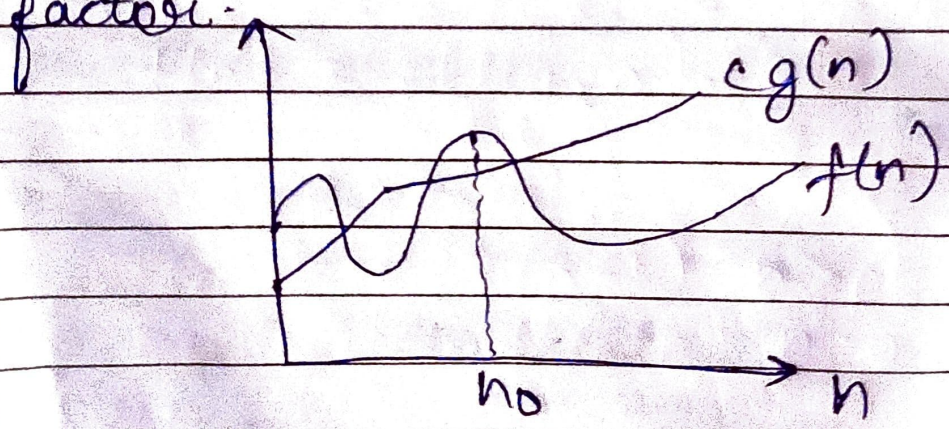
Name - Diya Rautela
University Roll no - 1961047
Sec - 'A'
Subject - DAA
Course - B.Tech (CSE)

(1) Asymptotic notation are used to represent the complexities of algori-thms for asymptotic analysis.

These notation are used for very large input.

1- <u>Big-oh (O)</u>-

It gives upper bound for a function f(n) to within a constant factor.

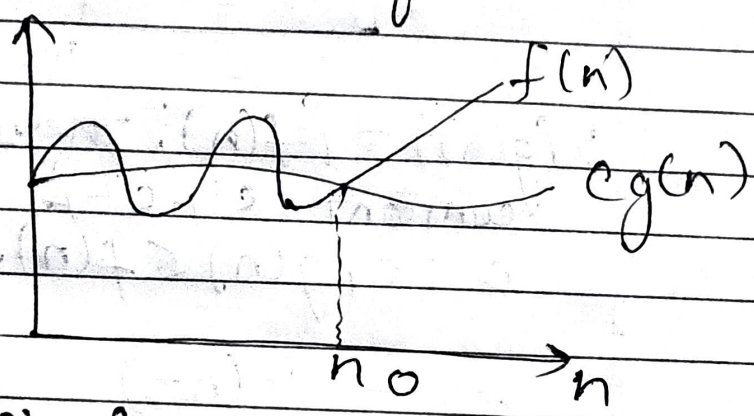


$f(n) \leq c \cdot g(n) \; \forall \; n \geq n_0, c > 0$

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

2- Big-omega Notation (Ω)

Big-omega(Ω) notation gives a lower bound for a $f(n)$ to within a constant factor.

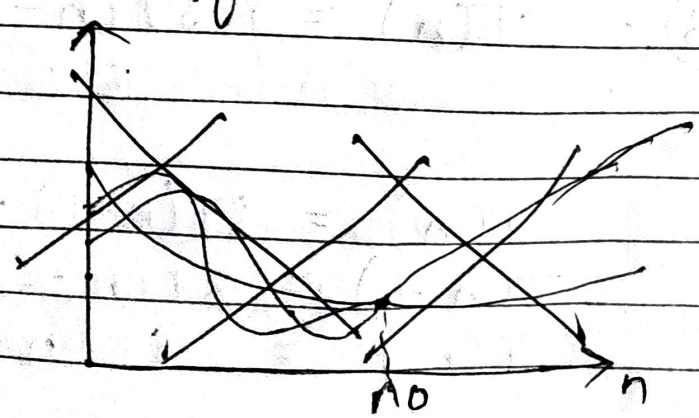


$\Omega(g(n)) = \{f(n) :$ There exist (+)ve constant c & n_0 such that $0 \leq cg(n) \leq f(n) \; \forall \; n \geq n_0\}$

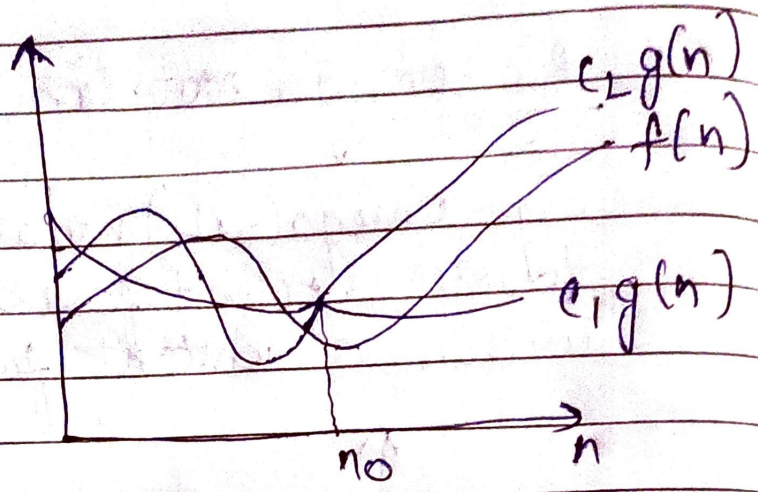
eg- $\Omega(n \log n)$

3- Big-Theta Notation (θ)

it gives bound of function within a constant factor.



J.K Papers
Date:
Page:



$\Theta(g(n)) = \{ f(n) :$ There exist (+)ve constant c_1, c_2 & n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n), \forall n > n_0$

eg - $\Theta(n^2)$

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

1, 2, 4, 8, ——— n

$T(n) = O(\log_2 n)$

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

$T(n) = 3T(n-1)$ — (1)
$T(n-1) = 3T(n-2)$
$T(n) = 9T(n-2)$ — (2)

$T(n) = 3^3 T(n-3)$ — (3)

$T(n) = 3^k T(n-k)$ — (4)

for $T(n-k) = T(0)$

$n-k = 0$

$n = k$

$T(n) = 3^n T(0)$

$T(n) = 3^n$

$T(n) = \Theta(3^n)$

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

$T(n) = 2T(n-1) - 1$ — (1)

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

$T(n) = 4T(n-2) - 1 - 2$ — (2)

$T(n) = 8T(n-3) - (1+2+4)$ — (3)

$T(n) = 2^k T(n-k) - (\underbrace{1+2+4+ \cdots 2^{k-1}}_{k\text{- terms}})$

$T(n-k) = T(0)$

$n = k$

$T(n) = 2^n T(0) - (\underbrace{1+2+4+ \cdots}_{k \text{ terms}})$

Its a GP

$a = 1$

$r = 2$

$$T(n) = 2^n - \left(\frac{1(2^n - 1)}{2-1}\right)$$

$$T(n) = 2^n - 2^n + 1$$
$$T(n) = 1$$
$$T(n) = \theta(1)$$

(5) -

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

1, 3, 6, 10, 15, ——— n

← K-terms →

Its kth term is $\frac{k(k+1)}{2} = n$

$$k = \sqrt{n}$$
$$T(n) = O(\sqrt{n})$$

(6) -

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

}

$T(n) = O(\sqrt{n})$

(7) - $T(n) = O(n * \log_2 n * \log_2 n)$

$T(n) = O(n * (\log_2 n)^2)$

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

(8) - function (int n) $T(n)$

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

$T(n-3)$

$T(n) = T(n-3) + n^2$ — (1)

$T(n-1) = T(n-4) + (n-1)^2$

$T(n) = T(n-4) + n^2 + (n-1)^2$

$T(n) = T(n-5) + n^2 + (n-1)^2 + (n-2)^2$

$T(n) = T(n-K) + (n^2 + (n-1)^2 + (n-2)^2 + \cdots)$

$(K-2)$ terms

for $T(n-K)=1$

$K=n-1$

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

$(n-3)$ terms

$$T(n) = T(1) + (4^2 + 5^2 + \text{---} n^2)$$

$$T(n) = T(1) + \left(\frac{(n-3)(n-2)(2n-5)}{6}\right)$$

$$T(n) = 1 + \left(\frac{2n^3 + \text{---}}{6}\right)$$

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

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

(9) -

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

$i=1$ n times

$i=2$ 1, 3, 5 --- n $\quad n$

$i=3$ 1, 4, 7, --- n $\quad n/2$

$\vdots$ $\quad n/3$

$i=n$ 0

$$T(n) = \left(n + \frac{n}{2} + \frac{n}{3} + \cdots\right)_{n\text{-times}}$$

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

(10) - for the function n^k & a^n, what is the relation

$k \geq 1$ & $a > 1$

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

(11) -

```
void fun(int n)
{
    int j=1, i=0;
    while (i<n)
    {
        i = i+j ;
        j++;
    }
}
```

$$\underbrace{0, 3, 6, 10, 15, \ldots \text{———} n}_{K \text{ terms}}$$

So for this series is

K^{th} term is $\frac{K(K+1)}{2}$

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

$$K \simeq \sqrt{n}$$

$$T = \theta(\sqrt{n})$$

(12) - Recurrence relation of fabonacci series is

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

$$T(n) = 2T(n-2) + 1$$
$$T(n) = 4T(n-4) + 3$$
$$T(n) = 8T(n-6) + 7$$
$$T(n) = 16T(n-8) + 15$$

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

for $T(n-2K) = T(0)$

$$n = 2K$$

$$K = \frac{n}{2}$$

$$T(n) = 2^{n/2} T(0) + (2^{n/2} - 1)$$
$$T(n) = 2^n - 1$$
$$T(n) = O(2^n)$$

hence space complexity of fabinacci series is $O(n)$ as it depends on height of recursive tree & it is equal to n in fabinacci series.

(13) - n (log n)

```
void fun { for (int i=0; j<n; i++)
    {
      for (int i=0; i<n; i = i*2)
      {
        print ("*");
      }
    }
}

void main()
{
  func();
}
```

n^3

```
# include <stdio.h>
void main()
{
  int n;
  cin>>n;
  for (int i=0; i<n; i++)
  {
    for (int j=0; j<n; j++)
    {
      for (int k=0; k<n; k++)
      {
        n++;
```

```
}
}
}
}
```

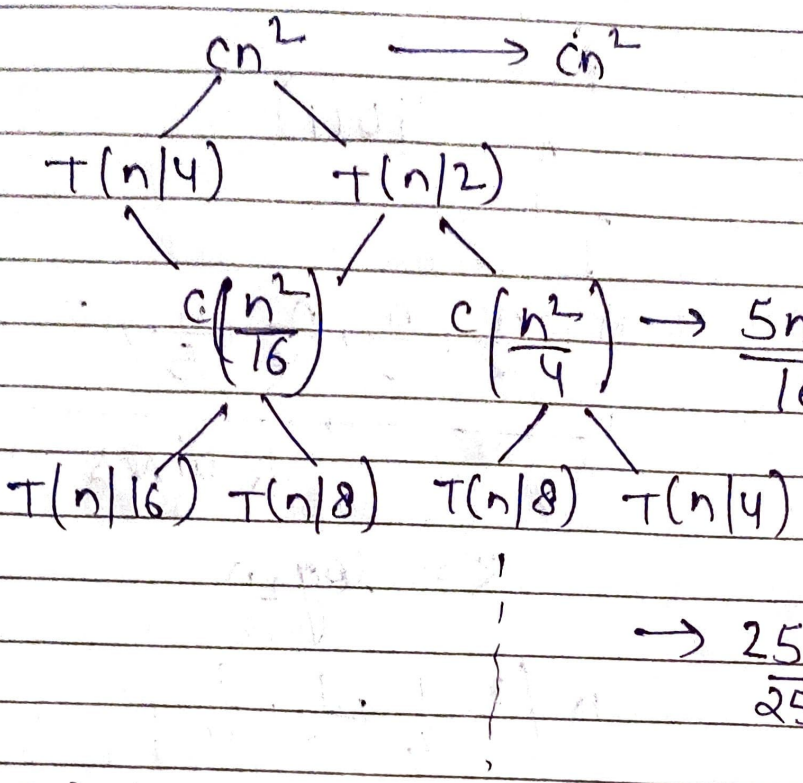
→ <u>log(log n)</u>

```
# include <bits/stdc++.h>
void fun (int n)
{
    if (n==2)
        return 1;
    else
    fun (sqrt (n));
}
void main( )
{
    fun (100);
}
```

(14) - $T(n) = T(n/4) + T(n/2) + cn^2$

$T(1) = 0$

$T(0) = 0$



$T(n)$ = cost of each level

$$T(n) = cn^2 + \frac{5cn^2}{16} + \frac{25cn^2}{256} + \cdots$$

it is a G.P

with $a = n^2$

$r = \frac{5}{16}$

so sum of s/p

$$T(n) = cn^2 \left(1 - \frac{5}{16}\right) = \frac{16cn^2}{11} = \frac{16cn^2}{11}$$

$$T(n) = O(n^2)$$

(15) - for (int i to n)

{

for (int j = 1; j < n; j += i)

{

} // O(1)

}

$$\underbrace{n, \frac{n}{2}, \frac{n}{3}, \frac{n}{4}, \frac{n}{5}, \ldots 1}_{k \text{ times}}$$

$k = \log_2 n$

$$n\left(1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots \frac{1}{n}\right)$$

$(n(\log n))$

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

(16) - for {int i = 2; i <= n; i = pow(i, k)}

{

} // O(1)

$$2, 2^k, 2^{(k)^2}, 2^{k^3}, \ldots n$$

It GP $a = 2$

$r = 2^k$

k^{th} term $= ar^{k-1}$

$$n = 2(2^K)^{K-1}$$

let $K^{(K-1)} = x$

$$K \log_K K = \log x$$

$$K = \log x \quad \text{--- (1)}$$

$$n = 2^x$$

$$\log_2 n = x \log_2 2$$

$$x = \log_2 n$$

$$\log x = \log(\log n)$$

from (1)

$$K = \log(\log(n))$$

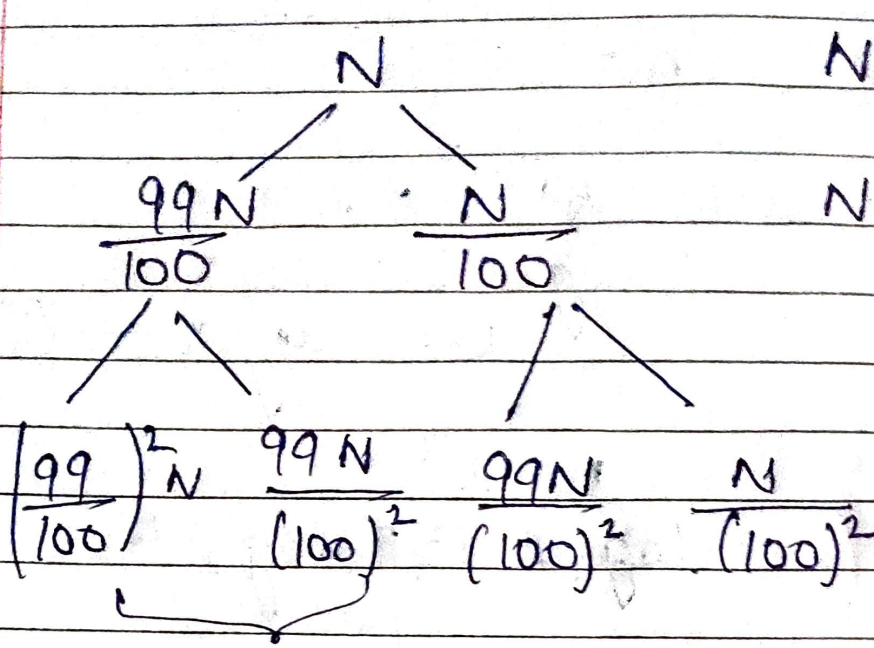
$$T(n) = O(\log(\log(n)))$$

(17) - hence pivot is divided in 99% & 1%

so

$$T(n) = T\left(\frac{99}{100} N\right) + T\left(\frac{N}{100}\right) + N$$

Now as here we can use 2 extreme ms of a tree where starting point is N



$$N\left(\frac{(99)(99)}{100\times100}+\frac{99(1)}{100\times100}\right)+\frac{100}{100\times100}N$$

$$=\frac{99}{100}N+\frac{N}{100}$$

$$=N.$$

$$=N$$

so cost of each level is N only

Total cost = height * cost of each level

so for 1st stream - $N, \frac{99N}{100}, \left(\frac{99}{100}\right)^2 N$

$$\left(\frac{99}{100}\right)^{n-1} N = 1$$

$$\left(\frac{99}{100}\right)^{n-1} = \frac{1}{N}$$

$$N = \left(\frac{100}{99}\right)^{n-1}$$

$$\log N = h \log(1)$$

$$h = \log N$$

or

$$h = \frac{\log N}{\log(100/99)} + 1$$

<u>height of 2nd stream</u>

$$N, \frac{N}{100} + \frac{N}{(100)^2} + \frac{N}{(100)^3} + \cdots 1$$

$$N\left(\frac{1}{100}\right)^{n-1} = 1$$

$$N = (100)^{n-1}$$

$$(n-1)\log 100 = \log N$$

$$n = \frac{\log N}{\log 100} + 1 \quad \& \quad h = \log N \text{ (approx)}$$

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

So time complexity is $O(N \log N)$

height of both extre is $\frac{\log N}{\log 100}$ +

1 of $\left(\frac{1}{100}\right)$

& $\frac{\log N}{\log\left(\frac{100}{99}\right)}$ + 1 of $\left(\frac{99}{100}\right)$

So we can conclude that if division is done more than height of tree will be &

when division ratio is less than height is less.

(18) - a) n, $n!$, $\log n$, $\log \log n$, root(n), $n \log n$

2^n, 2^{2n}, 4^n, n^2, 100

$\Rightarrow O(100) < O(\log \log n) < O(\log n) <$

$O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) <$

$O(2^n) < O(2^{2n}) < O(4^n)$

b) $2(2^n)$, $4n$, $2n$, 1, $\log(n)$, $\log(\log(n))$, $\sqrt{\log(n)}$, $\log 2n$, $2\log n$, n, $\log(n!)$, $n!$, n^2, $n\log(n)$

$O(1) < O(\log(\log(n))) < O(\log(n)) < O(\log 2n) < O(2\log n) < O(n) < O(n\log(n)) < O(\log(n!)) < O(2n) < O(4n) < O(n^2) < O(n!) < O(2(2^n))$

c) $8^{\wedge 2n}$, $\log_2 n$, $n\log_6(n)$, $n\log_2(n)$, $\log(n!)$, $\log_8(n)$, 96, $8n^2$, $7n^3$, 50

$\Rightarrow O(96) < O(\log_8(n)) < O(\log_2 n) < O(\log(n)) < O(n\log_6(n)) < O(n\log_2(n)) < O(5n) < O(8n^3) < O(7n^3) < O(n!) < O(8^{\wedge 2n})$

(19) - void linear search (int arr[], int n, int key)

```
{
  for (i=0 to i=n)
      if arr[i] == key
          cout << "found";
      else
          continue
}
```

J.K Papers
Date:
Page:

(20) - Iterative Insertion Sort

```
void Insertion sort (arr, n){

    int i, temp, j;
    for(i=1 to n)
    {
        temp = arr[i]
        j = i-1
        while j>=0 && arr[j]>temp
        {
            arr[j+1] = arr[j]
            j--
        }
        arr[j+1] = temp
    }
}
```

Recursive Insertion sort

```
insertion sort (arr, n)
{
    if n<=1
        return;

    insertion sort (arr, n-1);
    last = arr[n-1];
    j = n-2
```

```
while (j>=0 and arr[j] > last)
{
    arr[j+1] = arr[j]
    j--
}
arr[j+1] = last;
}
```

Insertion sort is called online, sorting because it don't know the whole input, it might make decision that later turn out to be not optimal.

Other algorithm are off-line algorithms that are discussed in lectures.

(22)

	inplace	Stable	online sorting
Bubble sort	Yes	Yes	NO
Selection sort	Yes	NO	NO
Insertion Sort	Yes	Yes	Yes
Merge Sort	NO	Yes	NO
Quick Sort	Yes	NO	NO
Heap sort	Yes	NO	NO

(21) - Time complexity

	Best	Avg	worst	Space
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	(n)	$O(n^2)$	$O(n^2)$	$O(1)$
merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$ {due to recursion
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$

(23) -

```
Binary search (arr, int n, Key)
{
    beg = 0
    end = n-1
    while (beg <= end)
    {
        mid = (beg + end)/2
        if [arr[mid] == Key]
```

found
else if arr[mid] < key
beg = mid + 1

else
end = mid - 1
}
}

Time complexity of linear search - O(n)
Space complexity of linear search - O(1)
Time complexity of Binary search - O(log n)
Space complexity of Binary search = O(1)

(24) $T(n) = \frac{T(n)}{2} + 1$