Searching & Sorting

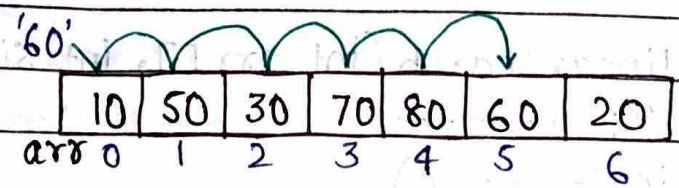
-By Nikhil Kumar

https://www.linkedin.com/in/nikhilkumar0609/

SEARCHING

* LINEAR SEARCH

• Find '60'



→ Program

```
bool search(int arr[], int size, int key){
    for(int i=0 ; i< size ; i++){
        if(arr[i] == key){
            return 1;
        }
    }
    return 0;
}
```

• Time Complexity = O(n)

• Approach :-

→ Start from the leftmost element of arr[] and one by one compare key with each element of arr[].

→ If key matches with an element, return true.

→ If key doesn't match with any of elements, return false.

- Linear Search through Recursion

```
bool linearSearch(int arr[], int size, int key) {
    // base case
    if (size == 0)
        return false;

    // Recursive Relation
    if (arr[0] == key) {
        return true;
    }
    else {
        bool remPart = linearSearch(arr+1, size-1, key);
        return remPart;
    }
}
```

* BINARY SEARCH

→ Used in Sorted array.
→ Time Complexity = O(log n).

• Program

```
int binarySearch(int arr[], int size, int key){
    int start = 0;
    int end = size-1;
    int mid = start + (end-start)/2;

    while (start <= end){
        if (arr[mid] == key){
            return mid;
        }
        // go to right wala part
        if (key > arr[mid]){
            start = mid+1;
        }
        // go to left wala part
        else { // key < arr[mid]
            end = mid-1;
        }
        // update mid
        mid = start + (end-start)/2;
    }
    return -1;
}
```

NOTE :- Hmlog "mid" aise find isliye Kr rhe hai kyuki agar mid = $\frac{start + end}{2}$

se Krnge toh sum max^m positive int value $(2^{31}-1)$ se greater bhi ho skta hai.

- Binary Search using Recursion

```
bool binarySearch (int arr[], int s, int e, int key){
    // base case → element not found
    if (s > e)
        return false;

    int mid = s + (e-s)/2;

    // element found
    if (arr[mid] == key)
        return true;

    if (arr[mid] < key) {
        return binarySearch (arr, mid+1, e, key);
    }
    else {
        return binarySearch (arr, s, mid-1, key);
    }
}
```

- Complexity Analysis

At each iteration, the array is divided by half.

At iteration 1, length of array = n

At iteration 2, length of array = $n/2$

At iteration 3, length of array = $n/2^2$

⋮

After k iterations, the length of array becomes1

Therefore, length of array $\Rightarrow \frac{n}{2^k} = 1$

$\Rightarrow n = 2^k$

Applying log on both sides.

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

$\Rightarrow \log_2(n) = k \log_2(2)$

$\Rightarrow \boxed{\log_2(n) = k}$

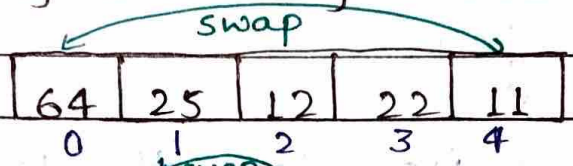
So, T.C = $O(\log n)$

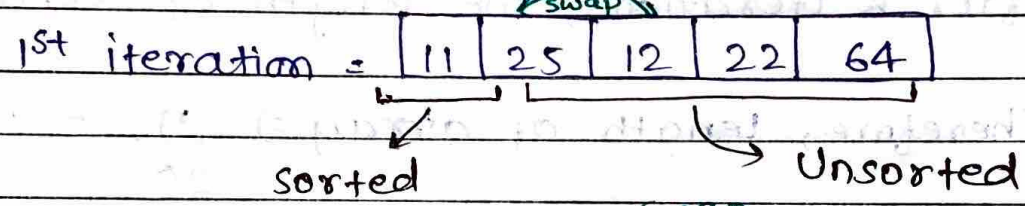
SORTING

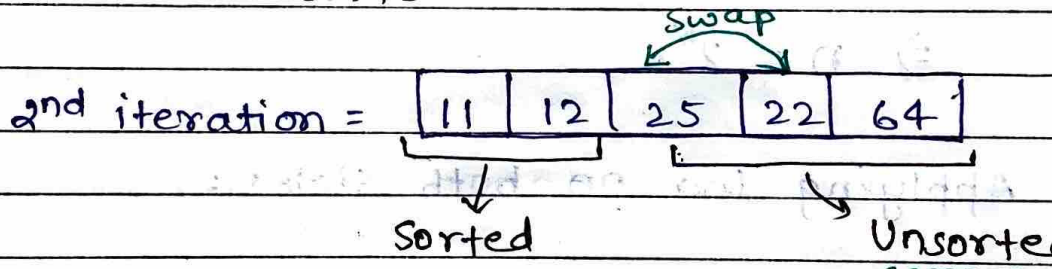
* SELECTION SORT

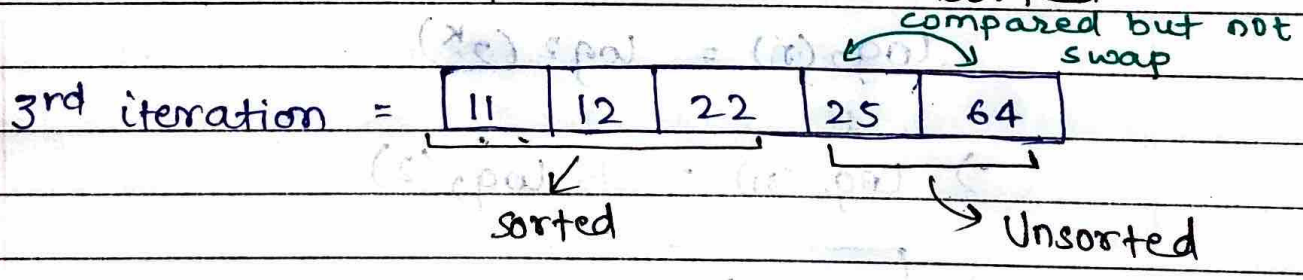
The Selection Sort algorithm sorts an array by repeatedly finding the minimum element from unsorted part and putting it at the beginning.

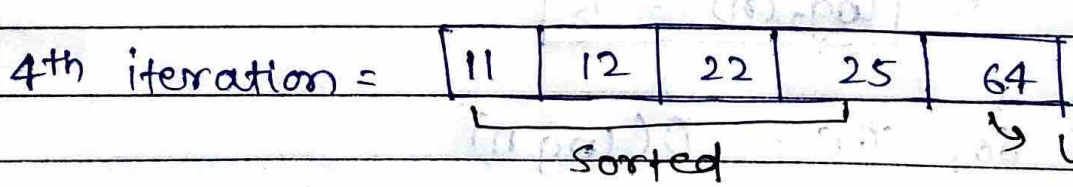
The algorithm maintains two subarrays :-
(i) The subarray which is already sorted.
(ii) Remaining subarray which is unsorted.











- Now, the first four elements are sorted, so, last element is automatically sorted.

- Program

```
void selectionSort (int arr[], int size){
    for (int i=0; i< size-1 ; i++){
        int minIndex = i;

        for (int j= i+1 ; j<n ; j++) {
            if (arr[j] < arr[minIndex])
                minIndex = j;
        }
        Swap (arr[minIndex], arr[i]);
    }
}
```

- Time Complexity = $O(n^2)$
- Auxiliary Space = $O(1)$

→ The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

- In Place = Yes, it doesn't require extra space.

→ Stable Algorithm

A Sorting algorithm is said to be Stable if two objects with equal or same keys appeared in the same order in sorted output as they appeas in the input array to be sorted.

- Selection sort works by finding the min^m element and then inserting it in its correct position by swapping with the element which is in position of this min^m element. This is what makes it Unstable.

for e.g. → i/p = | 4_A | 5 | 3 | 2 | 4_B | 1 |

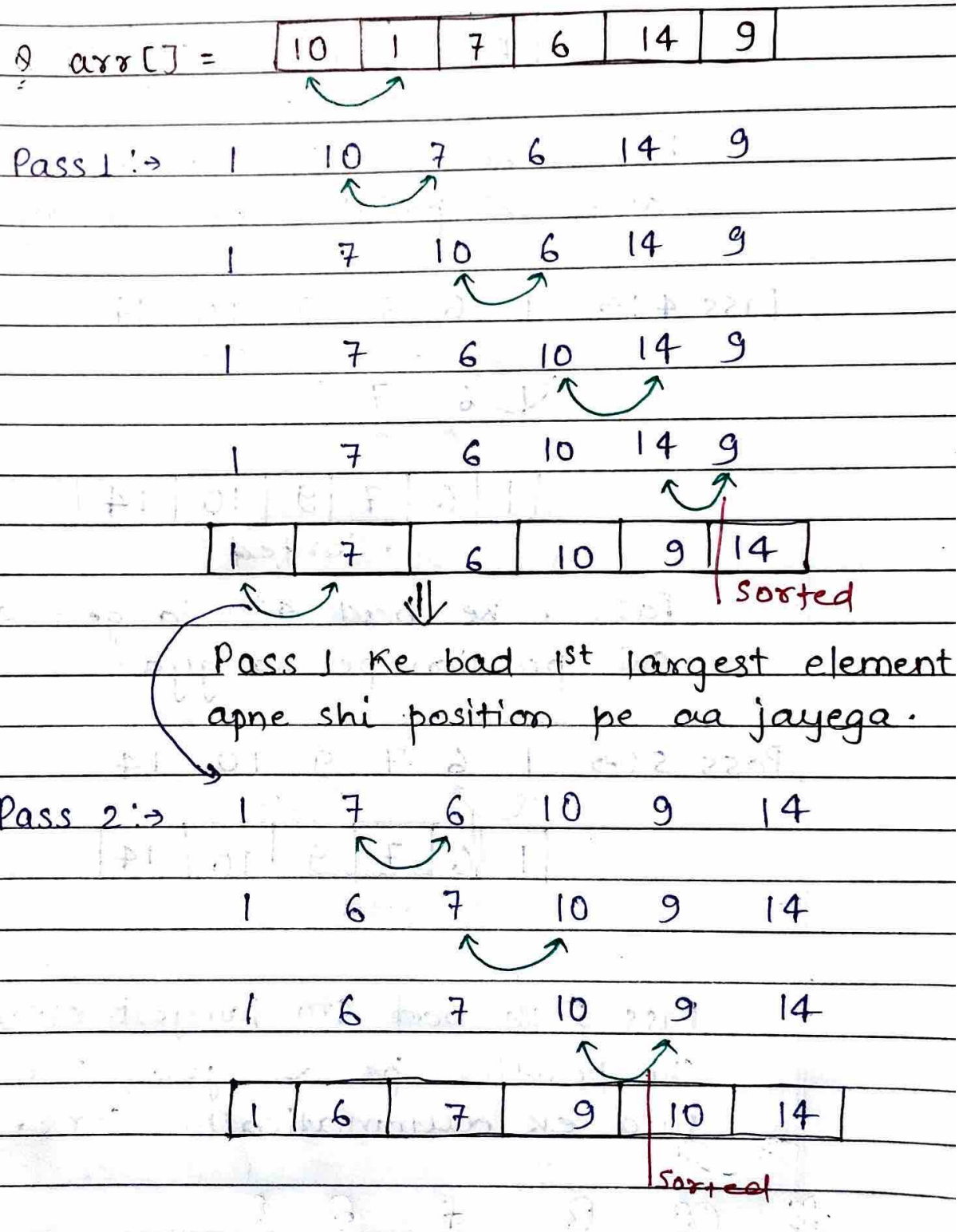
o/p = | 1 | 2 | 3 | 4_B | 4_A | 5 |

→ Selection Sort can be made stable if instead of swapping, the min^m element is placed in its position without swapping i.e: by placing the number in its position by pushing every element one step forward.

Stable Selection Sort o/p = | 1 | 2 | 3 | 4_A | 4_B | 5 |

* BUBBLE SORT

It is the simplest algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.



Pass 3:→ 1 6 7 9 | 10 14

Sorted

1 6 7 9

1 6 7 9

1	6	7	9	10	14

Sorted

Pass 3 ke bad 3rd largest element apne shi position pe aa gya.

Pass 4:→ 1 6 7 | 9 10 14

Sorted

1 6 7

1	6	7	9	10	14

Sorted

Pass 4 ke bad 4th largest apne shi position pe aa gya.

Pass 5:→ 1 6 | 7 9 10 14

Sorted

1	6	7	9	10	14

Sorted

Pass 5 ke bad 5th largest element shi position pe aa gya, toh bcha hua ek automatically sort.

- Program (Optimized Bubble Sort)

```
void bubbleSort (int arr[], int n) {
    for (int i = 1; i < n; i++) {
        // for pass 1 to n-1
        bool swapped = false;

        for (int j = 0; j < n-i; j++) {
            // process element till n-ith index
            if (arr[j] > arr[j+1]) {
                swap (arr[j], arr[j+1]);
                swapped = true;
            }
        }

        if (swapped == false) {
            // already sorted
            break;
        }
    }
}
```

- Time Complexity

Pass 1 → (n-1) Comparisons
Pass 2 → (n-2) "
Pass 3 → (n-3) "
⋮
Pass (n-1)th → 1 "

$$T.C = 1 + 2 + 3 + \ldots + (n-2) + (n-1)$$

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

where, n is size of array.

- Worst case :- $O(n^2)$, when array is reverse sorted.
- Best case :- $O(n)$, when array is already sorted.
- Stable = Yes
- Sorting in Place = Yes

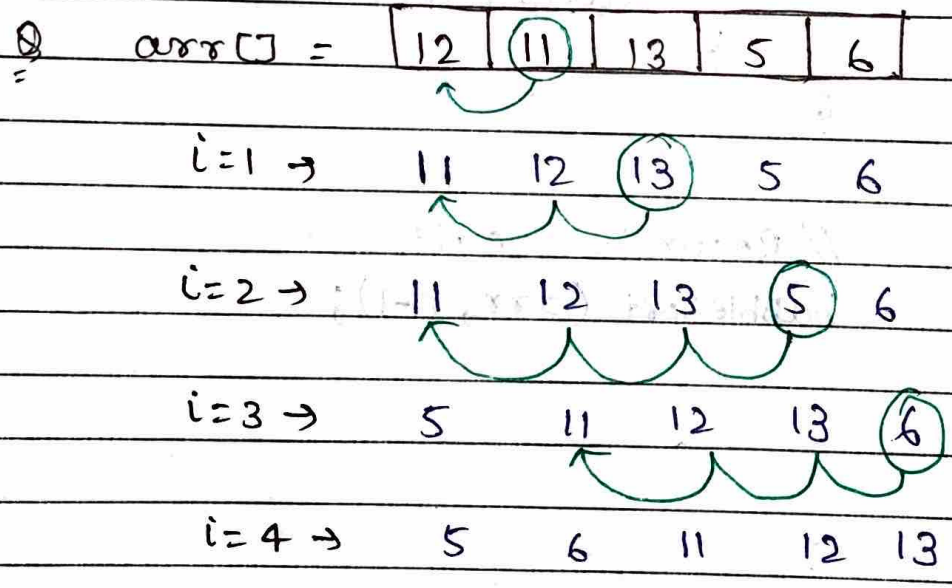
- Recursive Program

```
void bubbleSort(int arr[], int n) {
    // base case → already sorted
    if (n==0 || n==1)
        return;

    // 1 case solve krlo → largest element
       ko end mein rkh dega
    for(int i=0; i<n-1; i++){
        if (arr[i] > arr[i+1]){
            swap(arr[i], arr[i+1]);
        }
    }

    // Recursive call
    bubbleSort(arr, n-1);
}
```

* INSERTION SORT

It is a simple algorithm that work's similar to the way you sort playing cards in your hands.

→ The array is virtually split into a sorted and unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.



Program :-

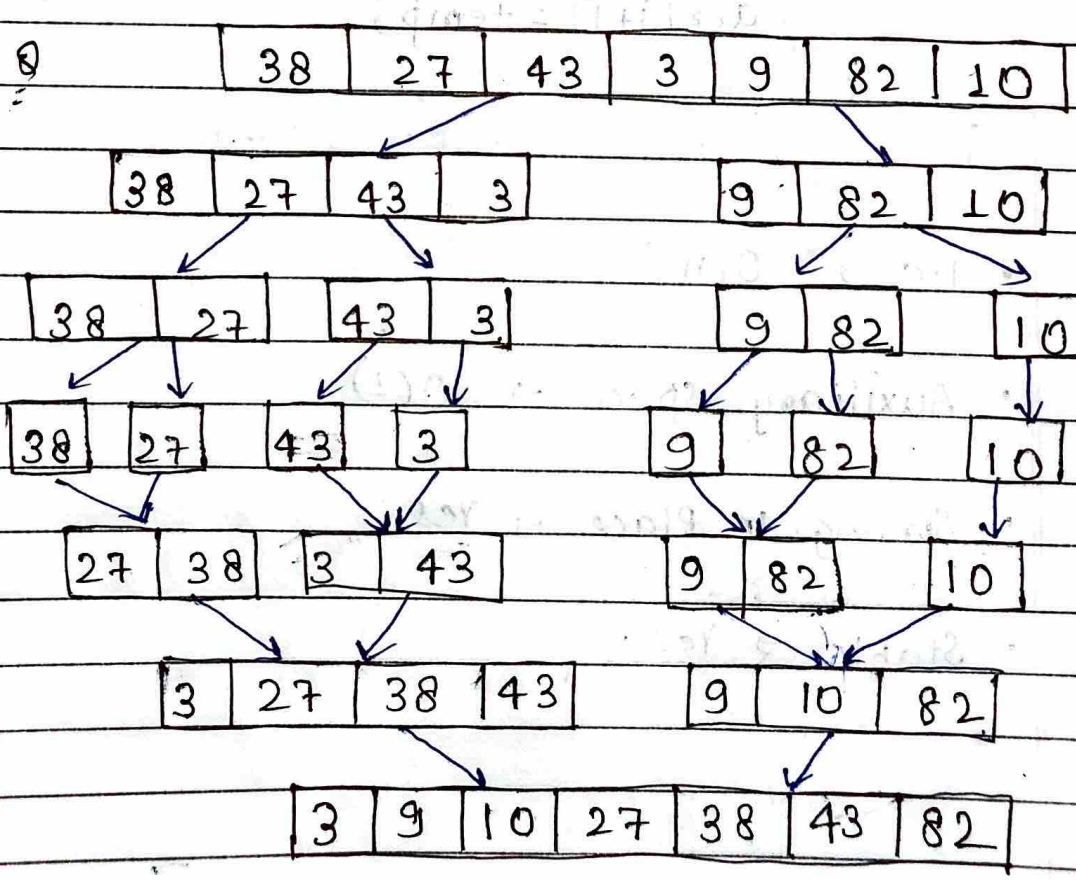
```
Void insertionSort (int arr [], int n){
    for (int i=1 ; i<n ; i++) {
        int temp = arr[i];
        int j = i-1 ;
        for (; j>=0 ; j--) {
            If (arr [j] > temp) {
                //shift
                arr [j+1] = arr [j];
            }
            else { //ruk jao
                break;
            }
        }
        // copy temp value
        arr [j+1] = temp;
    }
}
```

- T.C → $O(n^2)$
- Auxiliary space → $O(1)$
- Sorting In Place → Yes
- Stable → Yes.

* MERGE SORT

It is a sorting algorithm that uses the divide, conquer, and combine algorithm paradigm.

- Divide means partitioning the n-element array to be sorted into two sub-array of $n/2$ elements.
- Conquer means sorting the two sub-arrays recursively using merge sort.
- Combine means merging the two sorted sub-arrays of size $n/2$ to produce the sorted array of n elements.



- Program :-

```
void merge (int arr[], int s, int e){
    int mid = s+ (e-s)/2;
    int len1 = mid - s + 1;
    int len2 = e - mid;

    int *first = new int [len1];
    int * second = new int [len 2];

    // Copy values
    int mainArrayIndex = s;
    for (int i=0 ; i< len1 ; i++) {
        first[i] = arr [mainArrayIndex ++];
    }

    mainArrayIndex = mid+1;
    for (int i=0 ; i< len2; i++) {
        second [i] = arr [mainArrayIndex ++];
    }

    // merge 2 sorted arrays

    int index1 = 0;
    int index 2 = 0;
    mainArrayIndex = s;
```

```
    while (index1 < len1 && index2 < len2) {
        if (first[index1] < second[index2]) {
            arr[mainArrayIndex++] = first[index1++];
        }
        else {
            arr[mainArrayIndex++] = second[index2++];
        }
    }
    while (index1 < len1) {
        arr[mainArrayIndex++] = first[index1++];
    }
    while (index2 < len2) {
        arr[mainArrayIndex++] = second[index2++];
    }
    delete [] first;
    delete [] second;
}

void mergeSort(int arr[], int s, int e) {
    // base case
    if (s >= e)
        return;
    int mid = s + (e-s)/2
    // left part sort krna h
    mergeSort(arr, s, mid);
    // right part sort krna h
    mergeSort(arr, mid+1, e);
    // merge
    merge(arr, s, e);
}
```

- T.C = O(nlogn) → In all cases (worst, average, best).

Auxiliary space = O(n)

Sorting in Place = No in a typical implementation

Stable = Yes

* QUICK SORT

It is also a Divide and Conquer algorithm

→ It picks an element as pivot and Partitions the given array around the picked Pivot.

→ Pivot can be first element, last element, random element or median.

Q arr[] =

3	5	1	8	2	4
0	1	2	3	4	5

Pivot Count = 2 (smaller than Pivot)

Now, Put Pivot at its correct position in sorted array and Put all smaller elements (smaller than Pivot) before Pivot, and all greater than Pivot after Pivot.

0	1	2	3	4	5
1	2	3	8	5	4

less than Pivot Greater than Pivot

Pivot Index = Start + Count ⇒ 0+2 = 2

Now, 1<3 and 4>3

Then, 5>3 and 2<4 ⇒ Swap.

- Program :-

```
int partition (int arr[], int s, int e) {
    int pivot = arr[s];
    int count = 0;

    for (int i=0;i+1;
    for (int i = s+1 ; i <= e ; i++) {
        if (arr[i] <= pivot) {
            count ++;
        }
    }
    //Place Pivot at right position.
    int pivotIndex = s+count;
    swap (arr[pivotIndex], arr[s]);

    // left and right wala part smbhal lete hai
    int i=s, j=e ;
    while (i < pivotIndex && j > pivotIndex) {
        while (arr[i] <= pivot) {
            i++ ;
        }
        while (arr[j] > pivot) {
            j--
        }
        if (i< pivotIndex && j > pivotIndex) {
            swap (arr[i++], arr[j--]);
        }
    }
    return pivotIndex;
}
```

```
void quickSort (int arr[], int s, int e){

    //base case
    if (s >= e)
        return ;

    // Partition Krenge
    int p = partition (arr, s, e);
    // left part Sort Kro
    quickSort (arr, s, p-1);
    // right part Sort Kro
    quickSort (arr, p+1, e);
}
```