DESIGN AND ANALYSIS OF ALGORITHMS
ASSIGNMENT : 1

1. Asymptotic Notations are methods / languges using which we can define the running time of the algorithm based on input size. These notations are used to tell the complexity of an algorithm when the input is very large.

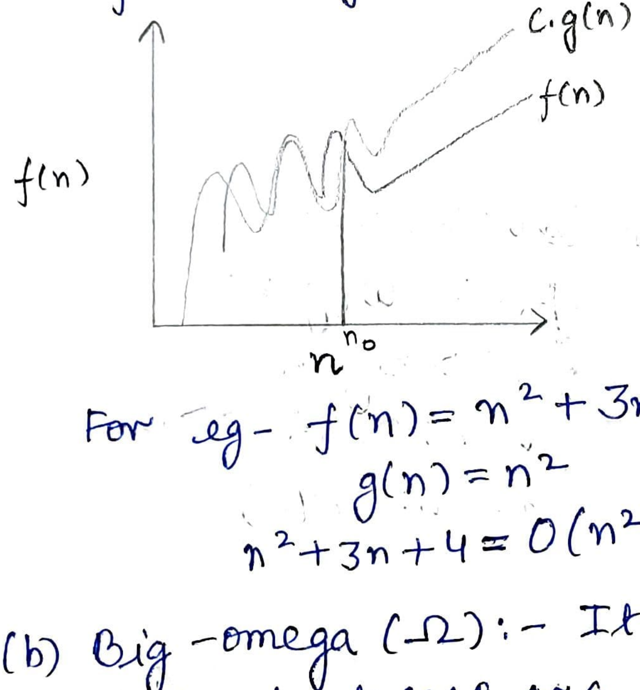
Suppose we have an algorithm as a function f and n as the input size, f(n) will be the running time of the algorithm. Using this we make a graph with y-axis as running time (f(n)) and x-axis as input size(n).

The different asymptotic notations are –

(a) Big-O notation

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

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



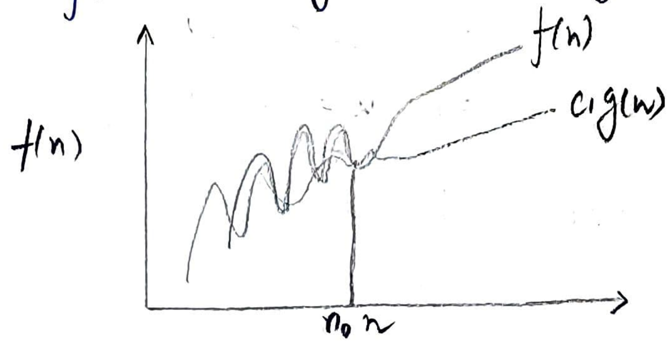
$f(n) = O(g(n))$
iff $f(n) \leq c.g(n)$
$\forall\ n \geq n_0$, some constant $c > 0$

For eg- $f(n) = n^2 + 3n + 4$
$g(n) = n^2$
$n^2 + 3n + 4 = O(n^2)$

Let $c = 100$.
$n^2 + 3n + 4 \leq 100.n^2$
$\forall\ n \geq 1$.

(b) Big-omega (Ω):- It is the asymptotic notation for the best case or a floor growth rate for a given fn.

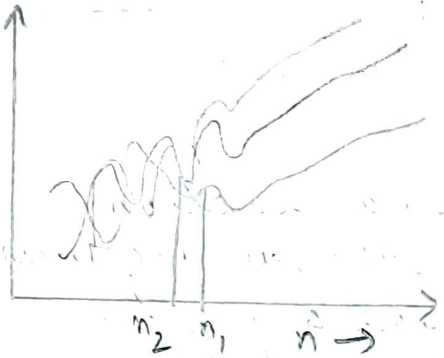
$f(n) = \Omega(g(n))$ where $g(n)$ is tight lower bound of $f(n)$.



$f(n) = \Omega(g(n))$
iff $f(n) \geq c\,g(n)$
$\forall\ n \geq n_0$ & $c > 0$

(c) Theta(Θ) is an asymptotic notation to ~~dentoe~~ 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 $c_1 g(n) \leq f(n) \leq c_2 g(n)$

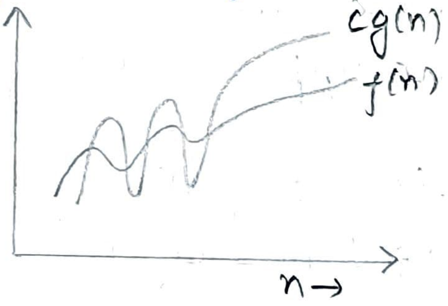
$\forall\ n \geq \max(n_1, n_2)$,

$c_1, c_2 > 0$

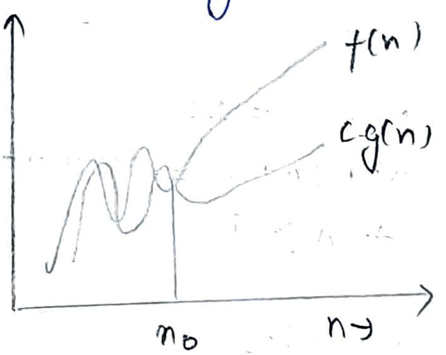
(d) Small-o denotes the upper bound (not tight) on the growth rate of runtime of an algorithm

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

Eg. $n = o(n^2)$



(e) Small-omega (ω) denotes the lower bound.



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

$f(n) > cg(n)$

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

eg. $n^2 = \omega(n)$

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

$i = 1, 2, 4, \ldots, n \rightarrow$ G.P.

$a = 1,\ r = 2,\ t_k = ar^{k-1}$

$\Rightarrow n = 1.(2)^{k-1}$

$\Rightarrow k = \log_2(2n)$
$= \log_2 n + 1$

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

3. $T(n) = \begin{cases} 3T(n-1) & \text{if } n > 0 \\ 1 & \text{otherwise} \end{cases}$

$\Rightarrow T(n) = 3T(n-1) \quad -(1)$

$T(0) = 1$

Using backward substitution,

Put $n = n-1$ in eq (1)

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

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

Put eq (2) in eq (1)

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

$T(n) = 9T(n-2) \quad -(3)$

Put $n = n-2$ in eq (1)

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

$T(n-2) = 3T(n-3) \quad -(4)$

Put in eq (3)

$T(n) = 9[3T(n-3)]$

$T(n) = 27T(n-3)$

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

$n-k = 0 \Rightarrow n = k$

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

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

$\Rightarrow$ Time complexity $\cancel{T(n)=}\ \Theta(3^n)$

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

Using backward substitution,

Put $n = n-1$ in (1).

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

Put (2) in (1),

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

Put $n = n-2$ in eq(1),

$T(n-2) = 2T(n-3) - 1$ — (4)

Put in eq(3)

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

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

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

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

$= 2^n - (2^n - 1)$ $\quad [2^{n-1} + 2^{n-2} + \ldots + 2^0 = 2^n - 1]$

$\Rightarrow T(n) = 2^n - 2^n + 1$

$T(n) = 1$

$\Rightarrow$ Time complexity is $O(1)$.

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

We can define the term 's' according to the relation $s_i = s_{i-1} + i$.

The value of i increases by 1 for each iteration.

The value contained in 's' at the i^{th} iteration is the sum of first i positive integers.

9

If k is the total no. of iterations taken by the program, then loop terminates if: $1+2+ \ldots + k$

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

$\Rightarrow$ So $k^2 + k > 2n$

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

Time Complexity $= O(\sqrt{n})$

6.
```
void function (int n){
    int i, count = 0;
    for (i=1, i*i <=n ; i++)
        count ++;
}
```
Loop ends if $i^2 > n$

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

7.
```
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)           ]  execute
            for (k=1; k<=n; k=k*2)      ]  log n times
                count ++;
```
Time complexity $= O(n \log^2 n)$

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

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

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
            printf("*");               with j increase by
    }                                  the rate of i.
}
```
$\Rightarrow$ Inner loop executes n/i times for each value of i.

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

$= O(n \log n)$

10. The asymptotic relationship between these functions n^k and a^n is

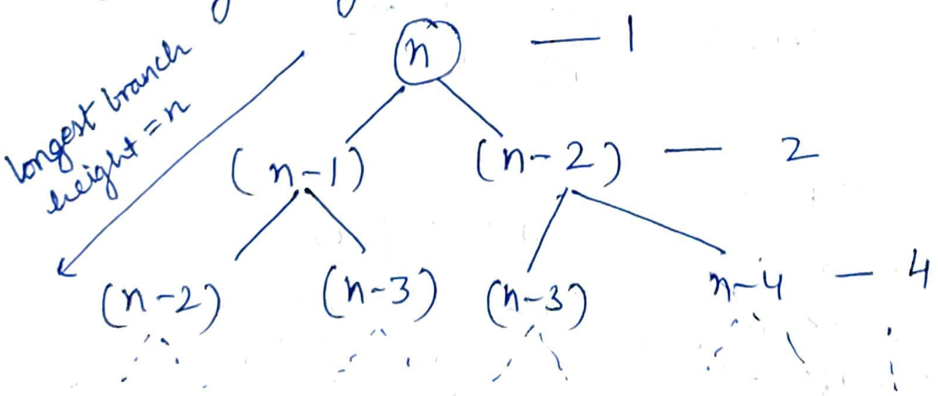
$$n^k = O(a^n) \qquad k>=1,\ a>1$$

$$n^k \le c \cdot a^n \quad \forall\ n \ge n_0$$

$$\Rightarrow \frac{n^k}{a^n} \le c$$

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, $0+1+\dots+k = \frac{k(k+1)}{2} > n$
$$\Rightarrow k = O(\sqrt{n})$$

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 -



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

$$T.C = O(2^{n+1}) = O(2 \cdot 2^n) = O(2^n)$$
Space complexity $= O(n)$ ($\because$ stack size never exceeds the depth of the call's tree shown above)

13. Programs with complexity –

i) $n \log n$ –

```
void func(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) $\log(\log n)$ → `for(int i=2; i<=n; i=pow(i,k)){//O(1)};`

Also, Interpolation search has this complexity.

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

Following is the initial recursion tree,

$$cn^2$$

$$T(n/4) \qquad T(n/2)$$

on further breaking down,

$$cn^2$$

$$cn^2/16 \qquad cn^2/4$$

$$T(n/16) \quad T(n/8) \quad T(n/8) \quad T(n/4)$$

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

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

G.P with ratio $5/16$

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

15. same as ques 9. Ans. $O(n \log n)$

16.
```
for (int i = 2 ; i <= n ; i = pow(i, k))
    {
        // O(1) expression
    }
```

In this case i takes values $2, 2^k, (2^k)^k, (2^{k^2})^k = 2^{k^3} \ldots$ $2^{k^{\log_k(\log(n))}}$. The last term must be less than or equal to n, we have

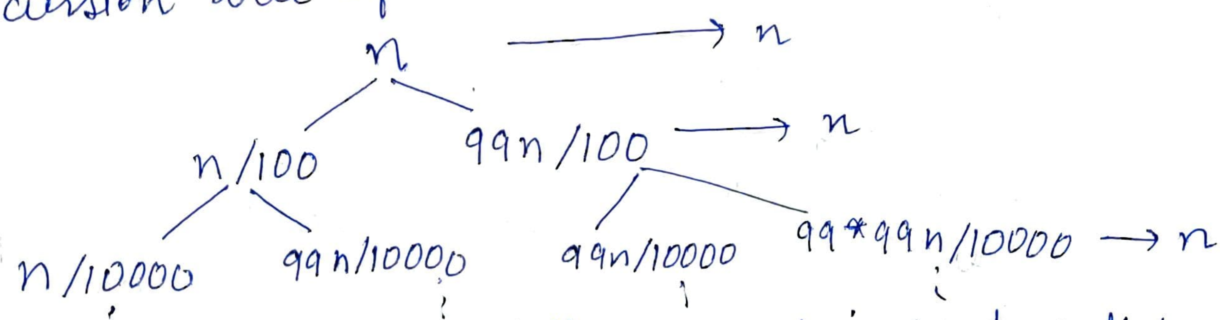
$$2^{k^{\log_k(\log(n))}} = 2^{\log n} = n \Rightarrow \text{It's true}$$

$\therefore$ There are total $\log_k(\log(n))$ many iterations and each iteration takes constant amount of time to run, $\therefore$ Total time complexity $= O(\log(\log n))$

17. 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 cn for all levels. This will follow until 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,

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

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

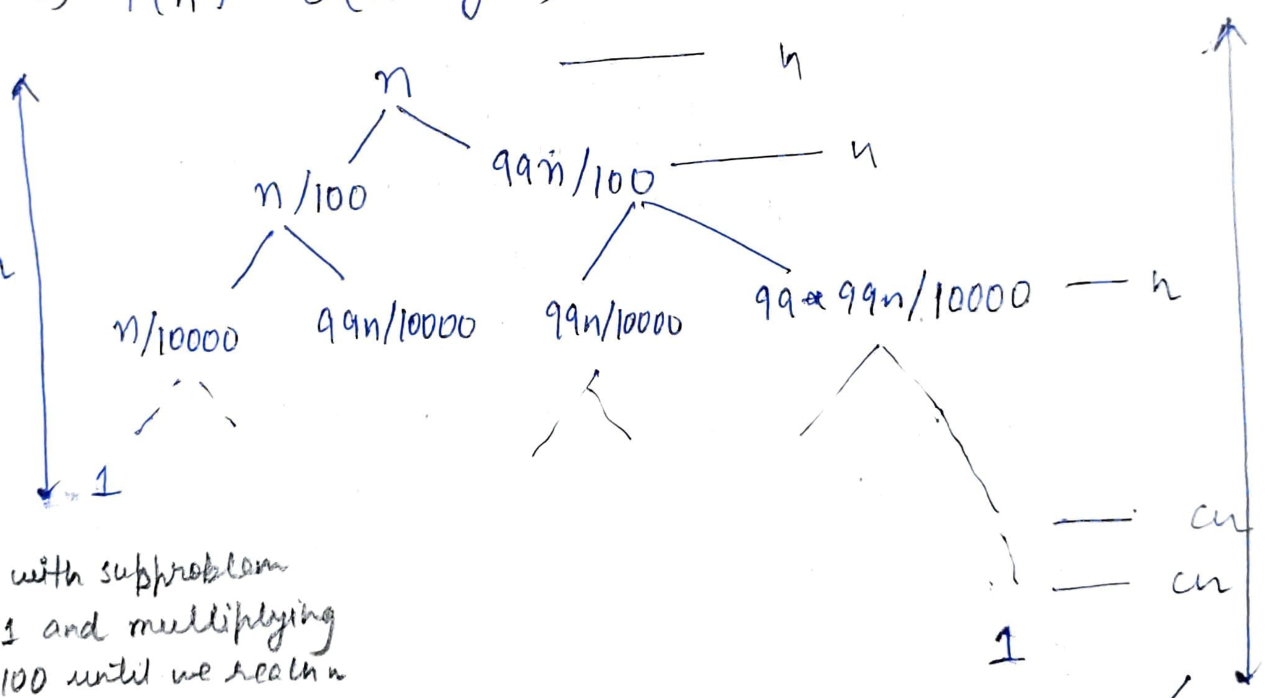
So, there are total $(\log_{\frac{100}{99}} n) + 1$ levels

Thus, $$T(n) = \left(\underbrace{cn + cn + \ldots + (< cn) + (< cn)}_{\log_{\frac{100}{99}} n + 1 \text{ times}}\right) < \left(\log_{\frac{100}{99}} n + 1\right) * cn$$

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

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

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



$\log_{100} n$ (left tree)

Starting with subproblem of size 1 and multiplying it by 100 until we reach n
$100^x = n \Rightarrow x = \log_{100} n$

$c \log_{\frac{100}{99}} n$ (right tree)

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

Multiplying the size by $\frac{100}{99}$ till n. $\left(\frac{100}{99}\right)^x = n$

18. 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) $96 < \log_8 n < \log_2 n < 5n < 8n^2 < 7n^3 <$~~

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

19. 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
             return -1;           if A[i] > data
                                  then no need to
                                  search the rest of
     }                            the array
 }
```

T.C: Best = $O(1)$, Avg, worst = $O(n)$; space = $O(1)$

20. 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;

    insertionSortRecursive(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 algorithm is one that will work if the elements to be sorted are provided one at a time with the understanding that the algorithm ~~th~~ must keep the sequence sorted as more and more elements are added in.
Insertion Sort considers one input element per iteration and produces a partial solution without considering future elements. Thus insertion sort is online.
Other algorithms like selection sort repeatedly selects the minimum element from the unsorted array & places it at the front which requires the entire input. Similary bubble, quick and merge sorts also require the entire input. Therefore they are offline algorithms or sorting.

21, 22 ↴

Sorting Algo	Time Best	Avg	Worst	Space complexity 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	$O(n\log n)$	"	$O(n\log n)$	$O(1)$	✗	✓

23. 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 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 $\Rightarrow$ Best : $O(1)$ & Avg, Worst = $O(\log_2 n)$

Space complexity $\Rightarrow$ Best : $O(1)$

Avg & worst $O(\log_2 n)$

programming stack used.

24. Recurrence relation for binary recursive search

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