Tutorial Sheet - 1 . Date: / /

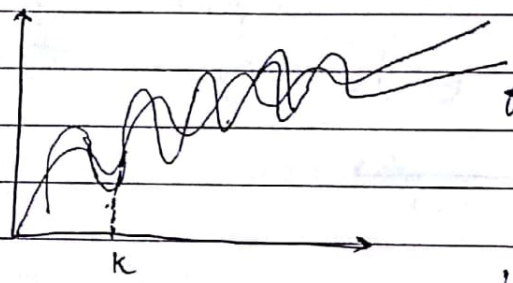
1- Asymptotic Notation (Types with example)

→ These notations are used to tell the complexity of an algo. when the input is very large.

→ It describes the algorithm efficiency & performance in a meaningful way. It describes the behaviour of time or space complexity for large instance characteristics

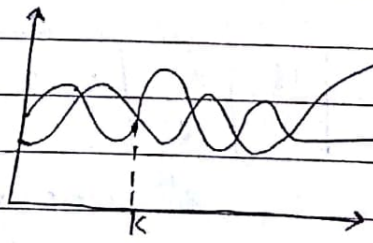
(5 types)

(i) Big Oh notation (O) : (Asymptotic upper Bound) The function $f(n) = O(g(n))$, if & only if there exist a +ve constant c & k such that $f(n) \leq c * g(n)$ for all n, $n \geq k$.



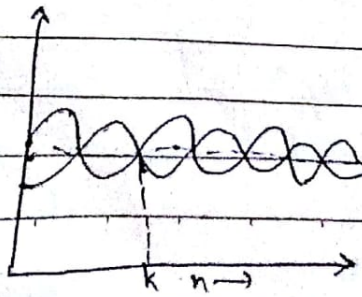
$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$ $\forall$ $n \geq n_0$, some constant $c > 0$

(ii) Big Omega notation (Ω) : (Asymptotic lower bound) The function $f(n) = \Omega(g(n))$, iff there exists a +ve constant c & k such that $f(n) \geq c * g(n)$ for all n, $n \geq k$



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

(iii) Big Theta notation (Θ) : (Asymptotic tight- bound) The function $f(n) = \Theta(g(n))$, iff there exists a +ve constant $c1, c2$ & k such that $c1 * g(n) \leq f(n) \leq c2 * g(n)$ for all n, $n \geq k$.

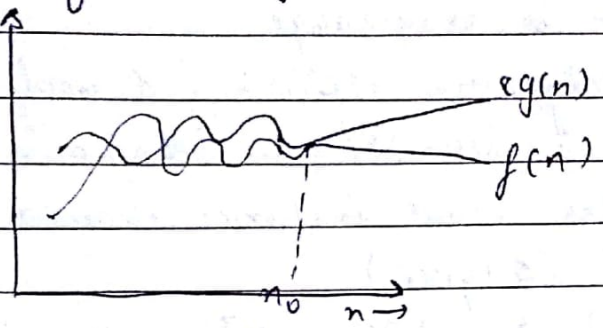


$f(n) = \Theta(g(n))$ iff $c_1 g(n) \leq f(n) \leq c_2 \cdot g(n)$ $\forall$ $n \geq \max(n_1, n_2)$

(iv) Small - oh (o) : o gives us upper bound.

$f(n) = o(g(n))$ $\quad f(n) \leq c\, g(n)$

$\forall\ n \geq n_0$ & $\forall\ c > 0$



$n = o(n^2)$

$n < 1n^2$

$2n^2$

$0.5n^2$

$n < 0.001 n^2 n_0$

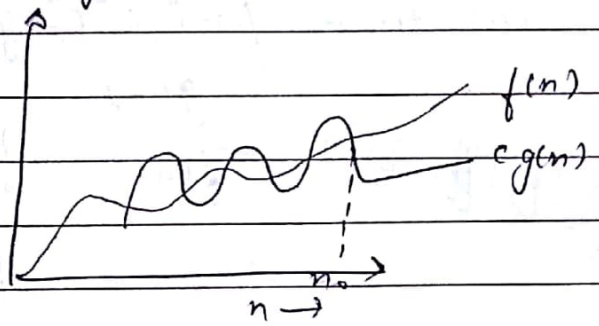
(v) Small - omega (ω) : lower bound

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

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

$\forall\ n > n_0$ & $\forall\ c > 0$

$n^2 = \omega(n)$



Q- time complexity ?

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

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

i	1	2	4	8	16	32	....	2^k
value	2^1	2^2	2^3	2^4	2^5	2^6	....	n

$i = 1, 2, 4, 8, 16, 32, \ldots, 2^k$ this means k times

i.e., $2^k = n$

$k \log_2 2 = \log_2 n$

$k = \log n$ $\quad [\log_2 2 = 1]$

$\boxed{T.C = O(\log n)}$

Date: / /

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

By forward substitution,

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

$T(0) = 3\ T(-1) = 0$

$T(1) = 3\ T(1-1) = 3T(0) = 3$

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

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

$\vdots$

$T(n) = 3^n$ ∴ $T.C = O(3^n)$

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

By forward substitution,

$T(0) = 1$

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

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

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

$\vdots$

$2^n - 2^{n-1} - 2^{n-2} - 2^{n-3} - \cdots - 2^2 - 2^1 - 2^0$

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

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

∴ $T.C = O(1)$

5-

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

$T(n)$

Date: / /

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

The value of 'i' increases by one for each iteration. The 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 terminate if:

$1+2+3+\ldots\ldots+K \quad \Rightarrow [K(K+1)/2] > n$

so, $K = O(\sqrt{n})$

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

6- void function(int n) T.C.?

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

$O(n) = T.C.$

7- void function(int n)

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

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

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

8-

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

function(n-3);
}

∴ T.C. = $O(n^2)$

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

∴ T.C = $O(n^2)$

10- For the function, n^k & c^n, what is the asymptotic relationship b/w these functions?
Assume that $k >= 1$ & $c > 1$ are constants. Find out the value of c & n_0 for which relation holds.

n^k is $O(c^n)$.