Name- Aayushi Agarwal

Sec- F

Roll NO- 03

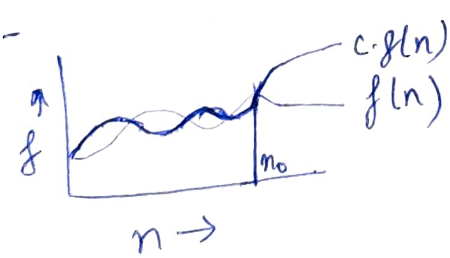
Subject- Design and Analysis of Algorithm

Univ Roll NO- 2014506

Ans 1) Asymptotic notations to analyse an algorithm running time identifying its behaviour as the input size for the algorithm increases. These notations are used to tell the complexity of an algorithm when input is very large.

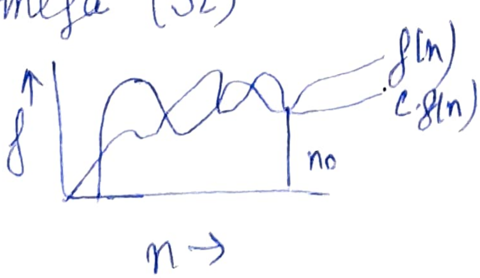
Types of asymptotic notations-

i) Big-Oh-



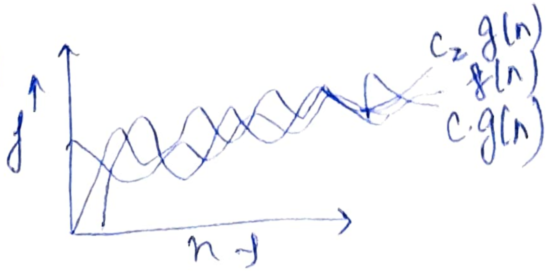
$f(n) = O(g(n))$
if and only if
$f(n) \leq c \cdot g(n)$
$\forall\ n \geq n_0$

(ii) Big Omega (Ω)



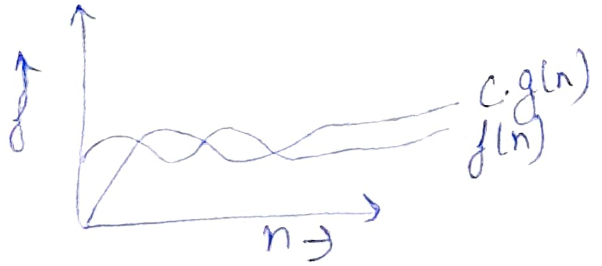
$f(n) = \Omega\, g(n)$
if & only if
$f(n) \geq c \cdot g(n)$
$\forall\ n \geq n_0$

(iii) Theta (θ)



$f(n) = \theta\, g(n)$
if & only if
$c \cdot g(n) \leq f(n) \leq c_2 g(n)$
$\forall\ n \geq \max(n_1, n_2)$

(iv) Small - Oh (o)



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

$f(n) < c \cdot g(n) \ \forall \ n > n_0$.

② $i = 1, 2, 4, 8 \dots n$

$2^0, 2^1, 2^2, 2^3 \dots 2^k$ — GP

$a = 1, \quad r = 2$

$t_n = a r^{k-1}$

$= 1 \times 2^{k-1}$

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

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

$= \log_2(n) + 1$

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

③ $T(n) = 3T(n-1)$ — ①

$n > 0$

$T(1) = 1$

put $n = n-1$ in Eq ①

$T(n-1) = 3T(n-2)$ — ②

Put $T(n-1)$ in Eq ①

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

$T(n) = 9T(n-2)$ — ③

Put $n = n-2$ in Eq ①

$T(n-2) = 3T(n-3)$ — ④

Put $T(n-2)$ in Eq ③

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

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

$T(n) = 3^k T(n-k)$ — 5

~~T(n) = 27 T(~~ $T(k) = 1$ $n-k = 1$

$\boxed{k = n-1}$ — 6

from 5 & 6

$T(n) = 3^{n-1} T(1)$

$T(n) = \frac{3^n}{3} \times 1$ = $\boxed{T.C = O(3^n)}$

(4) $T(n) = 2T(n-1) - 1$ — 1

$T(1) = 1$

Put $n = n-1$ in Eq 1

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

Put $T(n-1)$ in Eq 1

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

$T(n) = 4T(n-2) - 1$ — 2.

Put $n = n-2$ in Eq 1.

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

Put $T(n-2)$ in Eq 2.

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

$T(n) = 8T(n-3) - 4 - 2 - 1$ — 3

$T(n) = 2^k [T(n-k)] - 2^{k-1} - 2^{k-2} \ldots 2^1 - 2^0$ — 4

$T(1) = 1$

$n-k = 1$

$k = n-1$ — 6

from 4 & 5

$T(n) = 2^{n-1}[T(n-(n-1))] - 2^{n-2} - 2^{n-3} \text{———} 2^0$

$T(1) = 1$

$n-k = 1$

$k = n-1$ —— (6)

$= 2^{n-2} - 2^{n-1} - 2^{n-3} - 1$

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

$= \frac{1}{2} \times 1 = \frac{1}{2}$

$$\boxed{T.C = O(1)}$$

(5) $n = 1, 3, 6 \ldots k \rightarrow AP$

$T.C = \frac{k(k+1)}{2} = O\left(\frac{k^2+k}{2}\right) = O(k^2)$

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

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

$1 + 1 + (n+1)^2 + n + n \Rightarrow n^2 + 4n + 3$

$2 + n^2 + 2n + 1 + 2n$

$O(n^2 + 4n + 3) \Rightarrow O(n^2)$

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

(7)
```
void function (int n) {
    int i, j, k, count = 0;
for (i = n/2; i <= n; i++)          — O(n)
 for (j = 1; j <= n; j = j * 2)     — log(n)
 for (k = 1; k <= n; k = k * 2)     — log(n)
    count ++; }
```

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

$\log^2(n) \Rightarrow \boxed{O[\log^2(n)]}$

(8) function (int n) {
    if (n==1) return; — 1
for (i=1 to n)
    for (j=1 to n)] $n*n$
        printf ("*"); — 1
    }
} function (n-3) ⟶ $n*n^2$
}

$1+n^2+1+n^3 \Rightarrow n^3+n^2+2$

$\boxed{O(n^3)}$

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

i	j	times
1	$1 \to n$	$\frac{n+1}{2}$
2	$1 \to n$	$n+1/2$
⋮		
n	$1 \to n$	$n+1/2$

$T.C = \log n \cdot \frac{(n+1)}{2}$

$= O\left(\frac{n+1}{2} \log n\right) \Rightarrow O(n \log n)$

(10) $n^k \leq c a^n$

$a^n + n^k \leq c a^n \to a^n$

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

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

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

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

$c \geq 2 + \frac{n_0'}{1.5^n}$

$n_0 = 1$

$c \geq 2 + \frac{1}{1.5}$

$c \geq 3.0 + 1$

$c \geq 4$

(11) Time Complexity = O(n)

The execution of different code lines are -

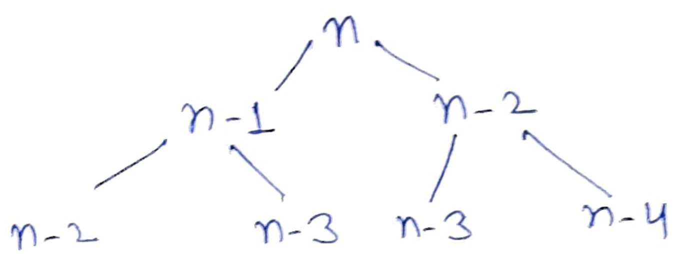
1. while $\Rightarrow (n-1)$
2. $i = i+j \Rightarrow (n)$
3. $j{+}{+} \Rightarrow (n)$

$$T.C = n + n + n - 1$$
$$= 3n - 1$$
$$TC = O(3n-1)$$

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

(12) The main working of fibonacci Series is

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



$$T(n) = 1 + 2 + 4 \cdots 2^n$$
$$a = 1, \quad r = 2$$
$$\frac{a(r^k - 1)}{r - 1} \Rightarrow \frac{1(2^{n+1} - 1)}{2 - 1} = 2^{n+1}$$
$$T(n) = O(2^{n+1}) = O(2^n * 2^1) \Rightarrow O(2^n)$$

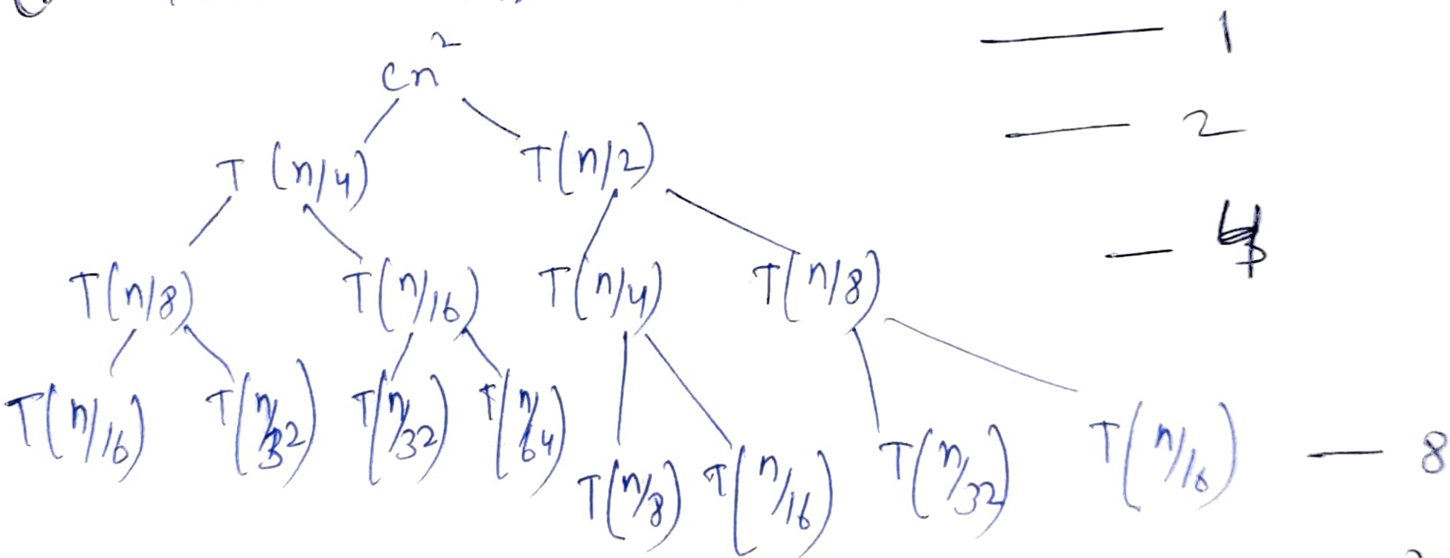
(13) $O(n(\log n))$

```
int n;
for (int i=0; i<n ; i++) {
    for (int j=n; j>0; j/=2) {
            printf ("*");
    }
}
```

$O(n^3)$

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

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



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

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

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

i	j	times
1	$1 - n$	n
2	$1 \to n$	$n/2$
$\vdots$		
n		n/n

$T(n) = n + \frac{n}{2} + \frac{n}{3} + \cdots \frac{n}{n}$

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

$\boxed{T(n) = n\log(n)}$

(16) $i = 2, 2^{c^1}, 2^{c^2}, 2^{c^3} \ldots\ldots 2^{c^{\log_c \log(n)}}$

The last term has to be $<= n$

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

There are in total $\log_c(\log(n))$ iterations each take a constant amount of time to run

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

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

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

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

(19)
```
Ins funcn (int arr[N], key) {
  for (i=0 to n-1) {
     if (A[i] = key) {
        return i; }
  }
  return -1;
}
```

(20) (a) Iterative Insertion Sort

```
void InsertionSort (int arr[], int n) {
      int i, temp, j;
   for (int i=1; i<=n-1; i++)
        temp = arr[i];
        j = i-1;
```

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

Recursive Inseotion Sort

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

(21)

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

(22)

Algorithm	In-place	Stable	Online
Bubble	✓	✓	✗
Selection	✓	✗	✗
Insertion	✓	✓	✓
Merge	✗	✓	✗
Quick	✗	✗	✗
Heap	✓	✗	✗

(23) Iterative Binary Search.

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

Recursive Binary Search

```
int Binary Search (int arr[], int l, int r, int n)
    if (l>r)
        return -1;
    int m= (l+r)/2;
    if (arr [m] =n;
        return m;
    else if (arr [m] <n)
    return BinarySearch (arr, m+1, r, n);
```

```
else
return BinarySearch(arr, l, m-1, n);
}
```

Time Complexity	Space complexity
Linear (Recursive) → O(n)	O(1)
Binary (Recursive) → O(n)	O(log n)
Linear (Iterative) → O(1)	O(1)
Binary (Iterative) → O(1)	O(1)

(24) Recurrence Relation for Binary search

$$\boxed{T(n) = T(n/2) + 1}$$