Name : Nandini Gupta

Section : D

University roll no : 2014421

Semester : 5

Design AND Analysis of Algorithms : Assignment-1

Ans 1 :- Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis.

⇒ Θ notation : It bounds a function from above & below so it defines exact asymptotic behaviour.

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 <= c_1 * g(n) <= f(n) <= c_2 * g(n)$ for all $n >= n_0\}$

⇒ Big O notation : It defines an upper bound of an algorithm it bounds function only from above.

$O(g(n)) = \{f(n)$: there exists positive constants c and n_0 such that $0 <= f(n) \leq c * g(n)$ for all $n >= n_0\}$

⇒ Ω notation : It provides asymptotic lower bound. It can be useful when we have lower bound on time complexity of an algorithm.

$\Omega(g(n)) = \{f(n)$: there exists positive constants c and n_0 such that $0 <= c * g(n) <= f(n)$ for all $n >= n_0\}$

Ans 2 : for(i = 1 to n) { i = i*2 }

Time complexity = $O(\log n)$

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

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

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

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

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

⋮

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

$= 3n(0)$

$= 3n$

$\therefore O(3^n)$

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

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

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

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

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

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

- - -

- - -

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

$= 2^n - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots - 2^2 - 2^1 - 2^0$

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

$T(n) = 1$

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

Ans 5:

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

$1+2+3+ \ldots + k \Rightarrow \frac{k(k+1)}{2} > n$

$k^2 + k > 2n$

$k^2 > n$

$k = \sqrt{n}$

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

Ans 6: void function(int n) {
```
void function(int n) {
  int i, count=0
  for (i=1; i*i<=n; i++)
    count ++
}
```
T.C $\boxed{O(n)}$

Ans 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)         // O(logn)
      for(k=1; k<=n; k=k*2)        // O(logn)
        count ++
}
```
T.C $= \boxed{O(n \log^2 n)}$

Ans 8:
```
function (int n) {
  if (n==1) return;
  for (i=1 to n) {
    for (j=1 to n) {
      printf ("*");
    }
  }
  function(n-3);
}
```
T.C = $O(n^2)$

Ans 9:
```
void function (int n) {
  for (int i= 1 to n) {
    for (j=1; j<=n; j=j+i)
      printf ("*");
  }
}
```