Name → Mamta Goyal
Course → B.Tech
Semester → 5
Section → C
Roll No → 1961093

Design and Analysis of Algorithms : Assignment 1

1) Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

The different asymptotic notations are :

i) Big Oh notation (O) : Asymptotic upper bound
In this the function $f(n) = O(g(n))$, if and only if there exists a positive constant c an n_0 such that $f(n) \leq c.g(n) \quad \forall \quad n \geq n_0$.

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

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

$2n^2 + n \leq c.n^2$

Now if $c = 3$

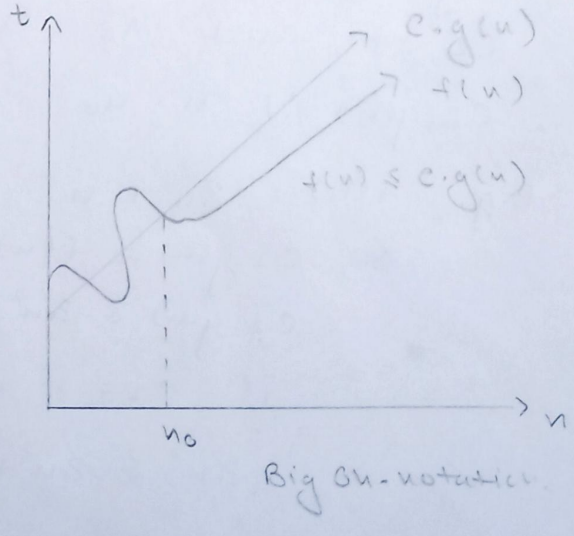
$2n^2 + n \leq c.n^2$

$2n^2 + n \leq 3n^2$

$n \leq n^2$.

$1 \leq n$

$\Rightarrow \quad n \geq 1$

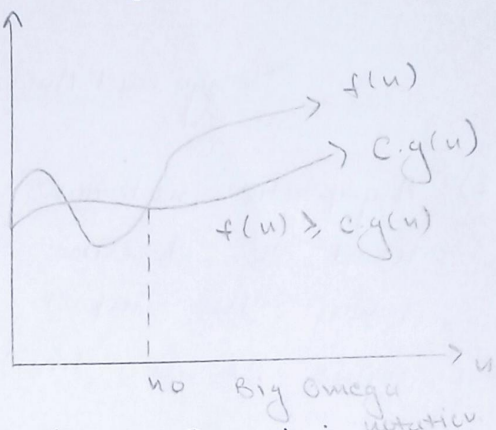


Big Oh - notation

So, for all values of $n \geq 1$ and $c = 3$, this condition will be true.

2) Big Omega notation (Ω): Asymptotic lower bound

In this the function $f(n) = \Omega(g(n))$, if and only if there exist a positive constant c and n_0 such that $f(n) \geq c.g(n)$ $\forall$ $n \geq n_0$.



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

So, $f(n) \geq c.g(n)$

$2n^2 + n \geq c.n^2$

if $c = 2$

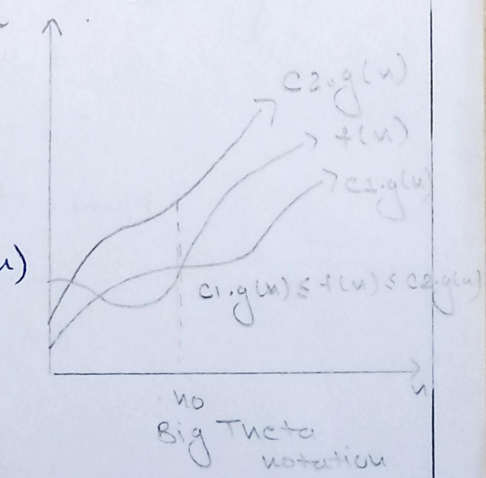
$2n^2 + n \geq 2n^2$

$n \geq 0$

So, for all values of $n \geq 0$ and $c = 2$, this condition will be true.

3) Big Theta notation (Θ): Asymptotic Tight bound

In this the function $f(n) = \Theta(g(n))$, if and only if there exists a positive constant c_1, c_2 and n_0 such that $c_1.g(n) \leq f(n) \leq c_2.g(n)$ $\forall$ $n \geq n_0$.



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

So, $c_1.g(n) \leq f(n) \leq c_2.g(n)$

$c_1.g(n) \leq 2n^2 + n \leq c_2.g(n)$

if $c_1 = 2$ and $c_2 = 3$

$2n^2 \leq 2n^2 + n \leq 3n^2$

So, for all values of $c_1 = 2$ and $c_2 = 3$, $n \geq 1$ this condition will be true.

4) Small oh(o) : Small o gives upper bound
In this the function $f(n) = o(g(n))$, if and only if there exists a positive constant c and n_0 such that $f(n) < g(n)$ $\forall$ $n > n_0$.

5) Small omega(ω) : Small omega gives lower bound.
In this the function $f(n) = \omega g(n)$, if and only if there exists a positive constant c and n_0 such that $f(n) > c.g(n)$ $\forall$ $n > n_0$.

2) Time complexity of :
for(i = 1 to n) { i = i * 2; }

→ Time complexity = $O(\log_2 n)$

3) $T(n) = 3T(n-1)$ if $n > 0$, otherwise 1.

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

Solving by substitution.

$T(n) = 3T(n-1)$ → (i)

$T(n-1) = 3T(n-2)$ → (ii)

Putting value of (ii) in (i)

$T(n) = 3[3T(n-2)]$ → (A)

$T(n-2) = 3T(n-3)$ → (iii)

Putting value from (II) in (A)

$T(n) = 3[3[3T(n-3)]] \rightarrow (B)$

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

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

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

$\vdots$

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

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

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

$T(n) = 3^n$

So, the complexity of this function is $O(3^n)$.

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

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

Solving by substitution.

$T(n) = 2T(n-1) - 1 \rightarrow (I)$

$T(n-1) = 2T(n-2) - 1 \rightarrow (II)$

Putting value from (II) in (I)

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

$T(n) = 2^2 T(n-2) - 2 - 1 \rightarrow (A)$

$T(n-2) = 2T(n-3) - 1 \rightarrow (III)$

Putting value from (III) in (A)

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

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

$\vdots$

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

$n - k = 0$

$n = k$

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

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

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

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

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

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

$T(n) = 1$

So, time complexity of this function is $O(1)$.

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

-> Time complexity of this function will be $O(\sqrt{n})$.

6)
```
int void function(int n){
        int i, count = 0;
            for(i = 1; i*i <= n; i++)
                    count ++;
}
```

-> Time complexity of this function will be $O(\sqrt{n})$.

7) Void function (int n){

```
int i, j, k, count = 0;
for( i = n/2 ; i <= n; i++)
    for( j = 1; j < n; j = j * 2)
        for( k = 1; k <= n; k = k * 2)
            Count ++;
}
```

-> The outermost loop is executing n times and the two inner loops will execute $\log n$ times each. So the total time complexity will be $O(n \log^2 n)$.

8) function (int n){

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

-> Time complexity for the above function will be $O(n^3)$.

9) void function(int n) {

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

→ Time complexity of this function will be :

$O(n \log n)$.

10) functions, n^k and a^n, $k >= 1$ and $a > 1$

Sol The asymptotic relationship between the functions n^k and a^n is,

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

$f(n) \leq c.g(n)$ for Big-oh notation.

So, $n^k \leq c.(a^n)$

If $k = 1$, $a = 2$, $c = 1$

$n^k \leq c.a^n$

$n \leq 2^n$

So, for all values of $n_0 \leq 2^n$ and $c >= 1$, this relation will hold true.

11) void fun(int n) {

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

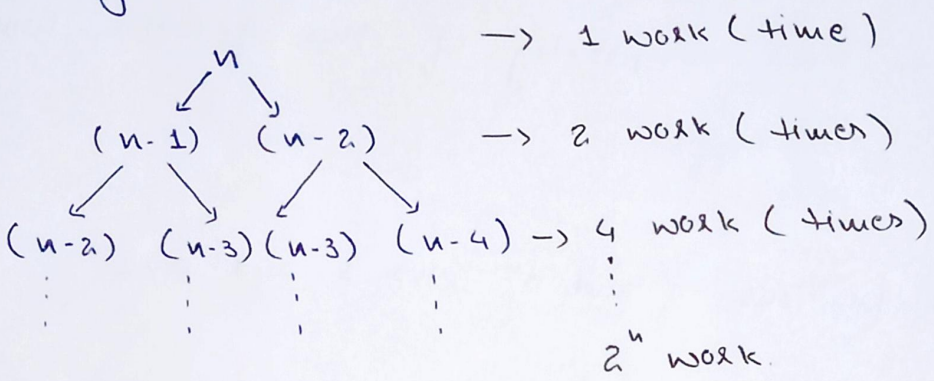
→ Time complexity for the above function will be :

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

12) Recurrence relation for the recursive function that prints Fibonacci series.

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

Solving it using tree method:



So, Time complexity, $T(n) = 1 + 2 + 4 + \ldots + 2^n$

$$T(n) = 1 + 2 + 4 + \ldots + 2^n$$

This is a G.P where $a = 1$, $r = 2$

So, Sum of G.P: $\dfrac{a(r^{\text{terms}} - 1)}{r - 1}$

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

$$= 2^{n+1} - 1$$

So, time complexity will be $O(2^{n+1}) = O(2^n \cdot 2)$

=> Time complexity for this function will be

$$\boxed{T(n) = O(2^n)}$$

The space complexity of this program will be O(n) because the function calls that are being executed recursively are actually being executed sequentially and sequential execution guarantees that the stack size will never exceed the depth of the calls which here is n. Only one recursion is evaluated at a time so we take the largest space taken by f(i-1) recursion into consideration which makes the **space complexity = O(n)**

13) Write programs which have complexity - n(log n), n^3, log(log n).

Sol 1) n(log n)

```
Void function(){
  for(int i=0; i<n; i++)
    {
      for(int j=0; j<n; j=j*2
        {
          printf("Hello");
        }
    }
}.
```

2) n^3

```
void function( ){
    for( int i=0; i<n; i++)
      {
        for( int j=0; j<n; j++)
          {
            for( int k=0; k<n; k++)
              {
                printf(" *");
              }
          }
      }
```

3) log(logn)

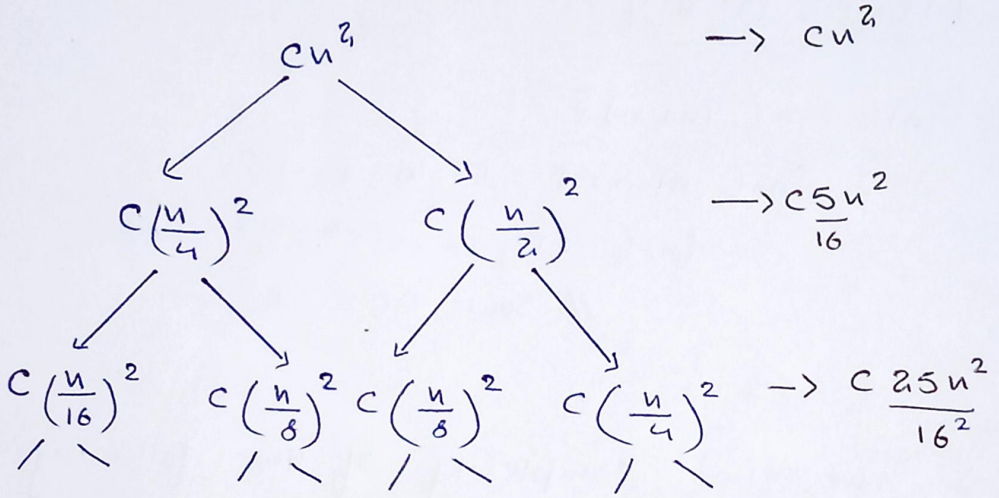
```
# include< bits/stdc++.h>
void fun( int n)
  {
    if( n== 2)
        return 1;
    else
        fun( sqrt(n));
  }

void main()
  {
    fun(100);
  }
```

(14) Solve: $T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + cn^2$

Sol Solving by recursion tree method.

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + cn^2$$



So, $T(n) = cn^2 + c\frac{5}{16}n^2 + c\frac{25}{256}n^2 + \ldots$

$$T(n) = c\left[n^2 + \frac{5}{16}n^2 + \frac{25}{256}n^2 + \ldots\right]$$

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

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

This is a g.p with $a = 1$, $r = \frac{5}{16}$

If $r < 1$, sum of G.P $= \frac{a}{1-r}$

$$\Rightarrow T(n) = cn^2\left[\frac{1}{1-\frac{5}{16}}\right]$$

$$T(n) = cn^2\left[\frac{16}{11}\right]$$

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

So, the time complexity of the given recurrence relation, $T(n) = T(\frac{n}{4}) + T(\frac{n}{2}) + n^2$ is

$$\boxed{T(n) = O(n^2)}$$

15)
```
int fun( int n) {
    for( int i = 1; i <= n; i++) {
        for( int j = 1; j < n; j += i) {
            // some O(1) task
        }
    }
}
```

→ The time complexity of the following function, fun() will be $O(n \log n)$.

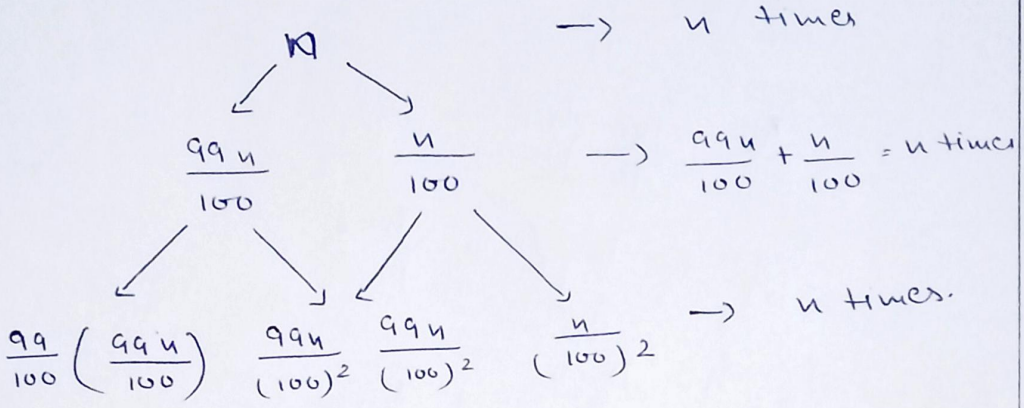
16)
```
for( int i = 2; i < n; i = pow(i, k))
{
    // some O(1) statement
}
```

→ The time complexity for the above function will be $O(\log\log n)$.

17) Recurrence relation when quick sort repeatedly divides the array in to two parts of 99% and 1% is,

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

Solving by recursion tree:



Cost of each level = n

Total cost = height * Cost of each level.

height = $\log n$

So, $T(n) = n + n + n + n + \ldots$

$T(n) = n * \log n$ (as height = $\log n$)

$T(n) = \underline{O(n \log n)}$

Time complexity for this recurrence relation will be $O(n \log_2 n)$

Now, height of first (left) stream :

For first stream, values at each level are

$N, \frac{99N}{100}, \left(\frac{99}{100}\right)^2 N, \ldots$

at h^{th} level, $n \left(\frac{99}{100}\right)^{h-1}$

So, $n \left(\frac{99}{100}\right)^{h-1} = 1$

$$\left(\frac{99}{100}\right)^{h-1} = \frac{1}{n}$$

$$n = \left(\frac{100}{99}\right)^{h-1}$$

$$\log n = \log\left(\frac{100}{99}\right)^{h-1}$$

$$h = \frac{\log n}{\log\left(\frac{100}{99}\right)} + 1.$$

For right stream, values at different levels are.

$n, \frac{n}{100}, \left(\frac{n}{100}\right)^2, \ldots 1.$

So, value at h^{th} level will be $\frac{n}{100^{n-1}}$

So, $n\left(\frac{1}{100}\right)^{h-1} = 1$

$$n = (100)^{h-1}$$

$$\log n = \log(100)^{h-1}$$

$$\log n = (h-1)\log(100)$$

$$h = \frac{\log n}{\log 100} + 1$$

with random data there will be a mix of good and bad splits throughout the recursion tree and a mixture of worst case and best case splits is asymptotically the same as best case. So, the heights of these two extreme parts is approximately the same, the only difference is with left stream where the height is a little more by a constant factor, but this constant disappears in the O analysis.

18) Arrange the following in increasing order of rate of growth:

a) $n, n!, \log n, \log\log n, \text{root}(n), \log(n!), n\log n, 2^n, 2^{2n}, 4^n, n^2, 100$

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

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

→ $O(1) < O(\log\log n) < O(\log n) < O(\log 2n) < O(2\log n) < O(n) < O(n\log n) < O(\log n!) < O(2n) < O(4n) < O(n^2) < O(n!) < O(2(2^n))$.

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

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

19) Linear search pseudocode to search an element in a sorted array with minimum comparisons:

```
Void LinearSearch(int arr[], int n, int key)
    for i <- 1 to n
        if arr[i] == key
            return i
    return -1
```

20) Pseudo code for iterative insertion sort:

```
void insertionSort(int* arr, int n)
    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;
        End while
        arr[j+1] <- temp
```

Pseudo code for recursive insertion sort

```
Void insertion_sort ( int * arr, int n)
    if n<= 1
       return
    insertion_sort ( arr, n-1)
    last <- arr[n-1]
    j <- n-2
    while j>=0 AND arr[j]> last
        arr[j+1] = arr[j]
        j--
    End while
```

Insertion sort is called online sorting because it processes its input piece by piece in a serial fashion i.e, in the order that the input is fed to the algorithm without having the entire input available from the start. Other sorting algorithms are different from insertion sort because unlike other sorting algorithms insertion sort does not waits and as soon as it encounters the elements, it sorts them.

21) Complexity of all sorting algorithms discussed in lectures.

Time Complexities

Sorting Algorithm	Best case	Average case	Worst case
1) Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
2) Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

3) Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
4) Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
5) Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
6) Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Space complexities

1) Insertion Sort : $O(1)$
2) Selection Sort : $O(1)$
3) Bubble Sort : $O(1)$
4) Merge Sort : $O(n)$
5) Heap Sort : $O(1)$
6) Quick Sort : Space complexity depends on the space used in the recursion stack. So, in average case, space complexity $= O(\log n)$ and in worst case, space complexity $= O(n)$.

22) Divide all the sorting algorithms into in place / Stable / Online sorting.

Algorithm Name	InPlace	Stable	Online
1) Insertion Sort	✓	✓	✓
2) Selection Sort	✓	✗	✗
3) Bubble Sort.	✓	✓	✗

4) Quick Sort	✓	×	×
5) Merge Sort	×	✓	×
6) Heap Sort	✓	×	×

23) Iterative and Recursive pseudocode for binary search.

Iterative :

```
int binarySearch( int arr[], int l, int r, int key)
{
    While( l <= r) {
        mid <- l + (r-l)/2
        if { arr[mid] == key
            return mid
        if arr[mid] > key
            r <- m - 1
        else
            l <- m + 1
    End while
    return - 1
```

Recursive :

```
int binarySearch ( int arr[], int l, int r, int key)
    if( r >= l
        mid <- l + (r-l)/2
        if arr[mid] == key
            return mid
        else if arr[mid] > key
```

```
        return binarySearch(arr, l, mid-1, key)
    else
        return binarySearch(arr, mid+1, r, key)
    return -1
```

Time complexity of linear Search:

-> Best case : O(1)
-> Average case : O(n)
-> Worst case : O(n)

Space complexity of linear search = O(1)

Time complexity of Binary Search (Iterative)

-> Best case : O(1)
-> Average case : O(log n)
-> Worst case : O(log n)

Space complexity of Binary Search (Iterative) = O(1)

Time Complexity of Binary Search (Recursive)

-> Best case : O(1)
-> Average case : O(log n)
-> worst case : O(log n)

Space complexity of Binary Search (Recursive)

-> Best case : O(1)
-> Average Case : O(log n)
-> Worst Case : O(log n)

24) Recurrence Relation for recursive binary Search.

Recurrence relation for recursive binary search is:

$T(n) = T\left(\frac{n}{2}\right) + 1$, where $T(n)$ is the time required for binary search in an array of size n.

Solving this using backward substitution method:

$T(n) = T\left(\frac{n}{2}\right) + 1$

$T(1) = 1$

$T(n) = T\left(\frac{n}{2}\right) + 1 \rightarrow (i)$

$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + 1 \rightarrow (ii)$

$T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + 1 \rightarrow (iii)$

Putting value from eq^n (ii) in (i)

$T(n) = T\left(\frac{n}{4}\right) + 1 + 1 \rightarrow (A)$

Putting value from (iii) in (A)

$T(n) = T\left(\frac{n}{8}\right) + 1 + 1 + 1 \rightarrow (B)$

$\vdots$

$T(n) = T\left(\frac{n}{2^k}\right) + k$

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

$n = 2^k$

$$\log_2 n = \log_2 2^k$$

$$k = \log_2 n$$

$$\therefore \quad T(n) = T\left(\frac{n}{2^{\log_2 n}}\right) + \log_2 n$$

using $a^{\log_a b} = b$

$$T(n) = T\left(\frac{n}{n}\right) + \log_2 n$$

$$T(n) = T(1) + \log_2 n$$

$$T(n) = 1 + \log_2 n$$

$\Rightarrow$ Time complexity $= O(\log_2 n)$

— x —