# ASSIGNMENT - 1

(1) Asymptotic Notations such as Big O, Big Omega and Big Theta, are used in computer science to describe the growth rate of algorithms' time or space complexity as the input size approaches infinity. They provide a concise way to analyze and compare the efficiency of algorithms without getting into specific implementation details.

Different asymptotic notations are :

(i) Big O (O)

$f(n) = O(g(n))$
$g(n)$ is tight upper bound of $f(n)$.
iff $f(n) \leq c\,g(n)$
$\forall\ n \geq n_0$ and for some constant, $c > 0$.

eg, $f(n) = 2n^2 + 3n + 1$ is $O(n^2)$.

(ii) Big Omega ($\Omega$)

$f(n) = \Omega(g(n))$
$g(n)$ is tight lower bound of $f(n)$
iff $f(n) \geq c\,g(n)$
$\forall\ n \geq n_0$ & for some constant, $c > 0$

eg, $f(n) = n^2$ is $\Omega(n)$

(iii) Theta ($\Theta$) notation

theta gives both tight upper + lower bound

$f(n) = \Theta(g(n))$
$[f(n) = O(g(n))\ \&\ f(n) = \Omega(g(n))]$
iff $c_2\, g(n) \leq f(n) \leq c_1\, g(n)$
$\forall\ n > \max(n_1, n_2)$ & for some constant. $c_1, c_2 > 0$

eg, $f(n) = n^2$ is $\Theta(n^2)$

(iv) Small (o)

$f(n) = o(g(n))$
$g(n)$ is upper bound of $f(n)$
iff $f(n) < c\,g(n)$
$\forall\ n > n_0,\ c > 0$.

eg, $f(n) = n$ is $o(n^2)$

(v) Small $\omega$

$f(n) = \omega(g(n))$
$g(n)$ is lower bound of $f(n)$
iff $f(n) > c\,g(n)$
$\forall\ n > n_0,\ c > 0$

eg, $f(n) = n^2$ is $\omega(n)$.

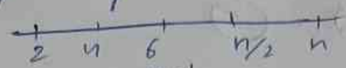
(2) for ( i = 1 ; i < n ; i ++ )

{

$i = i * 2;$

}

complexity ?




$a_n = ar^{n-1}$

$a_n = 2 \times 2^{k-1}$

$a_n = 2^k$

$n = 2^k$

$\log_2 n = \log_n 2^k$

$\log_2 n = k \log_2 2$

$k = \log_2 n$

$T.C = O(\log_2 n)$

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

$T(0) = 1$.

Put $n = n-1$ in ①

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

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

$T(n) = 9T(n-2)$ —②

put $n = n-2$ in ①

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

Put value of $T(n-2)$ in 2

$T(n) = 27T(n-3)$ — ③

$T(n) = 3^k T(n-k)$ — ④

$n - k = 0$

$n = k$

put this in 4

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

$= 3^n$

$\Rightarrow O(3^n)$

$= 2^n - 2^{n-1} - 2$

(4) $T(n) = 2T(n-1) - 1$ — (1) if $n > 0$ or $1$

$T(0) = 1$

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

$= 2 - 1 = 1$

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

$= 2 - 1 = 1$

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

$= 2 - 1 = 1$

$\Rightarrow O(1)$

(5)
```
int i=1, s = 1
while (s <= n){
    i++;
    s+=i;
    printf('#');
}
```
$\rightarrow (1, 2, 3, 4 \ldots k)$

| $i$ | $s$ |
|---|---|
| 1 | 1 |
| 2 | $1+1=2$ |
| 3 | $1+1+2=4$ |
| 4 | $1+1+2+3=7$ |
| $\vdots$ | $\vdots$ |
| $k$ | |

Assume $s > n$ (stopping condition)

$$1+1+2+3+4+\ldots+k$$
$$\therefore s = 1 + \frac{k(k+1)}{2}$$
$$\frac{1+k(k+1)}{2} > n \Rightarrow \frac{k(k+1)}{2} > n-1$$
$$k^2 > n-1 \Rightarrow k > \sqrt{n-1}$$
$$\Rightarrow T.C = O(\sqrt{n})$$

(6) Time complexity of
```
void function (int n){
    int i, count =0;
    for(i = 1, i*i < n ; i++)
        count ++ ;
}
```
The loop will run $\sqrt{n}$ times

$$i*i < n$$
$$i^2 < n$$
$$i < \sqrt{n}$$
$$\Rightarrow T.C = O(\sqrt{n})$$

(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 ++ ;
}
```
$i$ will run $n/2 + 1$ times

as for $j$ and $k$ –

$j$: 1, 2, $\vdots$, $k$

1  2  4  8 ... $n$

$$a_n = ar^{n-1}$$
$$n = 1 (2)^{k-1}$$
$$n = 2^k/2 \Rightarrow 2n = 2^k.$$

applying log on both sides

$$\log_2 2n = \log_2 2^k$$
$$\log_2 2 + \log_2 n = k \log_2 2$$
$$k = \log_2 n \Rightarrow O(\log_2 n)$$

$$T.C = O(n/2) \; O(\log_2^2 n) \Rightarrow T.C = O(\tfrac{n}{2} \log_2^2(n))$$

(8) Time complexity of -

```
function (int k) {
  if ( if(n==1) return ; — O(1)
    for (i = 1 to n) {          n times ] O(n^2)
      for (j = 1 to n) {        n times ]
        printf ("*");
  }.}
  function (n-3); T(n-3) times
}
```

The time complexity of both the inner loops is $O(n^2)$

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

as $T(1) = O(1)$

$T.C = O(n^2)$

| i | j | times |
|---|---|---|
| 1 | 1 → n | n times |
| 2 | 1 → n | n times |
| ⋮ | | |
| n | 1 → n | n times |
| | | $n * n$ |
| | | $\boxed{O(n^2)}$ |

~~Thus~~,

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

as $T(1) = O(1)$ $\Rightarrow T.C = O(n^2)$

(9) Time complexity of

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

for $j = \frac{n}{1} + \frac{n}{2} + \frac{n}{3} + \dots 1$

$n = 1 . 2^{k-1} \Rightarrow n = 2^k / 2 \Rightarrow 2n = 2^k$

$\log 2n = \log_2 2^k$

$T.C = O(\log_2 n)$

first loop runs for n times so

$T.C = O(n \log_2 n)$.

(10)

$n^k$ grows polynomially with n

$c^n$ grows exponentially with n

thus $c^n = n^k$

so, $n^k$ is $O(c^n)$

$c = ?$

$n_0 = ?$

$\log n^k = \log c * c^n$

$c \geq e, n_0 = k$