TUTORIAL – 1

(1). Asymptotic notations are mathematical notations that are used to represent the running time of an algorithm, when the input tends towards a particular value or a limiting value.

Following are the different asymptotic notations:

(i) Big-O notation ($O(fn)$)

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

e.g – Suppose we need to find an element in an array of 'n' numbers. Therefore, for worst-case the time required to find the key can be represented as "$O(n)$".

(ii) Omega Notation ($\Omega(fn)$)

→ It represents the lower bound of the running time of an algorithm. Hence, it provides the best case complexity of an algorithm.

e.g – for an array of 'n' elements, if we look for a key, the best case will be if we find the key at first index; that will be $\Omega(1)$. This is omega notation.

(iii) Theta Notation ($\theta(fn)$)

→ Theta function encloses the function from above & below. Hence, it's used to analyze the average-case complexity of an algorithm.

e.g – for an expression $8n^2+5n$ representing the time complexity of certain algorithm, we remove $5n$ and can represent the time complexity as $\theta(n^2)$ as it's the average time for that algorithm.

② for (i=1 to n)
{ i = i*2; }

i	power of 2
1	2^0
2	2^1
4	2^2
8	2^3
16	2^4
32	2^5
⋮	⋮
n	2^k

to reach the ~~program~~ 'n' the program will run k times.

Hence $n = 2^k$

take log

$\log n = k$.

Hence, time complexity is $O(\log(n))$.

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

<u>Using backward substitution</u>

$T(n) = 3T(n-1)$ — ①

put $n = (n-1)$ in ①

$T(n-1) = 3T(n-2)$ — ②

put ② in ①.

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

put $n = (n-3)$ in ①

$\cancel{T(n-3) = 9(3T(n-3))}$

$\cancel{T(n-3) = 27(T(n-3))}$

$\cancel{T(n-3) = 3^3(T(n-3))}$

$T(n-3) = 3T(n-3)$ — ④

put ④ in ①

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

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

$T(n) = 3^3T(n-3)$ — ⑤

⑤ can be re-written as

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

$n-k = 0$

$n = k.$

$T(n) = 3^nT(0)$

$= 3^n$

$T(0) = 1$

Hence, time complexity is $O(3^n)$.

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

$T(n) = 2T(n-1) - 1$ — ①

<u>Using backward substitution</u>

put $n = n-1$

$T(n-1) = 2T(n-2) - 1$ — ②

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

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

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

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

(5)

i	s
1	1
2	3
3	6
4	10
5	15
⋮	⋮

$$s = i_s + i_4 + i_3 \ldots i_1$$

Hence,

$$1 + 2 + 3 \ldots k > n \quad \text{(loop terminates)}$$

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

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

$$\frac{k^2}{2} + \frac{k}{2} > n$$

$$k = \sqrt{n}$$

Hence, the time complexity is $O(\sqrt{n})$.

(6).

i	$i*i$
1	1
2	2
3	6
4	24
5	120
⋮	⋮
k	n

$k(k+1) > n$ (terminating condition)

$k^2 + k > n$

put $k = n$

hence, time complexity is $O(n^2)$

(7).

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

Hence, time complexity is $O(n(\log n^2))$

(8)

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

$$\underbrace{(n-3)\ (n-6)\ (n-9)\ \cdots\ 1}_{k}$$

$\Rightarrow$ $a = n-3$ $\quad d = -3$

$1 = (n-3) + (k-1)(-3)$

$1 = n-3+(-3k)+3$

$3k = n-1$

$k = \frac{n-1}{3}$

$= O(n^3)$ $\qquad \{O(n*n^2)\}$

Time complexity is $O(n^3)$.

(9)

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

for $i=1 \longrightarrow j = n$ times

$i=2 \longrightarrow j = n/2$ times

$i=3 \longrightarrow j = n/3$ times

$\vdots$

$i=k \longrightarrow j = \frac{n}{k}$ times

$\therefore$ $i=n$ & $j = \frac{n}{n}$ times

Time Complexity $= \left(n + \frac{n}{2} + \frac{n}{3} \cdots \frac{n}{n}\right)$

$= n\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}\right)$

$= n\log n$

$\therefore$ Time Complexity $= O(n\log n)$

(10) $f(n) = n^k$, $g(n) = c^n$

where $k=1$ & $c=2$

$\Rightarrow f(1) = (1)^1$ $\quad g(1) = 2^1$

$f(1) < g(1)$

$\Rightarrow f(2) = (2)^1$ $\quad g(2) = 2^2 = 4$

$f(2) < g(2)$

satisfies O notation $\quad f(n) \leq g(n)$

$f(n_0) = C_0 . c^{n_0}$

$n_0^k = C_0 . c^{n_0}$

$$n_0^1 = c_0 \cdot 2^{n_0}$$

$$\left(\frac{n_0}{c_0}\right)^1 = 2^{n_0}$$

Comparing : $n_0 = 1$, $\frac{n_0}{c_0} = 2$

$$c_0 = \frac{1}{2}.$$

$$f(n) \leq 0.5\, g(n)$$

$$f(n) = O\, g(n).$$