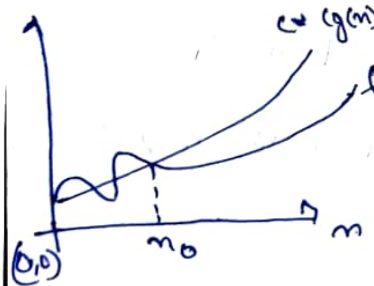
Q.1 ⟹ **Asymptotic Notation:** There are language to express the required time & space by an algorithm to solve a given problem.

(1) **Big - O Notation:** It is notation for the worst case analysis of an algorithm. (Upper bound)

According to it for a two func. $f(n)$ & $g(n)$, $f(n) = O(g(n))$, if and only if there exist n_0 & c such that.



$0 \le f(n) \le c * g(n)$ for all $n \geq n_0$

Ex ⟹ $n+n^2 = O(n^2)$

here $f(n) = n+n^2$, $g(n) = n^2$.

$n+n^2 \le n^2 + n^2$ ($\because n < n^2$, $n^2 = n^2$)

$n+n^2 \le 2n^2$ (here $c=2$) for $n_0 = 1$

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

or $n+n^2 = O(n^2)$

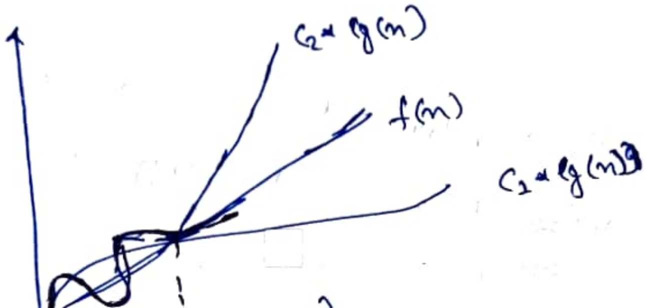
⊙ **Big theta (Θ);** For avg case time complexity (tightly bound)

for any two function $f(n)$ & $g(n)$

$f(n) = \Theta(g(n))$ if and only if there exists n_0, c_1, c_2

such that $0 \le c_1 * g(n) \le f(n) \le c_2 * g(n)$

[for n

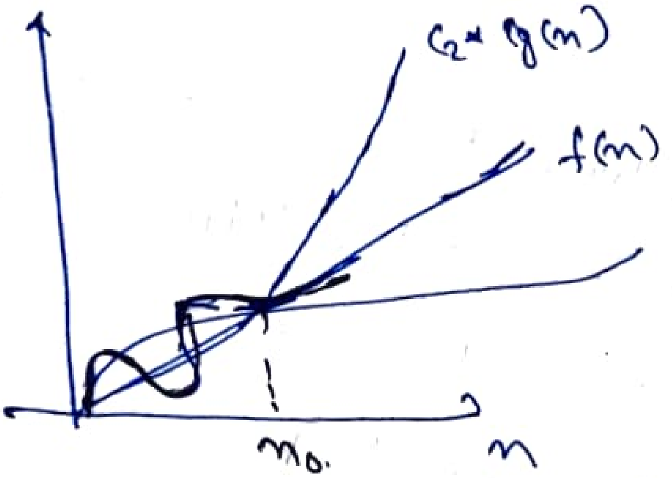


Big theta (Θ) : For avg case time complexity (~~low~~ tightly bound)

for any two function $f(n)$ & $g(n)$

$f(n) = \Theta(g(n))$ if and only if there exists n_0, c_1, c_2

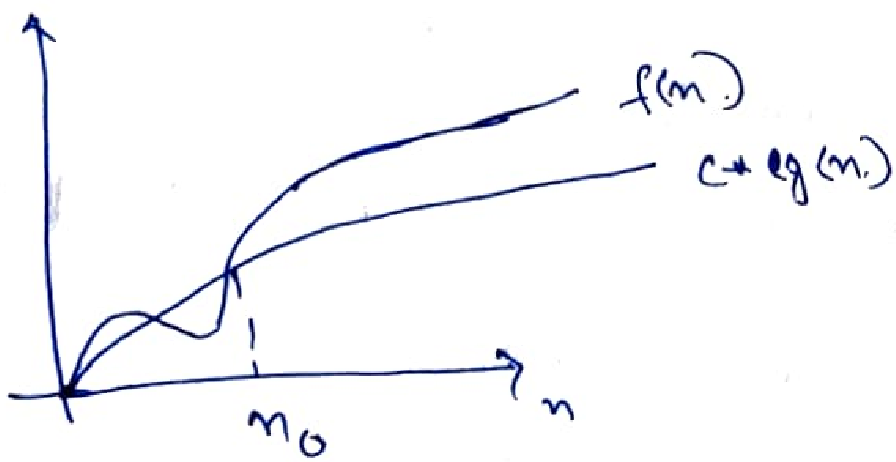
such that $0 \le c_1 * g(n) \le f(n) \le c_2 * g(n)$ [for $n \geq n_0$]



Big Omega (Ω) : for best case ~~tir~~ complexity (lower bound)

$f(n) = \Omega(g(n))$ iff $\exists\ n_0, c_1$

$\ni\ 0 \le c_1 * g(n) \le f(n)\ \forall\ n >= n_0$



Q.2 ⇒ T.C. of for (i=1 to n) { i=i*2 }

Series ⇒ 1, 2, 4, 8, 16, ---- n (G.P.)

$a = 1$, $r = 2$

$t_k = ar^{k-1}$ ⇒ $n = a \cdot 2^{k-1}$

⇒ $n = 2^{k-1}$

⇒ $2^k = 2n$

⇒ $k = 2 \log_2 n$

so T.C. ⇒ $O(\log_2 n)$

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

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

let $n = n-1$, $T(n-1) = 3T(n-2)$

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

or $T(n) = 3^3 T(n-3)$

or $T(n) = 3^n T(n-n)$

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

so T.C. ⇒ $O(3^n)$

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

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

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

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

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

let $n = n-2$, $T(n-2) = 2T(n-3) - 1$

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

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

or

$T(n) = 2^k T(n-k) - 2^{k-1} - 2^{k-2} \cdot \ldots - 2^1 - 2^0$

$T(0) = 1$, let $n-k=0$ so $k=n$

$T(n) = 2^n T(n-n) - 2^{n-1} - 2^{n-2} \ldots - 2^1 - 2^0$

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

$= 2^n - (2^{n-1} + 2^{n-2} + \ldots + 2^1 + 2^0) \}$ G.P

$T(n) = 2^n - \dfrac{1(2^n - 1)}{2-1} = \cancel{2^n} - \cancel{2^n} + 1$

so T.C. ⇒ $O(1)$

Q.5 ⇒

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

Series ⇒ 1, 3, 6, 10, 15, 21, 28 ------ n

1st iteration ⇒ s = s+1

2nd iteration ⇒ s = s+1+2

till ⇒ $1+2+3+\dots+k <= n$

$$\frac{k*(k+1)}{2} <= n$$

or $O(k^2) <= n$

or $k = O(\sqrt{n})$

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

Q6 ⇒ for (i=1 ; i*i <= n ; i++)

count ++

let loop run till k i=k

$k^2 <= n$

$k <= \sqrt{n}$

so T.C ⇒ $O(\sqrt{n})$

Q.7 ⇒ 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) $O(\log n)$

so T.C ⇒ $O(n \log^2 n)$

Q.8 ⇒ function (int n) {

if (n==1) return;

for (i=1 to n) {

for (j=1 to n) {

print ("*");

}

}

function (n-3);

}

Recurrence Relation $\Rightarrow$ $T(n) = T(n-3) + n^2$

or $T(n) = T(n-6) + 2*n^2$

$T(n) = T(n-9) + 3n^2$

or $T(n) = T(n-3k) + kn^2$

$T(1) = 0$, $n-3k = 1$ $\Rightarrow k = \frac{n-1}{3}$

so $T(n) = T(1) + \frac{(n-1)}{3} n^2$

so T.C. $\Rightarrow$ $O(n^3)$

Q.9 $\Rightarrow$

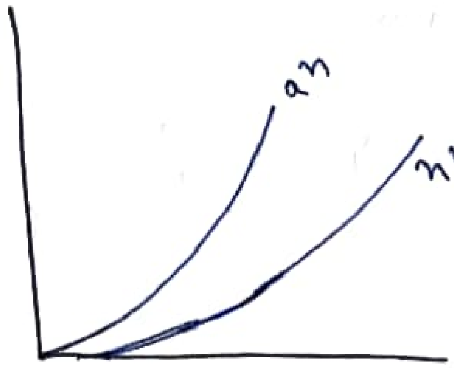
```
for (i=1 to n){
        for (j=1 ; j<=n ; j=j+i)
            printf ("*");
}
```

i	j	times
1	$1 \to n$	n
2	$1 \to n$	$\frac{n}{2}$
3	$1 \to n$	$n/3$
⋮	⋮	⋮
n	$1 \to n$	1
		$n \log n$

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

Q.14) Find asymptotic relation b/w n^k & a^n, $k>=1$ & $a>1$ are constants, find c & n_0 for which relations holds.

Sol)



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

$n^k \le a^n$, c $\forall$ $c>0$ & $n \geq n_0$

let $n = n_0$

$n_0^k \le c \cdot a^{n_0}$

[so let $k = a = 3$]

[$n_0^3 \le c \cdot 3^{n_0}$ so $c \geq 1$ & $n_0 \geq 1$]

Q.11 ⇒

```
void fun (int n){
    int i=0, j=1;
    while (i<n){
        i=i+j;
        j++;
    }}
```

Series ⇒ 0, 1, 3, 6, 10, 15 ...

let at last iteration :

$n = 0 + 1 + 2 + 3 + 4 + 5 + \dots + k$

$n = \frac{k(k+1)}{2}$

$n = \frac{k^2+1}{2}$

$n \cong k^2$

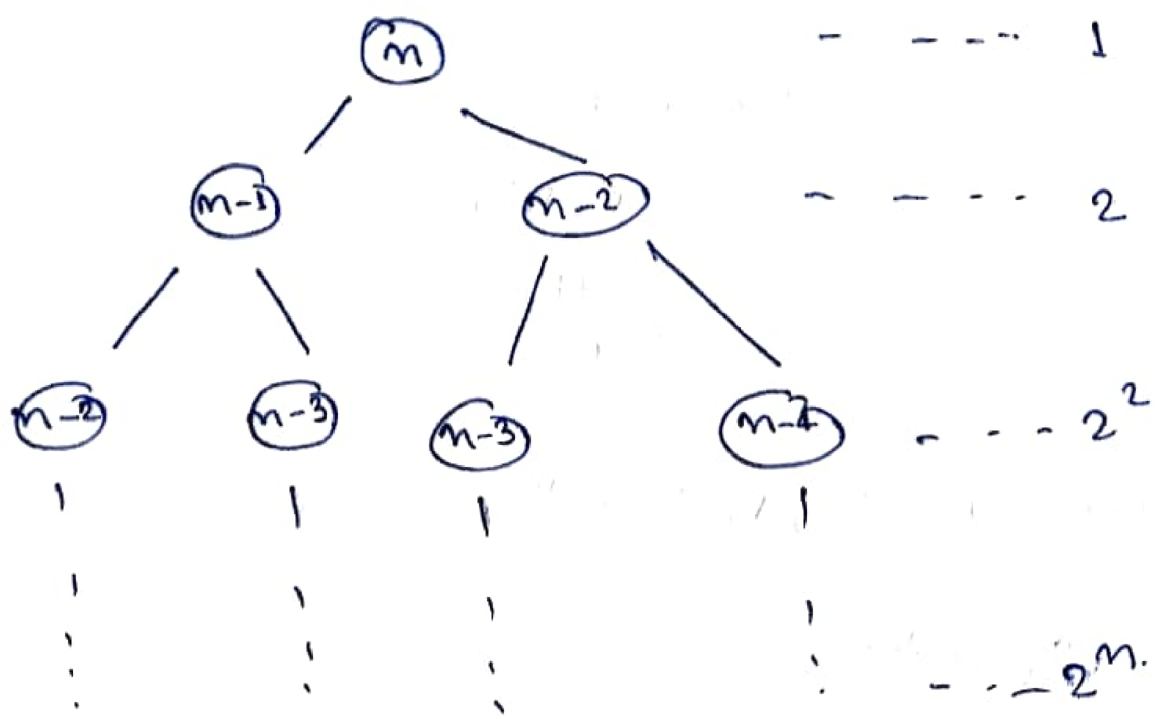
$k \cong \sqrt{n}$

so T.C. ⇒ $O(\sqrt{n})$.

Q.12 ⇒ Recurrence relation for fibonacci series..

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

using Recurrence tree method :



$$T.C. = 1 + 2 + 4 + \dots + 2^n = \frac{1\,(2^{n+1}-1)}{2-1} = 2^{n+1}-1$$

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

Space Complexity: Space complexity of fibonacci series using recursion is proportional to height of recurrence tree.

so $S.C. \rightarrow O(n)$

Q.13 ⇒ Write code for complexity.

(i) $n \log n$

```
for (i to n)
{
    for (j=1 , j<=n, j*=2)
        O(1) statements
}
```

(ii) n^3

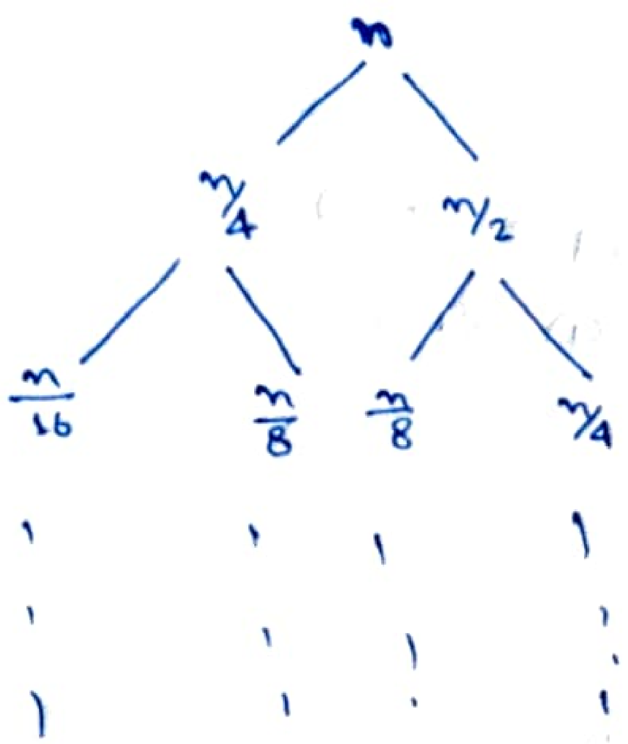
```
for (i to n)
    for (j to n)
        for (k to n)
            O(1) statements
```

(iii) $\log(\log n)$

~~for (int i=0 ; i<n ;~~

```
int i=n;
while (i>0)
{
    - - -}
    - - -}
    i = √i ;
}
```

Q.14) $T(n) = T(n/4) + T(n/2) + cn^2$



$\rightarrow cn^2$

$\rightarrow \frac{cn^2}{16} + \frac{cn^2}{4} = \frac{5cn^2}{16}$

$\rightarrow \frac{cn^2}{256} + \frac{cn^2}{64} + \frac{cn^2}{64} + \frac{cn^2}{16}$

$= \frac{25\,cn^2}{256}$

so $T(n) = cn^2 + \frac{5n^2}{16} + \frac{25n^2}{256} + \cdots)$

here $r = \frac{5}{16}$ so $S_n = \frac{1}{1-r}$

$T(n) = cn^2\left(1 + \frac{5}{16} + \frac{25}{256} + \cdots\right)$

$= cn^2\left(\frac{1}{1-\frac{5}{16}}\right) = cn^2 \times \frac{16}{11}$

so T.C $\Rightarrow$ $\Theta(n^2)$

Q.15 ⇒

```
int fun (int n)
{
    for (i to n)
        for (j=1 ; j<n ; j+=i) {
            O(1) task
    }
}
```

i	j	times	
1	1→n	n−1	
2	1→n	(n−1)/2	
3	1→n	(n−1)/3	
⋮	⋮	⋮	
n	1→n	n−1/n	
		n logn	

[T.C. ⇒ O(nlogn)]

Q16 ⇒ for (int i = 2; i <= n; i = pow(i, k))

{ O(1);

}

$i = 2, 2^{k}, 2^{k^2}, 2^{k^3} \ldots, 2^{k^x}$

$n = 2^{k^x}$

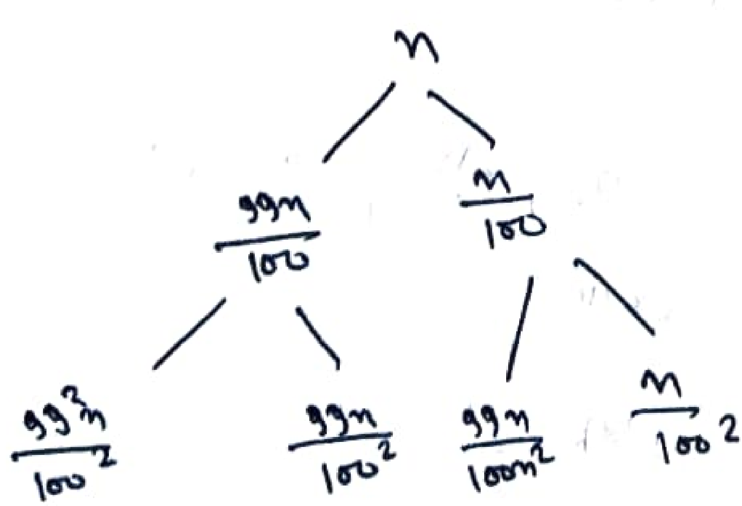
$\log n = k^x \log 2$

$\frac{\log \log n}{\log 2} = x \log k$

$x = \frac{\log \log n}{\log 2 + \log k}$

So T.C. ⇒ $O(\log \log n)$

Q17 ⇒ ~~$F(n) = T(n-1) + n$~~ $T(n) = T\left(\frac{99n}{100}\right) + T\left(\frac{n}{100}\right)$



If we take longer branch i.e. $\frac{99n}{100}$

T.C. $\Rightarrow \log_{\frac{100}{99}} n \simeq \log n$

$n = \left(\frac{99}{100}\right)^k$ $k = \log_{\frac{100}{99}} n$

$T(n) = n \left(\log \frac{100}{\frac{99}{100}}\right)^n = o(n \log_{99} n)$

Q18 $\Rightarrow$ Increasing of growth.

(a) $100 < \log\log n < \log n < \sqrt{n} < n < n\log n < n^2 < 2^n < 2^{2n} < 4^n < n!$

(b) $1 < \log\log n < \sqrt{\log(n)} < \log n$ ~~$< 2n < 4n < 2(2^n) < \log$~~ $< \log 2n < 2\log n < n < 2n < 4n < n^2\log n < n^2 < \log(n!) < 2^{2n} < n!$

(c) $96 < \log_8 n <$ ~~$\log$~~ $\log_2 n < 5n < n\log_8(n) < n\log_2 n < 8n^2 < 7n^3 <$ $\boxed{\cancel{8^n(2n)} \; \cancel{< \log(n!)} \; < n}$. $\log n! < 8^{2n} < n!$

Q.19 ⇒ Linear Search :

```
for (i=0 to k-1)
{   if ( ar[i] = key)
      { return i ;
      }
    return -1 ;
}
```

Q.20 ⇒ **Iterative Insertion Sort :**

~~void insertion_sort (arr , n)~~

~~loop from i=1 to n-1~~

~~pick element ar[i] & insert it into sorted into sorted sequence.~~

```
void insertion_sort ( int arr[] , int n)
{  int i, temp, j ;
   for i ← 1 to n
   {   temp ← arr[i] ;
       j ← i-1 ;
       while (j>=0 AND arr[j] > temp)
       {      arr[j+1] ← arr [j] ;
```

```
            j ← j-1 ;
        }
        arr [j+1] ← temp ;
    }
}
```

Recursive Insertion sort :-

```
void  recursive_insertion_sort ( int arr[] , int n)
{
      if (n <=1)
          return
      recursive_insertion_sort (arr , n-1)
      val = arr [n-1]
      pos = n-2
      while ( pos >=0  && arr [pos] > val) {
              arr[pos +1] = arr [pos]
              pos = pos -1
      }
      arr [pos+1] = val
}.
```

It is called online sorting because it provided ~~one~~ one sorted element at a time & ~~sequence of sorted as consider~~, produces a partial solution without considering future elements.

Q.21=>

Algorithm	Time complexity Best case	Average Case	Worst case
① Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
② Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
③ Merge sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
④ Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
⑤ Quick sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$
⑥ Heap sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$

Q.22=>

Algorithm	Inplace	Stable	Online Sorting
Bubble Sort	✓	✓	✗
Selection Sort	✓	✗	✗
Merge Sort	✗	✓	✗
Insertion Sort	✓	✓	✓
Quick Sort	✗	✗	✗
Heap Sort	✓	✗	✗

Q.23 ⇒ **Recursive Binary Search:**

```
int b_search ( int arr [], int l, int r, int x).
{   if (l>r)
        return -1;
    int m= (l+r)/2
    if (arr[m] = x)
            return m;
    else if (arr [m] <x)
            b_search (arr, m+1, r, x);
    else
            b_search (arr, l, m-1, x
}
```

Iterative Binary Search:

```
int binarysearch (int arr [], int n, int x)
{
    l=0 , r= n-1;
    while ( l<r)
    {   m= (l+r)/2
        if (arr [m] = x)        return m;
        else if ( arr [m] <x)       l= m+1;
        else       r= m-1;
    }
```

return -1 ;

}

Time & Space Complexity of Iterative Binary search ⇒ $O(\log n)$ & O(

Time & Space Complexity of Recursive Binary search ⇒ $O(\log n)$, $O(\log n)$

Q24⇒ Recurrence Relation for Binary Search ⇒

$$T(n) = T(n/2) + 1$$