29/08

Page No.:
Date:

Asymptotic Notations :-

→ Algorithm Run Time Analysis :

↳ How much time will the given algorithm will take to run.

- ↦ Worst case
- ↦ Average case
- ↦ Best case

→ Notations :-

→ Types of Asymptotic Notations) :-

i) Omega (Ω) :-

Tighter Lower bound

Algo not be "less than" given time.

ii) Big-o (O) :-

Tighter upper bound

Algo not be "more than" given time.

iii) Theta (Θ) :-

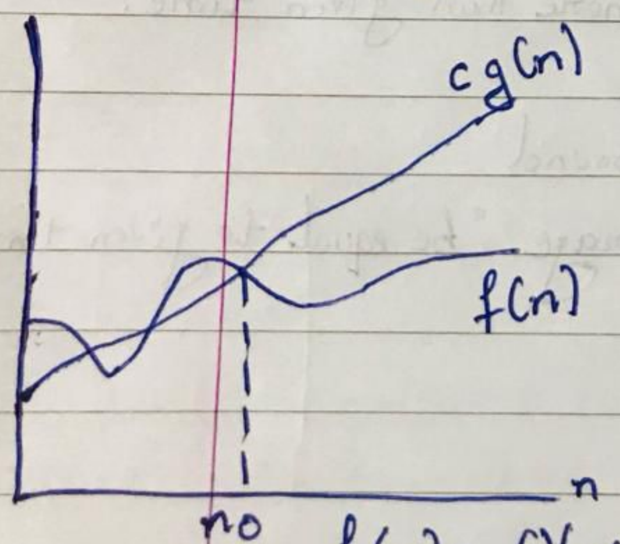
Upper & Lower bound

Algo " On an average " be equal to given tim

Common Asymptotic Notations :-

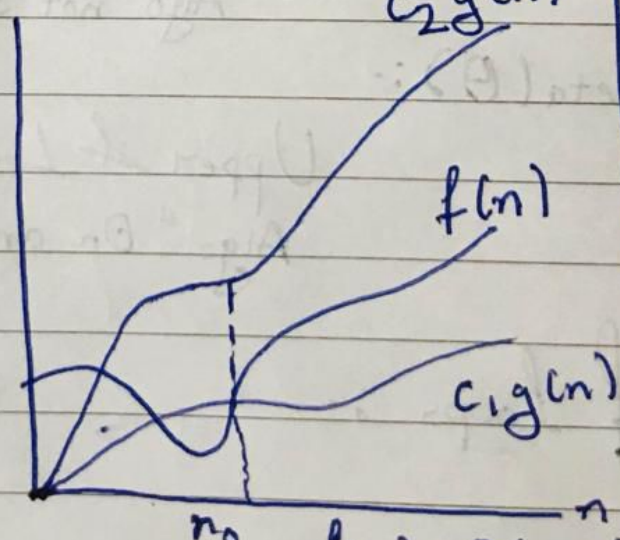
- Constant → $O(1)$
- Logarithmic → $O(\log n)$
- Linear → $O(n)$
- Quadratic → $O(n^2)$
- Cubic → $O(n^3)$
- Polynomial → $n^{O(1)}$
- Exponential → $2^{O(1)}$

Big-o (O)



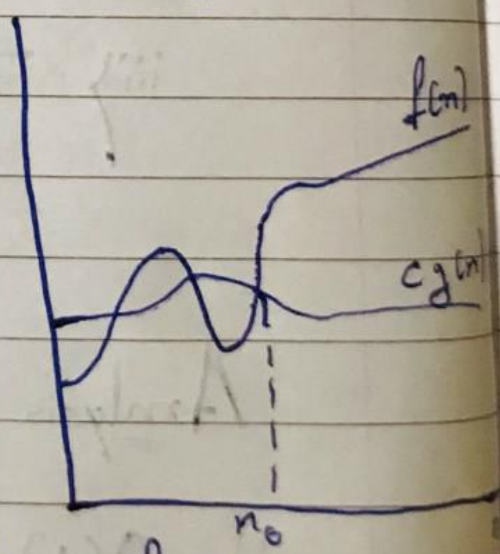
$f(n) = O(g(n))$

Theta (Θ)



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

Omega (Ω)



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

Master Theorem :-

- Direct way to get sol^n
- Only for recurrence relations

$T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$

n = size of input

a = no. of subproblem

n/b = size of subproblem

$f(n)$ = cost of work done outside recursive call.

i) if $f(n) = O(n^{\log_b a - c})$, then $T(n) = \Theta(n^{\log_b a})$

ii) if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} * \log n)$

iii) if $f(n) = \Omega(n^{\log_b a + c})$, then $T(n) = \Theta(f(n))$

$c > 0$ is a constant

Problems :- (Time Complexity)

I. Iterative Programs :-

1. for (i to n) → $O(n)$
   // Statement (O(1))

2. for (i to n)
   for (j to n) → $O(n^2)$
   // Statements (O(1))

3. i=1, s=1
   while (s <= n)
   {
   i++
   s = s + i
   }
   (GATE-1991)

S	1	3	6	10	....	n
i	1	2	3	4	....	K

Sum of Natural no. $\frac{K(K+1)}{2}$

$\frac{K(K+1)}{2} > n$

$\frac{K^2+K}{2} > n$

$K = O(\sqrt{2})$

Time complexity = $O(\sqrt{2})$

$k = O(\sqrt{2})$

4. $i=1,\ j=1,\ k=1 \ldots, n$

```
for (i to n)
    for (j to i)
        for (k to 100)
            // Statement
```

$i=1$	$i=2$	$\ldots$	$i=n$
$j=1$	$j=2$	$\ldots$	$j=n$
$k=100$	$k=200$		$k=n*100$

$= 100+200+300 \ldots + n*100$

$= 100(1+2+3+\ldots n)$

$= 100\left(\frac{n(n+1)}{2}\right) = O(n^2)$

Time complenity $= O(n^2)$

5.

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

$i = 1, 2, 4, 8 \ldots n$

$2^0\ 2^1, 2^3, 2^4 \ldots 2^k$

$2^k = n$

$k = \log_2 n$

Time complenity $= O(\log_2 n)$

6. for (i = n/2 to n) — $n/2$
for (j = 1 to n) j = 2 * j — $\log_2 n$ (for 5 no. problem)
for (k = 1 to n) k = k * 2 — $\log_2 n$

Time complexity = $O(n (\log_2 n)^2)$

7.
```
n = 2^2^k
for ( i=1 to n)
{   j=2
    while( j<=n)
    {
        j = j^2
    }
}
```

Let

k = 1	k = 2	k = 3
$n = 2^2 = 4$	$n = 16$	$n = 2^{2^3} = 2^8$
$j = 2, 4$	$j = 2, 4, 16$	$j = 2, 2^2, 2^4, 2^8$
n * 2 times	n * 3 times	n * 4 times

└ for loop

$n * (k+1)$

$n = 2^{2^k}$

$\log_2 n = 2^k (\log 2) \to 1$

$\log \log n = k (\log 2) \to 1$

$k = \log \log n$

$T(n) = n * (k+1)$

$= n * (\log \log n + 1)$

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