Assignment: 1

Q1 Assympotic notations are mathematical tools used for analysis of algorithms that describe their behaviours. They provide a way to express the time and space complexity of an algorithm.

a. Big O: Represents the upper bound of an algo.

e.g. if an algo has time complexity of $O(n^2)$ it means its worst-case running time grows quadratically with input size.

b. Omega Ω: Represents the lower bound of algo.

e.g. if an algo has time complexity of $\Omega(n)$ it means its best-case running times grows linearly with input size.

c) Theta notation Θ: Represents range (both upper & lower bound)

e.g. if an algo has a time complexity of $\Theta(n)$, it means ~~that~~ its running time grows linearly with input size, both best & worst cases.

Q2 for(i = 1 to n) — i=1, 2, 4, 8, 16 ... n/2 ... n
{
    i = i * 2;
}

$2^k \geq n$

taking log on both sides

$k \geq \log_2(n)$

Complexity = $O(\log_2 n)$.

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

$T(0) = 1$
$T(1) = 3T(0) = 3(1) = 3$
$T(2) = 3T(1) = 3(3) = 9$
$T(n) = 3^n$

verification :- $T(n) = 3T(n-1) = 3 \cdot 3^{n-1} = 3^n$

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

Q4 $T(n) = \begin{cases} 2T(n-1)-1 & \text{if } n>0 \\ 1 & \text{otherwise} \end{cases}$

$T(0) = 1$
$T(1) = 2T(0) - 1 = 2(1) - 1 = 1$
$T(2) = 2T(1) - 1 = 2(1) - 1 = 1$
$T(3) = 2T(2) - 1 = 2(1) - 1 = 1$
⋮

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

Q.5
```
int i=1, s=1;        ---- (1)
while (s<=n)
{
    i++;
    s=s+i;
    print ("#");
}
```

$i = 1, 2, 3, 4, 5, \ldots$
$S = 1, 3, 6, 10, 15, \ldots n$

$$s = \frac{i(i+1)}{2}$$

$$\frac{i(i+1)}{2} \leq n$$

$$i(i+1) \leq 2n$$

$$i^2 + i - 2n \leq 0$$

$$\therefore \; i < \frac{-1 + \sqrt{1+8n}}{2}$$

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

Q.6 void function (int n)
```
{
    int i, count = 0          ———— (1)
    for (i=1; i * i<=n; i++)
    {
        count ++
    }
}
```
$i = 1, 2, 3, 4, \ldots \sqrt{n}$
$i^2 = 1, 4, 9, 16 \ldots n^2$

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

Q7 void function (int n)
```
{
    int i, j, k, count = 0;
    for (i=n/2; i<=n; i++)          ———— n/2
      for (j=1; j<=n; j=j*2)        ———— log2(n)
          for (k=1; k<=n; k=k*2)    ———— log2(n)
              count ++
}
```

$\frac{n}{2} \times \log_2(n) \times \log_2(n)$

complexity = $O(n \log^2(n))$

Q8 function(int n) ——— $T(n)$

{

if (n==1) return;

for (i=i to n) { ——— n

for (j=1 to n) { ——— n^2

print("*"); ——— n^2

}

}

function (n-3); ——— $T(n-3)$

}

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

Time complexity = $O(n^2)$

Q.9 void function (int n)

{ for (i=1 to n) {

for (j=1; j<=n; j=j+i)

print("*");

}

}

$i = 1, 2, 3, 4, ----$

$j = 1, 3, 6, 10, ----$

$i = 1$, n times

$i = 2$, $n/2$ Times

$i = 3$, $n/3$ Times

$1 + n/2 + n/3 + ------ + 1$

complexity :- $O(n \log n)$

Q-10 n^k $(k \geq 1)$ $c^n (c > 1)$

c^n grows faster than n^k.