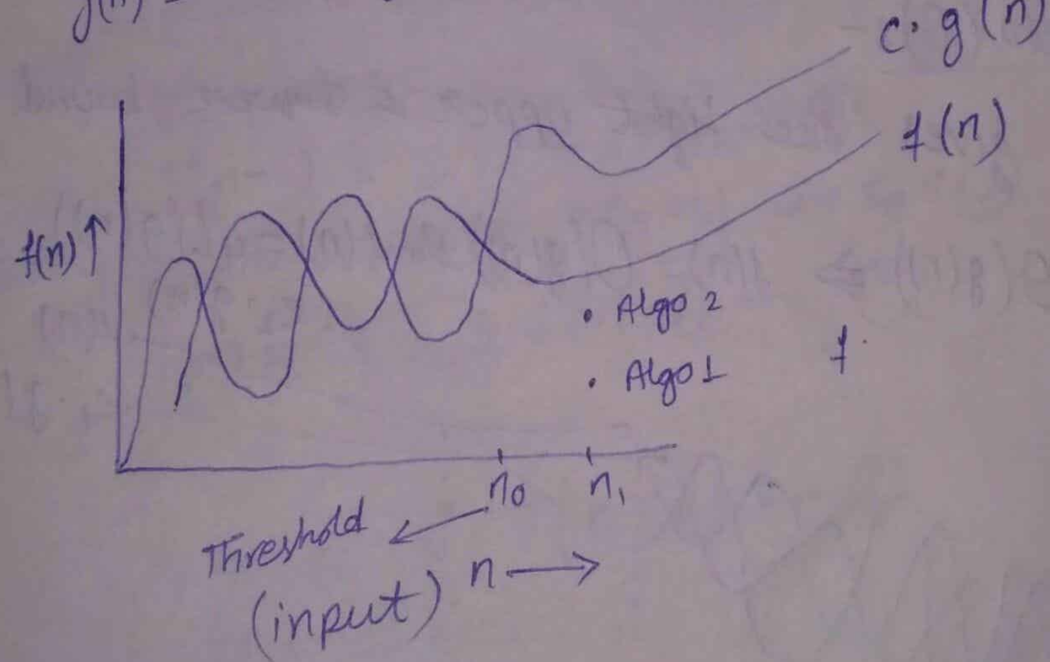
① → When we have analyse the complixity of any algorithm in term of time and space, we never provide an exact number to define the time required and the space required by algorithm, so, we use Asymptotic notation.

The three types of Asymptotic notation are:-

<1> **Big Oh(O)** :-

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

$g(n)$ = is "tight" upper bound of $f(n)$



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

if only if

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

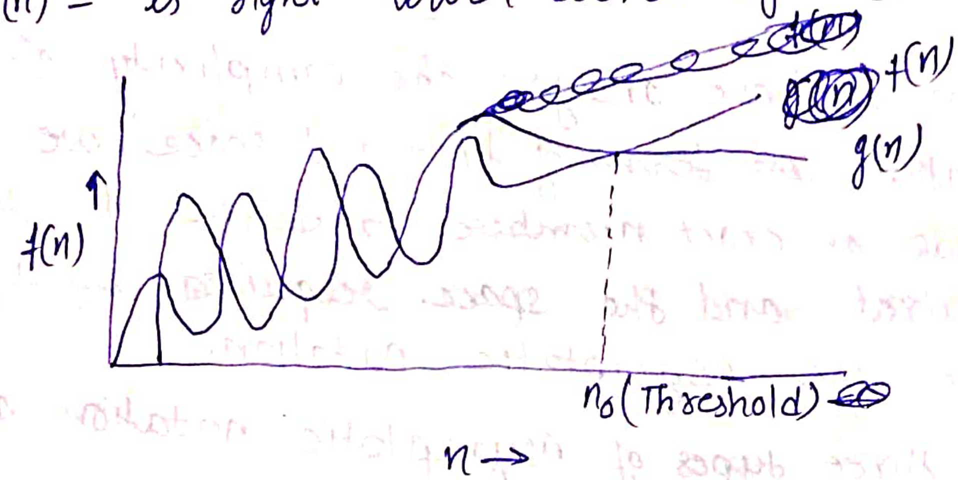
$\forall\, n \geq n_0$, some constant c, Ex:- Algo1 $\rightarrow O(n^2+3n+4) \longrightarrow O(n^2+n) \rightarrow O(n^2)$

Algo2 $\rightarrow O(2n^2+4) \longrightarrow O(n^2)$

(ii) <u>Big Omega (Ω):-</u>

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

$g(n)$ = is "tight" lower bound of $f(n)$.



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

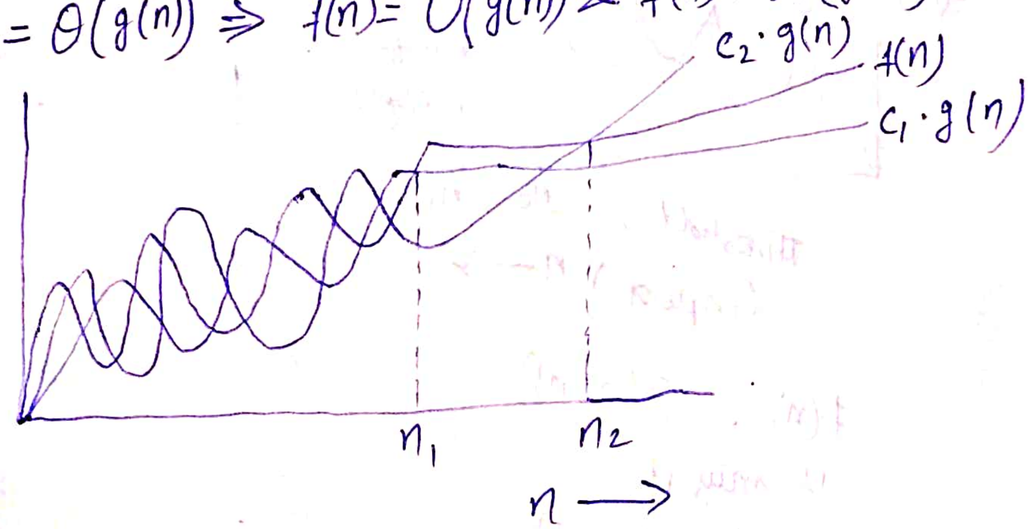
$f(n) \geq c \cdot g(n)$

$\forall\, n \geq n_0$ & some const $c > 0$

(iii) <u>Big Theta (Θ):-</u>

Theta gives the tight upper & lower bound both.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$



$f(n) = \Theta(g(n))$ if only if

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

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

② → for ($i = 1$ to n)
{ $i = i*2$
}

$1, 2, 3, \ldots n$

$n = 1 + (k-1) \cdot 1$

$n = k - 1 + 1$

$k = n$

Time complexity $= O(n)$.

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

base case $T(1) = 1$

$T(n) = 3T(n-1)$ —— ①

put $n = n-1$

$T(n-1) = 3T(n-2)$ —— ②

put the $T(n-1)$ in Eq^n ①

$T(n) = 3 \cdot 3 \cdot T(n-2)$ —— ③

Now put $n = n-2$ in Eq^n ①

~~T(n) = 3~~

$T(n-2) = 3 \cdot T(n-3)$ —— ④

put the value of $T(n-2)$ in Eq^n ③

$T(n) = 3 \cdot 3 \cdot 3 \cdot T(n-3)$ —— ⑤

⋮

$T(n) = 3^k \cdot T(n-k)$ —— ⑥

$T(1) = 1$

$\therefore$ $n - k = 1$

$k = n-1$

$T(n) = 3^{n-1} \cdot T(n-n+1)$

$T(n) = \frac{3^n}{3} \cdot T(1) \Rightarrow T(n) = \frac{3^n}{3} \cdot 1$

$O(n) = 3^n$ Ans

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

$T(1) = 1$ → base condition.

$T(n) = 2T(n-1) - 1$ —— (1)

put $n = n-1$ in Eq^n (1)

$T(n-1) = 2T(n-2) - 1$ —— (2)

putting $T(n-1)$ in Eq^n (1)

$T(n) = 2\{2 \cdot T(n-2) - 1\} - 1$

$T(n) = 2 \cdot 2 \cdot T(n-2) - 1 - 2$ —— (3)

Now, $n = n-2$ in Eq^n (1)

⋮

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

$T(n-2) = 2T(n-3) - 1$ —— (4)

put the value of $T(n-2)$ in Eq^n (3)

$T(n) = 2 \cdot 2 \cdot \{2T(n-3) - 1\} - 1 - 2$

$T(n) = 2 \cdot 2 \cdot 2 \cdot T(n-3) - 1 - 2 - 4$ —— (5)

⋮

$T(n) = 2^k T(n-k) - \{\underbrace{1 + 2 + 4 + \ldots 2^{k-1}}_{GP.}\}$

$r > 1$

Now $n - k = 1$

$k = n - 1$

$T(n) = 2^{n-1} T(1) - \{\frac{1(2^k - 1)}{2-1}\}$

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

$T(n) = O(1)$ Ans.

(5) →
```
int i=1, s=1;
while(s<=n)
{
    i++;
    s=s+i;
    print("#");
}
```

$T(n) = 1+2+3+\dots+n.$

$T(n) = \frac{n*(n+1)}{2} = \frac{n^2+n}{2} = O(n^2)$

(6) → $T(n) = 1+2+3+\dots+\sqrt{n}$

$\therefore \sqrt{n} = a+(n-1)d$

$\sqrt{n} = 1+(k-1)1$

$\sqrt{n} = k-1+1$

$k = \sqrt{n}$

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

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

$\therefore$ Time complexities $\to O\left(\frac{n}{2} * (\log n)^2\right)$

$\to O\left(n(\log n)^2\right)$

(8) → function(int n) {

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

Let T(n) be the time complexity of this function

$$T(n) = T(n-3) + n^2 \quad \text{—— (1)}$$
$$T(1) = 1$$

~~let~~ put n = n-3 in Eqn (1)

$$T(n-3) = T(n-6) + n^2 \quad \text{—— (2)}$$

putting T(n-3) in Eqn (1)

$$T(n) = T(n-6) + n^2 + n^2 \quad \text{—— (3)}$$

putting n = n-6 in Eqn (1)

$$T(n-6) = T(n-9) + n^2 \quad \text{—— (4)}$$

putting the value of T(n-6) in Eqn (3)

$$T(n) = T(n-9) + n^2 + n^2 + n^2 \quad \text{—— (5)}$$

⋮

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

∴ $n-3k = 1$

$k = \frac{n-1}{3}$

$$T(n) = T\left(n - ~~3~~n + 1\right) + \frac{(n-1) \times n^2}{3}$$

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

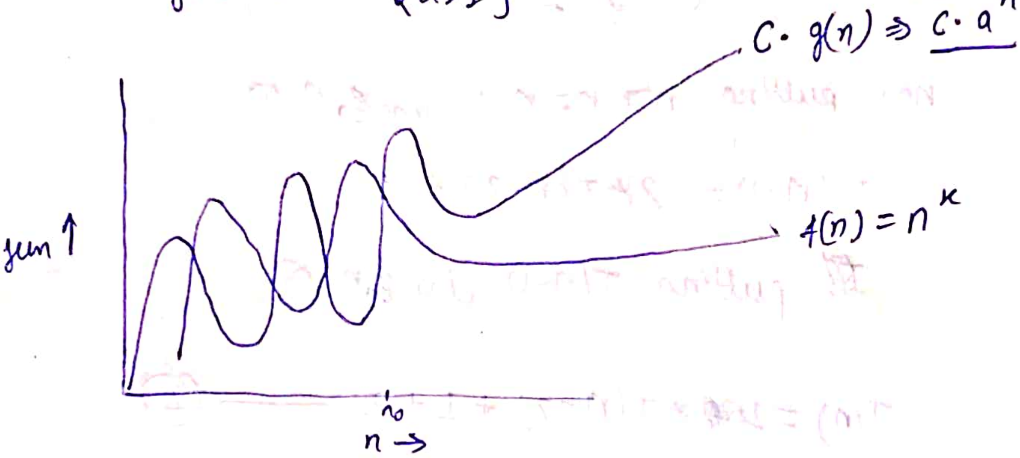
$\Rightarrow O(n^3)$

(9) → void function (int n)

```
for(i=1 to n) {          → O(n)
  for (j=1; j<=n; j=j+1). → O(n)
     print("*")
  }
}.
```

$\therefore$ Time complexity → $O(n^2)$.

(10) → $f(n) = n^k$ $\{k \geq 1\}$ ⎫ → are constant.
$\&\ g(n) = a^n.$ $\{a > 1\}$ ⎭



$\therefore$ $f(n) = O(g(n))$

only if

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

$n^k \leq c \cdot a^n \quad \forall\ n \geq n_0,$

some constant $c > 0$

(11) → void fun (int n) {

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

Time comp $= 1 + 2 + 3 + \dots + n$

$= O(n)$.

(12) $\rightarrow$

```
int ~~fib~~ fib(int n)
{
    if(n<=1)
        return;
    else
        return fib(n-1) + fib(n-2);
}
```

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

let assume $T(n-1) \approx T(n-2)$

$$T(n) = 2*T(n-1)+1 \quad \text{— (1)}$$

now putting $n=n-1$ in Eq^n (1)

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

putting $T(n-1)$ in Eq^n (1)

$$T(n) = 2*2*T(n-2) + 1 + 2 \quad \text{— (2)}$$

now putting $n=n-2$ in Eq^n (1)

$$T(n-2) = 2*2*T(n-3)+1$$

putting $T(n-2)$ in eq^n (2)

$$T(n) = 2*2*2*T(n-3) + 1+2+4$$

$\vdots$

$$T(n) = 2^k*T(n-k) + 1+2+\dots+2^{k-1}$$

$n-k=0$

$k=n$

$n-1$

$$T(n) = 2^{n-1}*T(1) + (2^k-1)$$

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

~~T(n) = 2^n~~ $O(2^n)$

(13) → (i) $n(\log n)$

```
int sum=0;
for(int i=1; i<=n; i*=2)
 {
    for (int j=1; j<=n; j+=2)
     {
        sum+=j;
     }
 }
```

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

(ii) n^3

```
int sum=0;
 for(int i=1; i<=n; i++){
   for(int j=1; j<=n; j++){
     for(int k=1; k<=n; k++){
        sum+=k;
     }
   }
 }
```

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

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

c is constant

```
for(int i=2; i<=n; i=pow(i,c)) {
    //O(1) expression
}
```

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

$T(\frac{n}{2}) > T(\frac{n}{4})$

$\Rightarrow$

$T(n) <= 2T(\frac{n}{2}) + cn^2$

Now from master's method.

$T(n) = aT(\frac{n}{b}) + f(n)$

$\therefore c = \log_b a = \log_2 2 = 1$

$n^c = n^1$

$f(n) = cn^2$

$\therefore f(n) > n^c$

$\Rightarrow T(n) = \theta(n^2)$

(15) →

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

$\frac{n-1}{1} + \frac{n-1}{2} + \frac{n-1}{3} + \cdots + \frac{n-1}{n-1} + 1$

$\frac{n}{1} + \frac{n}{2} + \frac{n}{3} + \cdots \frac{n}{n-1} - \log(n-1)$

$n\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1}\right\} - \log(n-1)$

$n\log(n-1) - \log(n-1)$

$n\log(n-1)$

$\underline{n \log n}$

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

{

// O(1) exp.

}

$\Rightarrow \quad 2^1, 2^k, (2^k)^k \ldots$

$2^{k^2}, (2^{k^2}) = 2^{k^3}, \ldots, 2^{k \log_k(\log(n))}$

$2^{k \log_k(\log(n))} = 2^{\log(n)} = n$

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

(18) → a) → $100 < \log\log n < \log n < root(n) < \log(n!) < n <$

$n \log n < n^2 < 2^n < 2^{2n} = 4^n$

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

$< n \log n$

c) → $96 < \log_8(n) < \cancel{\log_2 n} \; n\log_6(n) < \log_2 n < n\log_2 n < 5n <$

$8n^2 < 7n^3 < 8^{(2n)}$

(19) →

```
linearSearch(arr, n, key)
  if(arr[n-1]==key)
    return n;
  temp = arr[n-1]
  arr[n-1] = key.
  for(i=0, i++)
    if(arr[i]==key)
    {
      arr[n-1]= temp;
      return (i<n-1)
}
```

(20) →

```
Insertionsort(int arr[], int n)
{ for i=1 to i<n
 {int temp = arr[i];
  int j=i
  while (j>0 && arr[j-1] > temp)
        arr[j]=arr[j-1]
        j--
  arr[j] = value
 }
}
```

Iteration Insertion Sort

```
Insertionsort(int arr[], int i, int n)
{
    int temp = arr[i];
    int j = i
    while (j>0 && arr[j-1]>temp)
        arr[j] = arr[j-1]
        j--
    arr[j] = temp;

    if (i+1 <= n)
        insertionsort(arr, i+1, n);
}
```

Recursive Insertion sort

Insertion sort is an online algorithm which processes its input piece-by piece in a serial way, i.e. in the order that the input is fed to the algorithm, without having the entire input available from the begining.

Selection Sort → offline
Merge Sort → offline

(21) →

	Best	Average	Worst
Bubble Sort →	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort →	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort →	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort →	$O(n\log(n))$	$O(n\log n)$	$O(n\log n)$

(22) →

	Bubble	Quick	Selection	Insertion	Merge
Not Inplace / Inplace	Inplace	Inplace	Inplace	Inplace	Not Inplace
Instable / Stable	Stable	Instable	Instable	Stable	Stable
offline / online			Online	Online	Offline

(23) → Iterative Binary search

```
int binarySearch( int arr[], int l, int r, int x)
{ while (l<=r) {
     int m = (l+r)/2;
      if (arr[m]=x)
         return m;

      if (arr[m]<x)
          l = m+1;

      else
          r = m-1;
  }
  return -1;
}
```

Recursive Binary Search

```
int binarysearch(int arr[], int l, int r, int x)
{
    if(r>=l) {
      int mid = (l+r)/2;

      if (arr [mid] == x)
         return mid;
      elseif (arr[mid] > x)
         return binarysearch(arr, l, mid-1, x);
      else
         return binarysearch (arr, mid+1, r, x);
    }
    return -1;
}.
```

Recursive Binary Search

Time Complixity → $O(\log_2 n)$

Space Complixity → $O(\log_2 n)$

Iterative Binary search

Time → $O(\log_2 n)$

space → $O(1)$

Recursive linear Search

Time → $O(n)$

Space → $O(n)$

Iterative ~~Binary~~ Linear search

T → $O(n)$

Space → $O(1)$

24 $T(n) = T(\frac{n}{2}) + 1.$