

SORTING

→ Arrange in an order.

→ Basic sorting Algorithm:

- ◊ Bubble Sort
- ◊ selection sort
- ◊ Insertion sort
- ◊ counting sort

* Bubble Sort:

◊ idea: large elements come to the end of array by swapping with adjacent elements.

eg. $arr[] = \{5, 4, 1, 3, 2\}$ $\rightarrow n=5$

5, 4, 1, 3, 2	4, 1, 3, 2, 5	1, 3, 2, 4, 5	1, 2, 3, 4, 5
4, 5, 1, 3, 2	1, 4, 3, 2, 5	1, 3, 2, 4, 5	1, 2, 3, 4, 5
4, 1, 5, 3, 2	1, 3, 4, 2, 5	1, 2, 3, 4, 5	1, 2, 3, 4, 5
4, 1, 3, 5, 2	1, 3, 2, 4, 5	1, 2, 3, 4, 5	3rd turn
4, 1, 3, 2, 5	1st turn	2nd turn	0 to n-5
0th turn	0 to n-3	0 to n-4	
0 to n-2	yha swapping nhi karenge, aki last wala largest h, ye already hum nikal chuke h.		

Pseudo code:

// for turns.

for (turns = 0 to n-2) {

// for comparing & swapping

for (int j = 0 to n-2-turns)

{

if (arr[j] > arr[j+1]) {

int temp = arr[j]

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

code:

```
public static void main (String args[]) {
```

```
Scanner sc = new Scanner(System.in);
```

```
int n = sc.nextInt();
```

```
int arr[] = new int[n];
```

```
for (int i = 0; i < arr.length; i++) {
```

```
    arr[i] = sc.nextInt();
```

```
for (int turns = 0; turns < arr.length - 1; turns++) {
```

```
    int swap = 0;
```

```
    for (int j = 0; j < arr.length - 1 - turns; j++) {
```

```
        if (arr[j] > arr[j+1]) {
```

```
            int temp = arr[j];
```

```
            arr[j] = arr[j+1];
```

```
            arr[j+1] = temp;
```

```
            swap++;
```

```
        }
```

```
    } if (swap == 0) {
```

```
        break;
```

```
}
```

```
System.out.print(arr[j+1]);
```

```
}
```


* selection sort:

idea: pick the smallest (from unsorted), put it at the beginning.

eg. - $arr[] = \{5, 4, 1, 3, 2\}$

$\boxed{5, 4, 1, 3, 2}$ $n=5$

$\boxed{1}$ 5 4 3 2 $\rightarrow i=0$

$\boxed{1 \ 2}$ 5 4 3 $\rightarrow i=1$

$\boxed{1 \ 2 \ 3}$ 5 4 $\rightarrow i=2$

$\boxed{1 \ 2 \ 3 \ 4}$ 5 $\rightarrow i=3$
bad hum iske loop iterate nhi karenge, ki last wala to sorted hi hoga.

pseudo code:

```
for(int i=0 to n-2)
{
    minposition = i; max
    for(j=i+1 to n-1) min
    {
        unsorted array start
        if(arr[minp] > arr[j])
        {
            minp = j
        }
    }
    int temp = arr[minp];
    arr[minp] = arr[i];
    arr[i] = temp;
}
```

Actual code:

```
public static void selectionSort(int arr[]) {
```

```
    for (int i = 0; i < arr.length - 1; i++) {
```

```
        int minpos = i
```

```
        for (int j = i + 1; j < arr.length; j++) {
```

```
            if (arr[minpos] > arr[j]) {
```

```
                minpos = j;
```

```
            }
```

```
            // swap
```

```
            int temp = arr[minpos];
```

```
            arr[minpos] = arr[i];
```

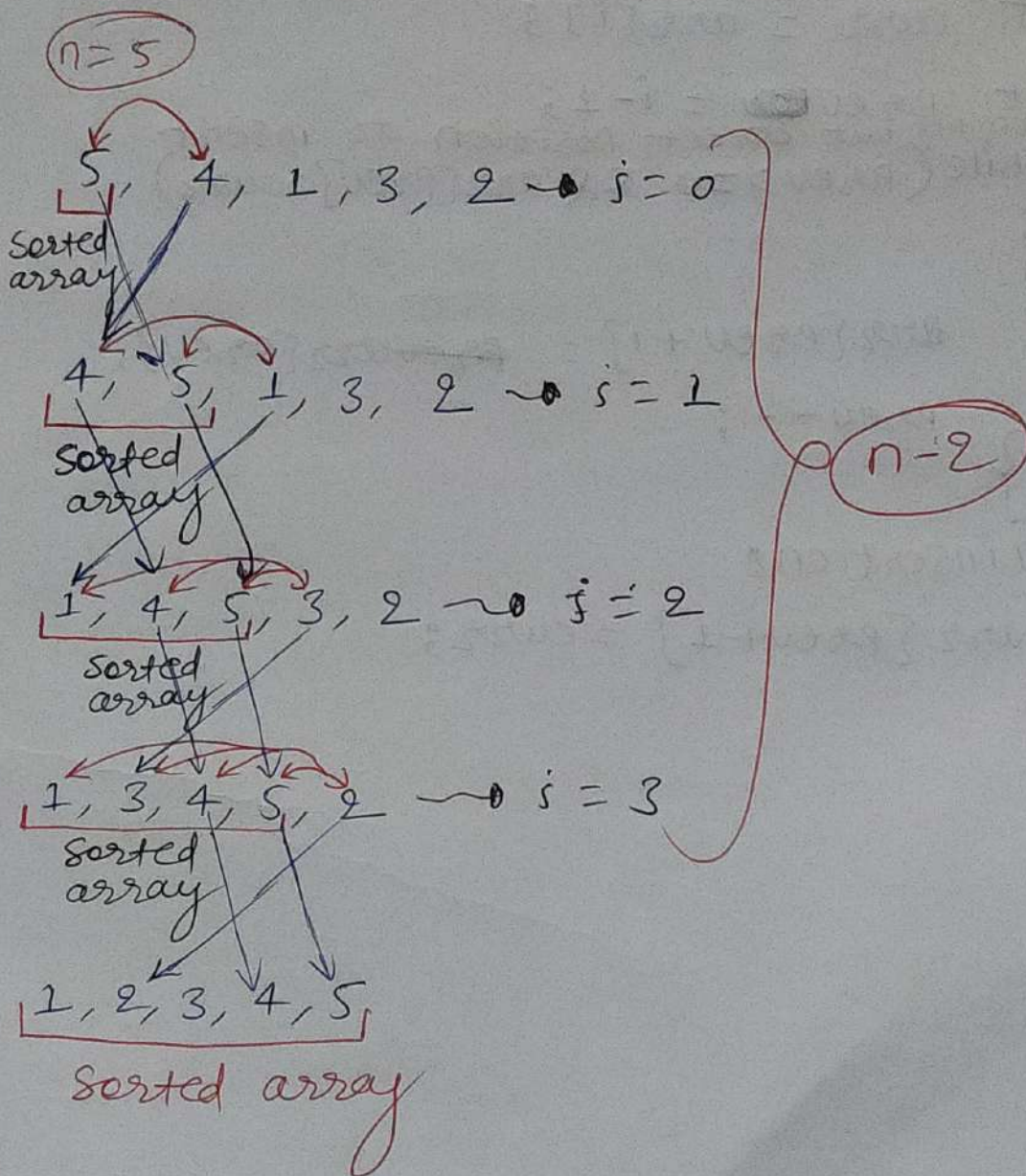
```
            arr[i] = arr temp;
```

```
        }
```


* Insertion sort:

idea: pick an element (from unsorted array) & place in the right pos. in sorted array.

eg. $arr[] = \{5, 4, 1, 3, 2\}$



Actual
def. code

```
Public static void insertionSort(int arr[])  
{
```

```
    for(int i=1; i<arr.length; i++){
```

```
        int curr = arr[i];
```

```
        int prev = i-1;
```

```
        // finding out correct position to insert  
        while(prev >= 0 && arr[prev] > curr)
```

```
        {
```

```
            arr[prev+1] = prevarr[prev];
```

```
            prev--;
```

```
        }
```

```
        // insertion
```

```
        arr[prev+1] = curr;
```

```
    }
```

```
}
```


* Inbuilt Sort :

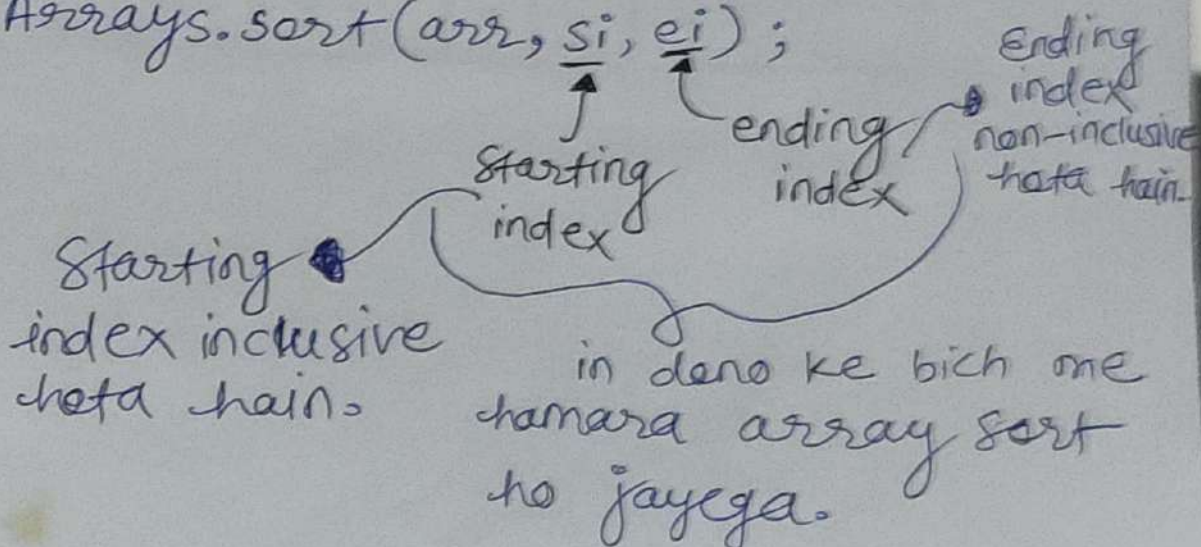
```
import java.util.Arrays;  
or, import java.util.*;
```

```
Arrays.sort(arr);
```

- bus ye function ko hame call karna hai, aur hamara array (arr) sort ho jayega. inc. order me
- Is function ki time complexity approx (baki) sorting algorithm se kam hoti hai.
- $O(n \log n)$;

* Aur agar hume array ko kisi particular index se lekar kisi particular index tak sort karna hai to uske like :

```
Arrays.sort(arr, si, ei);
```



⇒ for reverse order of sorting/decreasing order of sorting:

```
{ import java.util.Collections;  
  Arrays.sort(arr, Collections.reverseOrder());
```

ye comparator
ka use karta
hai, dec. sort
ke liye.

reverse order function
object type variable
ke upar kam karta
hai, ye primitive ke
upar kam nhi karta hai.

```
Arrays.sort(arr, si, ei, Collections.reverseOrder());
```


*→ counting sort:

*) This sorting will be beneficial to use for min. range.

ex:-

1, 3, 1, 3, 2, 4, 3, 7

max. no.
7
range = 7

make count array & frequency array
size of this array is as range+1.

0	2	1	3	1	0	0	1
0	1	2	3	4	5	6	7

now →

Sorting

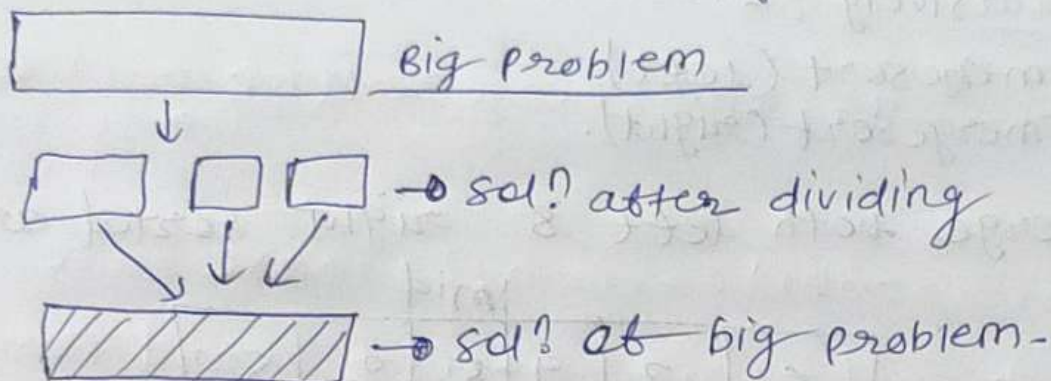
print the index no. and dec. element till 0.

1	1	2	3	3	3	4	7
---	---	---	---	---	---	---	---

pseudo
code

```
Public static void countingSort(int arr[]) {  
    int largest = Integer.MIN_VALUE;  
    for (int i = 0; i < arr.length; i++) {  
        largest = Math.max(largest, arr[i]);  
    }  
    int count[] = new int[largest + 1];  
    for (int i = 0; i < arr.length; i++) {  
        count[arr[i]]++;  
    }  
    // sorting  
    int j = 0;  
    for (int i = 0; i < count.length; i++) {  
        while (count[i] > 0) {  
            arr[j] = i;  
            j++;  
            count[i]--;  
        }  
    }  
}
```


⇒ Divide & conquer problem :



* merge sort :

unsorted

[6 | 3 | 9 | 5 | 2 | 8]

sorted

[2 | 3 | 5 | 6 | 8 | 9]

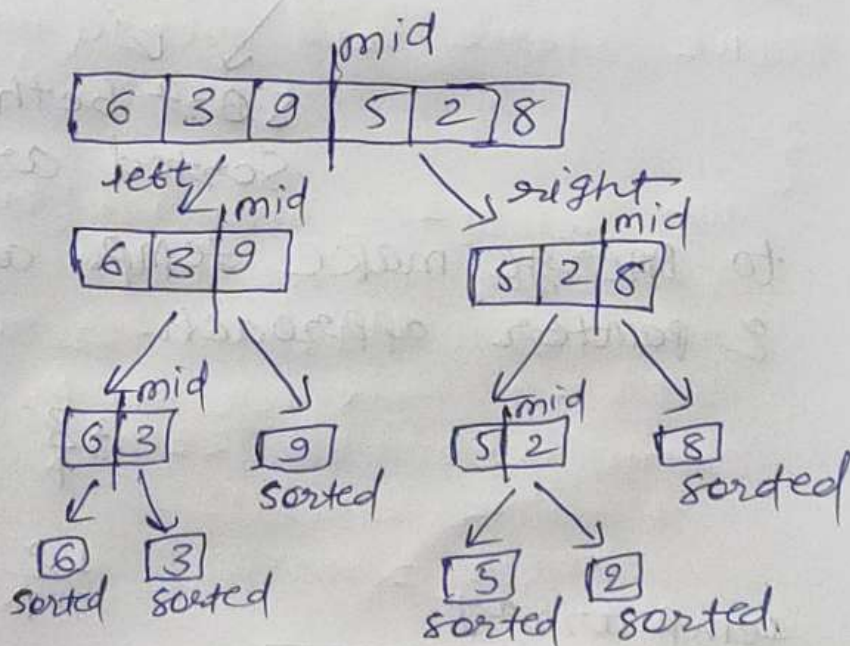
ascending order

- Approach :

① Divide

mid

$$\text{mid} = s + \frac{(e-s)}{2}$$

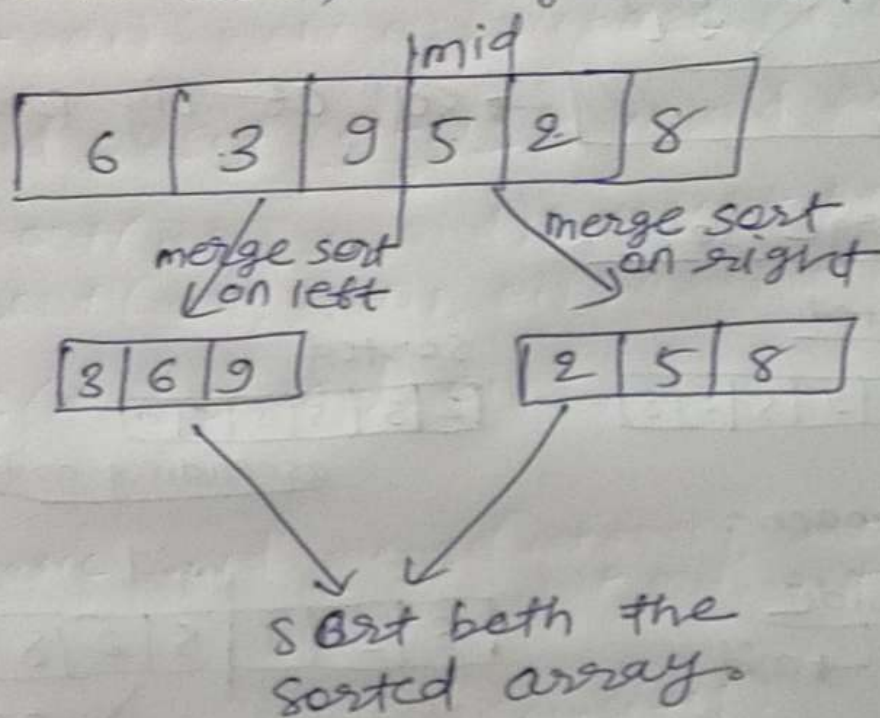


② Recursively :-

mergesort(left).

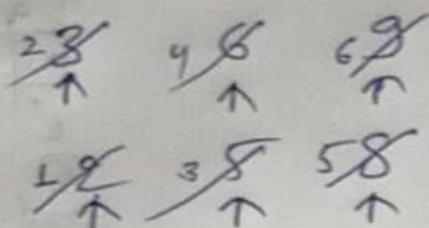
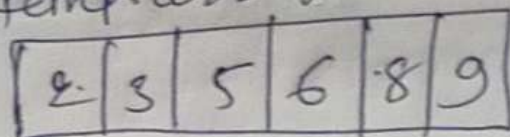
mergesort(right).

③ merge both left & right sorted array.

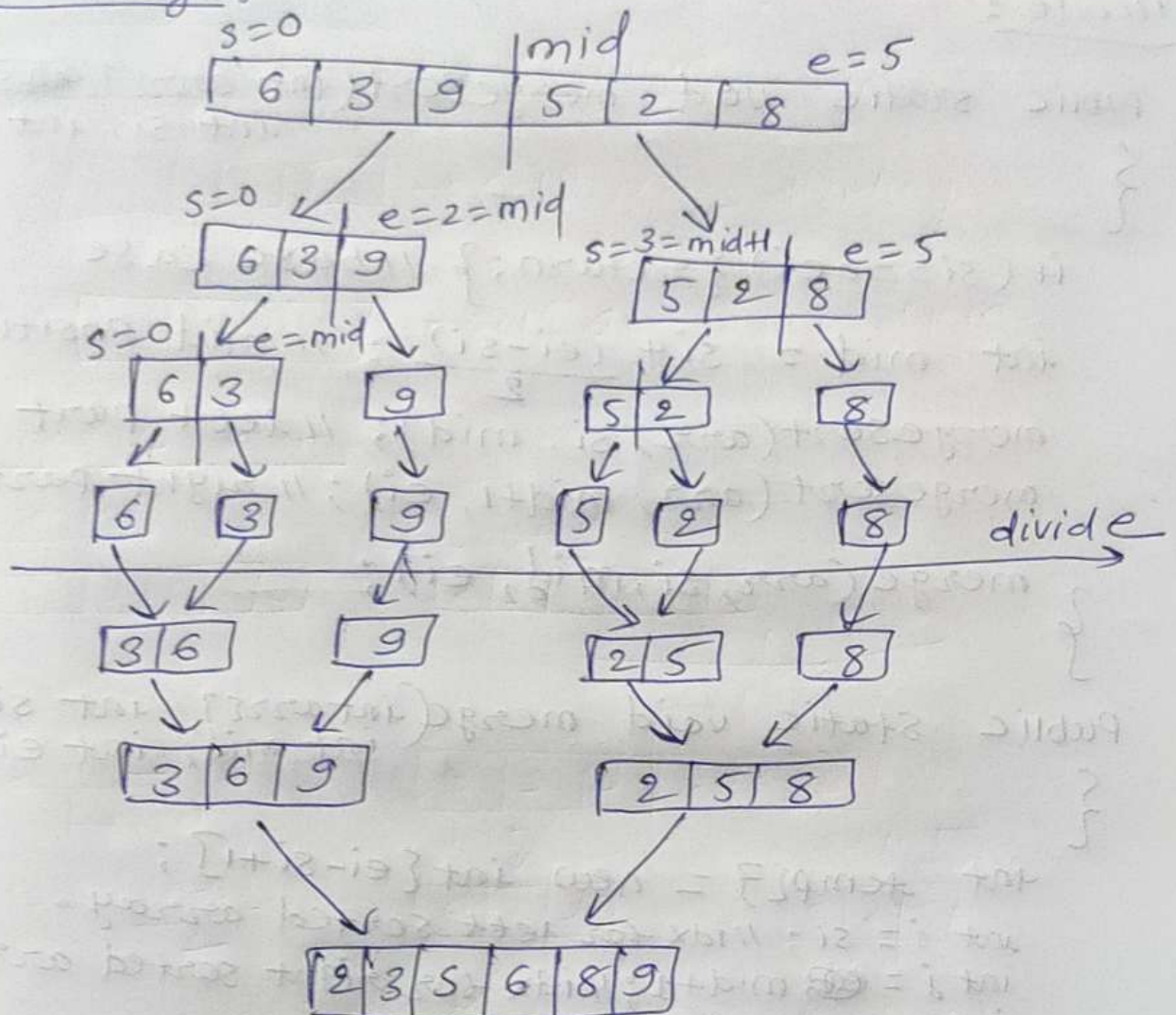


to merge make temp. array & use 2 pointer approach.

temp. array



summary:



Recursion:

① base case:

↳ ($s > e$ or $s == e$)

② kam:

{ divide
mergesort(left)
mergesort(right)
merge

//code:

```
public static void mergesort(int arr[], int
                                int si, int ei)
```

```
{
```

```
    if (si >= ei) { return; } // base case
```

```
    int mid = si +  $\frac{(ei - si)}{2}$ ; // mid position
```

```
    mergesort(arr, si, mid); // left part
```

```
    mergesort(arr, mid+1, ei); // right part
```

```
    merge(arr, si, mid, ei);
}
```

```
public static void merge(int arr[], int si,
                          int mid, int ei)
```

```
{
```

```
    int temp[] = new int[ei - si + 1];
```

```
    int i = si; // idx for left sorted array.
```

```
    int j = mid mid + 1; // idx for right sorted array.
```

```
    int k = 0; // idx for temp.
```

```
    while (i <= mid && j <= ei) {
```

```
        if (arr[i] < arr[j]) {
```

```
            temp[k] = arr[i];
```

```
            i++;
```

```
        } else {
```

```
            temp[k] = arr[j];
```

```
            j++;
```

```
        }
```

```
        k++;
```

```
    }
```


// for leftover elements of 1st sorted part
while($i \leq \text{mid}$) {

temp[k++] = arr[i++];
}

// for leftover elements of 2nd sorted part
while($j \leq \text{ei}$) {

temp[k++] = arr[j++];
}

// copy temp to original array:

for(int i = si, k = 0; k < temp.length; k++, i++) {

arr[i] = temp[k];
}

}

TC $\rightarrow n \log n$

Space $\rightarrow O(n)$

↑
because of temp array.

• depth first sorting

⇒ Quick sort :

↳ average → $O(n \log n)$

worst → $O(n^2)$

space → $O(1)$.

6	3	9	8	2	5
---	---	---	---	---	---

pivot & partition :

① pivot → element on which quick sort will sort the array.

→ let's take 8 as pivot element.

lesser element than 8 8 greater element than 8

2 3 5 6 8 9 → sort

→ random, medium, first, last

take last element 5 as pivot.

② partition : then partition :
↓
along parts.

3, 2,	< 5 <	6, 9, 8
	pivot	

↓
quick sort

↓
quick sort

③ quick sort (left)
quick sort (right)

↓ ↓
Base (single element)

Partition:

$si=0$ $ei=5$

6	3	9	8	2	5
---	---	---	---	---	---

pivot = 5

pivot index = $ei = 5$

$j = -1$ ← iterator for making places for elements which are lesser than pivots.

--	--	--	--	--	--

-1 0 1 2 3 4 5

(6) (3) (9) (8) (2) (5)

↑ ↑
6 > 5 3 < 5
 ↓ $j = -1$

↑ $i++$, and then swap it to first idx.

$j = 0$ // swap -

$int temp = arr[j]$

$arr[j] = arr[i]$

$arr[i] = temp;$

3	2	9	8	6	5
---	---	---	---	---	---

swap one more time,

3	2	5	8	6	9
---	---	---	---	---	---

//code:

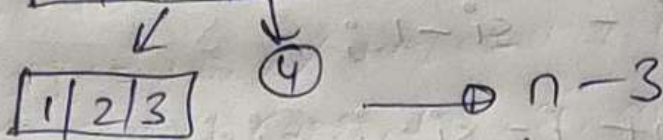
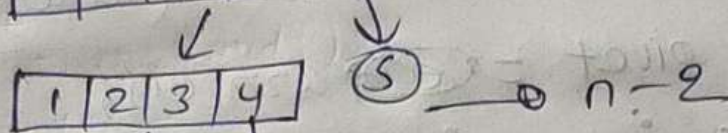
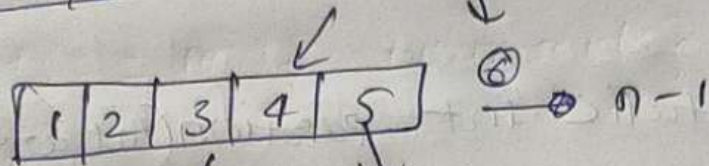
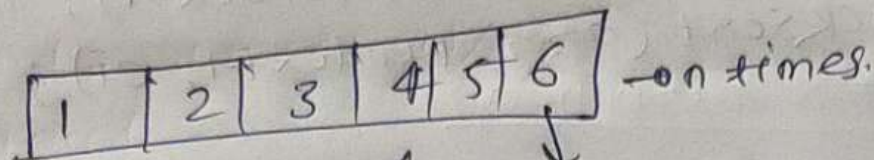
```
public static void quicksort(int arr[], int si, int ei)
{
    if (si >= ei) { return; }
    int pindex = partition(arr, si, ei);
    quicksort(arr, si, pindex-1); // left
    quicksort(arr, pindex+1, ei); // right
}
```

```
public static int partition(int arr[], int si, int ei) {
    int pivot = arr[ei] arr[si];
    int i = si-1;
    for (int j = si; j < ei; j++) {
        if (arr[j] <= pivot) {
            i++;
            // swap
            int temp = arr[j];
            arr[j] = arr[i];
            arr[i] = temp;
        }
    }
    i++;
    int temp = pivot;
    arr[ei] = arr[i];
    arr[i] = temp;
    return i; // pivot index
}
```


* worst case in quicksort:

1) worst case occurs when pivot is always the smallest or the largest element.

Ex: -



$$n + n-1 + n-2 + \dots$$

$$\Rightarrow \frac{n(n+1)}{2}$$

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

* Search in Rotated sorted array:

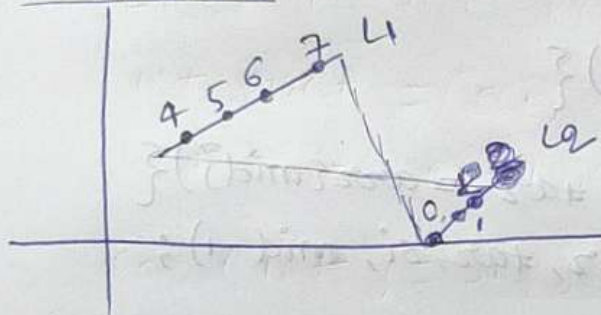
input: sorted, rotated array with distinct numbers (in ascending order) It is rotated at a pivot point. find the index of given element.

4	5	6	7	0	1	2
0	1	2	3	4	5	6

target: 0

Output = 4

1st Approach:



$$(si \leq target \leq mid)$$

left

Right (false)

$$(mid \leq target \leq ei)$$

right

left (false)

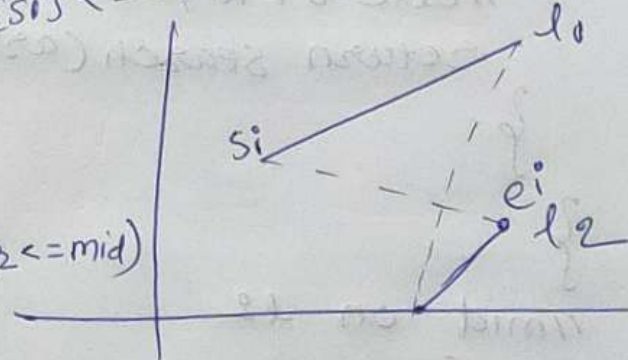
2nd Approach:

$$arr[si] \leq arr[mid]$$

Case-1 mid on l1

Case-a: l1 left ($si \leq tar \leq mid$)

Case-b: ~~l1~~ mid Right else



Case-2 mid on l2 $arr[mid] \leq arr[ei]$

Case-c: l1 Right ($mid \leq tar \leq ei$)

Case-d: mid left else

//code:

```
public static int search(int arr[], int tar, int si, int ei)
{
    if (si > ei) return -1; // base case
    // kam
    int mid = si + (ei - si) / 2;
    // case found
    if (arr[mid] == tar) { return mid; }

    // mid on left
    if (arr[si] <= arr[mid]) {
        // case a: left
        if (arr[si] <= tar && tar <= arr[mid]) {
            return search(arr, tar, si, mid - 1);
        } else {
            // case b: right
            return search(arr, tar, mid + 1, ei);
        }
    }

    // mid on right
    else {
        // case c: right
        if (arr[mid] <= tar && tar <= arr[ei]) {
            return search(arr, tar, mid + 1, ei);
        } else {
            // case d: left
            return search(arr, tar, si, mid - 1);
        }
    }
}
```