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

By forward substitution

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

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

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

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

$= 3*3$

$= 9$

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

$= 3T(2)$

$= 3*3^2$

$= 3^3$

$T(n) = 3^n$

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

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

By forward substitution

$T(0) = 1$

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

$= 2-1$

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

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

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

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

$\vdots$

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

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

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

$= 1$

T.C. $= O(1)$

(5)
```
int =1, s=1;
while (s<=n)
{ i++;
  s=s+i;
  printf("#");
}
```
The value of 'i' increase by one for each iteration. The value contained in 's' at the ith 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 terminate if:

$1+2+3+ \dots +k$

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

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++)
 {
     count++;
 }
```
$O(n)$ = T.C.

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

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

(8) function (int n)
```
{
  if(n==1)
     return;
  for(i=1 to n)              O(n)
  {
     for(j=1 to n)           O(n)
     {
        printf("*");
     }
  }
  function(n-3);
}
```
T.C. $= O(n^2)$

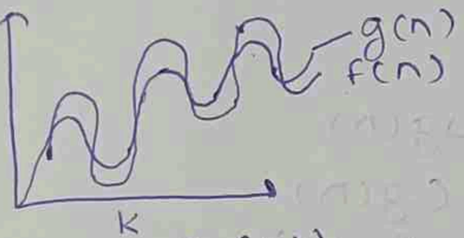
(9) void function (int n)
```
{
  for (i=1 to n )                     O(n)
   for(j=1; j<=n; j=j+1)              O(n)
        printf("*");
  }
}
```
T.C. $= O(n^2)$

(10) for the function, n^k & c^n, what is the asymptotic notations b/w these functions.
Assume, that $k>=1$ & $c>1$ are constants find out the value of c & n_0 for which relation holds.
n^k is $O(c^n)$.

ASSIGNMENT 01

① These notation are used to tell the complexity of an algorithm when the input is very large
→ It describe the algorithm efficiency and Performance in a meaningful way. It describes the behaviour of time or space complexity for large instance characteristics

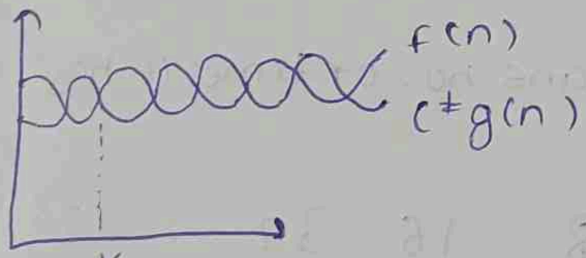
① big oh notation - The function $f(n) = O(g(n))$, if and only if there exist a positive constant c and k such that $f(n) \leq c * g(n)$ for all n, $n \geq k$



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

$f(n) = O(g(n))$
iff
$f(n) \leq c \cdot g(n)$
$\forall\ n \geq n_0$
So constant $c > 0$

② Big omega Notation → 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, $n \geq k$



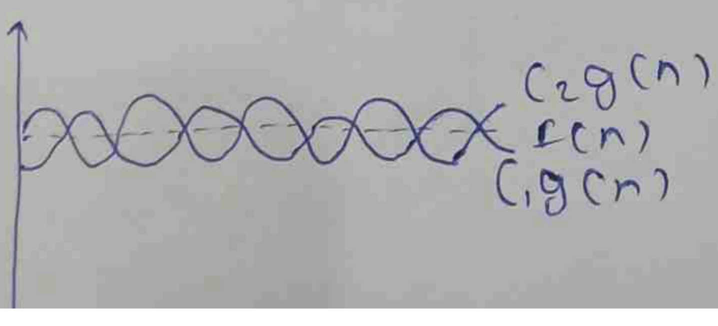
$f(n) = \Omega g(n)$
$f(n) \geq c \cdot g(n)$

$\forall\ n \geq n_0$ and same consta
$c > 0$

iii) Big theta notation

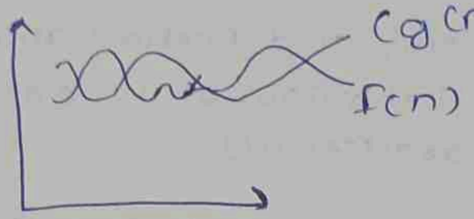
Similarly,

$f(n) = \theta g(n))$
if $c_1 g(n) \leq f(n) \leq c_2 g($
$\forall\ n \geq max(n_1, n_2)$



iv) Small oh notation - o gives us upper Band

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



$f(n) \leq c\,g(n)$

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

$n = o(n^2)$

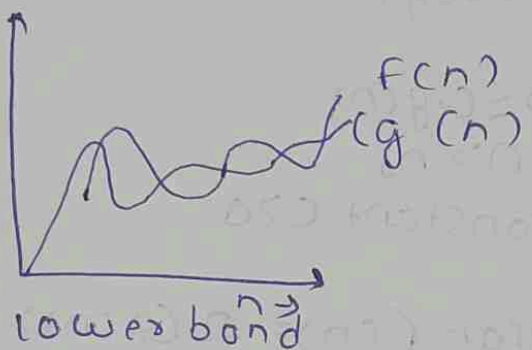
$n < 1n^2$

$2n^2$

$0.5n^2$

$n < 0.001 n^2 n_0$

v) Small omega (ω)



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

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

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

$n^2 = \omega(n)$

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	2^1	2^2	2^3	2^4	2^5	2^6	.... n

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

i.e., $2^k = n$

$k \log_2 2 = \log_2 n$

$k = \log n$

$\log_2 2 = 1$

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

By forward substitution

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

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

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

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

$= 3 * 3$

$= 9$

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

$= 3T(2)$

$= 3 * 3^2$

$= 3^3$

$T(n) = 3^n$

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

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

By forward substitution

$T(0) = 1$

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

$= 2 - 1$

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

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

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

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

$\vdots$

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

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

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

$= 1$

T.C. $= O(1)$

(5)
```
int i=1, s=1;
while (s<=n)
{ i++;
  s=s+i;
  printf("#");
}
```
The value of 'i' increase by one for each iteration. The value contained in 's' at the ith 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 terminate if:

$1+2+3+ \dots + k$

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

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++)
 {
    count++;
 }
```
$O(n)$ = T.C.

(7)
```
void function (int n)
{
 int j, k, i, 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. = $\log n * \log n$
$= O(n \log^2 n)$
$\therefore$ T.C. $= O(n \log^2 n)$

(8)
```
function (int n)
{
  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(j=1; j<=n; j=j+1)           O(n)
       printf("*");
  }
}
```
T.C. $= O(n^2)$

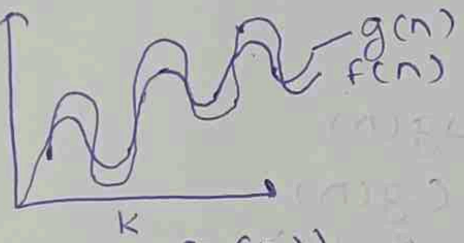
(10) for the function, n^k & c^n, what is the asymptotic notations b/w these functions.
Assume, that $k>=1$ & $c>1$ are constants find out the value of c & n_0 for which relation holds.

n^k is $O(c^n)$.

ASSIGNMENT 01

① These notation are used to tell the complexity of an algorithm when the input is very large
→ It describe the algorithm efficiency and Performance in a meaningful way. It describes the behaviour of time or space complexity for large instance characteristics

① big oh notation - The function $f(n) = O(g(n))$, if and only if there exist a positive constant c and k such that $f(n) \leq c^* g(n)$ for all $n, n \geq k$

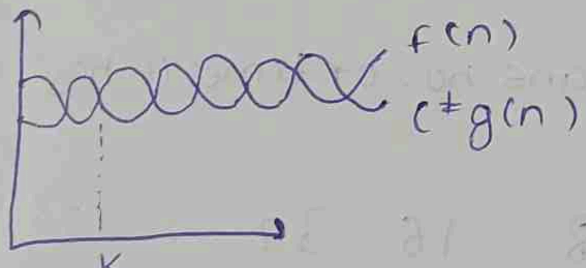


$f(n) = O(g(n))$
if

$f(n) = O(g(n))$
iff
$f(n) \leq c \cdot g(n)$
$\forall\ n \geq n_0$
So constant $c > 0$

② Big omega Notation → 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, $n, n \geq k$

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



$f(n) \geq c \cdot g(n)$

$\forall\ n \geq n_0$ and same consta

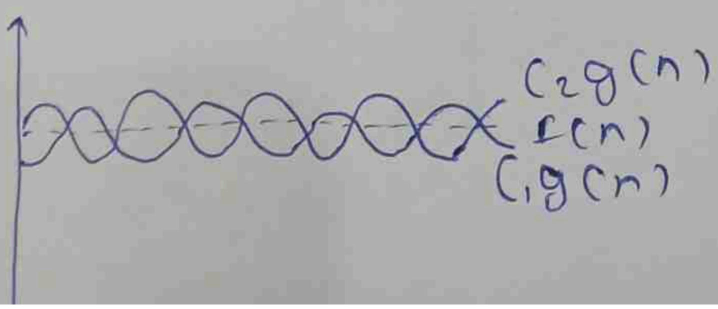
$c > 0$

iii) Big theta notation
Similarly,

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

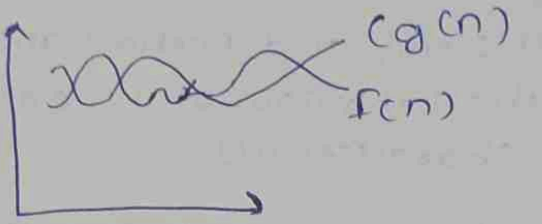
if $c_1 g(n) \leq f(n) \leq c_2 g(n)$

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



iv) Small oh notation - o gives us upper Band

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



$f(n) \le c\,g(n)$

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

$n = o(n^2)$

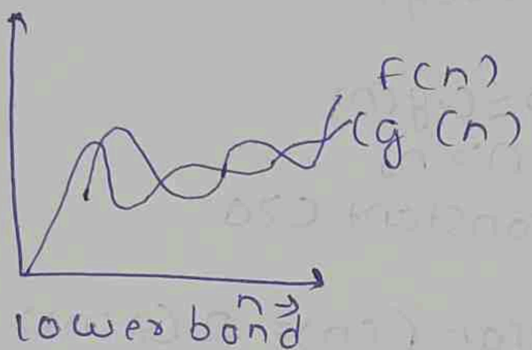
$n < 1n^2$

$2n^2$

$0.5n^2$

$n < 0.001 n^2 n_0$

v) Small omega (ω)



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

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

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

$n^2 = \omega(n)$

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	2^1	2^2	2^3	2^4	2^5	2^6	... n

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

i.e., $2^k = n$

$k \log_2 2 = \log_2 n$

$k = \log n$

$\log_2 2 = 1$