Tutorial 1

Shruti Agarwal
G-45

1) Asymptotic Notations:-

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

There are mainly three of asymptotic notations:
- Big-O notation
- Omega notation
- Theta notation

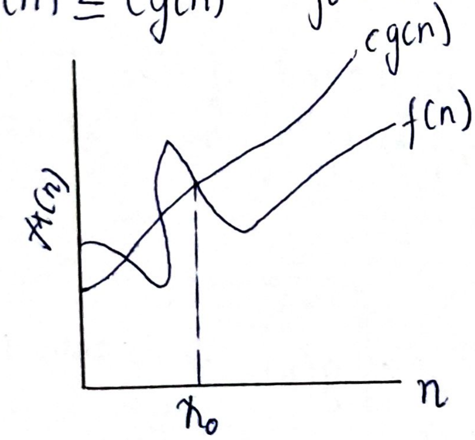
(i) Big O Notation:-
- gives upper bound of the running time of an algorithm
- gives worst-case complexity of an algorithm

Given two functions $f(n)$ & $g(n)$

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

iff

$f(n) \leq cg(n)$ for all $n \geq n_0$ $c>0$



Example: In bubble sort, When the input array is in reverse condition, the algorithm takes the maximum time (n^2) to sort the elements i.e the worst case.

(ii) Omega Notation (Ω):-

- represents the lower bound of the running time of an algorithm
- It provides the best case complexity of an algorithm

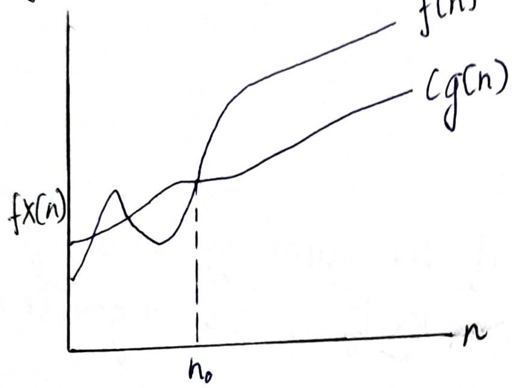
Given two functions

$f(n)$ and $g(n)$

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

iff

$f(n) \geq c\, g(n)$ for all $n \geq n_0$, $c > 0$



Example In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear ie best case.

(iii) Theta Notation (Θ):-

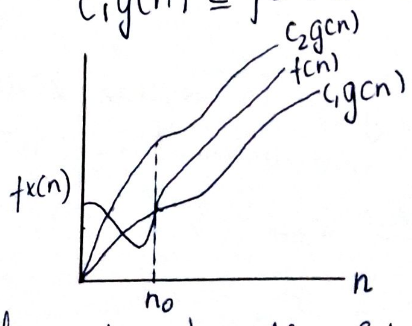
- represents the upper & lower bound of the running time of an al[gorithm]
- provides the average-case complexity of an algorithm

Given two fxⁿ

$f(n)$ & $g(n)$

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

iff $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$, $c_1, c_2 > 0$



Example:- In bubble sort, when the input array is neither sorted nor in reverse order, then it takes average-time.

for(i=1 to n)

{ $i = i * 2$; }

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

$n = ar^{k-1}$

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

$n = 2^{k-1}$

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

$2n = 2^k$

$\log(2n) = \log(2^k)$

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

$k = \log(2n)$

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

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

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

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

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

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

$\vdots$

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

$= 3^n T(0)$

$= 3^n$ [$\because T(0) = 1$]

$\Rightarrow$ complexity $= O(3^n)$

14 $T(n) = \{2T(n-1) - 1$ if $n>0$, otherwise $1\}$

$T(n) = 2T(n-1) - 1$ —(1)

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

$= 2T(0) - 1$

$T(1) = 2 - 1$

$T(1) = 1$ —(2)

putting $n = n-1$

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

$T(n) + 1 = 2(2T(n-2)\} - 1)$ (from (1))

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

putting $n = n-1$

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

$T(n) + 1 = 2^3 T(n-3) - 2^2 - 2$ (from (1))

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

$\vdots$

$T(n) = 2^k T(n-k) - 2^{k-1} - 2^{k-2} - 2^{k-3} \cdots - 2^2 - 2^1 - 2^0$

$T(1) = 1$ from (2)

$n - k = 1$

$T(n) = 2^{n-1} T(1) - [2^0 + 2^1 + 2^2 + \cdots + 2^{n-3} + 2^{n-2}]$

$= 2^{n-1} \times 1 - [2^{n-1} - 1]$

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

$T(n) = 1$

$\Rightarrow$ Complexity $= O(1)$

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

$$\begin{array}{l} S \\ 3 \\ 6 \\ 10 \\ 15 \\ 21 \\ \vdots \\ n \end{array} \quad \left.\begin{array}{l} \\ \\ \\ \\ \\ \\ \end{array}\right\} k \text{ times}$$

$$S_k = 3+6+10+15+21+ \cdots T_k$$

$$S_{k-1} = 3+6+10+15+ \cdots T_{k-1}$$

$$S_k - S_{k-1} = 3+3+4+5+6+ \cdots + T_k - T_{k-1}$$

$$T_k = 3 + [3+4+5+6+ \cdots + (k-1)\text{times}]$$

$$= 3 + \frac{(k-1)}{2}\left(6 + (k-2)\cdot 1\right)$$

$$= 3 + \frac{(k-2)}{2}(k+4)$$

(T_k) k^{th} term

for last iteration k^{th} term is n

$$T_k = n$$

$$\frac{(k-2)(k+4)}{2} + 3 = n$$

For time complexity removing lower order terms

$$k = \sqrt{n}$$

$\Rightarrow$ Complexity $= O(\sqrt{n})$

6)

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

i	times
1	$\sqrt{1}$
2	$\sqrt{2}$
3	$\sqrt{3}$
4	$\sqrt{4}$
⋮	
n	$\sqrt{n}$

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

7)

```
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	j	k
n	1	1
n-1	2	2
⋮	4	⋮
n/2	⋮	⋮
$\frac{n}{2}$ times	logn times	logn times

$= O\left(\frac{n}{2}(\log n)(\log n)\right)$

$\Rightarrow$ Complexity $= O(n \log n)^2)$

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

n	i	j
n	1	1 ... n
	⋮	1 ... n
	n	⋮ n
n-3	1	1 ... n
	⋮	⋮ ... n
n-6	⋮	⋮ n
⋮	⋮	⋮

$= n + n-3 + n-6 \cdots + 7 + 4 + 1$

or $\quad 1 + 4 + 7 + \cdots + n-3 + n$

$n = 1 + 3(k-1) \quad (\because n^{th} = a + (n-1)d)$

$k = \dfrac{n+2}{3}$

$\Rightarrow$ Time complexity of recursion $= O(n)$

i	j
1	1
⋮	⋮
n	n $\Rightarrow n^2$

$\therefore$ Total time complexity $= O(n\, n^2)$

$= O(n^3)$

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

i	j
1	1 ... n
2	1, 3, ... n
3	...
...	
n	n = n^2

$\Rightarrow$ Time complexity = $O(n^2)$