{ i = i * 2; }

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

For above loop, the loop will run for foll. 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

$i = 1, 2, 4, 8, \ldots, 2^k$ i.e k times

i.e $2^k = n$

$k \log_2 2 = \log_2 n$

$k = \log n \quad [\log_2 2 = 1]$

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

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

By forward substitution;

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

$T(0) = 1$

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

$T(\phi) = 3T(1-1)$

$= 3T(0)$

$= 3$

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

$= 3 * 3 = 3^2$

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

$|$

$T(n) = 3^n$

$\therefore T(C) = O(3^n)$

PAGE No.
DATE

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

By forward substⁿ,

$T(0) = 1$

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

$= 2-1$

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

$= 2T(1)-1$

$= 2(2-1)-1$

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

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

$= 2T(2)-1$

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

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

$\vdots$

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

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

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

$\therefore T(c)=1$

5.

```
int =1 , S=1;
while (S<=n)
{
    i++;
    S=S+i;
    printf("#");
}
```

PAGE No. DATE

$S_i = S_{i-1} + i$

The value of 'i' inc. by one for each value contained in 's' at the ith iteration is the sum of first 'i' +ve integers. If k is total no of iterations taken by any program then while loop terminates if : $1+2+3+ \cdots +k$

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

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

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

6.

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

Time complexity :- $O(n)$

7

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

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

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

PAGE No.
DATE

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

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

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

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

10. n^k is $O(c^n)$ ans
$n^k = O(c^n)$