Design & Analysis of Algorithm

Assignment - 1

Name: Muskan Agrawal

Sec: CST

Uni Rno: 2014419

Class Rno: 33

1. Asymptotic Notation:

Asymptotic Notation are methods / languages using which we can define the running times of the algorithm based on input size. The notation are used to tell the complexity of an algorithm when the input is very large.

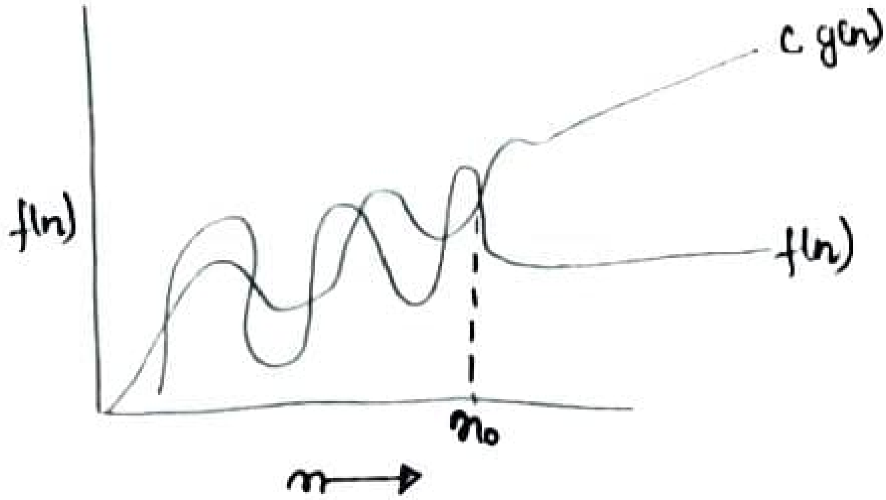
Suppose, we have an algo as a funcn 'f' and 'n' as input size, f(n) will be running time of algo using this we make a graph with y-axis as running time - (f(n)) and x-axis as input size (n)

The different Asymptotic Notations are :-

a) Big - O Notation:

It is an asymptotic Notation for worst case or the ceiling growth for a given function.

$f(n) = O(g(n))$, where $g(n)$ is tight upper bound of $f(n)$.



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

if

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

$\forall\ n \ge n_0$, some constant $c > 0$

Eg $f(n) = n^2 + 3n + 4 = O(n^2)$

$g(n) = n^2$

let $c = 100$

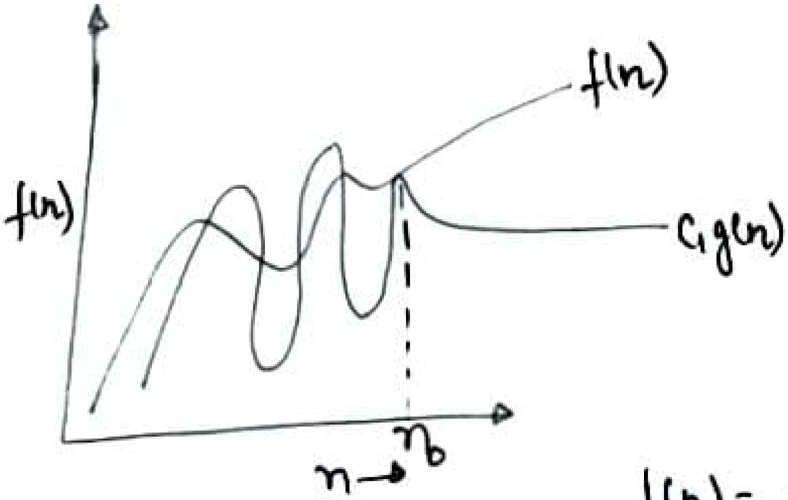
$n^2 + 3n + 4 \le 100 \cdot n^2$

$\forall\ n \ge 1$

b) **Big-omega (Ω):-**

It is the asymptotic notation for the best case or a floor growth rate for a given function.

$f(n) = \Omega(g(n))$, where $g(n)$ is tight lower bound of $f(n)$.



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

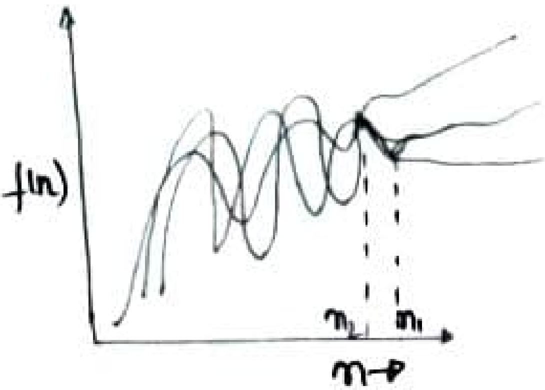
iff $f(n) \ge c\,g(n)$

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

c) **Theta (θ):-**

It is an asymptotic notation to denote the asymptotically tight bound on the growth rate of runtime of an algorithm.

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



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

iff $c_1 g(n) \le f(n) \le c_2 g(n)$

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

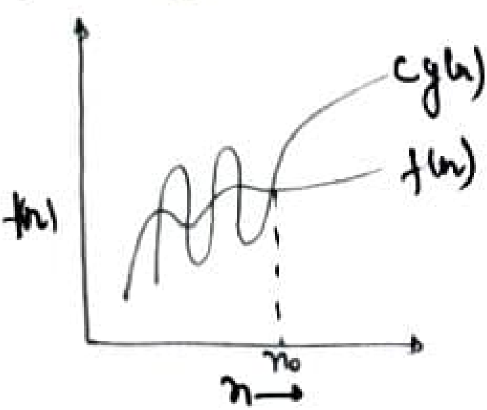
$c_1, c_2 > 0$

d) **Small-o:-**

Denote the upper bound (not tight) on the growth rate of runtime of an algorithm

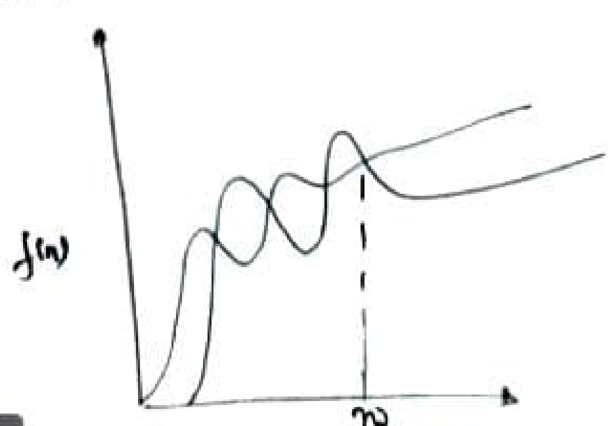
$f(n) = o(g(n)) \rightarrow f(n) < c\,g(n)\ \forall\ n > n_0$ & $c > c_0$

Eg $n = o(n^2)$



e) **Small-omega (ω):-**

denotes the lower bound.



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

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

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

Eg $n^2 = \omega(n)$

Ans2:- for(i=1 to n){
$i = i*2$
}

$i = 1, 2, 4,$ $n \rightarrow GP$

$t_k = a r^{k-1}$ $[a=1, r=2]$

$n = 1 \; 2^{k-1}$

$\log_2 n = (k-1)$

$k = \log_2 n + 1$

$T.C = O(\log_2 n + 1)$

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

Ans3:- $T(n) = \{3T(n-1)$ if $n>0$, otherwise $1\}$

$T(n) = 3T(n-1)$ —(i)

$T(0) = 1$

$n \rightarrow n-1$ in eqn (i)

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

$T(n-1) = 3T(n-2)$ —(ii)

put value of T(n-1) from (i) to eqn (ii)

$T(n) = 9T(n-2)$ —(iii)

$n \rightarrow n-2$ in eqn (iii)

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

$T(n-2) = 3T(n-3)$ —(iv)

put value of T(n-2) from eqn (iv) to (iii)

$T(n) = 27T(n-3)$ —(v)

$\rightarrow T(n) = 3^k T(n-k)$

$n-K=0 \rightarrow n=K$

$\rightarrow T(n) = 3^n T(n-n)$

$= 3^n T(0)$

$= 3^n$

$\underline{T.C = O(3^n)}$

Ans 9:- $T(n) = \begin{cases} 2T(n-1)-1 & \text{if } n>0 \\ 1 & \text{, otherwise} \end{cases}$

$T(n) = 2T(n-1)-1$ — (i)

$T(0) = 1$

put $n \rightarrow n-1$

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

$T(n-1) = 2T(n-2)-1$ — (ii)

from (i) & (ii) —

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

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

$T(n) = 4T(n-2)-2-1$ — (iii)

put $n \rightarrow n-2$ in eq^n (i)

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

$T(n-2) = 2T(n-3)-1$ — (iv)

from (iii) & (iv) —

$T(n) = 4[2T(n-3)-1]-2-1$

$T(n) = 8T(n-3)-4-2-1$

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

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

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

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

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

$= 1$

$T(n) = O(1)$

Ans5:-
```
int i=1, s=1;
while (s<=n) {
        i++;
      s=s+i;
   printf("#");
}
```

We can define the term 's' a/c to relation $S_i = S_{i-1}+1$. The value of 'i' increases by one for each iteration. The value contained in 's' at the i^{th} position iteration is the sum of the first 'i' the integers

If K is total no. of iterations taken by the program. then while loop terminates if:-

$1+2+3+ \cdots K = \frac{K(K+1)}{2} > n$

$\Rightarrow K = O(\sqrt{n})$

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

Ans6:-
```
void function (int n) {
    int i, count=0;
  for (i=1, i*i<=n ; i++)
          count++;
}
```

loop ends if $i^2 > n$

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

Ans7:-
```
void function (int n) {
   int i, j, k, count=0;
   for (i = n/2 ; i<=n ; i++) — n
       for (j=1 ; j<=n ; j=j*2)
```

for (k=1; k<=n; k=k*2)] execute log n times

count++;

Time complexity = $O(n\log^2 n)$

Ans 8:- void function (int n)

{

if (n==1) return, → constant time

for (i=1 to n){ ⟶ n times

for (j=1 to n) { → n times

printf ("*");

}

function(n-3);

}

Recurrence rel^n : $T(n) = T(n-3) + cn^2$

$\Rightarrow T(n) = \theta(n^3)$

Ans 9:- void function (int n) {

for (i=1 to n) { ⟶ This loop execute n times

for(j=1; j<=n; j=j+i) → This executes j times with j increase by the rate of i.

printf ("*");

}

}

⇒ Inner loop executes n/i times for each value of i.

Its running times is $n \times \left(\sum_{i=1}^{n} n/i\right)$

$= O(n\log n)$

Ans 10:- The asymptotic relationship b/w the functions n^k and a^n is

$$n^k = O(a^n) \qquad k >= 1,\ a > 1$$

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

$$\rightarrow \frac{n^k}{a^n} \leq c$$

Ans 11:- Same as ques 5.

i is increasing at the rate of j

$\Rightarrow$ If k is total no. of iterations, while loop terminates if,

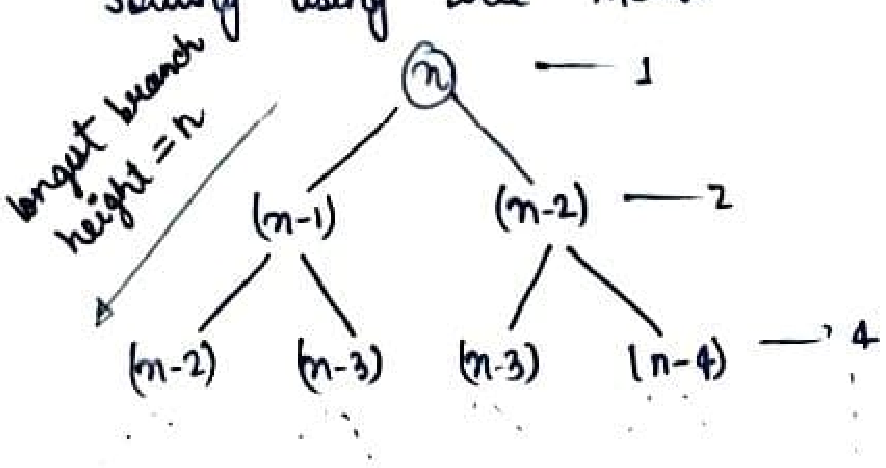
$$0 + 1 + \cdots + k = \frac{k(k+1)}{2} > n$$

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

Ans 12:- The recurrence relation for the recursive method of fibonacci series is -

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

Solving using tree method -



$$T.C = 1 + 2 + 4 \cdots + 2^n \qquad 2^n$$

$a=1,\ r=2 \qquad S = \dfrac{a(r^{\text{terms}}-1)}{r-1}$

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

$T.C = O(2^{n+1}) = O(2 \cdot 2^n) = O(2^n)$

Space complexity $= O(n)$ [∵ Stack size never exceeds the depth of the call's tree shown above]

Ans13:- Program with complexity –

i) $n \log n$ –

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

ii) n^3

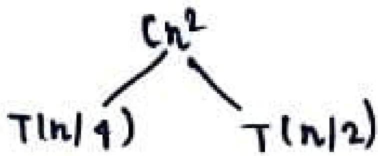
```
void function(int n){
    for(i=1; i<=n; i++){
        for(int j=1; j<=n; j++){
            for(int k=1; k<=n; k++){
                printf("#");
            }
        }
```

iii) $\log(\log n) \rightarrow$ for (int i = 2; i <= n; i = pow(i, k)) { //O(1) };

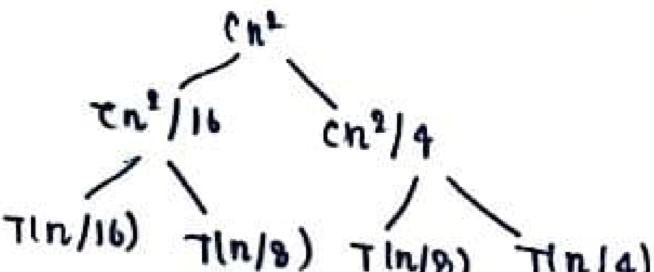
also, interpolation search has this complexity

Ans14:- $T(n) = T(n/4) + T(n/2) + cn^2$

Following is the initial recursion tree,



on further breaking down,



To know the value of T(n) we need to calculate the sum of tree nodes level by level.

$\Rightarrow T(n) = cn^2 + 5n^2/16 + 25n^2/256 + \dots$

GP with ratio $5/16$

$S_\infty = \dfrac{n^2}{1 - 5/16} \Rightarrow T.C = O(n^2)$

Ans15:- Same as ques 9

$\rightarrow O(n \log n)$

Ans16:-

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

In this case i takes value $2, 2^k, (2^k)^k, (2^{k^2})^k = 2^{k^3}$.

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

The last term must be less than or equal to n, we have,

$$2^{k^{\log_k(\log(n))}} = 2^{\log n} = n \text{, It's True}$$

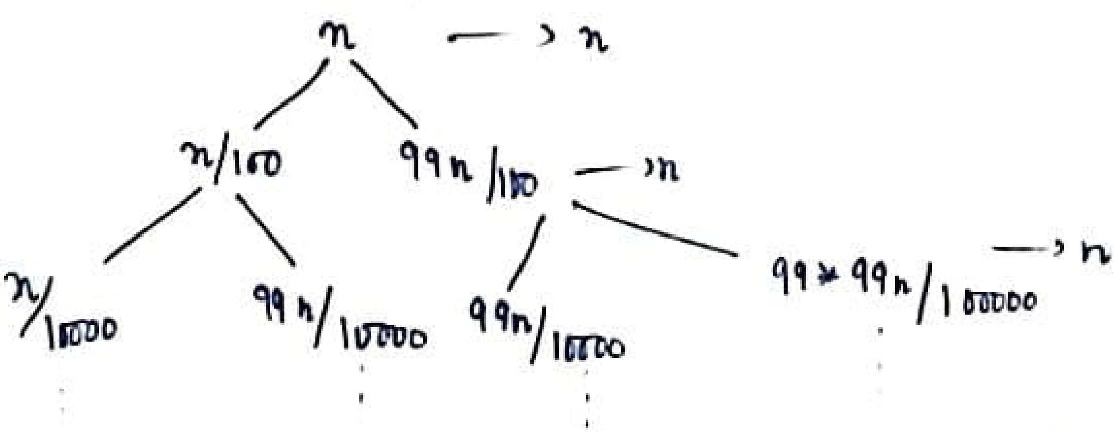
∴ There are total $\log_k(\log(n))$ many iterations and each iteration takes constant amount of time to run,

∴ Total times complexity $= O(\log(\log n))$

Ans17:- The running time when in quick sort when the partition is putting 99% of elements on one side and 1% elements on another in each repetition

$$T(n) = T\left(\frac{99n}{100}\right) + T\left(\frac{n}{100}\right) + cn$$

Recursion tree of the above equation is,



We can see that initially, the cost is n for all levels, This will follow untill the left most branch of the tree reaches its base case (size 1) because the left most branch has least elements in each division, so it'll finish first

. The rightmost branch will reach its base case at last because it has maximum no. of elements in each division

At level i, the rightmost node has $n*\left(\frac{99}{100}\right)^i$ elements,

for the last level,

$$n*\left(\frac{99}{100}\right)^i = 1$$

$$\rightarrow i = \cancel{\frac{\log_{100} n}{n}} \quad \log_{\frac{100}{99}} n$$

So, there are total $(\log_{\frac{100}{99}} n)+1$ levels

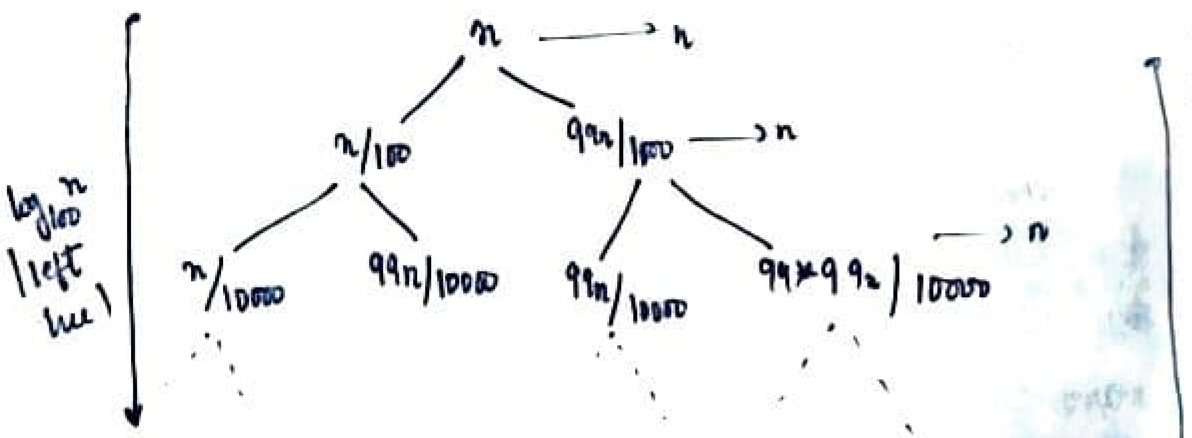
Thus,

$$T(n) = \Big(\underbrace{cn + cn + \cdots \le (cn) + \le (cn)}_{\log_{\frac{100}{99}} n \,+\, 1 \text{ times}}\Big) \le \left(\log_{\frac{100}{99}} n + 1\right) cn$$

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

$\left(\log_{\frac{100}{99}} n = \frac{\log_2 n}{\log_2 \frac{100}{99}}\right)$ Ignoring constant term by $\log \frac{100}{99}$

$$\Rightarrow T(n) = O(n \log n)$$



Starting with subproblem of size 1 & multiplying it by 100 until we reach relation

$$100^x = n$$

$$\rightarrow x = \log_{100} n$$

Right child is $\frac{99}{100}$ of size of nodes above ~~the size of nodes~~ it. Each parent is $\frac{100}{99}$ times the size of right child.

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

Ans18:- a) Increasing order of rate of growth —

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

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

c) $96 < \log_8 n < \log_2 n < 5n < \log(n!) < n\log_6(n) < n\log_2 n < 8n^2 < 7n^3 < n! < 8^{2n}$

Ans19:- Linear search in a sorted array with minimum no of comparisons —

```
int linearSearch (int A[], int n, int data){
    for i=0 to n-1 {
        if (A[i] = = data)
            return i;
    else if (A[i] > data)   // array is sorted if
        return -1;          A[i] > data then, no
                            need to search the rest
                            of the array
```

T.C ⇒ Best = O(1), Avg., worst = O(n)

Space = O(1)

Ans20:- Pseudo code for iterative insertion sort —

```
void insertionSort (int arr[], int n) {
    int i, temp, j;
    for i ← 1 to n
    {
        temp ← arr[i];
```

```
        j ← i-1;
    while (j>= 0 && arr[j] > temp){
            arr[j+1] = arr[j];
        }   j ← j-1;
    }   arr[j+1] = temp;
}
```

Pseudo code for recursive insertion Sort -

```
void insertionSortRecursive (int arr[], int n) {
        if (n <= 1)
            return;
        insertionSortRecursion (arr, n-1);
        int last = arr[n-1];
        int j = n-2;
    while (j >= 0 && arr[j] > last){
            arr[j+1] ← arr[j];
            j ← j-1;
        }
        arr[j+1] ← last;
}
```

An online sorting algo is one that will work if the elements to be sorted are provided one at a time with the understanding that the algo must keep the sequence sorted as more & more elements are added in. Insertion sort considers one input element per iterations & produces a partial solution without considering future elements. Thus insertion sort is online.

Other algo like selection sort repeatedly selects the minimum element from the unsorted array & places it at the first which requires the entire input. Similarly bubble, quick & merge sorts also require the entire input. Therefore they are offline algo

Ans 21,22:-

Sorting algo	Time Best	Avg	Worst	Space Worst	Stable	Inplace
Bubble	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	✓	✓
Selection	$O(n^2)$	"	"	"	✗	✓
Insertion	$O(n)$	"	"	"	✓	✓
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	✓	✗
Quick	$O(n \log n)$	"	$O(n^2)$	$O(n)$	✗	✗
Heap	$O(n \log n)$	"	$O(n \log n)$	$O(1)$	✗	✓

Ans23:- Iterative pseudo code for binary search:-

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

Time Complexity - Best time case: $O(1)$

Avg, worst: $O(\log_2 n)$

Space: $O(1)$

Binary search recursive code:-

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

T.C ⇒ Best: $O(1)$ & Avg, worst = $O(\log_2 n)$

Space Comp. ⇒ Best: $O(1)$

Avg, worst $O(\log_2 n)$

Ans24:- Recurrence relation for binary search

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