Design and Analysis of Algorithms

Assignment 1

Name - Priyanshu Goyal

Section - A

Class Roll No - 39

University Roll No - 2014787

Subject Code - TCS-505

Q1) What do you understand by Asymptotic notations. Define different Asymptotic notation with examples.

Ans) Asymptotic notations are used to tell the complexity of an algorithm when the input is very large.

- When we use asymptotic notation, there is an inherent assumption that our input size will be very large.
- The different asymptotic notations are :-

1) Big O (O) - The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.

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

means $g(n)$ is tight upper bound of $f(n)$

means $f(n)$ can never go beyond $g(n)$.

Sign
Priyanshu

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

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

$\forall\ n \geq n_0$, some constant $c > 0$.

↑ some threshold.

Ex- $f(n) = 3\log_2 n + 100$

$g(n) = \log_2 n$

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

have to find $c,\ n >= n_0$ such that $3\log n + 100 <= c * \log n$

If we take $c = 200$, $n_0 = 2$

$\Rightarrow 3\log_2 n + 100 <= 200 * \log_2 n$

$\forall n >= 2$, $c = 200$

$\therefore 3\log_2 n + 100 = O(\log_2 n)$

$\forall\ n >= 2$, $c = 200$.

2) Big Omega (Ω) : It is an asymptotic notation for the best case, or a floor growth rate for a given function. It provides us with an asymptotic lower bound for growth rate of run-time of an algorithm.

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

$f(n) >= c \cdot g(n)$

$\forall\ n >= n_0$ and some constant $c > 0$.

Ex- $f(n) = \log_2 n$

$g(n) = 3\log_2 n + 100$

Sign
Priyanshu

for $c = \frac{1}{200}$ (>0) , $n_0 = 2$

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

$\Rightarrow \log_2 n >= \frac{1}{200}(3\log_2 n + 100)$

$\forall n >= 2$

$\Rightarrow f(n) = \Omega(g(n)) \Rightarrow \log_2 n = \Omega(3\log n + 100)$

$\forall n >= 2$, $c = \frac{1}{200}$

3) Theta notation (Θ): This notation bounds a function from above and below, so it defines asymptotic behaviour.

Ex- $3n^3 + 6n^2 + 6000 = \Theta(n^3)$.

Dropping lower order terms is always fine because there will always be a number n after which $\Theta(n^3)$ has higher values than $\Theta(n^2)$ irrespective of constants involved.

$f(n) = \Theta(g(n))$, iff

$0 <= c_1 g(n) <= f(n) <= c_2 g(n)$

$\forall n >= \max(n_1, n_2)$, $c_1 \& c_2 > 0$

$\Rightarrow c_2 g(n)$ is greater than $f(n)$ after n_2.

$c_1 g(n)$ is lesser than $f(n)$ after n_1.

Sign
Priyanshu

4) Small o (o): It is an asymptotic notation to denote the upper bound on the growth rate of run time of an algorithm.

$f(n) = o(g(n))$, if

if for all real constants $c > 0$, $n > n_0$ (input size)

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

Ex: $f(n) = n$

$g(n) = n^2$ ⇒ can find $c > 0$, $n > n_0$ after which $n < n^2$

⇒ $n = o(n^2)$

Ex: if we take $c = 0.5$, $n_0 = 2.5$

⇒ $n < 0.5 \cdot n^2$

$\forall\ n > 2.5$ ⇒ $n = o(n^2)$

5) Small Omega (ω): - It is an asymptotic notation to denote the lower bound on growth rate of run time of an algorithm.

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

if for all real constants $c > 0$, $n > n_0$

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

Ex: $f(n) = n^2$

$g(n) = n$ ⇒ can find $c > 0$, $n > n_0$ after which $n^2 > n$

⇒ $n^2 = \omega(n)$.

if we take $c = 2$, $n_0 = 2.5$

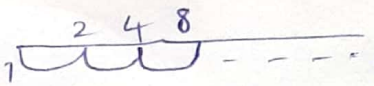
⇒ $n^2 > 2n$ $\forall\ n > 2.5$

⇒ $n^2 = \omega(n)$

Sign
Priyanshu

Q2) What should be time complexity of - for(i=1 to n) {i=i*2}.

A2)



1, 2, 4, 8, 16, ---- n ⇒ let be k steps

⇒ G.P ⇒ a=1, r = 2/1 = 2

k^{th} term in GP = $a r^{k-1}$

∴ k^{th} term = n ⇒

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

$\Rightarrow 2^k = 2n$

$k \log_2 2 = \log_2 2 + \log_2 n$

$\Rightarrow k = 1 + \log_2 n$

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

Q3) $T(n) = \{3T(n-1)$ if $n>0$ otherwise $1\}$

$T(n) = 3T(n-1)$ -①

$T(0) = 1$

Put $n = n-1$ in ① ⇒

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

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

$\Rightarrow T(n-1) = 3.3T(n-3)$

$\Rightarrow T(n) = 3.3.3.T(n-3)$

⋮

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

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

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

∴ $T(0) = 1$

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

Sign
Priyanshu

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

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

$T(0) = 1$

Using back substitution -

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

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

$\Rightarrow T(n) = 2.2T(n-2) - 2 - 1$

Put $n = n-2$ in (1) $\Rightarrow$

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

$\Rightarrow T(n) = 2.2.2T(n-3) - 4 - 2 - 1$

$\Rightarrow T(n) = 2^3 T(n-3) - 2^2 - 2^1 - 2^0$

$\vdots$

$T(n) = 2^k T(n-k) - 2^{k-1} - 2^{k-2} - 2^{k-3} \ldots - 2^2 - 2^1 - 2^0$

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

~~$\Rightarrow T(n) = 2^n T(0) \Rightarrow 2^k \left[1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3}\right]$~~

$\Rightarrow T(n) = 2^n T(0) - \underbrace{(2^{k-1} + 2^{k-2} + 2^{k-3} \ldots + 2^2 + 2^1 + 2^0)}_{\text{G.P}}$

$\Rightarrow T(n) = 2^n - \underbrace{(2^0 + 2^1 + 2^2 + \ldots 2^{k-3} + 2^{k-2} + 2^{k-1})}_{k \text{ terms}}$

$a = 1, \quad r = \frac{2}{1} = 2$

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

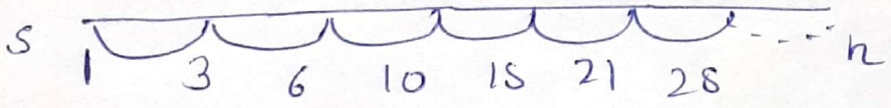
$\Rightarrow T(n) = 2^n - \frac{2^k}{2}$

$\Rightarrow T(n) = \frac{2.2^n - 2^n}{2}$

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

Sign
Priyanshu

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



1 3 6 10 15 21 28 ... n,
let be k steps — kth term

$S = 1+3+6+10+15+21+28+\cdots+T_k$ ——(1)
$S = 1+3+6+10+15+21+28+\cdots+T_{k-1}+T_k$ ——(2)

(1) − (2):
$\Rightarrow 0 = 1+2+3+4+5+6+7+\cdots-(T_k-T_{k-1})-T_k$
$\Rightarrow T_k = 1+2+3+4+5+6+7+\cdots+(T_k-T_{k-1})$
$\Rightarrow T_k = \frac{k(k+1)}{2}$

$\because$ k^{th} term $= n$
$\Rightarrow n = \frac{k(k+1)}{2} \Rightarrow n = \frac{k^2+k}{2}$

Neglecting lower order terms $\Rightarrow$
$n = k^2 \Rightarrow k = \sqrt{n}$
$\therefore$ T.C $= O(\sqrt{n})$

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

Sign
Priyanshu

$i^2 <= n$

$\Rightarrow i <= \sqrt{n}$ $\Rightarrow$ 

$\therefore T.C = O\sqrt{n}$

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 ++;

}

i	j	k	Times
$n/2$	$\log_2 n$	$\log_2 n$	$\log_2 n * \log_2 n$
$\frac{n}{2}+1$	$\log_2 n$	$\log_2 n$	$\log_2 n * \log_2 n$
$\frac{n}{2}+2$	$\log_2 n$	$\log_2 n$	$\log_2 n * \log_2 n$
⋮	⋮		⋮
$\frac{n}{2}+\frac{n}{2}=n$	$\log_2 n$	$\log_2 n$	$\log_2 n * \log_2 n$

$n/2$ times

Sum $= \frac{n}{2} * \log_2 n * \log_2 n$

$= \frac{n}{2} (\log_2 n)^2$

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

Sign
Priyanshu

8] function (int n) {

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

A8] Recurrence relation :-

$$T(n) = T(n-3) + n^2 \quad -①$$

$$T(1) = 0 \qquad n<=1$$

Using backward substitution :

Put $n = n-3$ in ① ⇒

$$T(n-3) = T(n-6) + (n-3)^2$$

$$\Rightarrow T(n) = T(n-6) + (n-3)^2 + n^2$$

$$\Rightarrow T(n-6) = T(n-9) + (n-6)^2$$

$$\Rightarrow T(n) = T(n-9) + (n-6)^2 + (n-3)^2 + n^2$$

$$\Rightarrow T(n) = T(n-3k) + \underbrace{(n-3(k-1))^2 + (n-3(k-2))^2 + \dots n^2}_{(k \text{ terms})}$$

let $n-3k = 0$

$\Rightarrow n = 3k \Rightarrow k = \frac{n}{3}$

$$\Rightarrow T(n) = T(0) + n^2 + (n-3)^2 + (n-6)^2 + \dots (n-(n-1))^2$$

$$\Rightarrow T(n) = 0 + n^2 + (n-3)^2 + (n-6)^2 + \dots (n-n+3)^2$$

$$\Rightarrow T(n) = \underbrace{n^2 + (n-3)^2 + (n-6)^2 + \dots (n-n+3)^2}_{k \text{ terms (from above)}}$$

Neglecting lower order terms ⇒

$$T(n) = n^2 + n^2 + n^2 + \dots \text{ k terms}$$

Sign
Priyanshu

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

$\Rightarrow k = \frac{n}{3}$

$\Rightarrow T(n) = \frac{n3}{3} \Rightarrow \boxed{T.C = O(n^3)}$

9)

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

i	j	times
1	1 → n	n
2	1 → n	$\left(\frac{n+1}{2}\right) \approx \frac{n}{2}$
3	1 → n	$\frac{n}{3}$
4	1 → n	$\frac{n}{4}$
⋮		
n	1 → n	$\frac{n}{n}$

$$\text{Sum} = n + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} + \cdots \frac{n}{n}$$

$$= n\left(\underbrace{1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots \frac{1}{n}}_{\text{Harmonic progression}}\right)$$

$d = 1, \; a = 1$

$$\Rightarrow \text{Sum} = \frac{1}{d} \log_e \left[\frac{2a + (2n-1)d}{2a - d}\right]$$

$$\Rightarrow \text{Sum} = \log_e \left[\frac{2 + 2n - 1}{1}\right] = \log_e (2n+1)$$

$\Rightarrow$ Sum $= n \log_e (2n+1)$

$\Rightarrow T.C = O(n \log_e (2n+1))$

Sign
Priyanshu

10) asymptotic relationship between n^k and $a^n \Rightarrow$

Given, Assume $k >= 1$ and $a > 1$ are constants.

$n^k \rightarrow$ polynomial complexity

$a^n \rightarrow$ Exponential complexity

$\Rightarrow$ $\boxed{n^k = O(a^n)}$

$\Rightarrow n^k <= c \cdot a^n$

$\forall\ n >= n_0$

If $c = 100$, $n_0 = 2 \Rightarrow n^k <= 100 \cdot a^n, \forall n >= 2$

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

$\because c = 100, n_0 = 2.$

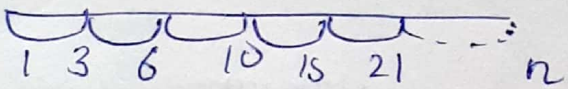
11) void fun (int n){

int j=1, i=0;

while (i<n){

i = i+j;

j++; }}



1 3 6 10 15 21 28 .... n

let k steps

k^{th} term.

$\because$ $S-1 =$ ~~1~~ + 3 + 6 + 10 + 15 + 21 + ... T_k ①

$-S = -1 + 3 + 6 + 10 + 15 + 21 + \cdots T_{k-1} + T_k$ ②

① - ② $\Rightarrow -1 + T_k = 2+3+4+5+6+\cdots(T_k - T_{k-1})$

$\Rightarrow T_k = 1+2+3+4+5+6+\cdots(T_k - T_{k-1})$

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

$\because$ k^{th} term $= n \Rightarrow n = \frac{k(k+1)}{2} \Rightarrow n = \frac{k^2+k}{2}$

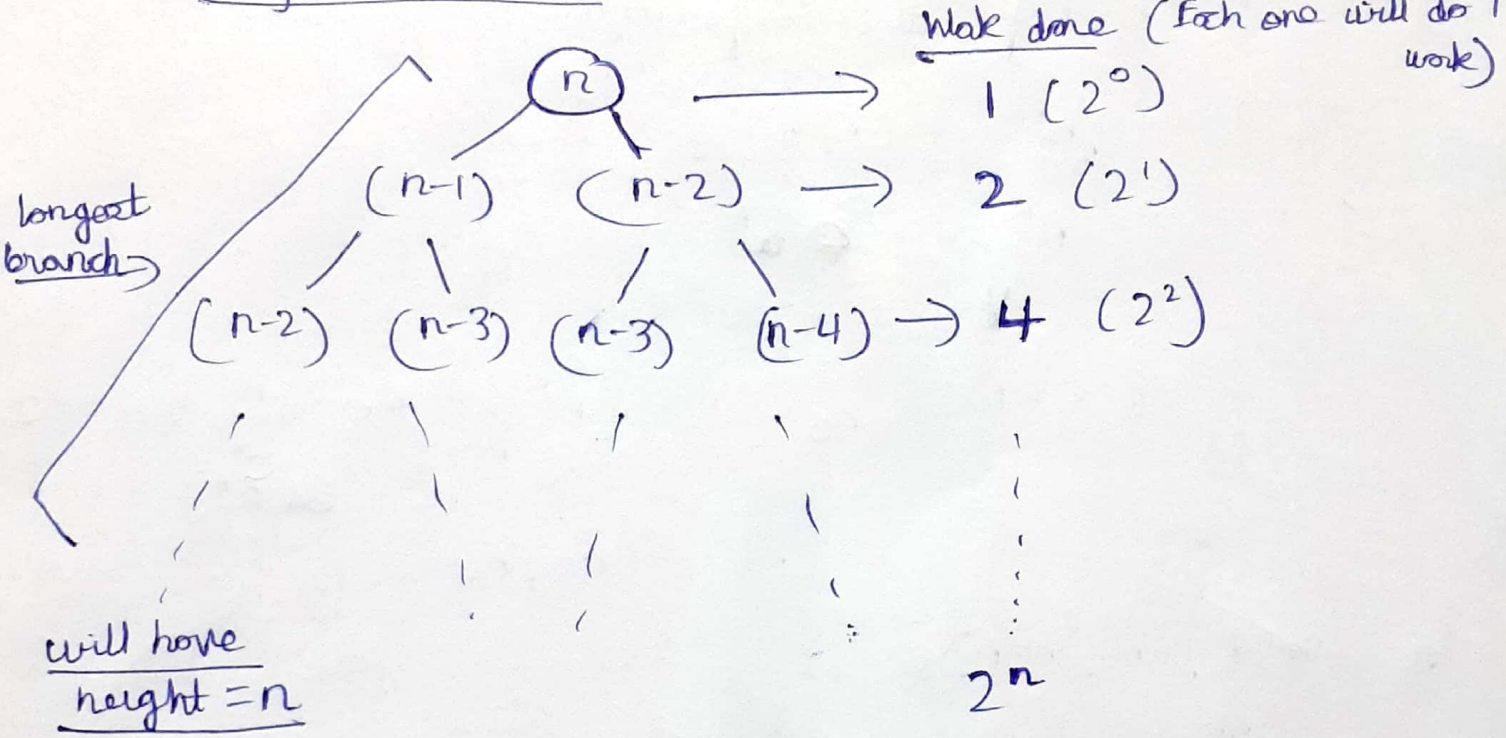
$\Rightarrow k \propto \sqrt{n} \Rightarrow \boxed{T.C = O\sqrt{n}}$

Sign
Priyanshu

(2) Recursive Fibonacci Series - Recurrence Relation =

$$\Rightarrow T(n) = T(n-1) + T(n-2) + 1 \quad , \; n >= 2$$
$$\qquad\qquad\qquad\quad 1 \qquad\qquad\quad n < 2$$

Using Tree Method:



$\Rightarrow$ Complexity = $1 + 2 + 4 + \dots 2^n$ (G.P)

$a = 1$, $r = \frac{2}{1} = 2$, no of terms = $n+1$ (from 0 to n)

$$\Rightarrow S = \frac{1.(r^{n+1} - 1)}{r-1} = \frac{2^{n+1} - 1}{2-1} = 2^{n+1} - 1$$

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

- Space Complexity = $O(n)$

because of recursion stack, as there are n recursive calls there will be n stack frames.

Sign
Priyanshu

A13) a) $n(\log n)$

```
#include <iostream>
using namespace std;
int main()
{
    int i, j;
    int n;
    cin >> n;
   for ( i=1 ; i<=n ; i++)
    {
      for(j=1 ; j<=n ; j=j*2)
       {
         cout << "Hello";
       }
    }
}
```

b) n^3

⇒

```
void fun(int n)
{
   int i, j;
   int k;
  for ( i=1 ; i<=n ; i++)
  {  for (j=1 ; j<=n ; j++)
   {  for(k=1 ; k<=n ; k++)
      {  cout << "Hello";
      }
   }
}
```

Sign
Priyanshu

c) $\log(\log n)$

```
void fun(int n)
{
    int i;
    for(i=n; i>=2; i=sqrt(i))
    {
        cout << "Hello";
    }
}
```

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

Assume, $T(n/2) >= T(n/4)$

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

Applying Master's Theorem $\Rightarrow$

$a = 2, b = 2 \Rightarrow \log_b^a = \log_2^2 = 1$

$\Rightarrow n^{\log_b^a} = n^1 = n$

$f(n) = cn^2$

$f(n) > n^{\log_b^a} \Rightarrow$ Time Complexity $= \theta(f(n))$

$= \theta(cn^2) = \theta(n^2)$.

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

Sign
Priyanshu

⇒

i	j	times
1	1 → n	n
2	1 → n	$\frac{n}{2}$
3	1 → n	$\frac{n}{3}$
4	1 → n	$\frac{n}{4}$
⋮		
n	1 → n	$\frac{n}{n}$

$$\text{Sum} = n + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} \cdots \frac{n}{n}$$

$$= n\left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots \frac{1}{n}\right)$$

harmonic series

here, sum $\simeq \log_e n$

$$\Rightarrow \text{Sum} = n \log_e n$$

$$\therefore \text{Complexity} = O(n \log_e n)$$

16)

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

$$i \Rightarrow 2, 2^k, 2^{k^2}, 2^{k^3} \cdots 2^{k^{\log_k(\log(n))}}$$

last term must be .<= n

Here, last term $= 2^{k^{\log_k(\log_2 n)}}$

$$= 2^{\log_2 n} = n$$

$\Rightarrow$ last term $= n$

So, total iterations $= \log_k(\log n)$

Sign
Priyanshu

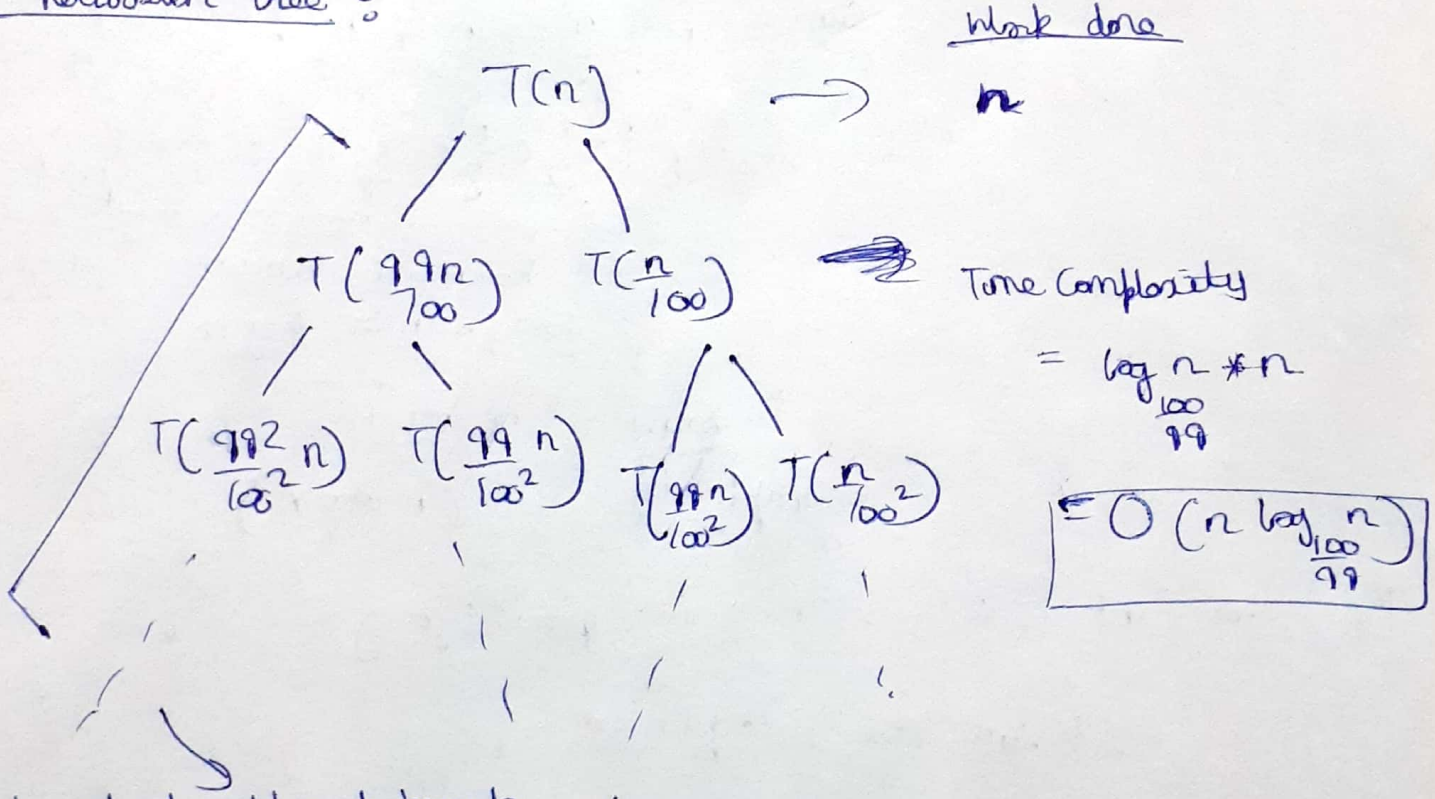
$\Rightarrow$, in each iteration $\Rightarrow O(1)$ time

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

17) Recurrence Relation $\Rightarrow$

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

Recursion tree :



Height of leftmost branch $= \log_{\frac{100}{99}} n$ ($\because T\left(\frac{99n}{100}\right) = T\left(\frac{n}{\frac{100}{99}}\right)$)

Height of rightmost branch $= \log_{100} n$

$\therefore$ Difference in heights $= \log_{\frac{100}{99}} n - \log_{100} n$

Sign
Priyanshu

Ans (8) a) $100 < \log\log n < \log n < \text{root}(n) < n < \log(n!)$
$< n\log n < n^2 < 2^n < 2^{2n} < 4^n < n!$

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

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

Ans (9) Linear Search Pseudocode in sorted array:

```
int linear(int arr[], int key, int n)
{
    i=0
    while(i is less than n)
    {
        if(arr[i] = key)
            return i;
        if(arr[i] > key)
            return -1;
    } i++;
}
```

Sign
Priyanshu

A20) Iterative Insertion sort →

```
void insertion (int arr[], int n)
{
    for ( i=1 ; i<n ; i++)
    {
        j = i-1;
        key = arr [i-1];
        while ( j>=0 and arr [j] > key)
        {
            arr [j+1] = arr [j]
            j--;
        }
        arr [j+1] = key
    }
}
```

b) Recursive Insertion Sort →

```
void insertion ( int arr [] ; int n)
{
    if ( n<=1)
        return ;
    insertion ( arr, n-1) ;
    l = arr [n-2];
    j = n-2;
    while ( j>=0 and arr [j] > l)
    {
        arr [j+1] = arr [j];
        j--;
    }
    arr [j+1] = l;
}
```

Sign
Priyanshu

- Insertion Sort is called online sorting because it can accept new data while sorting procedure is ongoing without imparting the ongoing process.

- Bubble Sort, Selection Sort, Quick Sort, Merge Sort ⇒ Offline sorting algorithm

21)

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

22)

	Inplace	Stable	Online
• Bubble Sort	✓	✓	×
• Selection Sort	✓	×	×
• Insertion Sort	✓	✓	✓
• Quick Sort	✓	×	×
• Merge Sort	×	✓	×

23) Iterative Binary Search ⇒

```
int binarysearch(int arr[], int l, int r, int R)
{
  while (l<=r)
  {
    m = ~~(l+r)/2~~ l+(r-l)/2;
```

Sign
Priyanshu

```
if ( arr[m] = k)
    return m;
if ( arr[m] < k)
    l ← m+1;
else
    r ← m-1;
}
return -1;
}
```

Recursive BinarySearch, →

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

- Time Complexity →
  - Iterative Binary Search → $O(\log_2 n)$
  - Recursive Binary Search → $O(\log_2 n)$
  - Iterative Linear Search → $O(n)$
  - Recursive Linear Search → $O(n)$
- Space Complexity →
  - Iterative Binary Search → $O(1)$
  - Recursive Binary Search → $O(\log n)$
  - Iterative Linear Search → $O(1)$
  - Recursive Linear Search → $O(n)$

Sign
Priyanshu

24] Recurrence Relation for binary Recursive Search ⇒

$$\begin{cases} T(n) = T(n/2) + 1 & -(1) \\ T(1) = 1 \end{cases}$$

$\Rightarrow T(n/2) = T(n/4) + 1 \quad -(2)$

$\Rightarrow T(n) = T(\frac{n}{4}) + 1 + 1 \quad -(3)$

$\Rightarrow T(n/4) = T(\frac{n}{8}) + 1 \quad -(4)$

$\Rightarrow T(n) = T(\frac{n}{8}) + 1 + 1 + 1$

$\Rightarrow T(n) = T(\frac{n}{2^3}) + 3 \quad -(5)$

$\vdots$

$T(n) = T(\frac{n}{2^k}) + k \quad -(6)$

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

$\Rightarrow \log_2 n = k \log_2 2$

$\Rightarrow \boxed{k = \log_2 n}$

Put in (6)

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

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

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

$\Rightarrow T(n) = O(\log_2 n)$

Sign
Priyanshu