Assignment-1

Date
Page

① What do you understand by Asymptotic notations. Define diff. Asymptotic notations with examples.

Asymptotic notations are methods/languages using which we can define the running time of the algorithm based on input size.
for example - In bubble sort, when the input array is already sorted, the time taken by algorithm is linear i.e the best case -

There are mainly three asymptotic notations:-
- Big-O notation
- Omega notation
- Theta notation

→ Big-O notation - represents the upper bound of the running time of an algorithm. Thus provides worst case complexity.
$O(g(n)) = \{ f(n)$: there exist positive constants c & n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0 \}$

Example - $f(n) = 3\log n + 100$ $g(n) = \log n$
lets take $c = 200$
$3\log n + 100 <= 200 * \log n \ \forall \ n > 2$ (undefined at $n=1$)

Big-Omega Notation (Ω-notation) ⇒ Ω represents the lower bound of the running time of an algorithm.
Thus, provides the best case complexity.
$\Omega(g(n)) = \{ f(n)$: there exist positive constants c & n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$

Example - $4n+3 \geq cn$ for some $c > 0$ & some $n_0 \geq 1$
when $c=1$ & $n_0 = 1$ for any $n \geq 1$ this $4n+3 \geq n$ is true
Thus we say if $f(n) = 4n+3$ & $g(n) = n$
$f(n) = \Omega(g(n))$ or simply $4n+3 = \Omega(n)$

Date
Page

→ Theta Notation (Θ-notation) Theta notation encloses the function from above & below. Since it represents the upper & the lower bound of the running time of an algo, it is used for analyzing the average-case complexity of an algo.

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2 & n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$

Date
Page

2- What should be complexity of

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

$i = 1, 2, 4, 8, \cdots n$

$2^0, 2^1, 2^2, \cdots 2^k$

$a = 1, r = \frac{t_2}{t_1} = \frac{2}{1} = 2$

$t_k = ar^{k-1}$

$n = 1 * 2^{k-1}$

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

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

$k = 1 + \log_2(n)$

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

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

$T(n) = 3T(n-1) \quad -①$

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

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

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

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

$= 3^n T(0)$

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

$T(n) = 3T(n-1) \quad -①$

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

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

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

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

Complexity-

Date
Page

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

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

$T(n) = 2T(n-1)-1$ — ①
$T(n) = 2[2T(n-2)-1]-1$
$T(n) = 2^2 T(n-2)-2-1$ — ②
$= 2^2[2T(n-3)-1]-2-1$
$T(n) = 2^3 T(n-3) - 2^2-2-1$ — ③
⋮
$T(n) = 2^k T(n-k) - 2^{k-1} - 2^{k-2} - 2^{k-3} \cdots - 2^2 - 2 - 1$ — ④

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

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

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

5- Time complexity of

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

i	s
1	1
2	3
3	6
⋮	⋮

$\Rightarrow 1+3+6+10 \cdots k = n$
$\frac{k(k+1)(k+2)}{6} = n \Rightarrow O(k^3) = n$
$k = \sqrt[3]{n}$

Date
Page

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

⇒ i * i <= n
1 <= n
4 <= n
9 <= n
16 <= n
$k^2 <= n$
$k = \sqrt{n}$
So. 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 ++;
}
}

$i \rightarrow n/2$
$j \rightarrow \log n$
$k = \log n$

$O(\log n) * \log(n)$
T.C ⇒ $O(\log(\log(n)))$

Date
Page

8- function(int n) {
if (n==1) return;
for (i=1 to n) {
for (j=1 to n) {
print("*");
}
}
function(n-3);
}

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

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

9- void function(int n){
for(i=1 to n) { —— (n times)
for(j=1; j<=n; j=j+i)
print("*");
}

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

$\left(n * \sum_{i=1}^{n} n/i\right)$

$\Rightarrow O(n \log n)$

10- n^k is $O(a^n)$

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

$a^n + n^k \leq c \cdot a^n - a^n$

$a^n + n^k \leq a^n(c-1)$

$\frac{a^n + n^k}{a^n} \leq (-1)$

$c \geq 1 + \frac{n_0^k}{a^{n_0}} + 1$

let k=1, a=1.5

$c \geq 2 + \frac{n_0^k}{a^{n_0}} \Rightarrow c \geq 2 + \frac{n^1}{1.5^n}$

$n_0 = 1$

$c \geq 3 + 1 \quad c \geq 4$

Date
Page

11-
```
void fun (int n)
{
  int j=1, i=0;
  while (i<n)
    {
      i=i+j;
      j++;
    }
}
```

while executes (n-1) times, $\Rightarrow$ 'i=i+j' executes n times
& j++ execute the same.
So the time complexity is O(n)

12-

```
fib(n):
    if n<=1
        return 1
    return fib(n-1) + fib(n-2)
```

for $n>1$

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

let's say $c = 4$

for lower bound establish that $T(n-1) \sim T(n-2)$

though $T(n-1) \geq T(n-2)$, hence lower bound.

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

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

$= 2^*(2T(n-4)+c)+c$

$= 4T(n-4)+3c$

$= 8T(n-6)+7c$

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

$n - 2k = 0$

$k = n/2$

$T(n) = 2^{\wedge}(n/2)^*T(0) + (2^{n/2} - 1)^*c$

$= 2^{n/2} * (1+c) - c$

$T(n) \sim 2^{n/2}$

for upper bound $T(n-2) \sim T(n-1)$

$T(n-2) \leq T(n-1)$

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

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

$= 2^*(2T(n-2)+c)+c$

$= 4T(n-2)+3c$

$= 8T(n-3)+7c$

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

$n-k=0$

$n=k$

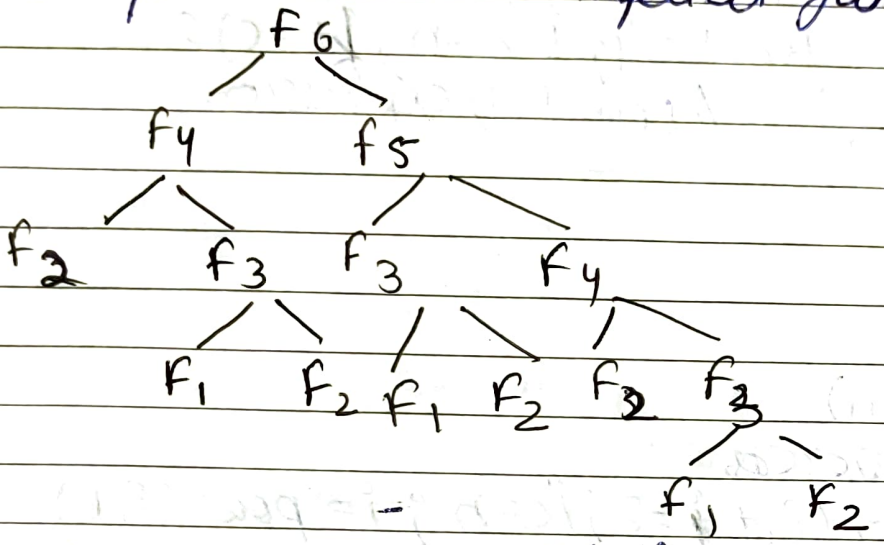
$= 2^n(1+c) - c$

$T(n) \sim 2^n$

Date
Page

Hence the time complexity is $O(2^n)$

Space Complexity :-
for fibonacci recursive implementation the space required is proportional to the maximum depth of the recursion tree because that is the maximum number of elements that can be present in the implicit func^n call stack



Hence Space complexity $O(N)$.

13- Programs with complexity :-
(i) $n(\log n)$:-
Merge Sort
Heap Sort
Quick Sort
Certain Divide & Conquer Algorithms based on optimizing $O(n^2)$ algo

(ii) n^3 - nested loops
①
```
for (int i=1 ; i<=n ; i+=c) {
  for (int j = 1; j<=n; j+=c) {
```

Date
Page

```
for(int k=1 ; k<=n ; k+=c){
    // some O(1) expressions
}
}
}
```

②
```
for(int i=n ; i>0 ; i-=c){
    for(int j=i+1 ; j<=n ; j+=c){
        for(k=j ; k<=n ; k+=c){
            // some O(1) expressions
        }
    }
}
```

(iii) log(log n)

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

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

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

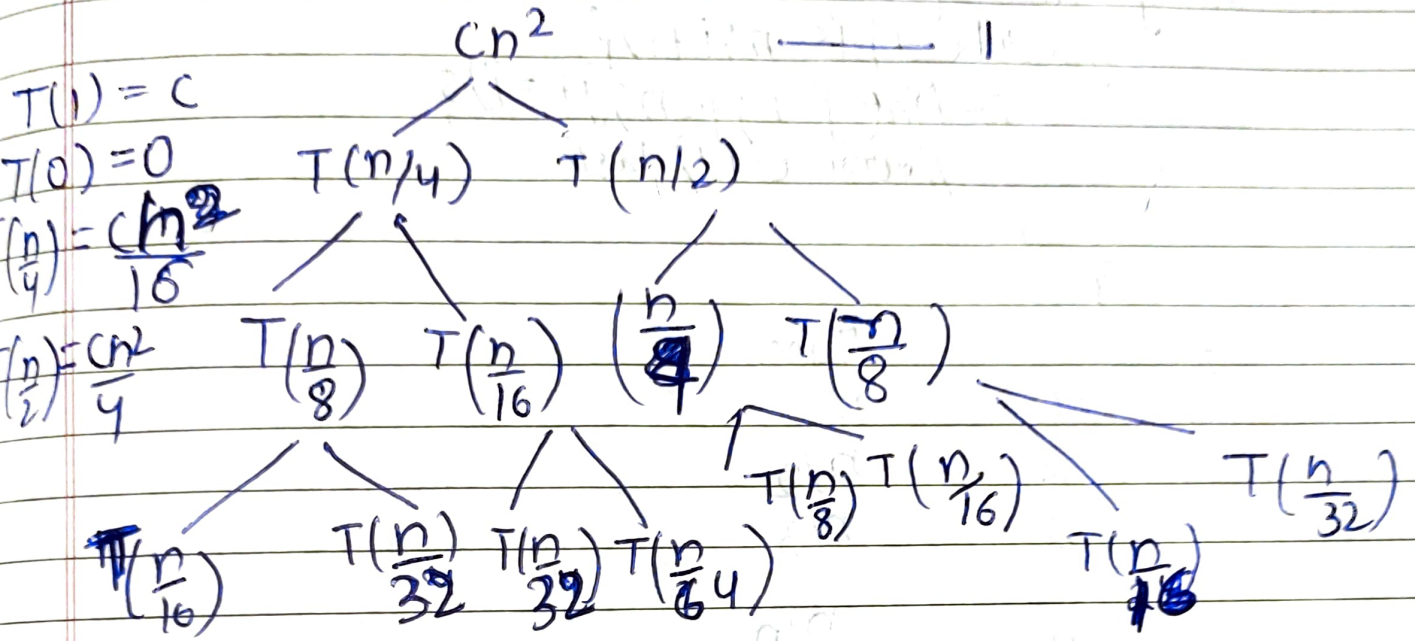
Therefore total T.C = O(log(log n))

②
```
for(int i=n ; i>1 ; i=func(i)) // func() is any constant root func
{
    // some O(1) expression
}
```

$i = n, n^{1/k}, n^{1/k^2}, \ldots, n^{\frac{1}{k}\log_k(\log(n))}$ which gives O(log(log n))

Date
Page

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



$T(1) = c$

$T(0) = 0$

$T(\frac{n}{4}) = \frac{cn^2}{16}$

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

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

G.P ratio

ratio $\Rightarrow \frac{5}{16}$

$$\frac{n^2}{(1 - 5/16)} = O(n^2)$$

Date
Page

15- time complexity fun()

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

	Times
for i=1 —	n
i=2	n/2
i=3	n/3
⋮	
i=n	n/n

$T.C = (n + n/2 + n/3 + \cdots n/n)$

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

$\Downarrow$

$\log(n)$

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

16- T.C

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

k = const;

$i = 2, 2^k, 2^{k^2}, 2^{k^3} \cdots 2^{k^{\log_k(\log(n))}}$ (last term must be $<= n$)

$\Downarrow$

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

total $\log_k \log(n)$ many iterations, each iteration take $O(1)$

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

18-

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

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

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

19- Linear Search Pseudocode for searching an element in Sorted array.

```
void LinearSearch(int arr[], int n, int key)
{
    for (i=0; i<n; i++)
    {
        if (arr[i]==key)
        {
            return i;
        }
    }
 return -1;
}
```

Date
Page

20- Recursive Insertion Sort :-

```
Void RecursiveInsertionSort(int arr[], int i, int n)
{
    int value = arr[i];
    int j = i;
    while (j > 0 && arr[j-1] > value)
    {
        arr[j] = arr[j-1];
        j--;
    }
    arr[j] = value;
    if (i+1 <= n)
    {
        RecursiveInsertionSort(arr, i+1, n);
    }
}
```

Iterative Insertion Sort :-

```
Void InsertionSortIterative(int arr[])
{
    int
    for(i=0; i<=arr.length(); i++)
    {
        value = arr[i];
        int j = i;
    }
    while (j!=0 && arr[j-1] > value)
    {
        arr[j] = arr[j-1];
        j--;
    }
    arr[j] = value;
}
```

Date
Page

Insertion sort is online sorting algorithm because insertion sort considers one input element per iteration and produces a partial solution without considering the future elements.

Sorting algorithm discussed :-

Selection Sort :- It sorts array by repeatedly finding the minimum element from the unsorted part & putting it at the beginning & repeat the process till the array is sorted.

Algorithm -

```
void SelectionSort (int arr[], int n)
{
  int i, j, temp, min;
  for(i=0; i<n-1; i++)
    {
      min=i;
    for(j=i+1; j<=n; j++)
      {
        if(arr[j]<arr[min])
          {
            min=j;
          }
      temp=arr[i];
      arr[i]=arr[min];
      arr[min]=temp;
}
}
```

Date
Page

→ Bubble Sort Algorithm :-

It works by repeatedly swapping the adjacent elements if they are in wrong order & repeats the process till array is sorted.

Algorithm -

```
void BubbleSort(int arr[], int n)
{
    int i, j, swap;
    for(i=0; i<=n; i++)
    {
        swap=0;
        for(j=0; j<n-i-1; j++)
        {
            if(arr[j] > arr[j+1])
            {
                swap(arr[j], arr[j+1]);
                swap=1;
            }
        }
        if(swap==0)
            break;
    }
}
```

Date
Page

21- Complexity

Bubble Sort

$T.C = \sum_{i=0}^{n-1} \left(\sum_{j=0}^{n-i-1} 1 \right)$

$= \sum_{i=0}^{n-1} (n-i) = n + (n-1) + \dots + 1$

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

Best case - $O(n)$

Average case - $O(n^2)$

Worst case - $O(n^2)$

Space complexity $O(1)$

Selection sort

Time complexity

Best :- $O(n^2)$

Average - $O(n^2)$

Worst - $O(n^2)$

Space complexity $O(1)$

Insertion Sort

Time Complexity

Best - $O(n)$

Avg - $O(n^2)$

Worst - $O(n^2)$

Space complexity $O(1)$

time can be reduce via flag variable so best case is $O(n)$

22-

Algorithm	Bubble Sort	Selection Sort	Insertion Sort
In-place	✓	✓	✓
Stable	✓		✓
Online			✓

Date
Page

23- Recursive Binary Search Pseudocode -

```
int BinarySearch (int arr[], int l, int r, int x)
{
    if (r >= l)                                   ——— O(1)
    {
O(1) ——— int mid = (l+r)/2;
        if (arr[mid] == x)
        {
O(1) ——— return mid;
        }
        else if (arr[mid] > x)
        {
T(n/2) ——— return BinarySearch (arr, l, mid-1, x);
        }
        else
        {
T(n/2) ——— return BinarySearch (arr, mid+1, r, x);
        }
    return -1;
}
```

Iterative Binary Search Pseudocode -

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

```
    else if (arr[mid] < x)
    {
        l = mid + 1;
    }
    else
    {
        r = mid - 1;
    }
}
return -1;
}
```

Space complexity of Binary search
(Best) SC - O(1) (iterative)
(Avg, worst) SC - O(logn) (Recursive)

Time complexity of Binary search
Best case - O(1)
Average case - O(logn)
Worst case - O(logn)

Space complexity of Linear search O(1)
Time complexity linear search -
Best - O(1)
worst - O(n)
Avg - O(n)

24- Recurrence Relation for Binary search -

$$T(n) = \begin{cases} c & \text{if } x >= l. \\ T(n/2) + c & \text{otherwise} \end{cases}$$

c ⇒ constant.