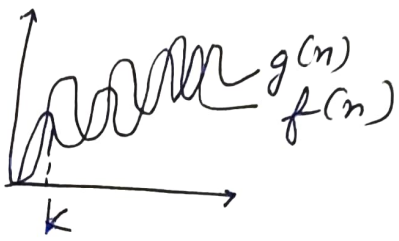
# Tutorial Sheet-1

**Sol 1:** Asymptotic Notation: These notations are used to tell the complexity of an algorithm when the input is very large. It describes the algorithm efficiency and performance in a meaningful way.

The asymptotic notation of an algorithm is classified into 5 types :-

(i) Big-oh notation (O):




$f(n) = O(g(n))$

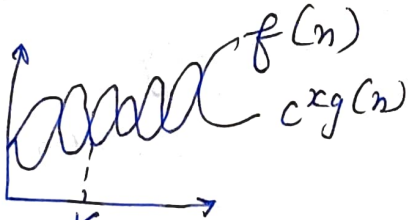
iff

$f(n) \leq c \cdot g(n)$

$\forall\ n \geq n_0$,

some constant $c > 0$

(ii) Big Omega notation ($\Omega$): (A symptotic lower bound).

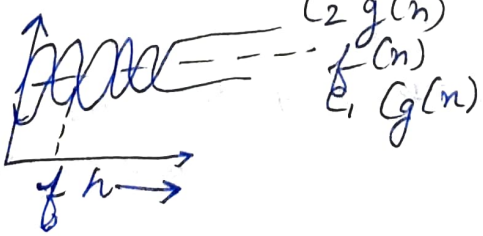



$f(n) = \Omega\ g(n)$ iff

$f(n) \geq c \cdot g(n)$

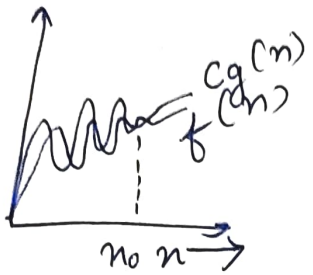
$\forall\ n \geq n_0$ for some const $c > 0$.

(iii) Big theta notation ($\Theta$): (A symptotic tight bound)



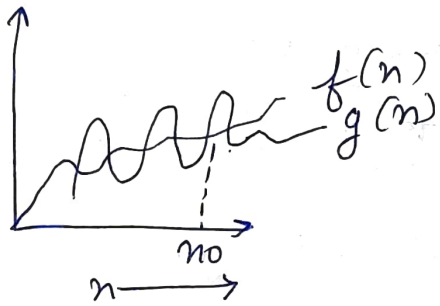
$f(n) = \Theta(g(n))$ iff $c_1 g(n) \leq f(n) \leq c_2 \cdot g(n)$
$\forall\ n \geq \max(n_1, n_2)$.

(iv) Small-oh (o):




$f(n) < cg(n)$
$\forall\ n > n_0$ & $\forall c > 0$
$n = o(n^2)$
$n < 1/n^2$
$2n^2$
$0 \leq n^2$
$n < 0.001/n^2 n_0$.

(v) Small-omega ($\omega$):




lower bound
$f(n) = \omega g(n)$
$f(n) > c \cdot g(n)$
$\forall\ n > n_0$ & $\forall c > 0$
$n^2 = \omega(n)$

Sol2: for (i=1 to n)
{ i = i * 2;
}

Time complexity for a loop means no. of times a loop has run.

→ for the above loop, the loop will run for the following values of i :-

| i | 1 | 2 | 4 | 8 | 16 | 32 | ---- | $2^k$ |
|---|---|---|---|---|---|---|---|---|
| values | $2^0$ | $2^1$ | $2^2$ | $2^3$ | $2^4$ | $2^5$ | --- | n |

i = 1, 2, 4, 8, 16, 32, ---$2^k$ these means k times

i.e. $\log_2 2 = \log_2 n$

$k = \log n$ ($\log_2 2 = 1$

∴ T.C. $= O(\log n)$.

Sol 3: $T(n) = \begin{cases} 3T(n-1), & n>0 \\ 1. & \end{cases}$

By forward substitution,

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

$T(0) = 1$

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

$= 3T(0)$

$= 3$

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

$= 3T(1)$

$= 3 * 3 = 3^2$

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

$= 3T(2)$

$= 3.3^2 = 3^3$

$T(n) = 3^n$.

Sol 4: $T(n) = \begin{cases} 2T(n-1) - 1, & n > 0 \\ 1. \end{cases}$

By forward substitution,

$T(0) = 1.$

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

$(2-1)$

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

$= 2T(1) - 1$

$= 2(2-1) - 1$

$= 2^2 - 2^1 - 1$

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

$= 2T(2) - 1$

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

$= 2^3 - 2^2 2^1 - 1$

$\vdots$

$2^n - 2^{n-1} - 2^{n-2} - 2^{n-3} - \cdots$

$- 2^2 - 2^1 - 2^0$

$\Rightarrow 2^n - (2^n - 1)$

$= 2^n - 2^n + 1 - 1$

$\therefore$ **TC = 1**

Sol 5:

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

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

The value of 'i' increases by one for each value contained in 'S' at the $i^{th}$ iteration is the sum of the first 'i' +ve integers. If k is the total no. of iterations taken by any program then while loop terminates if : $1+2+3+\cdots+k$

$$= [k(k+1)/2] > n.$$

i.e. $k = O(\sqrt{n})$

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

**Sol 6:-**

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

Time complexity : $O(n)$

**Sol 7:**

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

T.C. $= \log n * \log n = O(n \log^2 n)$

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

**Sol 8:-**

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

Time complexity = $O(n^2)$

**Sol 9:-**

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

$T_o C_o = O(n) * O(n) = O(n^2)$

$T_o C_o = O(n^2)$

**Sol 10:-** $n^k$ is $O(c^n)$ ans

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