Assignment

1. What do you understand by Asymptomatic notations? Define different asymptomatic notations with examples.

Ans → It is a mathematical notation that describes the behaviour of a function as its input size approaches infinity. It is used to analyze the time and space complexity of algorithms.

Different Types of Asymptomatic Notations are:

i) Big Oh (O) - it is used to describe the upper bound of the running time or space complexity of algorithms. it is the worst case scenario of an algorithm.

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

Iff $f(n) \leq c\, g(n)$

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

ii) Big Omega (Ω) - it is used to describe the lower space complexity of an algorithm. it is the best case scenario of an algorithm.

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

if $f(n) \geq c\, g(n)$

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

iii) Theta (Θ) - it is used to describe the tight bound of the running time or space complexity of an algorithm. it is the average case scenario of an algorithm.

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

iff $c_1 g(n) \leq f(n) \leq c_2 g(n)$

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

iv) Small Oh (o) - it is used to describe the strict upper bound of running time or space complexity of an algorithm. it is a more strict version of Big oh notation.

$$f(n) = o(g(n))$$
$$\text{iff } f(n) < c\,g(n)$$
$$\forall\ n > n_0, \forall\ c > 0$$

v) Small Omega (ω) - it is used to describe the strict lower bound of running time or space complexity of an algorithm. it is more strict version of Big-Omega notation.

$$f(n) = \omega(g(n))$$
$$\text{iff } f(n) > c[g(n)]$$
$$\forall\ n > n_0, \forall\ c > 0$$

2- What should be time complexity of

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

Soln →

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

k- terms

it is a G.P.

$$a_n = ar^{n-1}$$
$$n = 1.2^{k-1} \qquad a_n = n,\ a = 2,\ n = k$$
$$2n = 2^k$$
$$\log(2n) = k\log(2)$$
$$k = \log(2) + \log(n)$$
$$k = 1 + \log(n)$$

$\therefore\ O(\log_2(n))$

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

Sol^n→ By forward Solution

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

$T(0) = 1$

$T(1) = 3T(1-1) = 3T(0) = 3$

$T(2) = 3T(2-1) = 3T(1) = 3.3$

$T(3) = 3T(3-1) = 3T(2) = 3.3.3$

$\therefore O(3n) = O(n)$

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

Sol^n→ Here, $a = 2$, $b = 1$, $c = \log_2(2) = 1$

$f(n) = 1$

$n^c = f(n)$

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

5- What should be time complexity of

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

Sol^n→ The input size is n

i	s
2	3
3	6
4	10
5	15

k^{th} terms

After k^{th} term, relation S call become $\frac{k(k+1)}{2}$

The loop will stop when

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

$$k^2 + k > 2n$$

$$k^2 > 2n$$

$$k = \sqrt{2}\, n$$

Since, ignoring constants,

$\therefore O(\sqrt{n})$

6- Time complexity of

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

Sol^n → The input size is n

i		
1	→	1
2	→	4
3	→	9
4	→	16
5	→	25

loop will terminate when

$$k^2 > n$$

$$k^9 > \sqrt{n}$$

$\therefore O(\sqrt{n})$

7- Time complexity of -

```
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++;
}
```

Sol? → Time complexity of inner most loop

k = 1 to n ; k = k * 2

⇒ 1, 2, 4, 8, ---- , k^{th} term

now,

k^{th} term = 2^{k-1}

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

$2n = 2^k$

$\log(2n) = k \log(2)$

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

$k = 1 + \log(2n)$

it means for each value of j, this loop runs $(1 + \log_2 n)$ times

Time Complexity of middle loop

j = 1 to n ; j = j * 2

Means for each value of i, this loop runs $(1 + \log_2 n)$ times

Complexity of outer-most loop

i = n/2 to n ; i = i + 1

⇒ n/2, n/2 + 1, n/2 + 2, n/2 + 3, ---, k^{th}

k^{th} term = $\frac{n}{2} + k$

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

this loop will run $n/2$ times.

Total Complexity $= \frac{n}{2} + (1 + \log_2 n) + (1 + \log_2 n)$

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

$\Rightarrow O(n(\log_2 n)^2)$

9- Time Complexity of

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

Sol^→ The function consists of two nested loops that iterates over the variable i and j. The outer loop iterates over n times and inner loop iterates n/i times.

$n + n/2 + n/3 + \cdots$ k terms

this is Hermonic Series

$\log(n) + 0.572 + O(1/n)$

∴ Time Complexity is $O(n \log(n))$

Q. Time Complexity of -

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

Sol^n→ A recursive function is given with two nested loop that iterates over i and j. The outer loop iterates n times and inner loop iterates n times. i.e $n*n$ times.

At each recursive call, value of n decreased by 3. function will be called total of $n/3$ times.

∴ Time Complexity = $O(n^2 (n/3)^2)$