Sorting Algorithm

① Bubble Sort

Bubble sort is the simplest sorting algorithm that works by repeatedly swaping the adjecent elements if they are in wrong order.

7	11	9	2	17	4
0	1	2	3	4	5

1st pass

(n-1) Comp. | 7 | 11 | 9 | 2 | 17 | 4 | → 0, 1 → i=0

5 Comp. & 5 possible swap

7 9 11 2 17 4 → 1, 2 → i=1
7 9 2 11 17 4 → 2, 3 → i=2
7 9 2 11 17 4 → 3, 4 → i=3
7 9 2 11 4 17 → 4, 5 → i=4

Sorted

2nd pass

	0	1	2	3	4	5
4 Comp. 4 possible swap.	7	9	2	11	4	17
	7	9	2	11	4	17 → 0, 1
	7	2	9	11	4	17 → 1, 2
	7	2	9	11	4	17 → 2, 3
	7	2	9	4	11	17 → 3, 4

3rd pass

0	1	2	3	4	5	
7	2	9	4	11	17	
2	7	9	4	11	17	→ 0, 1
2	7	9	4	11	17	→ 1, 2
2	7	4	9	11	17	→ 2, 3

3 comparison & 3 possible swap.

4th pass

0	1	2	3	4	5	
2	7	4	9	11	17	
2	7	4	9	11	17	→ 0, 1
2	4	7	9	11	17	→ 1, 2

2 comparison & 2 possible swap.

5th pass

0	1	2	3	4	5	
2	4	7	9	11	17	
2	4	7	9	11	17	→ 0, 1

1 comparison & 1 possible swap.

⇓ sorted array.

no. of passes = len. of array − 1.

Total no. of comparision

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

$= O(n^2)$

checking for Stability

7, 8, 2, 7*

7, 8, 2, 7*
7, 2, 8, 7*
7, 2, 7*, 8
2, 7, 7*, 8

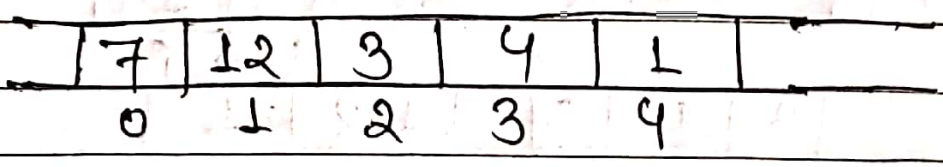
order is same as the order in input array. so, Bubble sort is stable sorting algorithm.

Bubble sort is not a Adaptive in nature But we can make Bubble sort as Adaptive. (Having Benefits of sorted array).

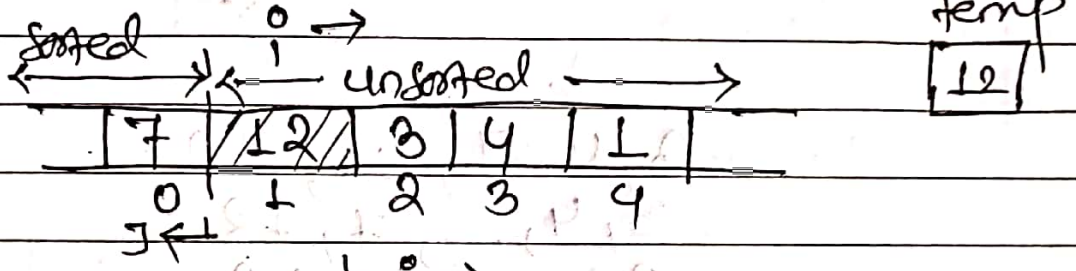
// program for Bubble Sort

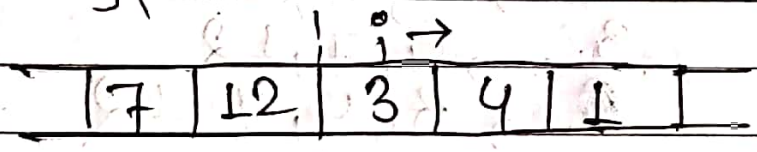
```
void Bubble_sort(int Arr[], int size){
    for(int j=0; j<size-1; j++){
        for(int i=0; i<size-j-1; i++){
            if(Arr[i] > Arr[i+1]){
                swap(Arr[i], Arr[i+1]);
            }
        }
    }
}
```

(11) Insertion Sort

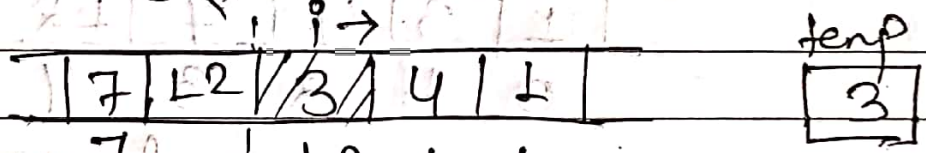


1st pass

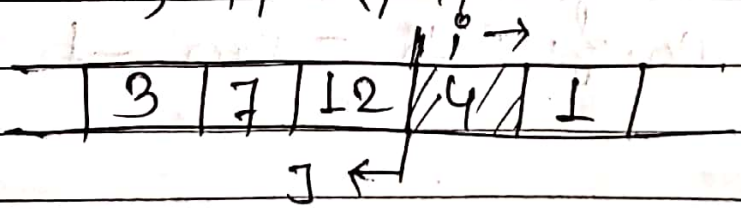




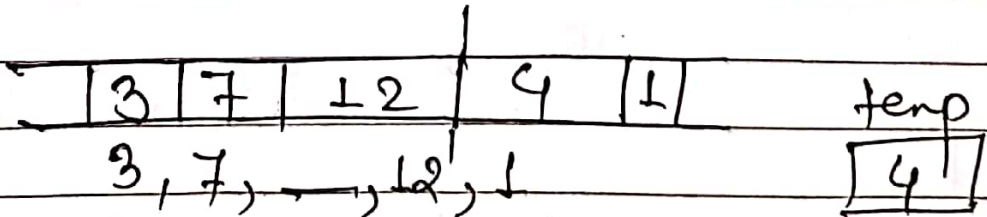
2nd pass



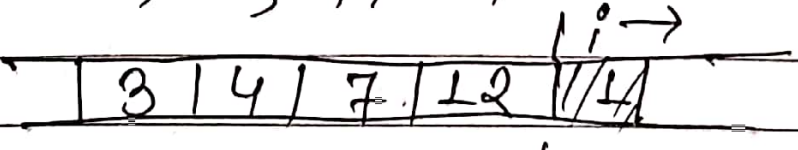
7, —, 12, 4, 1
—, 7, 12, 4, 1
3, 7, 12, 4, 1



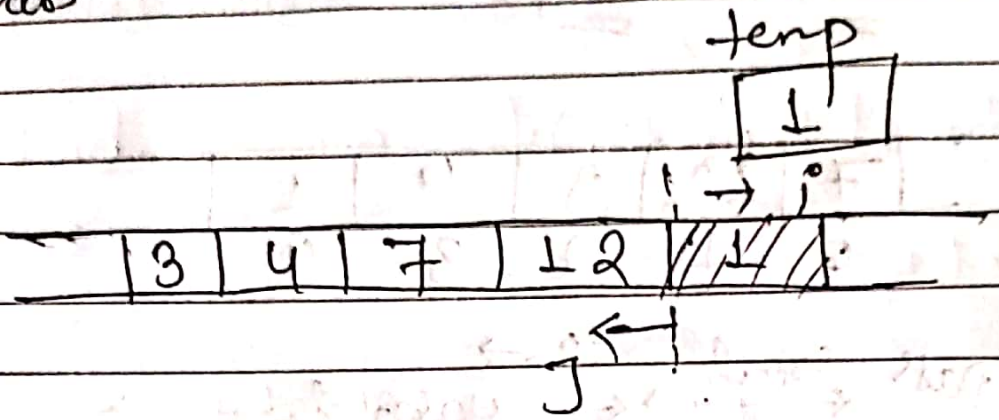
3rd pass



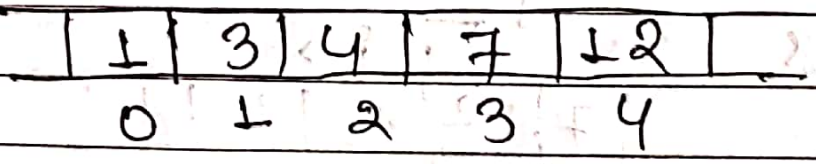
3, 7, —, 12, 1
3, —, 7, 12, 1



4th pass



3, 4, 7, —, 12
3, 4, —, 7, 12
3, —, 4, 7, 12
—, 3, 4, 7, 12



⇒ sorted array.

no. of pass = len − 1 = n − 1.

Total number of comparision/swap

$= 1+2+3+4+\dots+(n-1)$

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

$= O(n^2)$

for Best case

Total comparision $= 1+1+1+1 = 4$

$= n-1$

$= O(n)$

Analysis

(i) Total pass $= n-1$

total comparision $= O(n^2)$

(ii) Best case $\rightarrow O(n)$ (when array is sorted)

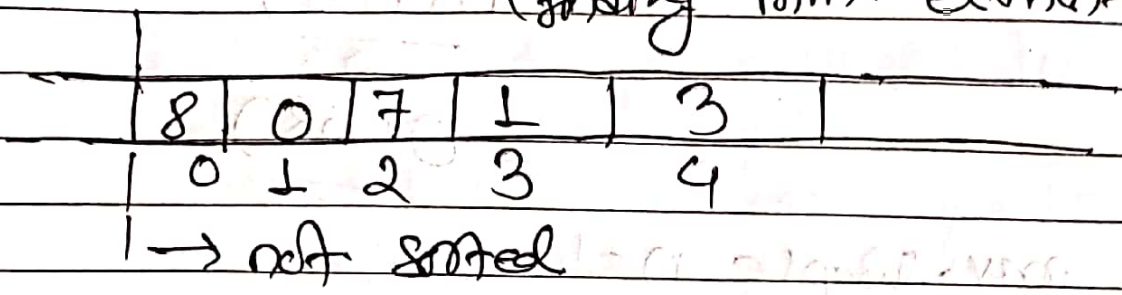
(iii) Stable $\rightarrow$ yes

(iv) Adaptive $\rightarrow$ yes

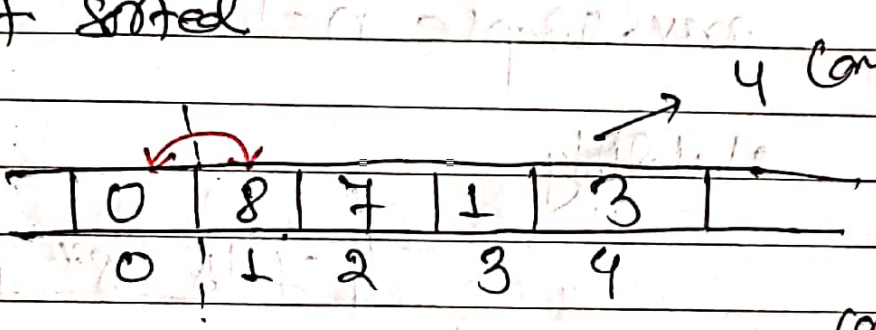
// program for Insertion sort

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

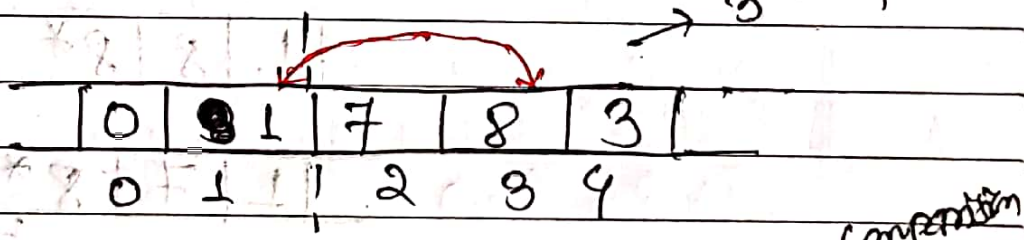
Selection Sort
(finding min. element).



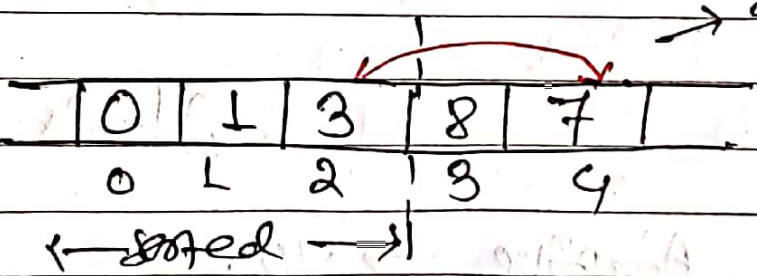
1st pass :-



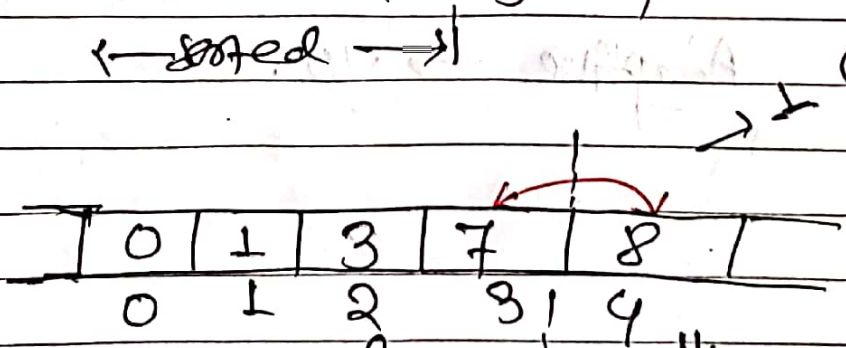
2nd pass :-



3rd pass :-



4th pass :-



only one element left, so no need to do 5th pass.

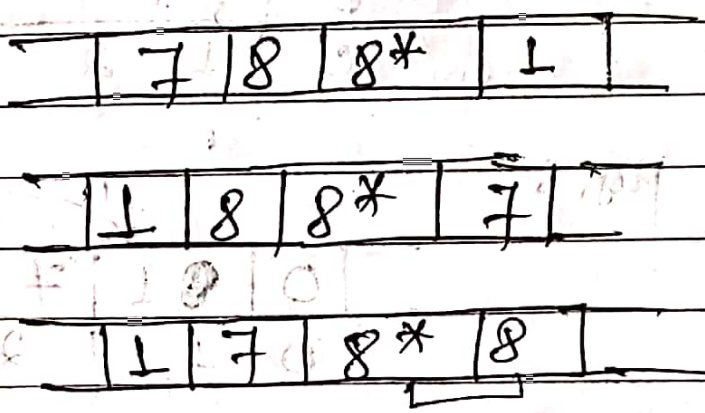
Total Comparision = $1+2+3+\cdots+(n-1)$

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

$= O(n^2)$

max-swaps = $n-1$

Stability



Not sorted.

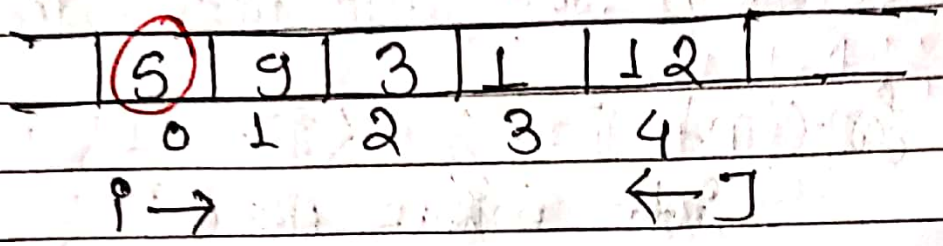
So, it is not a stable algorithm.

Adaptive → NO.

// program for selection sort

```
void selection_sort(int arr[], int size){
    for(int i=0; i<size-1; i++){
        int min = arr[i];
        for(int j=1; j<size; j++){
            if(min > arr[j]){
                min = arr[j];
            }
        swap(arr[i], min);
        }
    }
}
```

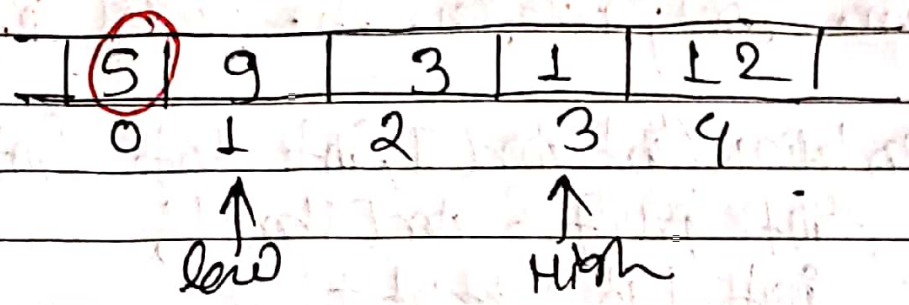
Quick Sort



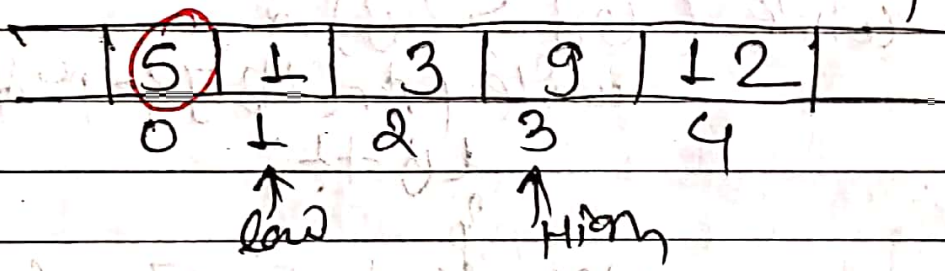
partition 1	Pivot	partition 2
values < pivot		values >= pivot

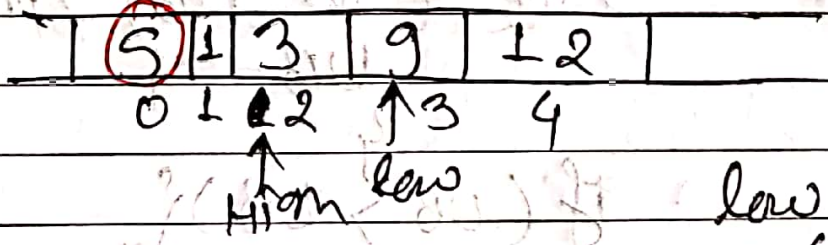
Algo:- partitioning

(i) i = low
(ii) j = High
(iii) pivot = low
(iv) i++ until element > pivot is found.
(v) j-- until element < pivot is found.
(vi) swap A[i] & A[j] & repeat (iv) & (v). until j ≤ i.
(vii) swap pivot and A[j].



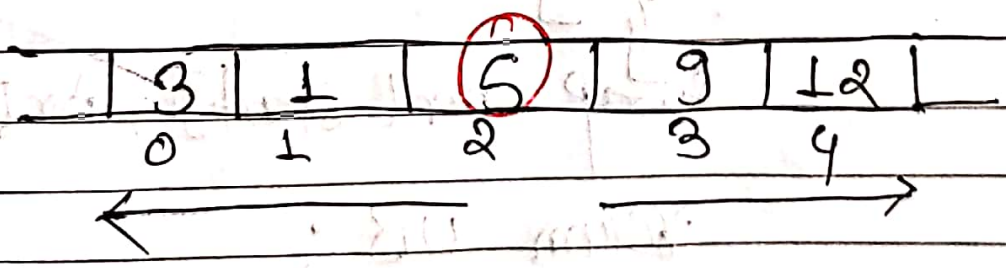
high > low . so swap (high, low)





low > High

so, swap (pivot, High).



// Program for partition in quick sort.

```
int partition (int Arr[], int start, int end){
    int pivot = Arr[start];
    int LB = start+1;
    int UB = end;
  while (UB > LB){
      while (Arr[LB] <= pivot){
          LB++;
      }
      while (Arr[UB] > pivot){
          UB--;
      }
      if (UB > LB){
        swap (Arr[LB], Arr[UB]);
      }
  }
  swap (Arr[start], Arr[UB]);

  return UB;
}
```

// program for quick sort.

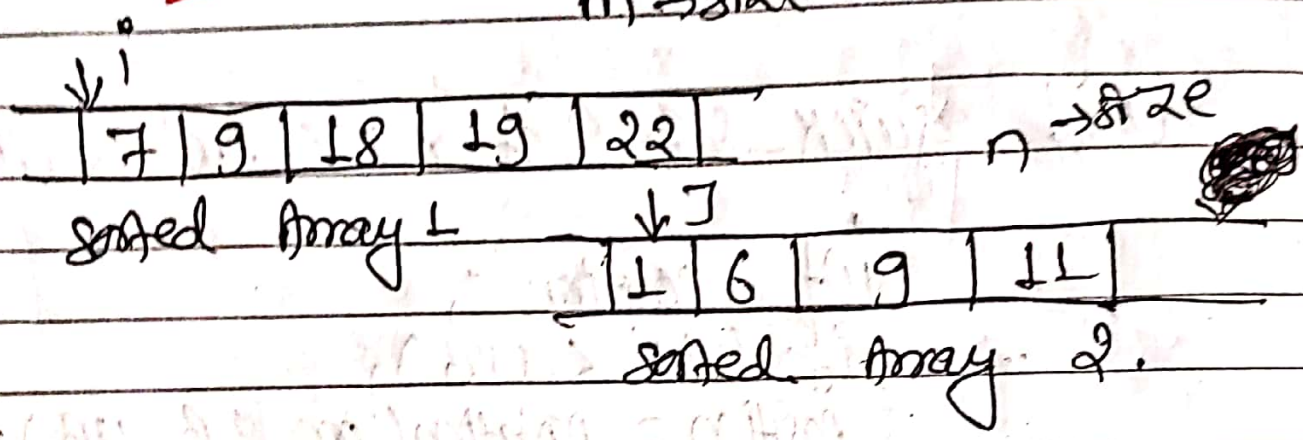
```
void quick_sort (int arr[], int LB, int UB){
    int position;
    if (LB < UB){
    position = partition(arr, LB, UB);

    quick_sort(arr, LB, position-1);
    quick_sort(arr, position+1, UB);
    }
}
```

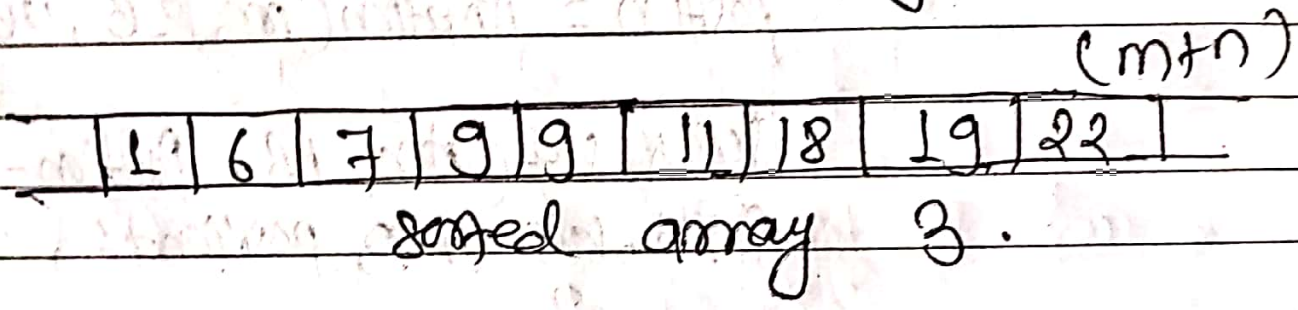
Analysis

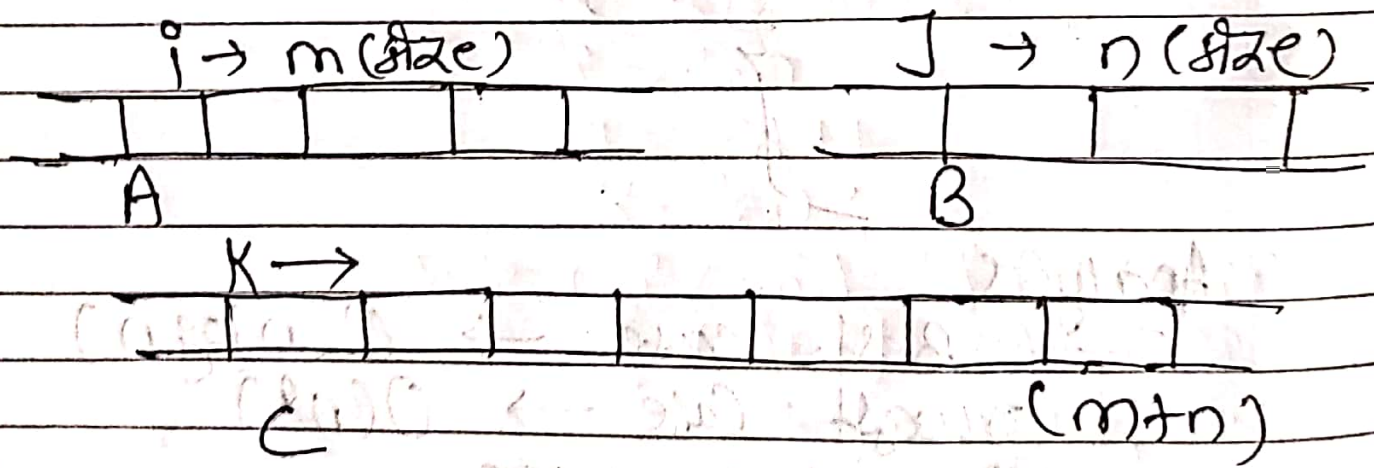
(i) Best case → O(n log n)
(ii) worst case → O(n²)
(iii) Stable → NO

Merge Sort

$i = j = k = 0$

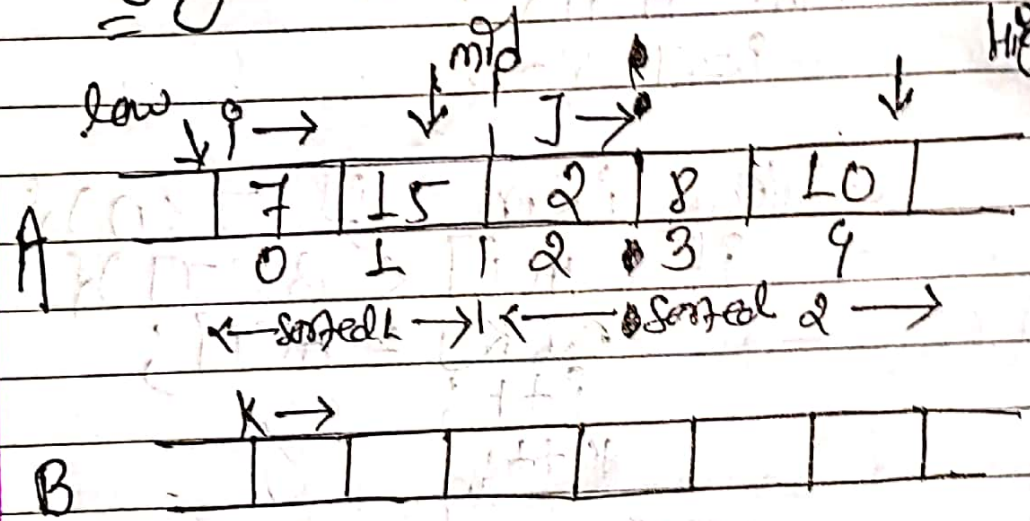






```
void merge(int A[], int B[], int C[], m, n){
    int i, j, k;
    i = j = k = 0;

    while(i<m && j<n){
        if(A[i] < B[j]){
            C[k] = A[i];
            i++;
            k++;}
        else{
            C[k] = B[j];
            j++;
            k++;
        }
    while(i<m){          // Copy all remaining element from A to C.
        C[k] = A[i];
        i++;
        k++;
    }
    while(j<n){          // Copy all remaining element from B to C.
        C[k] = B[j];
        j++;
        k++;
    }
}
```

Merging in a single Array




```
void merge (int A[], int mid, int low, int high){
    int i, J, K;
    int B[high+1];
    i = low
    J = mid+1
    K = low;
while (i<=mid && J<=high){
    if (A[i] < A[J]){
        B[K] = A[i];
        i++;
        K++;
    }
else {
    B[K] = A[J];
    J++;
    K++;
}
}
```

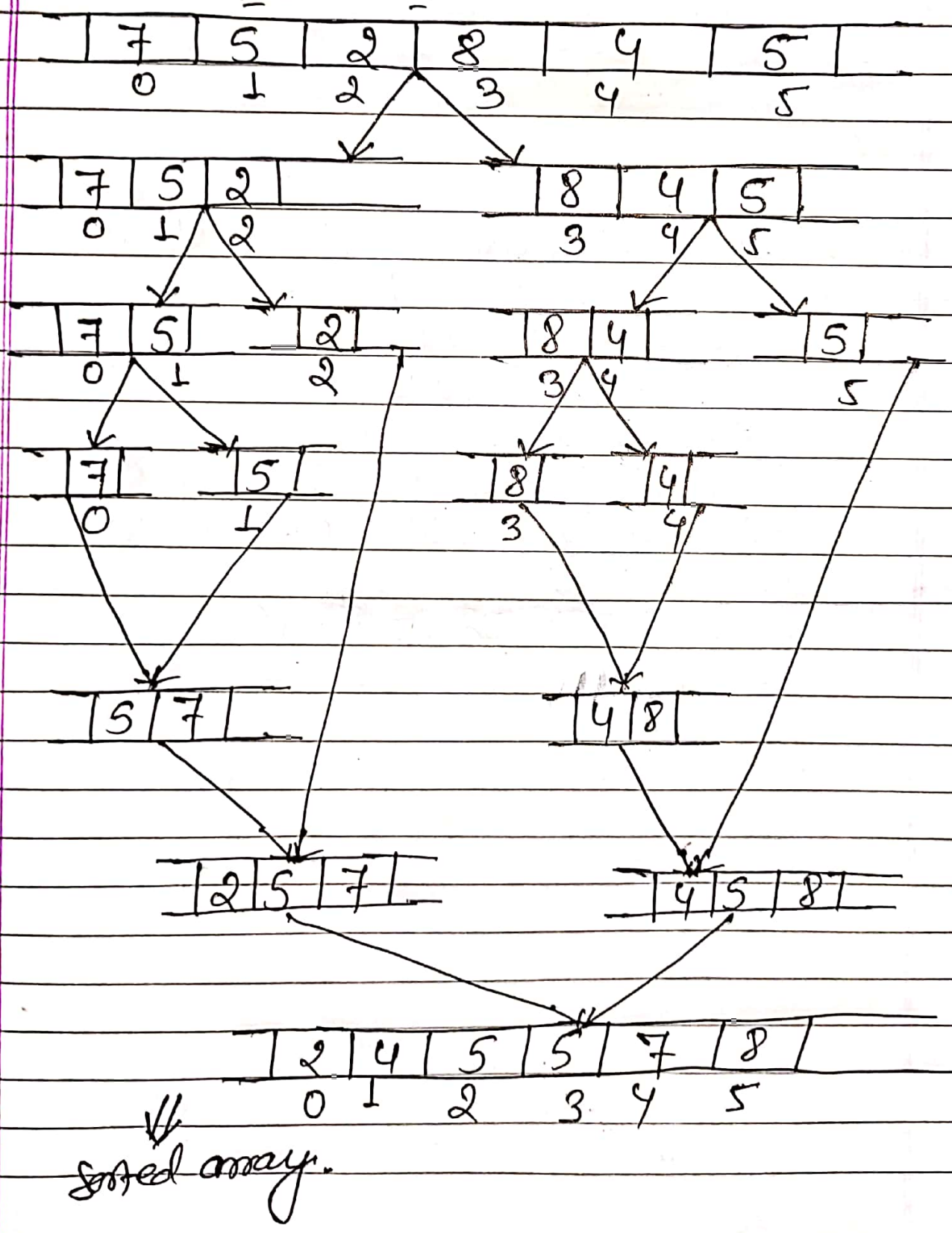
```
while (i<=mid){          // Copy all remaining
    B[k] = A[i];         elements from A
    k++;                 to C.
    i++;
}

while (j<=high){         // Copy all remaining
    B[k] = A[j];         elements from
    k++;                 B to C.
    j++;
}
// Copy Arr B to A.
for (i=low; i<=high; i++){
    A[i] = B[i];
}
}
```

// merge sort program.

```
void merge_sort (int A[], int low, int high){
    if (low < high){
        mid = (low + high) / 2;
        merge_sort (A, low, mid);
        merge_sort (A, mid+1, high);
        merge (A, mid, low, high);
    }
}
```

O(n log n)
//

merge sort



Sorted array.