Assignment

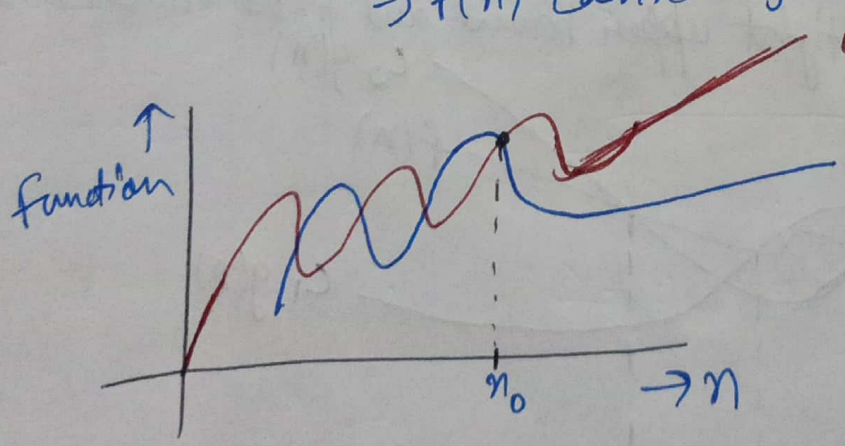
Name : Shashank Roll NO. : 2015277
Section : ML Subject : DAA

Q1 Asymtotic Notions are the mathematical notions used to describe running time of an algorithm when the input tends towards a particular limiting value. These notions are generally used to determine the running time of an algorithm and how it grows with amount of input. Types of asymtotic notions:

i) Big Oh (O) Notation:

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

$\Rightarrow$ g(n) is tight upper bound of f(n)
$\Rightarrow$ f(n) cannot go above g(n)



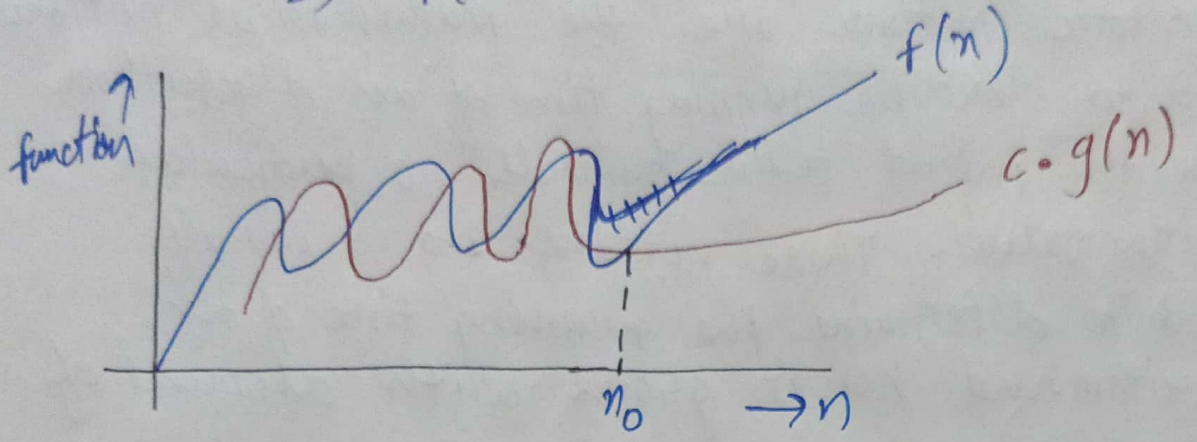
$$f(n) = O(g(n))$$
iff
$$f(n) \leq c \cdot g(n) \quad \forall \; n \geq n_0$$

ii) Big Omega (Ω):

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

$\Rightarrow$ $g(n)$ is "tight lower" bound of $f(n)$

$\Rightarrow$ $f(n)$ will not go below $g(n)$



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

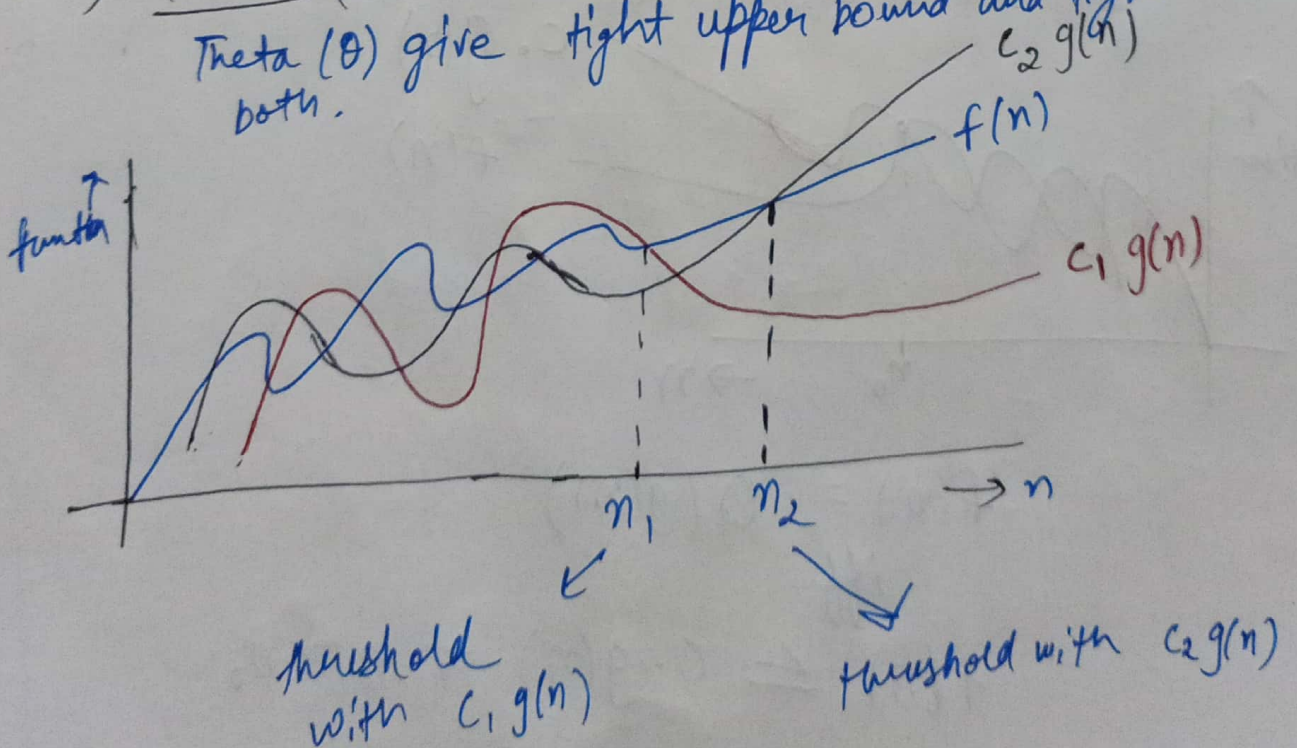
iff

$f(n) \geq c \cdot g(n) \quad \forall \quad n \geq n_0$

iii) Theta (Θ)

Theta (Θ) give tight upper bound and tight lower bound.

both.



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

iff

$c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall \; n \geq \max(n_1, n_2)$

$\{c_1, c_2 > 0\}$

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

$\Rightarrow f(n) = O(g(n)) \;\&\; f(n) = \Omega(g(n))$

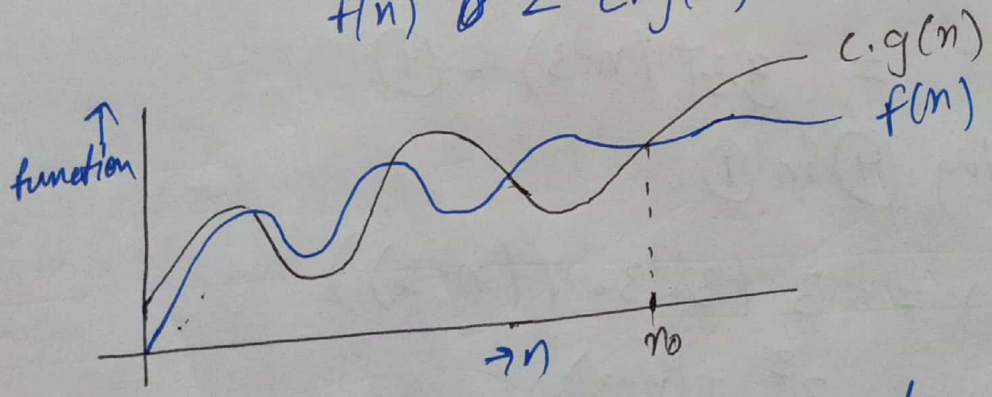
iv) small-oh (o)

gives upper bound

$f(n) = o(g(n))$

iff

$f(n) < c \cdot g(n) \quad \forall \; n > n_0 \;\&\; c > 0$

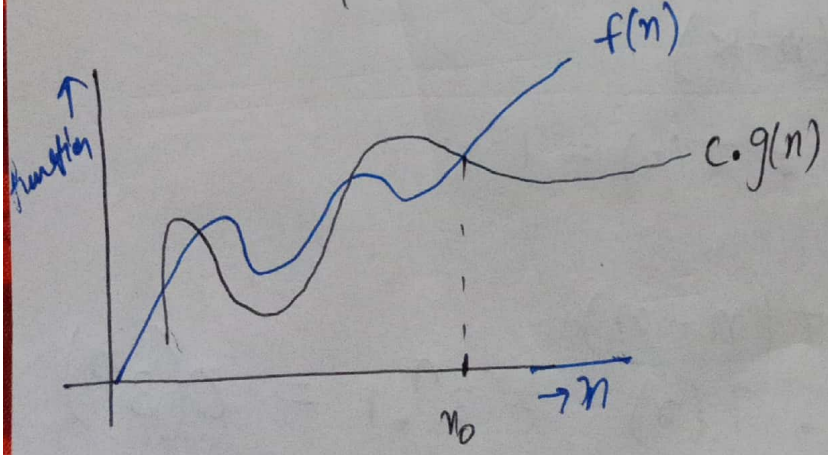


v) small omega (ω) → It gives lower bound

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

iff

$f(n) > c \cdot g(n) \quad \forall n > n_0 \;\&\; c > 0$



Q2. $O(\log n)$

Q3. $$T(n) = \begin{cases} 3T(n-1) & n>0 \\ 1 & n \le 0 \end{cases}$$

Sol. using back substitution

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

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

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

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

putting (3) in (2)

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

$$= 3.3.T(n-3) \quad -(4)$$

putting (4) in (1)

$$T(n) = 3(3.3.T(n-3))$$

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

so in middle any where

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

$\because$ base case $T(0) = 1$

at $k = n$

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

$$= 3^n . T(0) = 3^n . 1 = O(3^n)$$

Q4) $T(n) = \begin{cases} 2T(n-1) - 1 & n > 0 \\ 1 & n \leq 0 \end{cases}$

Using back substitution

$T(n) = 2T(n-1) - 1$ — (1)

$T(n-1) = 2T(n-2) - 1$ — (2)

putting (2) in (1)

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

$T(n) = 2^2 T(n-2) - 2 - 1$ — (3)

from (1)

Now $T(n-2) = 2T(n-3) - 1$ — (4)

putting (4) in (3)

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

$= 2^3 T(n-3) - 4 - 2 - 1$

$= 2^3 T(n-3) - 7$

at any middle value K

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

base condition

$T(0) = 1$

$\Rightarrow n - K = 0$

$K = n$

4)

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

$= 2^n T(0) - 2^n + 1$

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

$= O(1)$

5) TC of

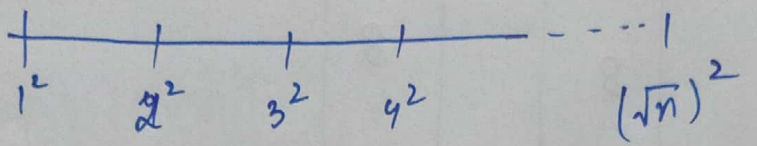
```
int i=1, s=1
while (s<=n){
    i++;
    s=s+i;
    printf("#");
}
```

i	s
1	1
2	3
3	6
4	10

s → 1 + 3 + 6 + 10

Q6 TC of

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



1, 2, 3, 4 . . . $\sqrt{n}$

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

Q7 TC of

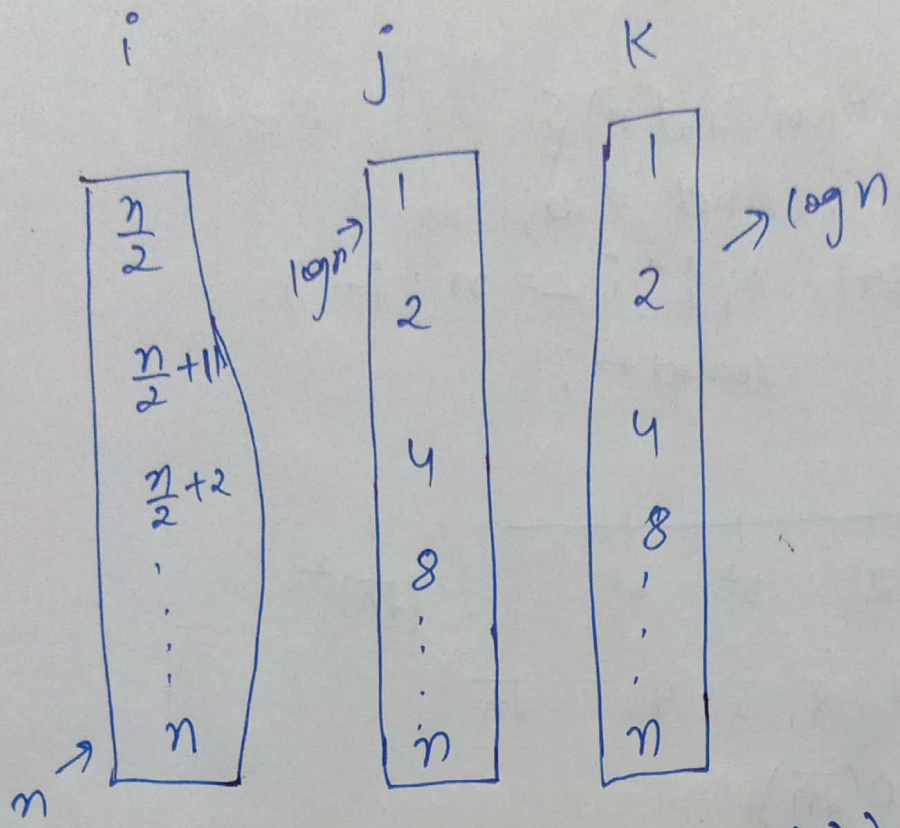
```
void function (int n){
    int i, j, k, count=0;
    for(i=n/2; i<=n; i++){
        for(j=1; j<=n; j=j*2){
            for(k=1; k<=n; k=k*2){
                count++;
            }
        }
    }
```

i j k

1

2

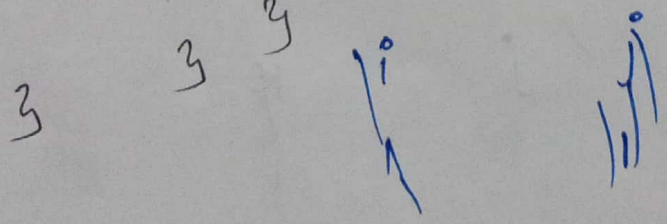
4



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

Q TC of

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



1st time $j \to 1$ to n in n times
2nd time $j \to 1$ to n in $\frac{n}{2}$ times
3rd time $j \to 1$ to n in $\frac{n}{3}$ time
⋮

$$= n + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} + \cdots \frac{n}{n}$$

$$= n\left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots \frac{1}{n}\right)$$

↓

approaching $\log(n)$

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

Q8 TC of

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

Sol. the $T(n)$ will be

$$T(n) = \begin{cases} T(n-3) + n^2 & , n > 1 \\ 0 & , n \leq 1 \end{cases}$$

by back substitution

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

$$T(n-2) = T(n-5) + (n-2)^2 \quad -(2)$$

$$T(n-3) = T(n-6) + (n-3)^2 \quad -(3)$$

$$T(n-6) = T(n-9) + (n-6)^2 \quad -(4)$$

putting (3) in (1)

$$T(n) = T(n-6) + (n-3)^2 + n^2 \quad -(5)$$

putting (4) in (5)

$$T(n) = T(n-9) + (n-6)^2 + (n-3)^2 + n^2$$

at middle value K

$$T(n) = T(n-3K) + (n-3(K-1))^2 + (n-3(K-2))^2 + \dots + n^2$$

$\because$ base condition

$$T(1) = 0$$

$$\Rightarrow n-3K = 1$$

$$\Rightarrow 3K = n-1$$

$$K = \frac{n-1}{3}$$

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

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

$\Rightarrow$

$n - 3k = 1$

$k = \frac{n-1}{3}$

$\Rightarrow$

$T(n) = T(1) + (4)^2 + (7)^2 + (10)^2 + (13)^2 + \ldots (n)^2$

$= 0 + 4^2 + 7^2 + 10^2 + 13^2 + \ldots n^2$

$\downarrow$

add n^2 terms $\frac{n}{3}$ times

$\Rightarrow O(n^3)$

Q 10.

$f(n) = n^k \quad k \geq 1$

$g(n) = a^n \quad a > 1$

$\because$ exponential grows faster than polynomial

$O(a^n) > O(n^k)$ for $a \geq p$ & $k \geq q$

let $p = 3$ and $q = 3$

$f(n) = n^3$ and $g(n) = 3^n$

$\Rightarrow \log(f(n)) = \log(n^3)$ & $\log(g(n)) = \log 3^n$

(10) let $f(n) = n^k$

and $g(n) = a^n$

∵ exponential grows faster than polynomial

$$n^k = O(a^n)$$

$\{ f(n) = O(g(n))$ iff $f(n) \leq cg(n)$ $\forall\ n \geq n_0$ & $c > 0 \}$

$\Rightarrow n^k \leq c \cdot a^n$

at $n = n_0$

$n_0^k \leq c \cdot a^{n_0}$

let $k = a = 2$

$2^k \leq c \cdot 2^{n_0}$

$n_0^2 \leq c \cdot 2^{n_0}$

this is true $\forall\ c \geq 1,\ n_0 \geq 0$

(11) TC of

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

→

i	j
0	1
1	2
3	3
6	4
10	5
15	6
21	7

$S = 0, 1, 3, 6, 10, 15 \cdots n$ — ①

$S = 0, 1, 3, 6, 10, \cdots\cdots + k$ — ②

① − ②

$0 = 0, 1, 2, 3, 4, 5, 6, \cdots + k - n$

$n = 0 + 1 + 2 + 3 + 4 + \cdots k$

$n = \frac{k(k-1)}{2}$

$\Rightarrow n \simeq k^2$

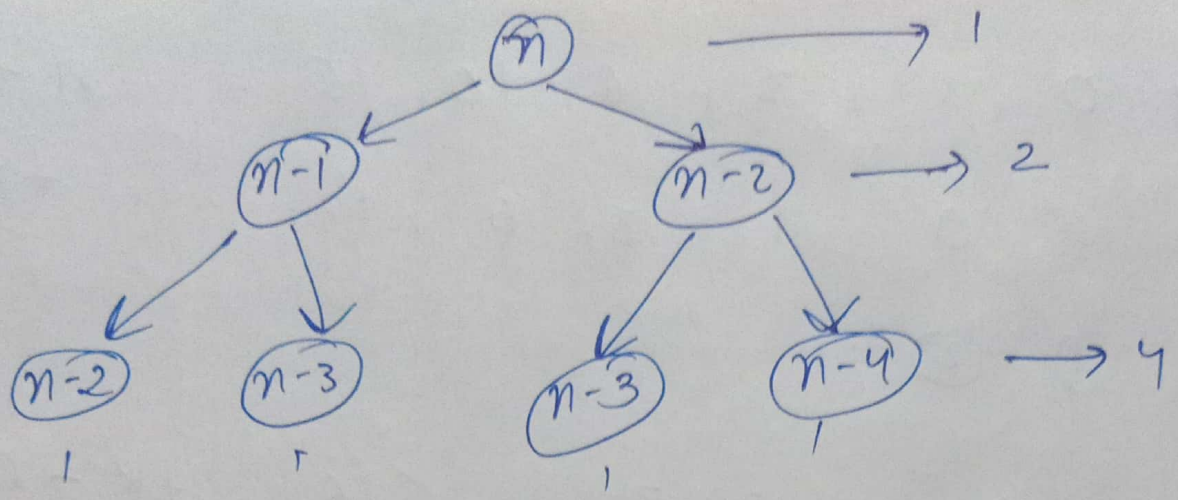
$k = \sqrt{n}$

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

(2) fibonacci series

0 1 1 2 3 5 8 13 - - - .

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



$T(n) = 1 + 2 + 4 + 8 + \ldots + 2^n$

{ n+1 terms, 2^0 to 2^n, n-0+1 = n+1

This is G.P

$$\text{sum} = \frac{a(r^{\text{no. of terms}} - 1)}{r-1} = \frac{1(2^{n+1}-1)}{2-1}$$

$$= 2^n \cdot 2$$

$$= O(2^n)$$

(3)

i)
```
for(int i=1; i<=n; ++i){
    for(int j=1; j<=n; j*=2){
        printf("#");
    }
}
```

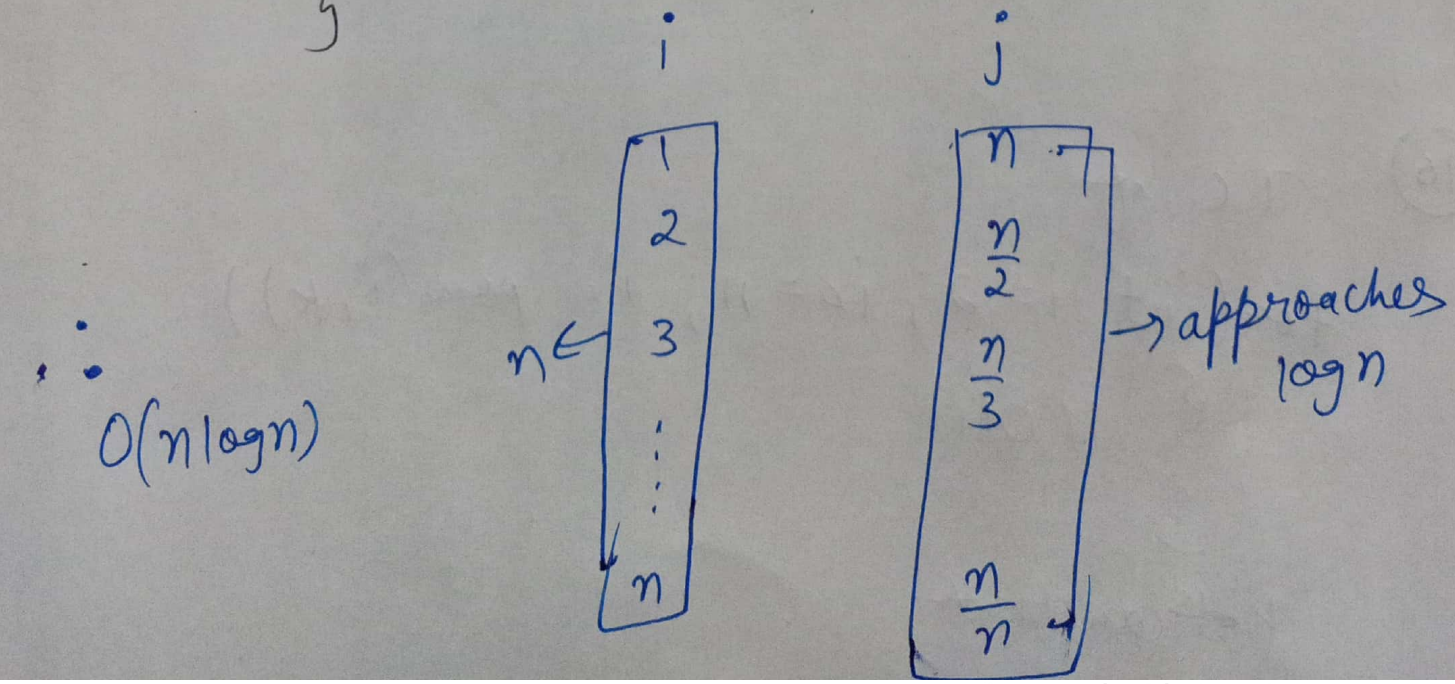
$O(n \log n)$;

(ii)
```
for(int i=0; i<=n; i++){
    for(int j=0; j<=n; ++j){
        for(int k=0; k<=n; k++){
            printf("##");
        }
    }
}
```
$O(n^3)$

(iii) ~~for(int i=1; i<=~~

(15) TC of

```
int fun(int n){
    for(int i=1; i<=n; i++){
        for(int j=1; j<n; j*=i){
            //O(1)
        }
    }
}
```



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

It can be assumed

$T(\frac{n}{2}) \geq T(\frac{n}{4})$

$\Rightarrow T(n) \; 2T(\frac{n}{2}) + cn^2$

by masters theorem

$f(n) = cn^2$

$n^k = n^2$

$\Rightarrow k = 2$

$c = \log_b^a = 1$

$\Rightarrow n^c = n^1$

$\Rightarrow f(n) > n^c$

$\Rightarrow O(n^k) = O(n^2)$

(16) TC of

```
for(int i = 2; i <= n; i = pow(i, k))
    { // O(1)
    }
```

k is constant

$i = 2, 2^k, 2^{k^2}, 2^{k^3}, \ldots\ldots 2^{k^x}$

$n = 2^{k^x}$

$\log_2 n = k^x \log_2 2$

$\log_2 n = k^x$

$\log_2(\log_2 n) = x \log_2 k$, $\log_k(\log_2 n) = k \log_k k$

$x = \frac{\log_2(\log_2 n)}{\log_2 k}$ $\Rightarrow k = \log_k(\log_2 n)$

$x = \log_k(\log_2 n)$

$n = 2^{k^x}$

$\log_2 n = k^x \log_2 2$

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

$\log_k(\log_2 n) = x$

$x = \frac{\log_2(\log_2 n)}{\log_2 k}$

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

(18)

a) $100 < \log(\log n) < \log n < \sqrt{n} < n < \log(n!) < n \log n < n^2 < 2^n < 2^{2n} < 4^n < n!$

b) $1 < n < 2n < 4n < \log(\log n)$

$1 < \log(\log n) < \sqrt{\log n} < \log n < \log 2n < 2 \log n < n < 2n < 4n < \log(n!) < n \log n < n^2 < 2 \cdot 2^n < n!$

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

(19)

```
input arr[n];
input key;
for (i=0 to n-1) {
    if (arr[i] == key) {
        cout << i;
    }
}
```

(20)

Iterative:

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

Recursive:

```
void insertionSort (int arr[], int i) {
    if (i<=0)
        return;
    insertionSort (arr, i-1);
    int j=i;
    while (j>0 && arr[j] < arr[j-1]){
        swap (arr[j], arr[j-1]);
        j--;
    }
}
```

It is called online sorting algorithm because it does not have the constraint of having entire input available at the beginning like other sorting algorithms as buble sort or selection sort. It can handle data piece by piece.

21)

Bubble Sort - $O(n^2)$
selection Sort - $O(n^2)$
insertion sort - $O(n^2)$
Merge sort - $O(n \log n)$
Quick Sort - $O(n \log n)$

22) Inplace → Bubble sort, selection sort, QuickSort, Insertion sort

Stable → Bubble sort, Insertion sort, Merge sort

Online → Insertion Sort

23) Iterative Binary search

```
int BinarySearch (int arr[], int l, int r, int x){
    while (l <= r) {
        int m = (l + r) / 2;
```

```
        if (arr[m] == x)
            return m;
        else
        if (arr[m] < x)
            l = m+1;
        else
            r = m-1;
    }
    return -1;
}
```

Recursive

```
int BinarySearch (int arr[], int l, int r, int key) {
    if (l <= r) {
        int m = (l+r)/2;
        if (arr[m] == key)
            return m;
        else if (key > arr[m])
            return BinarySearch(arr, mid+1, r, key);
        else
            return BinarySearch(arr, l, m-1, key);
    }
    return -1;
}
```

	Iterative BinarySearch	Recursive Binary Search
Space complexity	$O(1)$	$O(\log n)$
Time complexity	$O(\log n)$	$O(\log n)$

Linear Search

iterative
time → $O(n)$
space → $O(1)$

recursive
time → $O(n)$
space → $O(n)$

24) Recurrence relation for recursive Binary search

$$T(n) = T(n/2) + 1$$