7-March-2022

TUTORIAL - 1

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

2. What should be time complexity of:

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

3. $T(n) = 3T(n-1)$ if $n>0$,
   $T(1) = 1$

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

5. What should be time complexity of:

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

6. Time complexity of:

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

Sekhar.

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 ++;
}
```

8. Time complexity of

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

9. Time complexity of:

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

10. For the functions, n^k and c^n, what is the asymptotic relationship between these functions?

Assume that $k >= 1$ and $c > 1$ are constants. Find out the values of c and n_0 for which relation holds.

Solutions.

1. When we are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound, is known as asymptotic efficiency of algorithms.

Asymptotic notations: The notations used to described the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers.

The different asymptotic notations are:

(i) Ω - notation

Omega notation represents the lower bound of the running time of an algorithm.

Thus, it provides the best case complexity of an algorithm.

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0, such that $0 \leq c\,g(n) \leq f(n)$ $\forall\ n \geq n_0 \}$

(ii.) Theta - Notation (Θ -notation)

It encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

$\Theta(g(n)) = \{ f(n)$: there exist positive constant c_1, c_2 and n_0, such that $0 \leq c_1\,g(n) \leq f(n) \leq c_2\,g(n)$ $\forall\ n \geq n_0 \}$

(iii.) Big - O Notation (O-notation)

It represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n), \forall n \ge n_0 \}$

3. $T(n) = 3T(n-1)$ ——— (1)

put $n = (n-1)$ in (1)

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

$= 3T(n-2)$ ——— (2)

put value of $T(n-1)$ in (1),

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

$= 9T(n-2)$ ——— (3)

put $n = (n-2)$ in (1),

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

$= 3T(n-3)$ ——— (4)

put value of $T(n-2)$ in (3),

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

$= 27T(n-3)$ ——— (5)

put $n = (n-3)$ in (1),

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

$= 3T(n-4)$ ——— (6)

put value of $T(n-3)$ in (5),

$T(n) = 27[3T(n-4)]$

$= 81T(n-4)$ ——— (7)

General form :

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

Let, $n-k=1$

$n-1=k$

$\Rightarrow 3^{n-1} T(n-n+1)$

$\Rightarrow (3^n/3^1) T(1)$

$\Rightarrow \{3^n/3(1)\}$

$\Rightarrow O(3^n/3)$

$\Rightarrow \underline{O(3^n)}$.

5. $T(n) = 2T(n-1) - 1$ ——— (1)

put $n=(n-1)$ in (1),

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

$= 2T(n-2)-1$ ——— (2)

put value of $T(n-1)$ in (1),

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

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

$= 4T(n-2)-3$ ——— (3)

put $n=(n-2)$,

$$T(n-2) = 2T(n-2-1) - 1$$
$$= 2T(n-3) - 1 \quad \text{——— (4)}$$

put value of $T(n-2)$ in (3),

$$T(n) = 4[2T(n-3)-1] - 3$$
$$= 8T(n-3) - 4 - 3$$
$$= 8T(n-3) - 7 \quad \text{——— (5)}$$

General form:

$$T(n) = 2^k T(n-k) - (2^k - 1) \quad \text{——— (6)}$$

Let, $n-k = 1$

$n-1 = k$ put value of k in (6),

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

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

$$= \frac{2^n}{2} - (2^{n-1} - 1) \quad = 2^{n-1} - 2^{n-1} + 1$$

$$\Rightarrow O\left(\frac{2^n}{2} - (2^{n-1} - 1)\right) \quad O(1)$$

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

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

$\Rightarrow \sum_{i=1}^{n} (i = i*2)$ 1 + 1 + 1 + . . . k times

$\therefore 2^k >= n$

$2^k = n$

$\Rightarrow$ taking log on both sides,

$k \log_2 2 = \log_2 n$

$k = \log_2 n$

$\Rightarrow O(\log_2 n)$

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

$\Rightarrow$ k = total number of iterations

i = increases by one for each iteration.

$S_i = S_{i-1} + i$

= sum of first 'i' positive integers.

$i = 1, 2, 3, 4, 5, \ldots$ $\quad s = 1 + 3 + 6 + 10 + \ldots$

while loop terminates if:

$1 + 2 + 3 + \ldots + k = [k(k+1)/2] > n$

$\Rightarrow \frac{k(k+1)}{2} <= n$

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

$\Rightarrow O(k^2) <= n$

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

$\Rightarrow T(n) = O(\sqrt{n})$.

6. $i^2 = n$

$i = \sqrt{n}$

$i = 1, 2, 3, 4, \ldots, \sqrt{n}$

$\sum_{i=1}^{n} 1 + 2 + 3 + 4 + \ldots + \sqrt{n}$

$\Rightarrow T(n) = \frac{\sqrt{n}(\sqrt{n}+1)}{2}$

$= \frac{n + \sqrt{n}}{2}$

$\Rightarrow T(n) = O(n)$.

7. for $k = k * 2$

$k = 1, 2, 4, 8, \ldots, n$

GP = $1, 2, 4, 8, \ldots, n$

$a = 1$

$r = 2$

$$\text{Sum}_{\text{n-terms}} = \frac{a(r^n - 1)}{r - 1}$$

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

$$n = 2^k - 1$$

Taking log on both sides,

$$\log_2 n = k \log_2 2 - \log_2 1$$

$$\log_2 n = k$$

⇒

i	j	k
1	log n	log n * log n
2	log n	⋮
⋮	⋮	
n	log n	log n * log n

$n*$

⇒ $O(\log n * \log n) \Rightarrow O(n \log^2 n)$

8.
```
function (int n)
{
  if (n==1)
    return;                          // O(1)
  for (i=1 to n)                     // i = 1, 2, 3, 4, ... n => O(n)
  {
    for (j=1 to n)                   // j = 1, 2, 3, 4, ... n => O(n^2)
    {
      printf ("*");
    }
  }
  function (n-3);                    // T(n/3)
}
```

Using Master's Method.

$\Rightarrow T(n) = T(n/3) + n^2$

$a = 1 \quad b = 3 \quad f(n) = n^2$

$c = \log_3 1 = 0$

$\Rightarrow n^c = 1 < f(n)$

$\Rightarrow T(n) = \underline{\Theta(n^2)}$.

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

for $i = 1 \rightarrow j = 1, 2, 3, 4, \ldots, n$
$= n$

$i = 2 \rightarrow j = 1, 3, 5, \ldots, n$
$= n/2$

$i = 3 \rightarrow j = 1, 4, 7, \ldots, n$
$= n/3$

$i = n \rightarrow j = 1; \ldots \quad n$
$= n/n$

$$\Rightarrow \sum_{j=n}^{1} n + \frac{n}{2} + \frac{n}{3} + \ldots + 1$$

$$\sum_{j=n}^{1} n\left[1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}\right]$$

$$\sum_{j=n}^{1} n\,[\log n]$$

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

10. Given:

n^k and c^n

and:

$$n^k = O(c^n)$$

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

$\forall\ n \geq n_0$ and some constant $a > 0$

for $n_0 = 1$

$c = 2$

$\Rightarrow 1^k \leq a\, 2^1$

$n_0 = 1$ and $c = 2$.