Asymptotic Notation

are used to tell the complexity of an algorithm when the input is very large.

i) Big O Notation

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

g(n) is 'tight' upper bound of f(n)

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

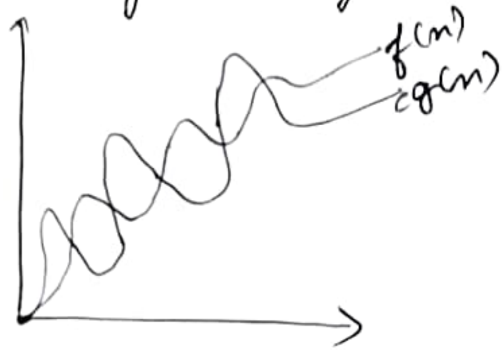
iff $f(n) \le c\,g(n)$

$\forall\, n \geq n_0$ and some constants $c > 0$

ii) Big Omega Notation (Ω)

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

g(n) is 'tight' lower bound of f(n)



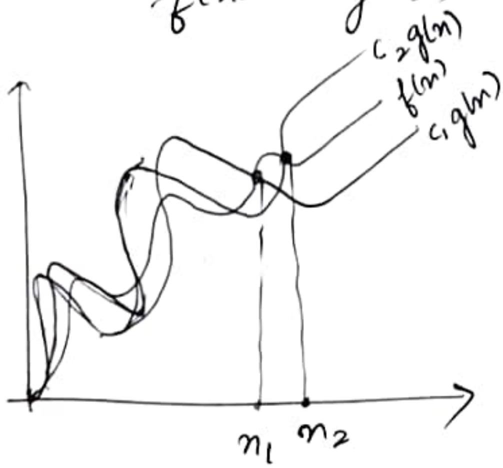
$f(n) = \Omega(g(n))$
iff $f(n) \geq c g(n)$
$\forall$ $n \geq n_0$ & some constant $c > 0$

iii) Theta Notation (θ)

$$f(n) = \theta(g(n))$$

theta given both 'tight' upper & 'tight' lower boun

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$



$f(n) = \theta(g(n))$

iff $c_2 g(n) \geq f(n) \geq c_1 g(n)$
$\forall$ $n \geq \max(n_1, n_2)$ and some constants $c_1, c_2 > 0$

iv) Small o Notation

g(n) is upper bound of f(n) $\quad f(n) = o(g(n))$

$f(n) < c g(n)$

$\forall$ $n \geq n_0$ and ~~some constant~~ for all constant c >

v) Small Omega Notation (ω)

g(n) is lower bound of f(n) $\quad f(n) = \omega(g(n))$

$f(n) > c g(n)$

for all constant $c > 0$

A 2.

```
for (i=1; i<=n; i=i*2)
```

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

$$t_k = a r^{k-1}$$

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

$$n = \frac{2^k}{2}$$

$$2n = 2^k$$

$$\log_2 (2n) = k \log_2 2$$

$$k = \log_2 2n$$

$$= \log_2 2 + \log_2 n$$

$$k = 1 + \log_2 n$$

$$O(\log_2 n)$$

{

for(i=1 to n)
{

for(j=1; j<=n; j=j+1){
print("*")
}

}

}

$i = 1, 2, 3, 4 \ldots$

$j = 1, 3, 6, 10 \ldots$

$i = 1$, n times

$i = 2$, $n/2$ times

$i = 3$, $n/3$ times

$$1 + \frac{n}{2} + \frac{n}{3} + \cdots + 1$$

time complexity = $O(n\log n)$

5.

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

i = 1, 2, 3, 4, 5, ... n

s = 1, 3, 6, 10, 15 ... n

$$s = \frac{i(i+1)}{2}$$

$$\frac{i(i+1)}{2} \le n$$

$$i(i+1) \le 2n$$

$$i^2 + i - 2n \le 0$$

$$\therefore i \le \frac{-1+\sqrt{1+8n}}{2}$$

time complexity $= O(\sqrt{n})$

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

loop continues until i^2 becomes greater than n

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

$i^2 = 1, 4, 9, 16 \ldots n^2$

time complexity $= O\sqrt{n}$

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

$T(0)=1$

$n=1 \Rightarrow T(1) = 3T(0) = 3$
$n=2 \Rightarrow T(2) = 3T(1) = 3^2$
$n=3 \Rightarrow T(3) = 3T(2) = 3\cdot 3^2 = 3^3$
$n=k \Rightarrow T(k) = 3^k$
$T(n) = 3^n$

time complexity $= O(3^k)$

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

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

$T(n) = 2^k T(n-k) - (2^0 + 2^1 + \dots + 2^{k-1})$

When, $n-k=0$
$k=n$

Base case $\Rightarrow T(0)=1$
$\therefore n-k=0 \Rightarrow n=k$

Substitute $k=n$

$T(n) = 2^n T(0) - (2^0 + 2^1 + \dots + 2^{n-1})$

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

time complexity $= O(1)$

PAGE No
DATE

7.

```
void function(int n)
{
  int i, j, k, c=0;
  for (i=n/2; i<=n; i++)                 — n/2
   for (j=1; j<=n; j=j*2)                — log2(n)
  for (k=1; k<=n; k=k*2)                 — log2(n)
   c++;
}
```

$n/2 * \log 2(n) * \log 2(n)$

time complexity = $O(n * \log^2(n))$.

8.

```
fn(int n)                                 — T(n)
{
  if (n==1) return;
  for (i=1 to n)                          — n
  {
    for (j=1 to n)                        — n2
    {
      print ("*");                        — n^2
    }
  }
  fn(n-3);                                — T(n-3)
}
```

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