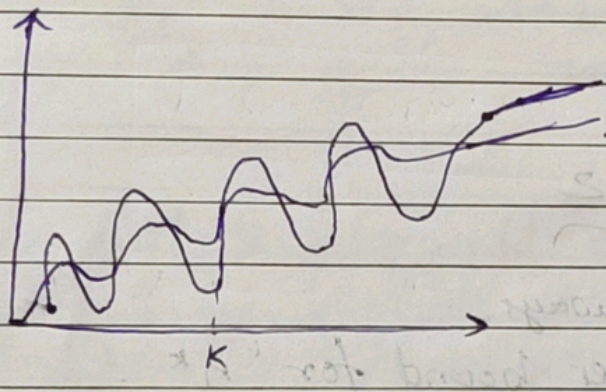
Date

Assignment - 1

Ans 1: These notations are used to tell the complexity of an algorithm when the input is very large.

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

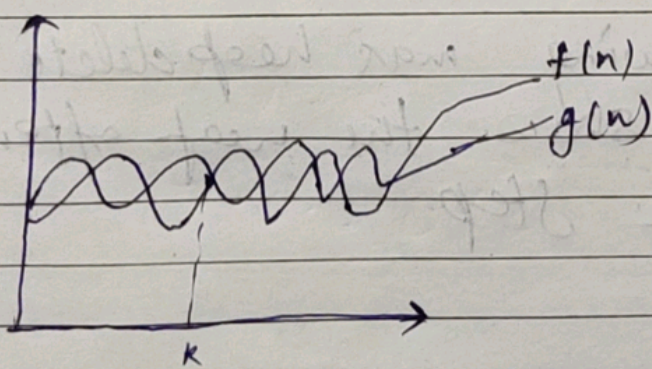
(i) **Big Oh notation (O):** (Asymptotic upper bound) The function $f(n) = O(g(n))$, if and only if there exist a +ve constant c & K such that $f(n) \leq c * g(n)$ for all n, $n \geq K$.



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

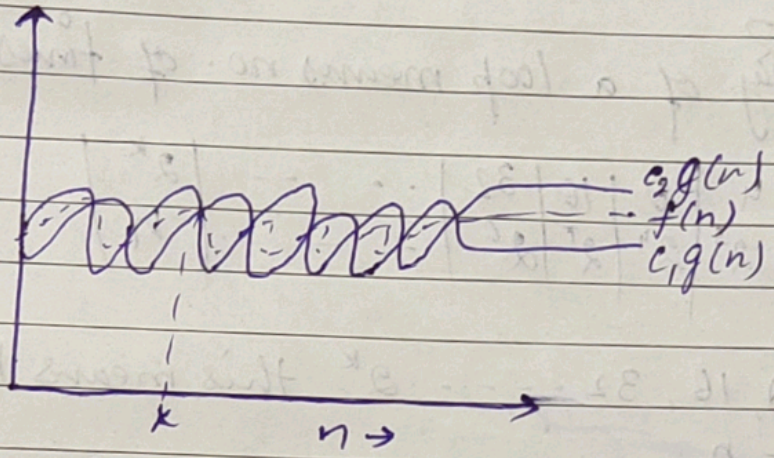
The function $f(n) = \Omega(g(n))$, iff there exists a +ve constant c & K such that $f(n) \geq c * g(n)$ for all n, $n \geq k$



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

(iii) Big theta notation (θ) : (Asymptotic light bound)

$f(n) = \theta(g(n))$



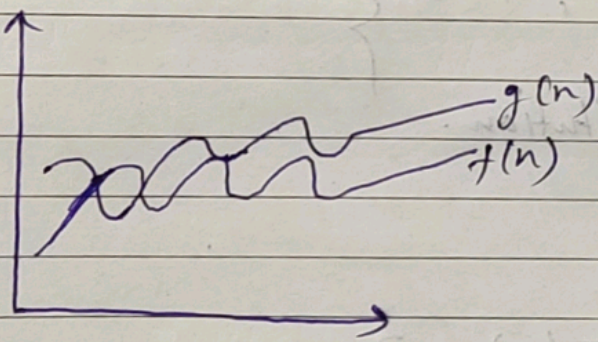
$f(n) = \theta(g(n))$ iff

$c_1 g(n) \leq f(n) \leq c_2 \cdot g(n)$

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

(iv) Small oh (o): o gives us upper bound.

$f(n) = o(g(n))$



$f(n) \leq o(g(n))$

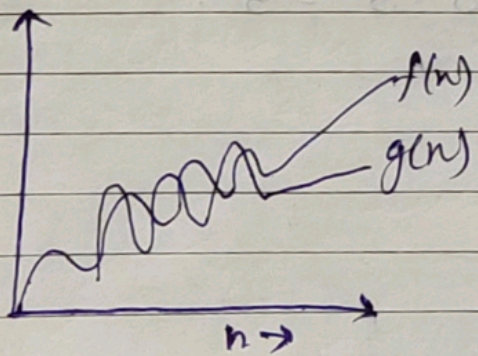
$\forall\ n > n_0$ & $\forall\ c > 0$

$n = o(n^2)$

$n < 1n^2$

$n < 0.001 n^2 n_0$

(v) Small omega (ω) : lower bound



$f(n) = \omega g(n)$

$f(n) > c g(n)$

$\forall\ n > n_0$ & $\forall\ c > 0$

$n^2 = \omega(n)$

Page No.
Date

Ans 2 : for (i=1 to n):
{ i = i * 2; }

time complexity of a loop means no. of times it has run.

i	1	2	4	8	16	32	------	2^k
value	2	2^2	2^3	2^4	2^5	2^6	------	n

i = 1, 2, 4, 8, 16, 32 ------ 2^k this means k times.

ie $2^k = n$

$k \log_2 2 = \log_2 n$

$k = \log n \quad (\log_2 2 = 1)$

T.C. = O(log n)

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

By forward substitution.

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

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

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

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

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

$T(n) = 3^n$

T.C. = O(3^n)

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

By forward subst.

$T(0) = 1$

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

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

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

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

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

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

$T.C := O(1)$.

5)
```
int i=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 iteration the value contained in 's' at the i^{th} iteration is the sum of the first 'i' the integers. If k is the total no. of iteration taken by any prog. then while loop terminate if.

$1+2+3+\ldots\ldots+k \Rightarrow (k(k+1)/2) > n$

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

Ans 6 :

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

$O(n) = T.C.$

Ans 7 :

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

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

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

Page No.
Date

Ans 8:

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

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

9)

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

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

Page No.
Date

Q) for the function, n^k & c^n, what is the asymptotic relationship b/w these fun.

- Assume that $k >= 1$ & $c > 1$ are constants.

	n^k	c^n
	$k >= 1$	$c > 1$
let	$k = 1$	$c = 2$
	n^1	2^n

as $c^n > n^k$ always that means c^n grows at a faster rate than n^k

thus, we can say c^n will act as an upper bound to n^k.