Analysis of Algo

Algorithm : step by step solutions/instructions to solve any problem(well defined computurized problem) sequence of steps/instructions .

Input —--->**Algorithm—--->**Output
|
|
>Sequence of steps

*Effieciency matters : 1)Time complexity 2)Space Complexity

LINEAR SEARCH :

CODE :

```
for(int i = 0;i<arr.length;i++){
if(arr[i]==target){printf(“%d”,i);break;};
}
```

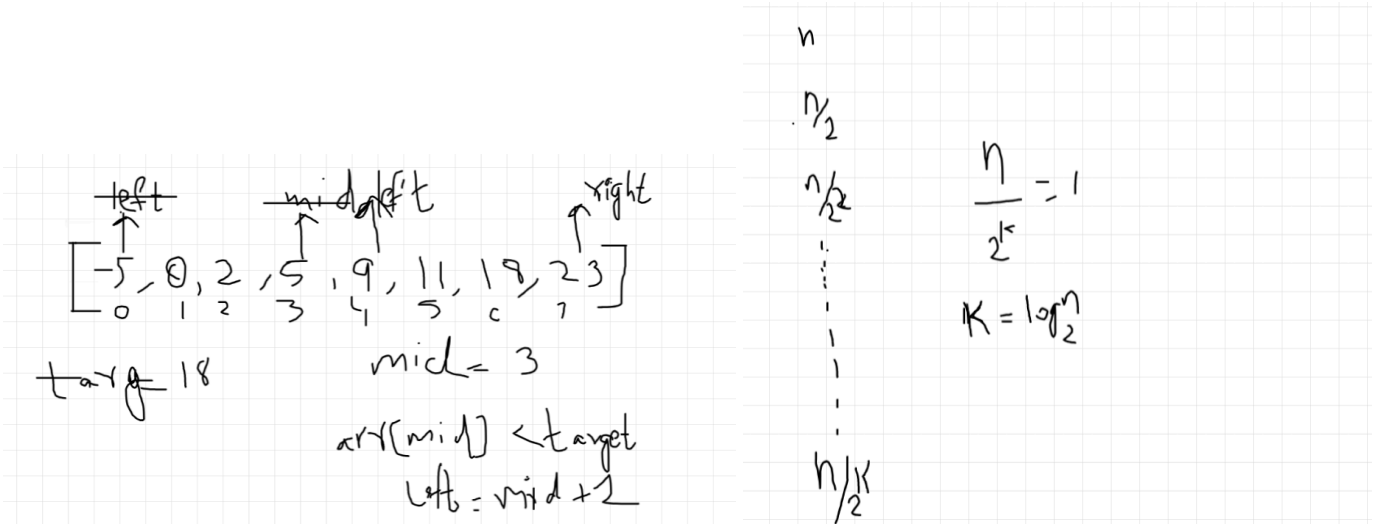
Time Complexity :

Best Case : Ω(1)
Worst Case : O(n)
Average Case : n(n+1)/2*n == O((n+1)/2)==Θ(n)

BINARY SEARCH : (Divide and conquer)

Code :
***Works for sorted arrays only**

```
Int binarysearch(int[]arr,int l,int r,int target){
while(l<=r){
int mid = l + (r-l)/2 ;
if(arr[mid]== target){return mid;}
else if(arr[mid] < target){l = mid+1;}
else{r = mid-1;}
}
return -1;
}
```



Time Complexity :

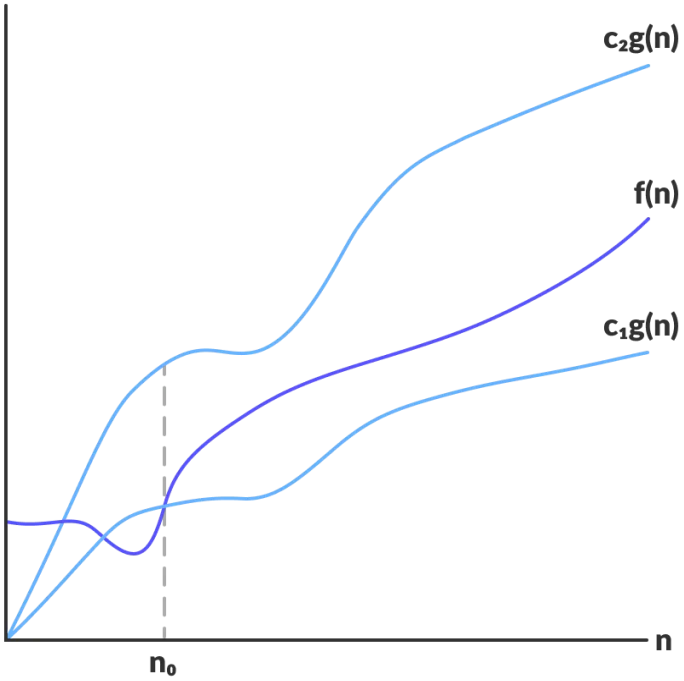
Best Case : O(1)
Worst case : O(logn)
Average Case : O(logn)

Space - iteration O(1) & recursive O(log n)
Recursive formula : T(n) = T(n/2) + c;

<u>ASYMPTOTIC NOTATIONS :</u>
=> Growth of Functions
=>compares algorithms
=>running time/space complexity.

1) **<u>theta(Θ) Notation : (Average /exact case)</u>**
   =>Tighter Bound
   For any given functions of (n)
   f(n) = Θ(g(n))

   Θ(g(n)) = f(n) : there exists positive constants c1 and c2 ,and n0 such that
   0 <= c1.g(n) <= f(n) <=c2.g(n) for n>=n0

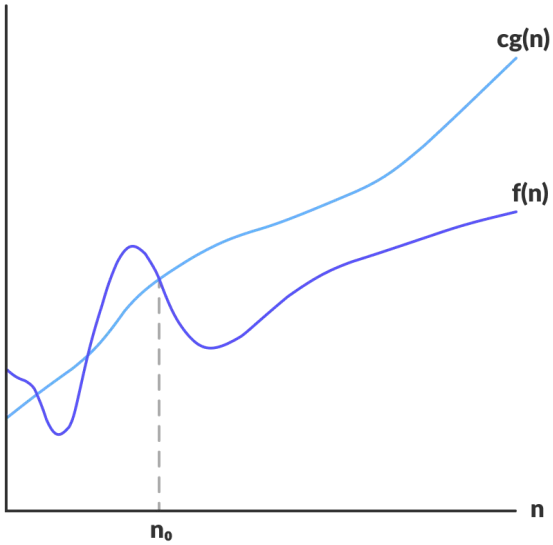


f(n) = Θ(g(n))

2) <u>Big Oh Notaion(O)</u> :

=> Upper Bound(Worst Case)

=>There exists +ve values c and n0 such that 0 <= f(n)<= cg(n) for all n>=n0

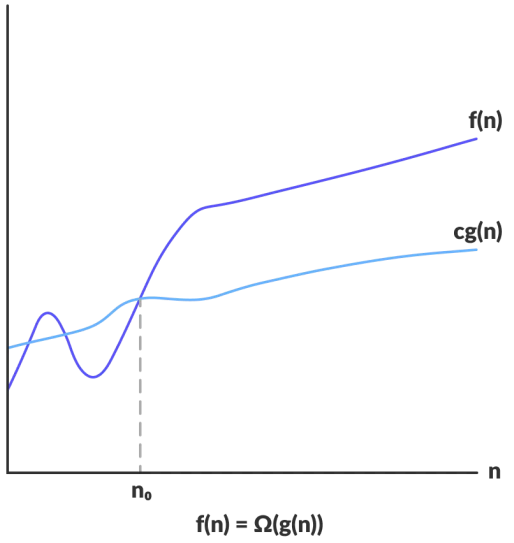


f(n) = O(g(n))

3) **Big omega (Ω) :**

=>lowerbound (Best Case analysis)

=>There exists +ve c and n0 such that 0 <= cg(n) <= f(n) for all n >=n0



4) **Small O notation (o) :**

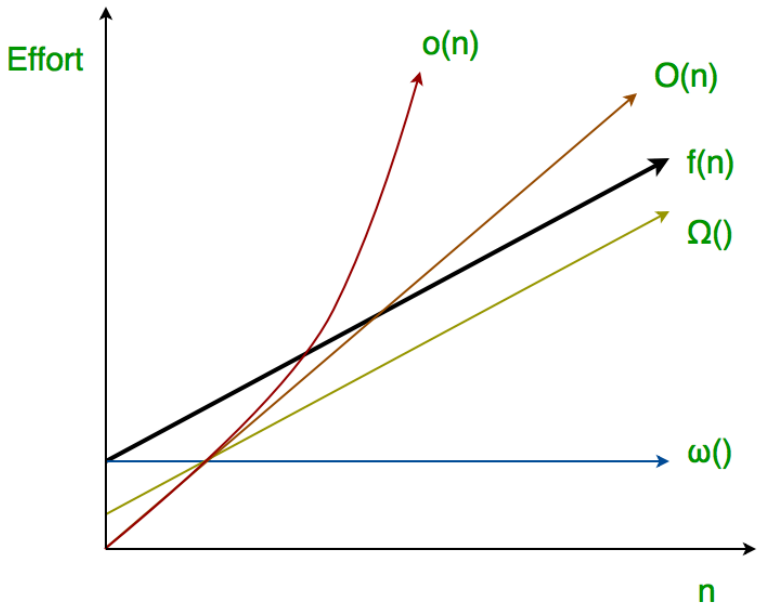
=> Loose upper bound

=> There exists +ve values c and n0 such that 0 <= f(n)< cg(n) for all n>=n0

5) **Small omega notaion (w) :**

=> Loose Lower Bound

=>There exists +ve c and n0 such that 0 <= cg(n) < f(n) for all n >=n0



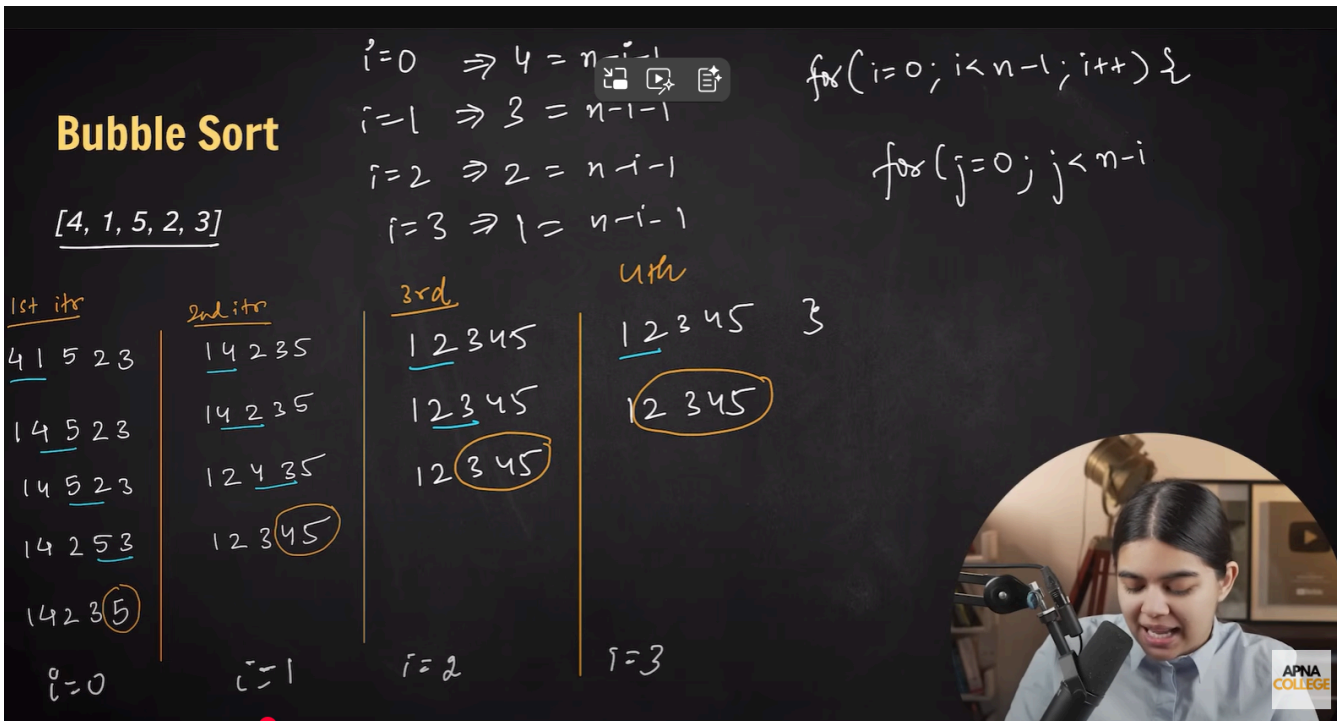
BUBBLE SORT :

//you can do this by pushing larger at end or smaller at beginning

CODE :

```
for(int i = 0;i<arr.length-1;i++){
Boolean swapped = false;
for(int j  =0;j<arr.length-i-1;j++){
if(arr[j]>arr[j+1]){int temp = arr[j];arr[j] = arr[j+1];arr[j+1] = temp;swapped = true;}
};
if(!swapped){break;}
}
```

(boolean swapped used for optimization to break for a sorted array)



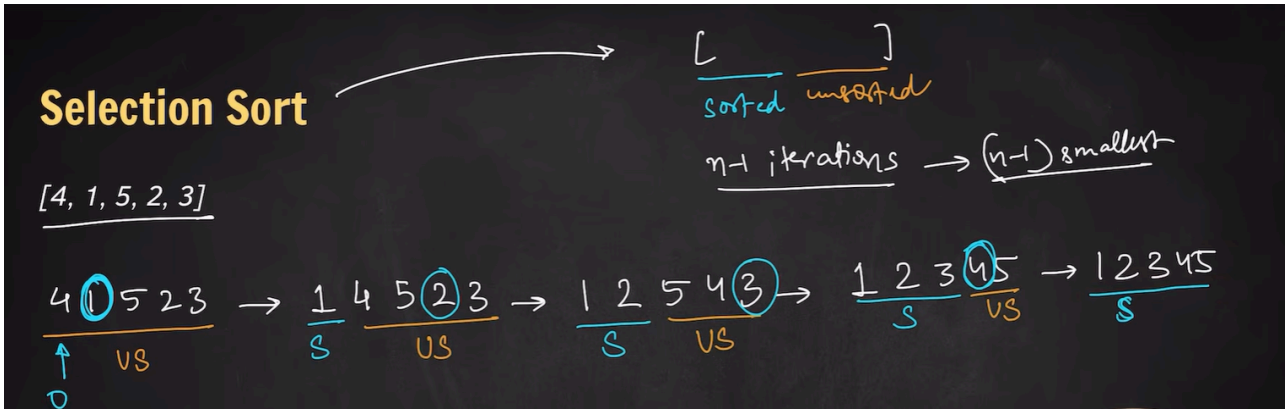
Time complexities :

Best Case : O(n) with optimization,O(n^2) without optimization
Average case,worst case : O(n^2)

SELECTION SORT :

Code :

```
void selectionsort(int [] arr){
for(int i = 0;i<arr.length - 1;i++){
Int smallestindex = i;
for(int j = i+1;j<arr.length;j++){
if(arr[j]<arr[smallestindex]){smallestindex = j;}
}
Int temp = arr[i];
arr[i] = arr[smallestindex];
arr[smallestindex] = temp;
}
}
```



Time complexities :

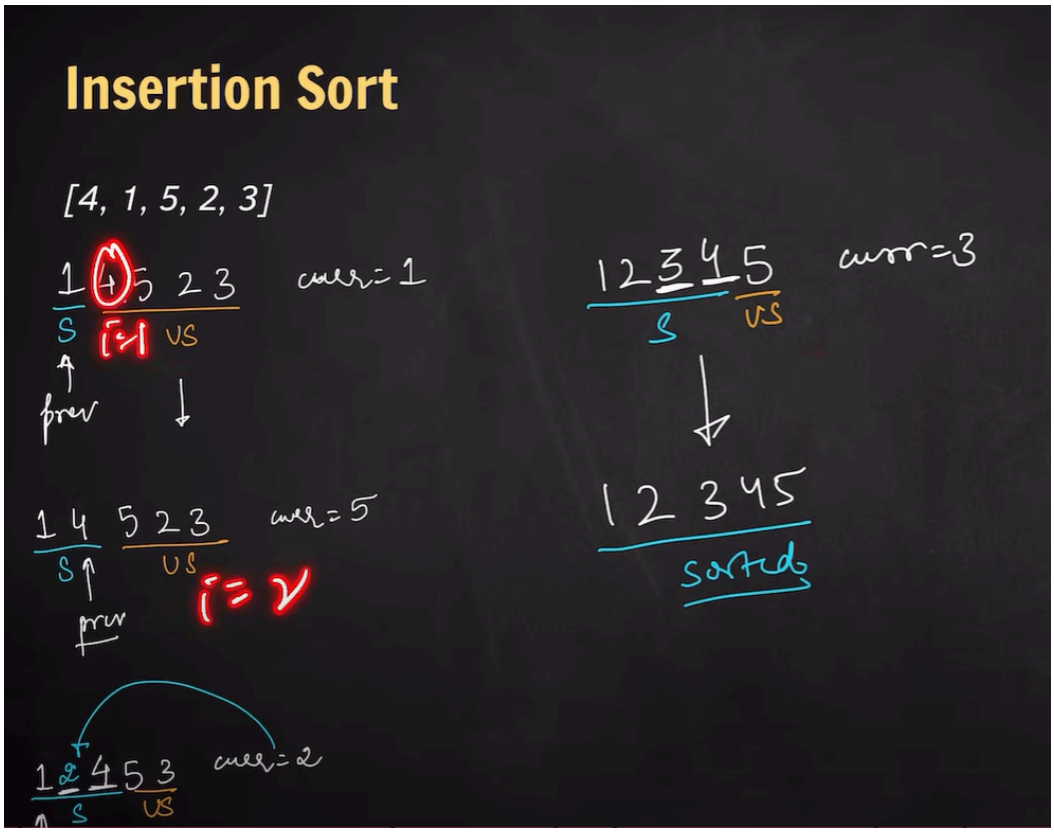
Best Case,Average case,worst case : O(n^2)

INSERTION SORT :

CODE :

```
Void insertionsort(int[]arr){
        for(int i =1 ;i<arr.length;i++){
        Int curr = arr[i];
        Int prev = i-1;
        while(prev>=0 && arr[prev]>curr){
        arr[prev+1]=arr[prev];
        prev--;
        }
        arr[prev+1] =curr;
```

```
}
}
```



Time complexities :

Best Case : O(n)
Average case,worst case : O(n^2)

MERGE SORT :

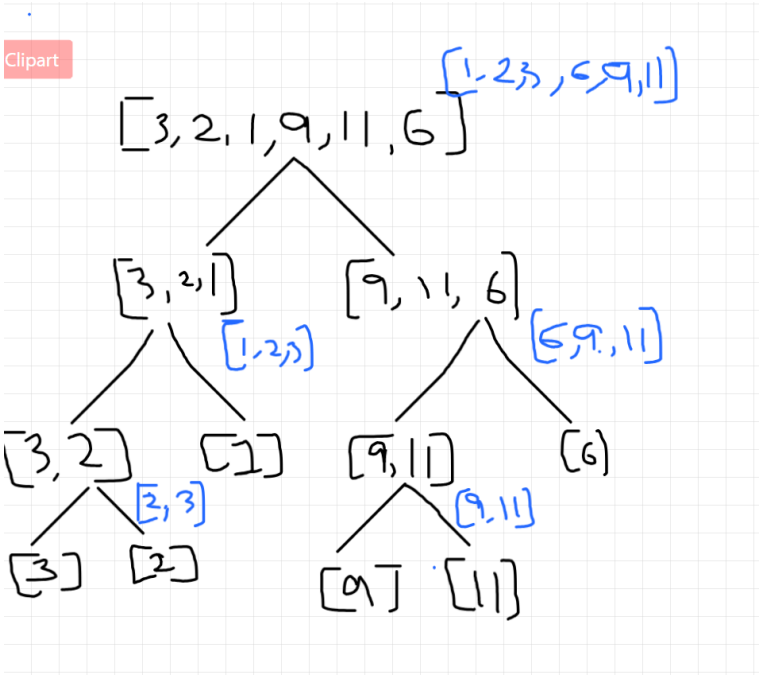
```
void mergeSort(int[] arr, int l, int r) {
    if (l < r) {
        int mid = l + (r - l) / 2;
        mergeSort(arr, l, mid);
        mergeSort(arr, mid + 1, r);
        merge(arr, l, mid, r);
    }
}
void merge(int[] arr, int l, int mid, int r) {
    int n1 = mid - l + 1;
    int n2 = r - mid
    int[] left = new int[n1];
    int[] right = new int[n2];
    for (int i = 0; i < n1; i++) left[i] = arr[l + i];
    for (int j = 0; j < n2; j++) right[j] = arr[mid + 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
```

```
 if (left[i] <= right[j]) {
            arr[k++] = left[i++];
        } else {
            arr[k++] = right[j++];
        }
    }
    while (i < n1) arr[k++] = left[i++];
    while (j < n2) arr[k++] = right[j++];
}
```

Time complexities :

Best case,worst Case,Average Case: O(nlogn)

Recurrence relation : T(n)=2T(n/2)+O(n)

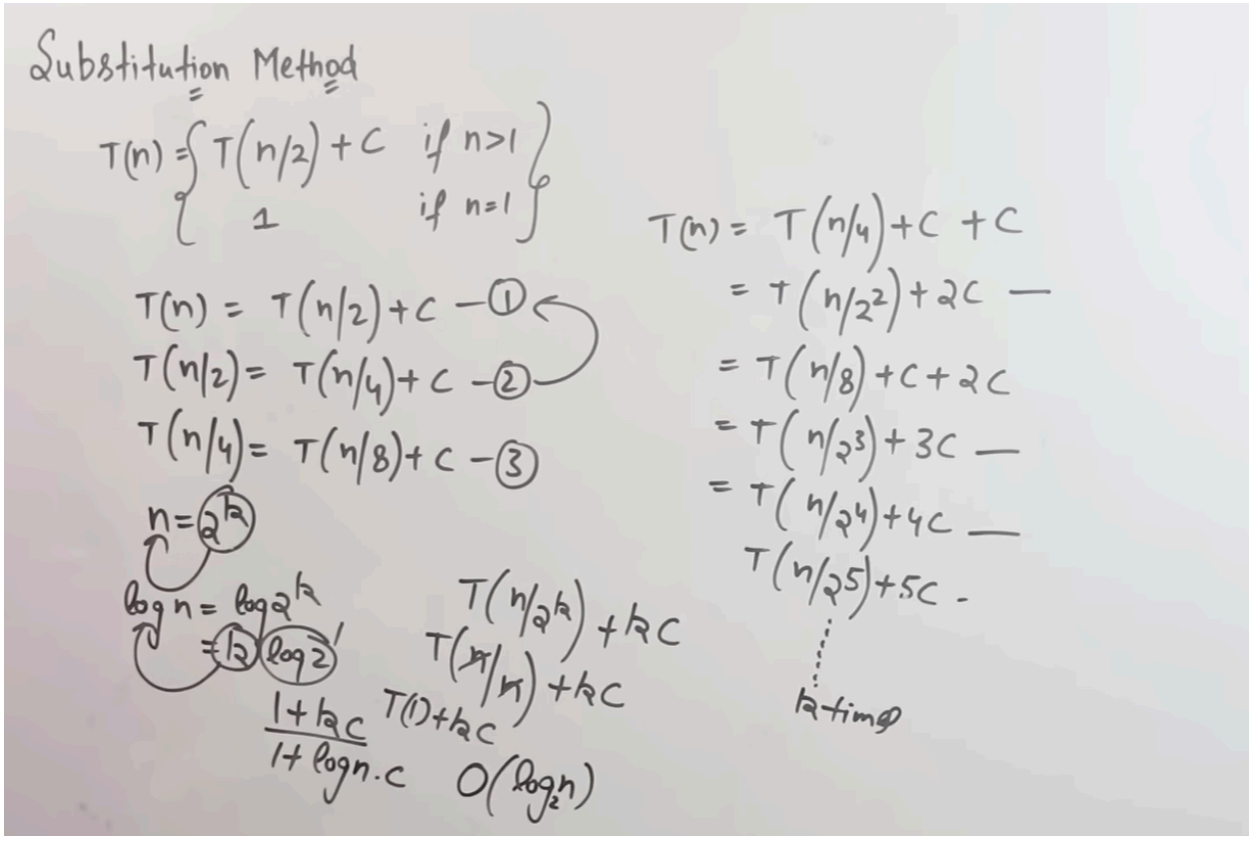


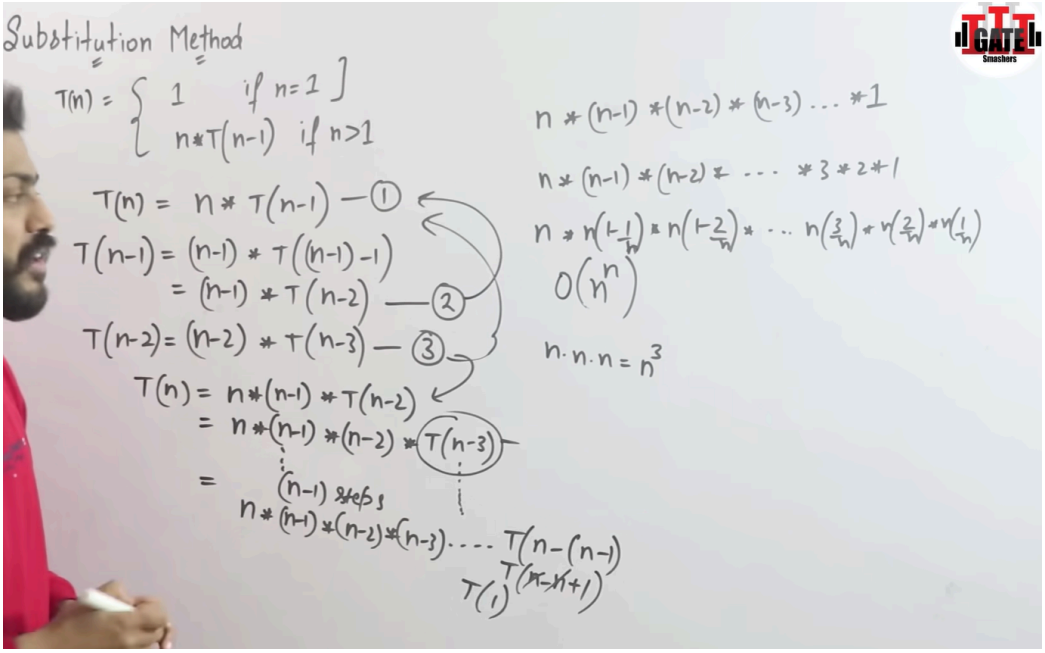
METHODS OF SOLVING RECURSION TREE :

1) **Substitution/iteration method :**

   =>Use mathematical induction

   =>Use base case such as T(1) = 1





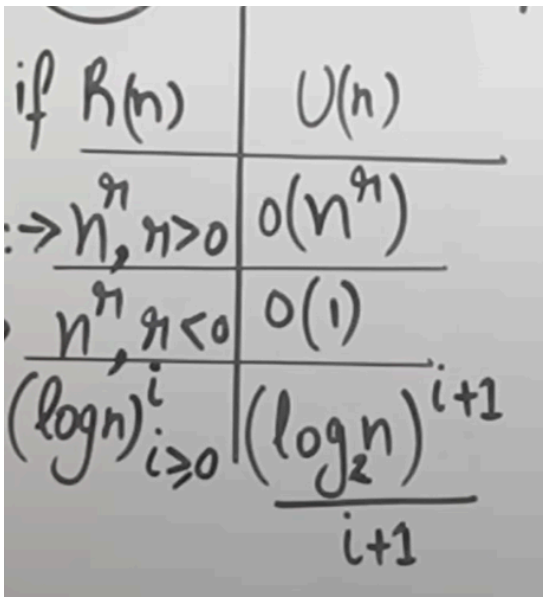
2) MASTER METHOD :

T(n) = aT(n/b) + f(n) ;
a>=1;b>1;

Solution is

T(n) = [n^(loga b)][U(n)] ;
U(n) depends on h(n);

h(n) = f(n) / [n^(loga b)];



DEFINATION 2 :

📌 Master Method — Quick Reference

General Form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- $a \geq 1$: number of subproblems
- $b > 1$: input size shrink factor
- $f(n)$: cost of dividing + combining results

We compare $f(n)$ with:

$$n^{\log_b a}$$

= work done by recursion per level.

🔹 3 Cases of Master Theorem

Case 1 (Recursion dominates):

If

$$f(n) = O\left(n^{\log_b a - \epsilon}\right), \quad \epsilon > 0$$

→

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

Case 2 (Balanced):

If

$$f(n) = \Theta\left(n^{\log_b a}\right)$$

→

$$T(n) = \Theta\left(n^{\log_b a} \log n\right)$$

Case 3 (Extra work dominates):

If

$$f(n) = \Omega\left(n^{\log_b a + \epsilon}\right), \quad \epsilon > 0$$

and **regularity condition** holds:

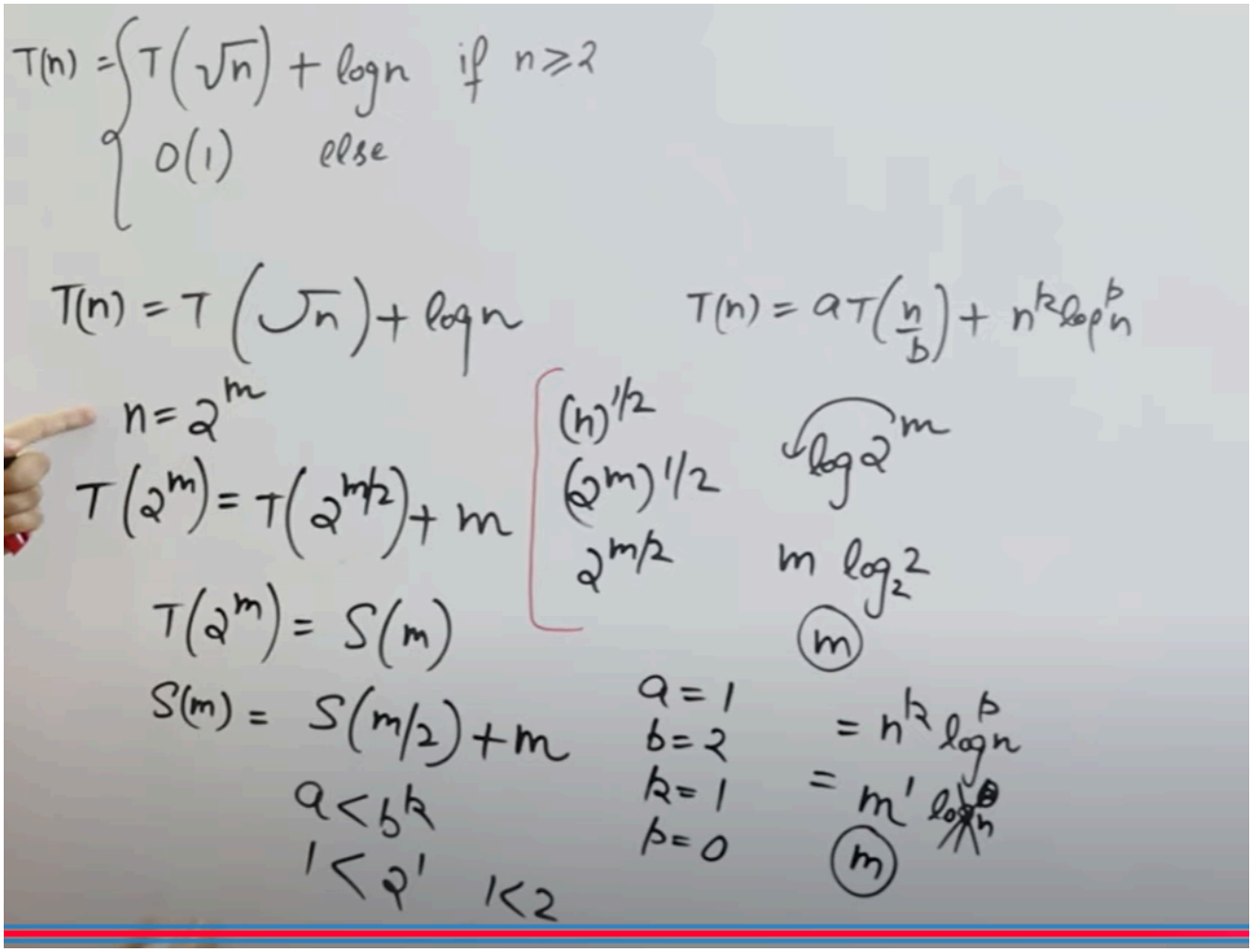
$$a \cdot f(n/b) \leq c \cdot f(n), \quad c < 1$$

→

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

If all three of them fail ,then the method cannot be applied.

3) changing variables



HEAP :

CODE 👍

IF ARRAY IS GIVEN ALREADY WITH ELEMENTS :

```
    static void  minheapify(int[]arr,int i){
        int largest = i;
        int left = 2*i+1;
        int right = 2*i+2;
        if(left < arr.length && arr[largest]>arr[left]){largest = left;}
        if(right<arr.length && arr[largest]>arr[right]){largest = right;}

        if(largest!=i){
            int temp = arr[i];
            arr[i]=arr[largest];
            arr[largest] = temp;
            minheapify(arr, largest);
```

```
        }
    }
    void display(){
        for(int i = 0;i<=size;i++){
            System.out.println(heaparray[i]);
        }
    }
    static void minheaparrayn (int[]arr){
        for(int i = arr.length/2 -1 ;i>=0;i--){
            minheapify(arr,i);
        }
    }
```

IF ARRAY GENERATED IN A CLASS AND CLASS IS USED WITH INSERT AND EXTRACT PROPERTY :

```
public class heap {
    int[] heaparray;
    int maxsize;
    int size =-1;
    heap(int maxsize){
        this.maxsize = maxsize;
        heaparray = new int[maxsize];
    }
    void mininsert(int x){
        if(size+1==maxsize){System.out.println("maxsize reached");return;}
        heaparray[++size] = x;
        int curr = size;
        int parent = (curr-1) /2;
        while(curr>=0&&heaparray[parent]>heaparray[curr]){int temp =
heaparray[parent];heaparray[parent] = heaparray[curr];heaparray[curr]=temp;curr =
parent;parent = (curr-1) /2;}
    }

    void extractmin(){
        if(size==-1){return;}
        if(size==0){System.out.println(heaparray[0]);--size;}
        int min = heaparray[0];
        heaparray[0] = heaparray[size];
        --size;
        minheapify(heaparray,0);
        System.err.println(min);
    }
    static void  minheapify(int[]arr,int i){
        int largest = i;
        int left = 2*i+1;
```

```
        int right = 2*i+2;
        if(left < arr.length && arr[largest]>arr[left]){largest = left;}
        if(right<arr.length && arr[largest]>arr[right]){largest = right;}

        if(largest!=i){
            int temp = arr[i];
            arr[i]=arr[largest];
            arr[largest] = temp;
            minheapify(arr, largest);
        }
    }
```

1. **Constructor**
   - `Heap(int maxsize, boolean isMinHeap)` → initializes heap array and sets type (min/max).
2. **Insert**
   - `insert(int x)` → adds a new element and maintains heap property (bubble up).
3. **Heapify**
   - `heapify(int i)` → fixes heap property from node `i` downward (bubble down).
4. **Build Heap**
   - `buildHeap(int[] arr)` → transforms any array into a valid heap.
5. **Extract Root**
   - `extractRoot()` → removes and returns min (for min-heap) or max (for max-heap).
6. **Peek Root**
   - `peek()` → returns root value without removing it.
7. **Delete Element**
   - `delete(int i)` → removes element at index `i`.
8. **Display Heap**
   - `display()` → prints heap elements in array form.
9. **Get Size**
   - `getSize()` → returns the number of eleme↓n heap.

Operation	Best Case	Average Case	Worst Case
Constructor	O(1)	O(1)	O(1)
Insert	O(1)	O(log n)	O(log n)
Heapify (down)	O(1)	O(log n)	O(log n)
Build Heap	O(n)	O(n)	O(n)
Extract Root	O(log n)	O(log n)	O(log n)
Peek Root	O(1)	O(1)	O(1)
Delete Element	O(log n)	O(log n)	O(log n)
Display Heap	O(n)	O(n)	O(n)
Get Size	O(1)	O(1)	O(1)
Check Empty	O(1)	O(1)	O(1)

QUICKSORT :

```
    static int pivotend(int[]arr,int left,int right){
        int val = arr[right];
        int i = left;
        for(int j = left;j<right;j++){
            if(arr[j]<=val){
                int temp = arr[j];
                arr[j] = arr[i];
                arr[i] = temp;
                i++;
            }
        }
        int temp = arr[right];
        arr[right] = arr[i];
        arr[i] = temp;
        return i;
    }
```

```java
  // return pivot at start
static int pivotstart(int[]arr,int left,int right){
    int val = arr[left];
    int i = left+1;
    for(int j = left+1;j<=right;j++){
        if(arr[j]<=val){
            int temp = arr[j];
            arr[j] = arr[i];
            arr[i] = temp;
            i++;
        }
    }
    int temp = val;
    arr[left] = arr[i-1];
    arr[i-1] = temp;
    return i-1;
}



// return pivot at random
static int pivotrandom(int[]arr,int left,int right){
    int random = (int)(Math.random() * (right - left + 1)) + left;
    int val = arr[random];
    arr[random] = arr[left];
    arr[left] = val;
    int i = left+1;
    for(int j = left+1;j<=right;j++){
        if(arr[j]<=val){
            int temp = arr[j];
            arr[j] = arr[i];
            arr[i] = temp;
            i++;
        }
    }
    int temp = val;
    arr[left] = arr[i-1];
    arr[i-1] = temp;
    return i-1;
}
```

```java
// return pivot at median
static int pivotmedian(int[]arr,int left,int right){
    int mid = (right+left)/2;
    int pivotindex = 0;
    if ((arr[left] - arr[mid])*(arr[right] - arr[left]) >= 0)
    pivotindex = left;
    else if ((arr[mid] - arr[left])*(arr[right] - arr[mid]) >= 0)
    pivotindex = mid;
    else
    {pivotindex = right;}

    int val = arr[pivotindex];
    arr[pivotindex] = arr[left];
    arr[left] = val;
    int i = left+1;
    for(int j = left+1;j<=right;j++){
        if(arr[j]<=val){
            int temp = arr[j];
            arr[j] = arr[i];
            arr[i] = temp;
            i++;
        }
    }
    int temp = val;
    arr[left] = arr[i-1];
    arr[i-1] = temp;
    return i-1;
}
```

```java
static void quicksortend(int[]arr,int left,int right){
    if(left<right){
    int pi = pivotend(arr,left,right);
    quicksortend(arr, left, pi-1);
    quicksortend(arr, pi+1, right);}
}
```

```
    //quicksort for pivot at start
    static void quicksortstart(int[]arr,int left,int right){
        if(left<right){
        int pi = pivotstart(arr,left,right);
        quicksortstart(arr, left, pi-1);
        quicksortstart(arr, pi+1, right);}
    }
    //quicksort for pivot at random pivot
    static void quicksortrandom(int[]arr,int left,int right){
        if(left<right){
        int pi = pivotrandom(arr,left,right);
        quicksortrandom(arr, left, pi-1);
        quicksortrandom(arr, pi+1, right);}
    }
    //quicksort for pivot at median
    static void quicksortmedian(int[]arr,int left,int right){
        if(left<right){
        int pi = pivotmedian(arr,left,right);
        quicksortmedian(arr, left, pi-1);
        quicksortmedian(arr, pi+1, right);}
    }
```

Time Complexities :

Pivot Choice	Best Case	Average Case	Worst Case	Notes
First element	O(n log n)	O(n log n)	**$O(n^2)$**	Worst case if array is already sorted (or reverse sorted).
Last element	O(n log n)	O(n log n)	**$O(n^2)$**	Same weakness as first element.
Random element	O(n log n)	O(n log n)	$O(n^2)$ (very rare)	Randomization makes worst case unlikely.
Median-of-three	O(n log n)	O(n log n)	O(n log n) (practically)	Picks a "balanced" pivot most of the time, avoids sorted-input worst case.

Simple but slow → Bubble, Selection, Insertion
Divide & Conquer → Merge, Quick
Heap-based → HeapSort
Specialized (integers) → Counting, Radix, Bucket
Practical hybrids → TimSort, IntroSort

🔹 1. Comparison-based Sorting

These compare elements to decide order.

Algorithm	Best Case	Average Case	Worst Case	Stable?	Notes
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	✅ Yes	Simple, good for teaching, not used in practice.
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	❌ No	Always $O(n^2)$, selects min and swaps.
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	✅ Yes	Efficient for small or nearly sorted arrays.
Merge Sort	O(n log n)	O(n log n)	O(n log n)	✅ Yes	Divide & conquer, needs extra space.
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	❌ No	In-place, fastest in practice with good pivoting.
Heap Sort	O(n log n)	O(n log n)	O(n log n)	❌ No	Uses heap data structure, good worst-case bound.

HEAP SORT :

```
//For sorting in decreasing order change "max" to "min" and
// < to > in (arr[largest],arr[left]) (arr[largest],arr[right])

public class heapsort {
    static void maxheapify(int[]arr,int i,int end){
        int largest = i;
        int left = 2*i+1;
        int right = 2*i+2;
        if(left < end && arr[largest]<arr[left]){largest = left;}
        if(right<end && arr[largest]<arr[right]){largest = right;}
        if(largest!=i){
            int temp = arr[i];
            arr[i]=arr[largest];
            arr[largest] = temp;
            maxheapify(arr, largest,end);
        }

    }
    static void maxheaparrayn (int[]arr){
```

```
            for(int i = arr.length/2 -1 ;i>=0;i--){
            maxheapify(arr,i,arr.length-1);
        }
    }
    static void heapsortalgo(int[]arr,int end){
        if(end==-1||end==0){return;}
        int max = arr[0];
        arr[0] = arr[end];
        arr[end] = max;
        --end;
        maxheapify(arr,0,end);
        heapsortalgo(arr, end);
    }
    public static void main(String[] args) {
        int[]arr = {-1,2,2,3,0,5,2,6,-10,9,4,7};
        maxheaparrayn(arr);
        heapsortalgo(arr,arr.length-1);
        for(int i = 0;i<arr.length;i++){
            System.out.println(arr[i]);
        }
    }
}
```

Time Complexity :

Average,worst,best : O(nlogn)

Counting Sort :

```
//elemEnts have to be>0 for this code -> TIME COMPLEXITY O(N+MAX);

public class countingsort {
    static void countsort(int[]arr){
        int max = arr[0];

        for(int i = 1;i<arr.length;i++){
            if(max<arr[i]){max = arr[i];}
        }//fetch max element O(N)

        int[]count = new int[max+1];//declare size for 0 to max indexing O(MAX)

        for(int i = 0;i<arr.length;i++){
            count[arr[i]]++;
        }//store freqencies O(N)
```

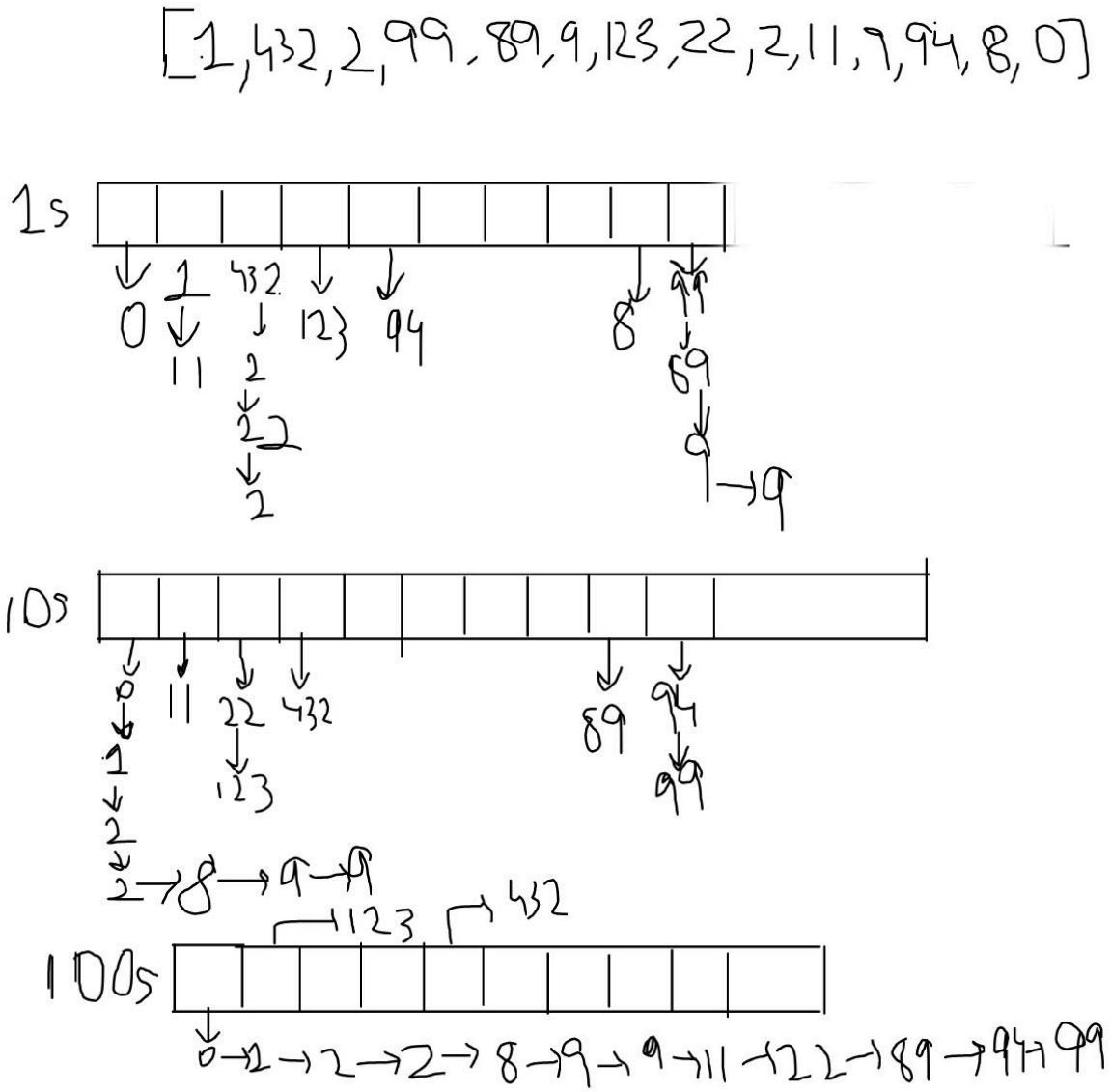
```
        //do cummulatives of indexies O(MAX)
        for(int i = 1;i<count.length;i++){
            count[i]+=count[i-1];
        }
        int []output = new int[arr.length];

        //fetch the output by traversing org array from end and storing it in index acc to
countarray map-1 while decrementing O(N)
        for(int i = arr.length-1;i>=0;i--){
            output[count[arr[i]]-1] = arr[i];
            count[arr[i]]--;
        }
        //transfer the output to org O(N)
        for(int i = arr.length-1;i>=0;i--){
            arr[i] = output[i];
        }
    }
    public static void main(String[] args) {
        int arr[] = {0,2,1,3,3,9,0,6,4,6,3,8};
        countsort(arr);
        for(int i = 0;i<arr.length;i++){
            System.out.println(arr[i]);
        }
    }
}
```

Time Complexity 👍

Average,best,Worst Case : O(N+MAX);
Non-comparision and stable sort;

RADIX SORT :



CODE :

```
//radix sort has time Complexity of O(d*(n+max)) ~ O(n)

public class radixsort {
    static int getmax(int[]arr){
        int max = arr[0];
        for(int i = 1;i<arr.length;i++){
            if(max<arr[i]){max=arr[i];}
```

```
        }
        return max;
    }//to get max

    static void countingsort(int arr[],int exp){

        int[]count = new int[10];//declare size for digits(MAX){(0-9) in this case}

        for(int i = 0;i<arr.length;i++){
            count[(arr[i]/exp)%10]++;
        }//store freqencies O(N) //(arr[i]/exp)%10 gives the value at an injdex of
1/10s/100s/100s......

        //do cummulatives of indexies O(MAX)
        for(int i = 1;i<count.length;i++){
            count[i]+=count[i-1];
        }
        int []output = new int[arr.length];

        //fetch the output by traversing org array from end and storing it in index acc to
countarray map-1 while decrementing O(N)
        for(int i = arr.length-1;i>=0;i--){
            output[count[(arr[i]/exp)%10]-1] = arr[i];
            count[(arr[i]/exp)%10]--;
        }

        //transfer the output to org O(N)
        for(int i = arr.length-1;i>=0;i--){
            arr[i] = output[i];
        }
    }
    static void radixsorting(int[]arr){
        int max = getmax(arr);
        for(int exp = 1;max/exp>0;exp =exp*10){
            countingsort(arr,exp);
        }
    }//perform on all digits that is digits present at max number O(d)
    public static void main(String[] args) {
        int [] arr = {1,432,2,99,89,9,123,22,2,11,94,8,0};
        radixsorting(arr);
        for(int i = 0;i<arr.length;i++){
            System.out.println(arr[i]);
        }
    }
}
```

Time Complexity;

Best,average,worst : O(d*(n+max)) ~ O(d*n) ~ O(n)
Stable and non comparison sort;