COURSE NAME: ALGORITHMS
COURSE CODE: CIS 212

SYED TANGIM PASHA

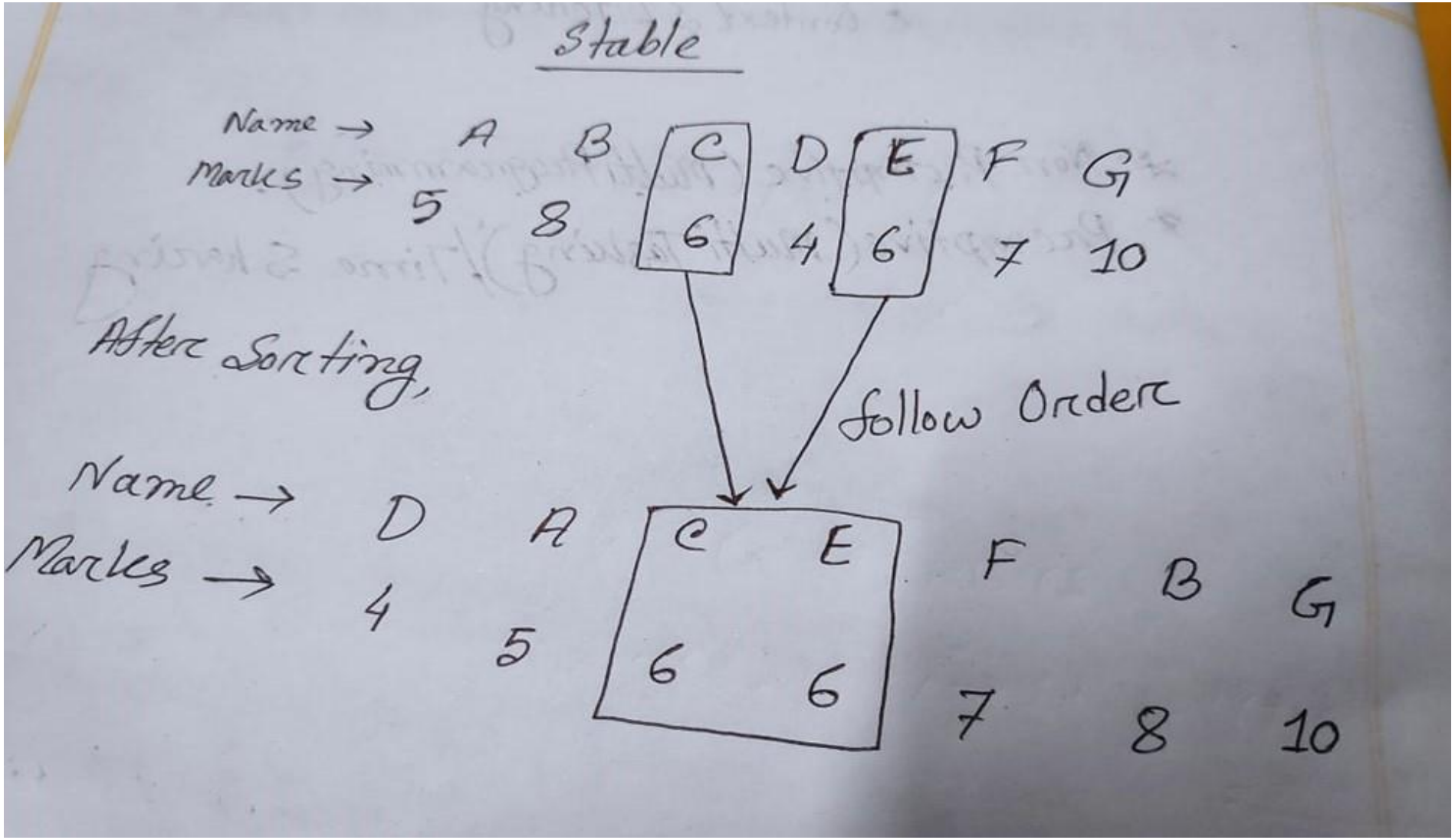
LECTURER,

DEPARTMENT OF COMPUTING AND INFORMATION SYSTEM (CIS)

DAFFODIL INTERNATIONAL UNIVERSITY (DIU)

DHAKA,BANGLADESH

SORTING


Stable
Name →
Marks →
A B C D E F G
5 8 6 4 6 7 10
After Sorting,
follow Order
Name →
Marks →
D A C E F B G
4 5 6 6 7 8 10

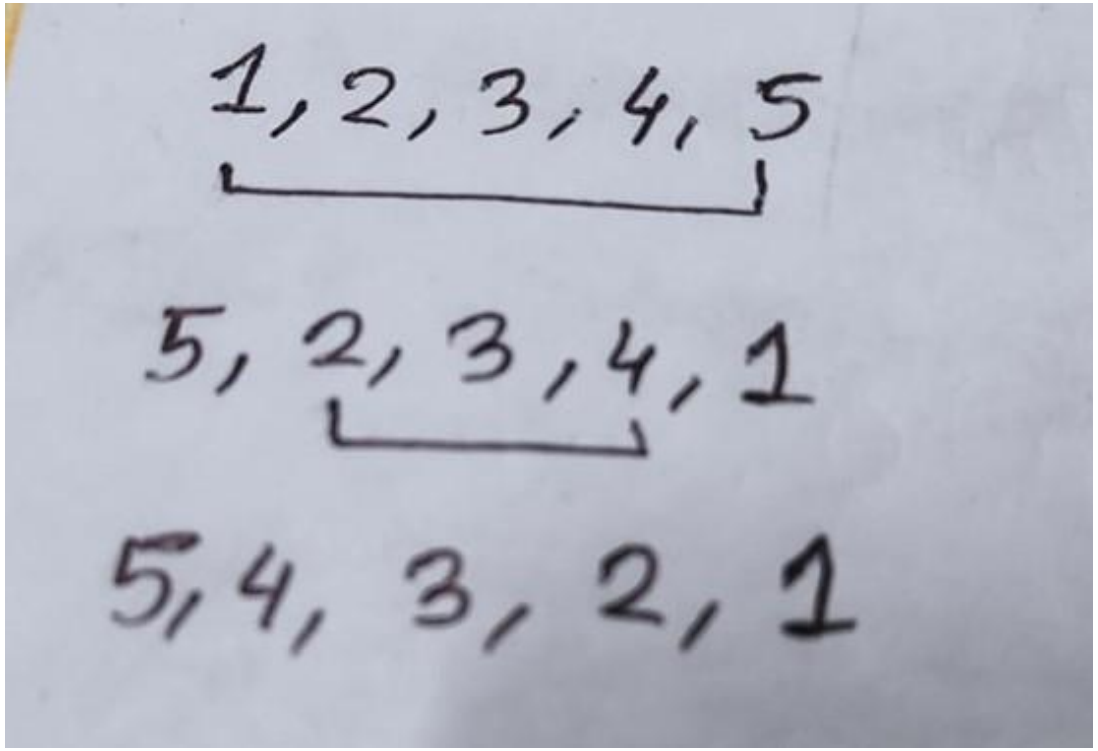
SORTING

ADAPTIVE

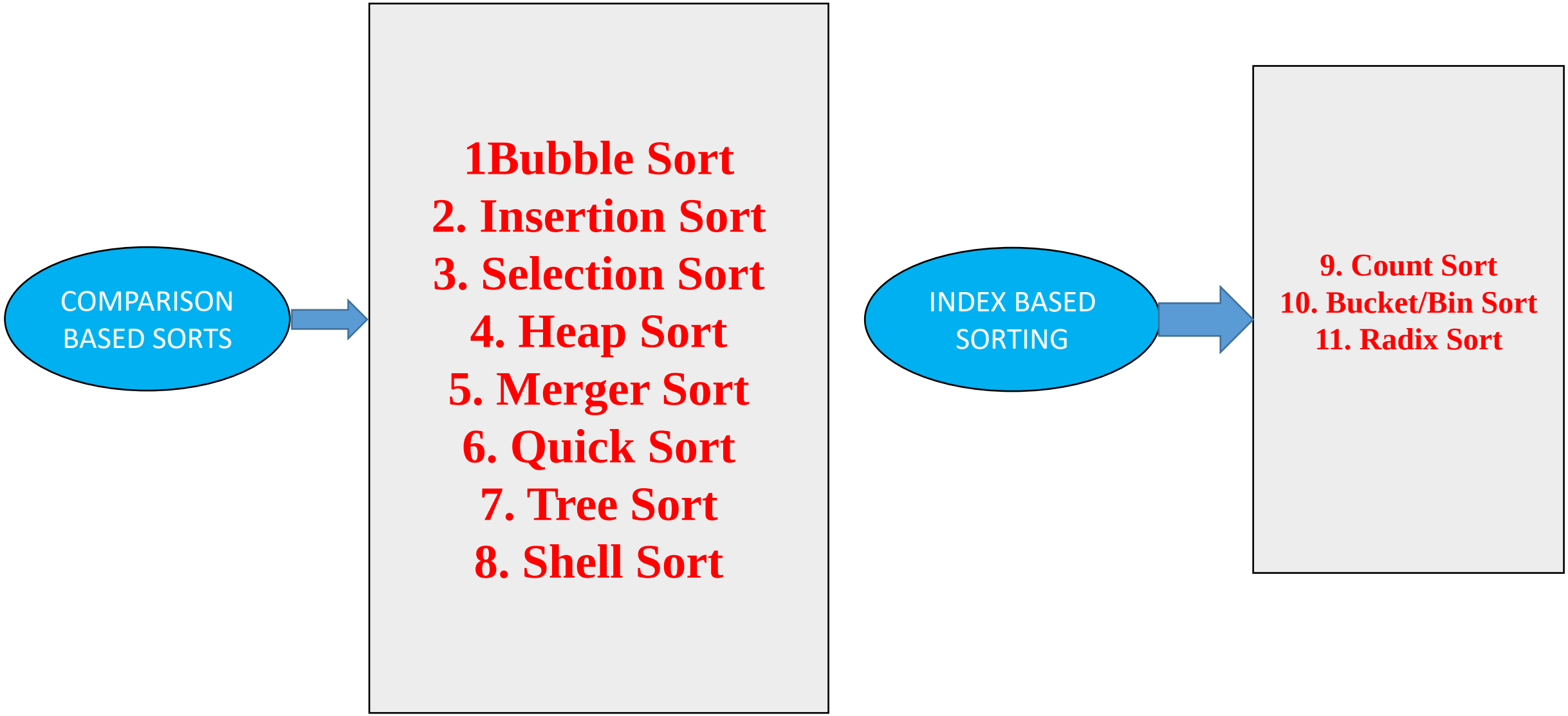
- A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.
- A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sorted order.

SORTING

IN PLACE



SORTING


COMPARISON BASED SORTS
1Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Heap Sort
5. Merger Sort
6. Quick Sort
7. Tree Sort
8. Shell Sort
INDEX BASED SORTING
9. Count Sort
10. Bucket/Bin Sort
11. Radix Sort

SORTING

1.BUBBLE SORT
2. SELECTION SORT
3. INSERTION SORT

O(n^2)

1. HEAP SORT
2. MERGE SORT
3. QUICK SORT
4. TREE SORT

O(nlogn)

SHELL SORT

O(n^3/2)

1. COUNT SORT
2. BUCKET/BIN SORT
3. RADIX SORT

O(n)

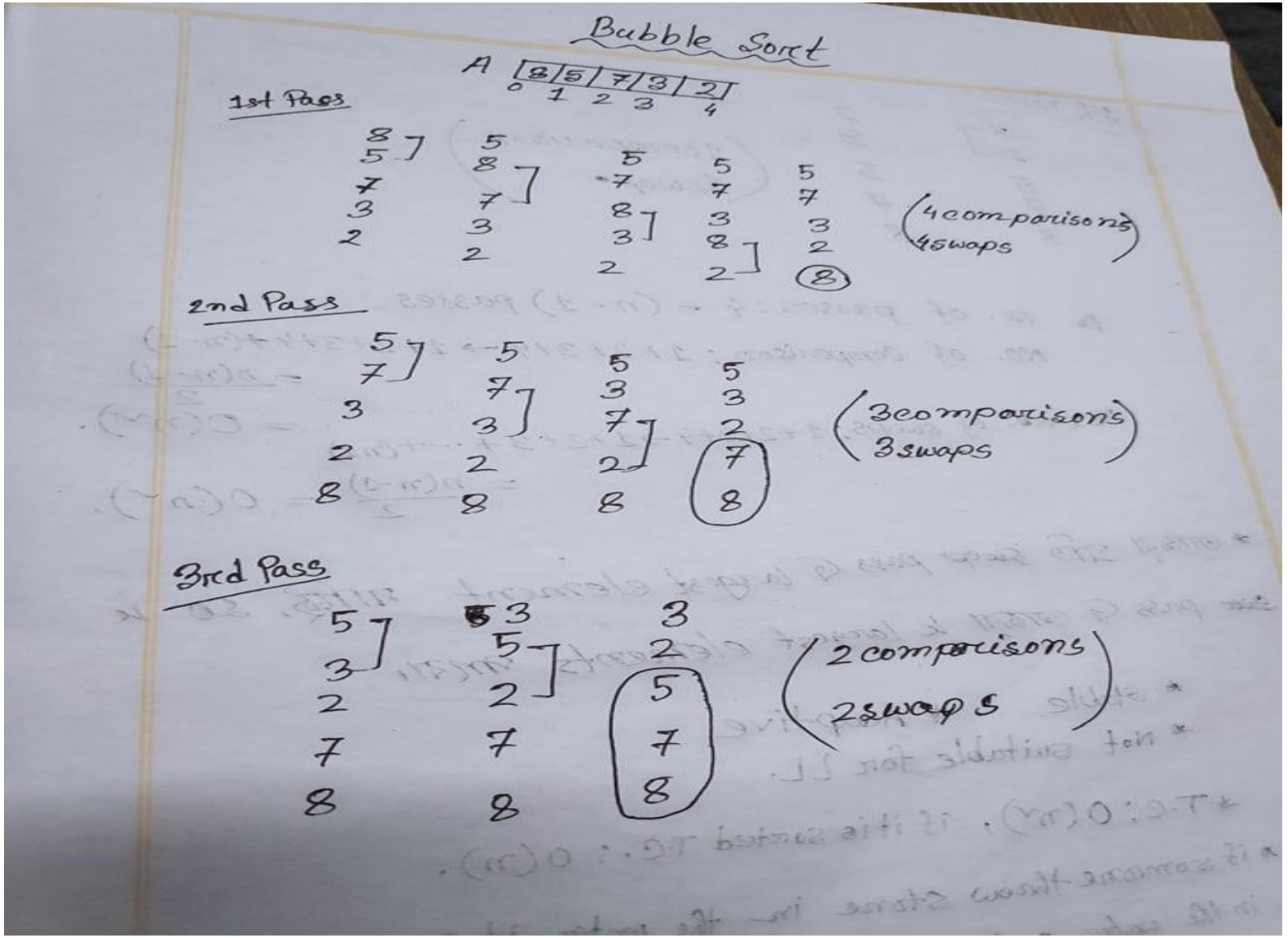
BUBBLE SORT

ALGORITHM

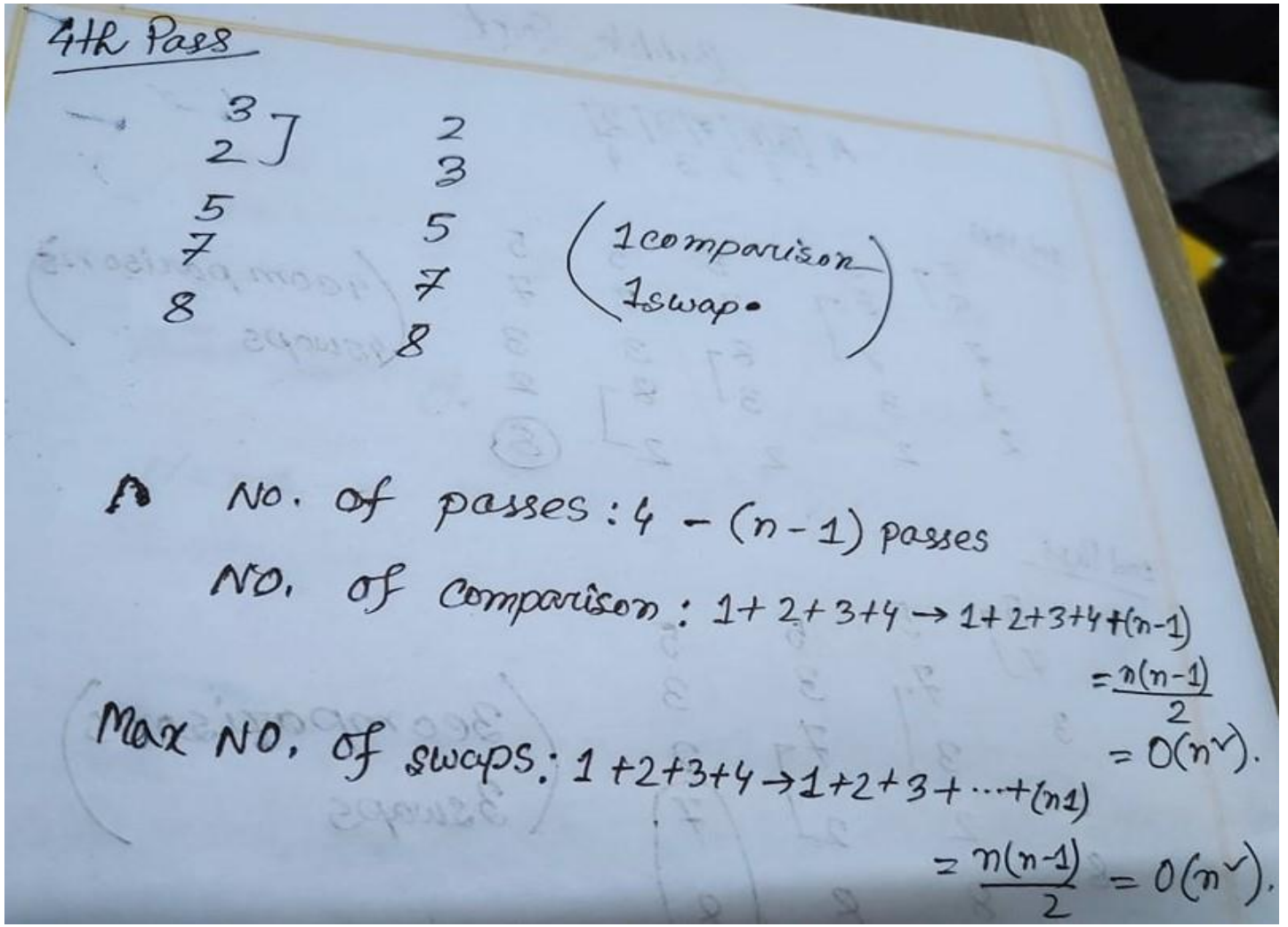
- For all elements of array

```
        if array[i] > array[i+1]
            swap(array[i], array[i+1])
        end if
    end for
```

BUBBLE SORT


Bubble Sort
A
8 5 7 3 2
0 1 2 3 4
1st Pass
4 comparisons
4 swaps
2nd Pass
3 comparisons
3 swaps
3rd Pass
2 comparisons
2 swaps

BUBBLE SORT


4th Pass
3 2 5 7 8
2 3 5 7 8
(1 comparison
1 swap.)
No. of passes : 4 - (n-1) passes
No. of comparison : 1+2+3+4 → 1+2+3+4+(n-1)
= n(n-1)/2
= O(n²).
Max NO. of swaps: 1+2+3+4 → 1+2+3+...+(n1)
= n(n-1)/2 = O(n²).

BUBBLE SORT

- Adaptive
- In Place
- Stable
- Worst Case T.C. : O(n^2)
- Best Case, If it is sorted, T.C. : O(n)
- S.C. : O(1)

INSERTION SORT

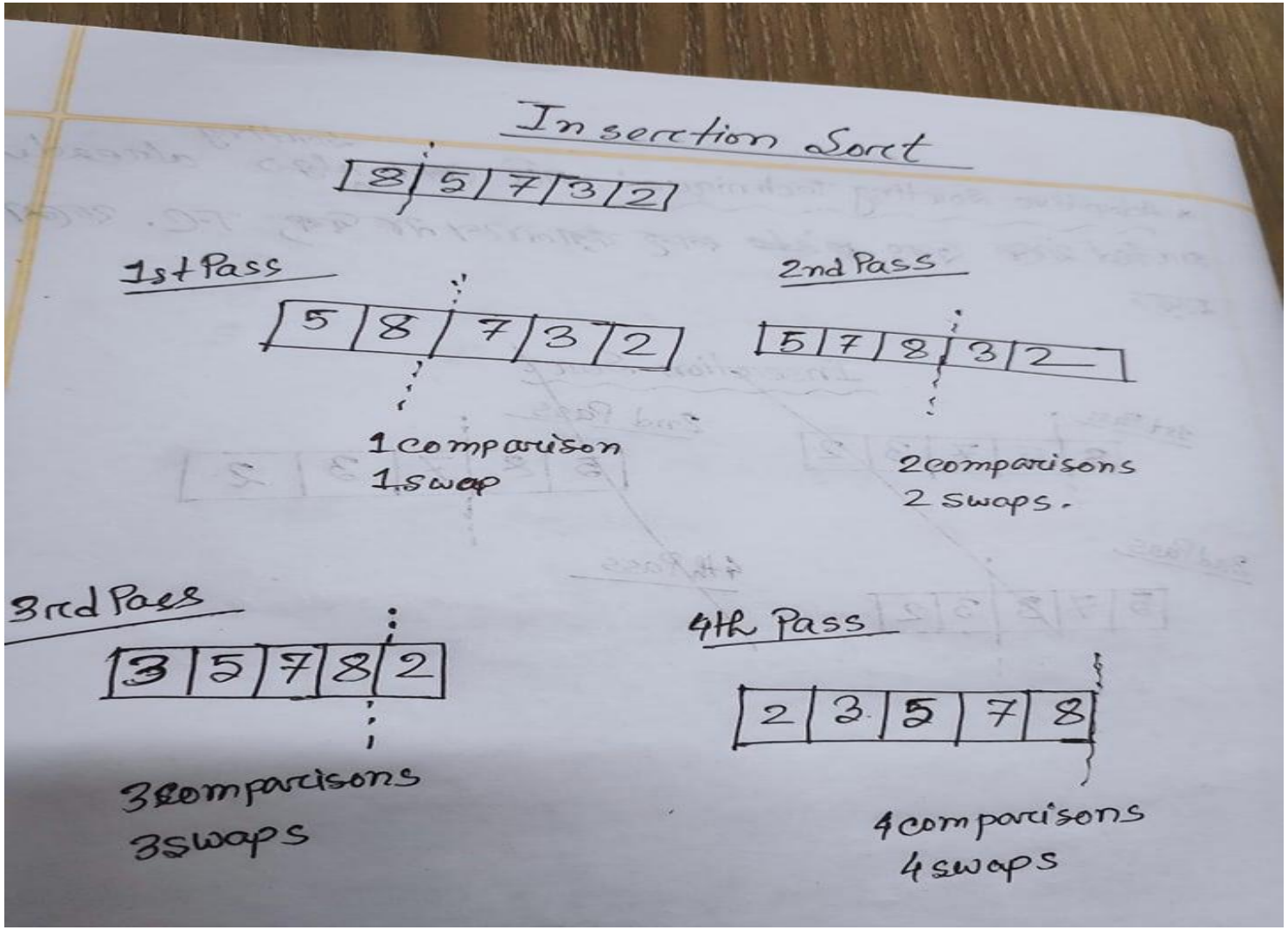
ALGORITHM

1: Iterate from arr[1] to arr[n] over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

INSERTION SORT



INSERTION SORT

- No. of Pass: (n-1)
- Adaptive
- In Place
- Stable
- Worst Case T.C. : O(n^2)
- Best Case T.C. : O(n)
- S.C. : O(1)

SELECTION SORT

- Step-1: Set MIN to location 0
- Step-2: Search the minimum element in the list
- Step-3: Swap with value at location MIN
- Step-4: Increment MIN to point to next element
- Step-5: Repeat until list is sorted

SELECTION SORT

Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

SELECTION SORT

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process -

SELECTION SORT

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process –

SELECTION SORT

- Not Adaptive
- In Place
- Not Stable
- Worst Case T.C. : O(n^2)
- Best Case T.C. : O(n^2)
- S.C. : O(1)

MERGE SORT

1. Find the middle point to divide the array into two halves:

   middle m = (left+right)/2

2. Call mergeSort for first half:

   Call mergeSort (array, left, middle)

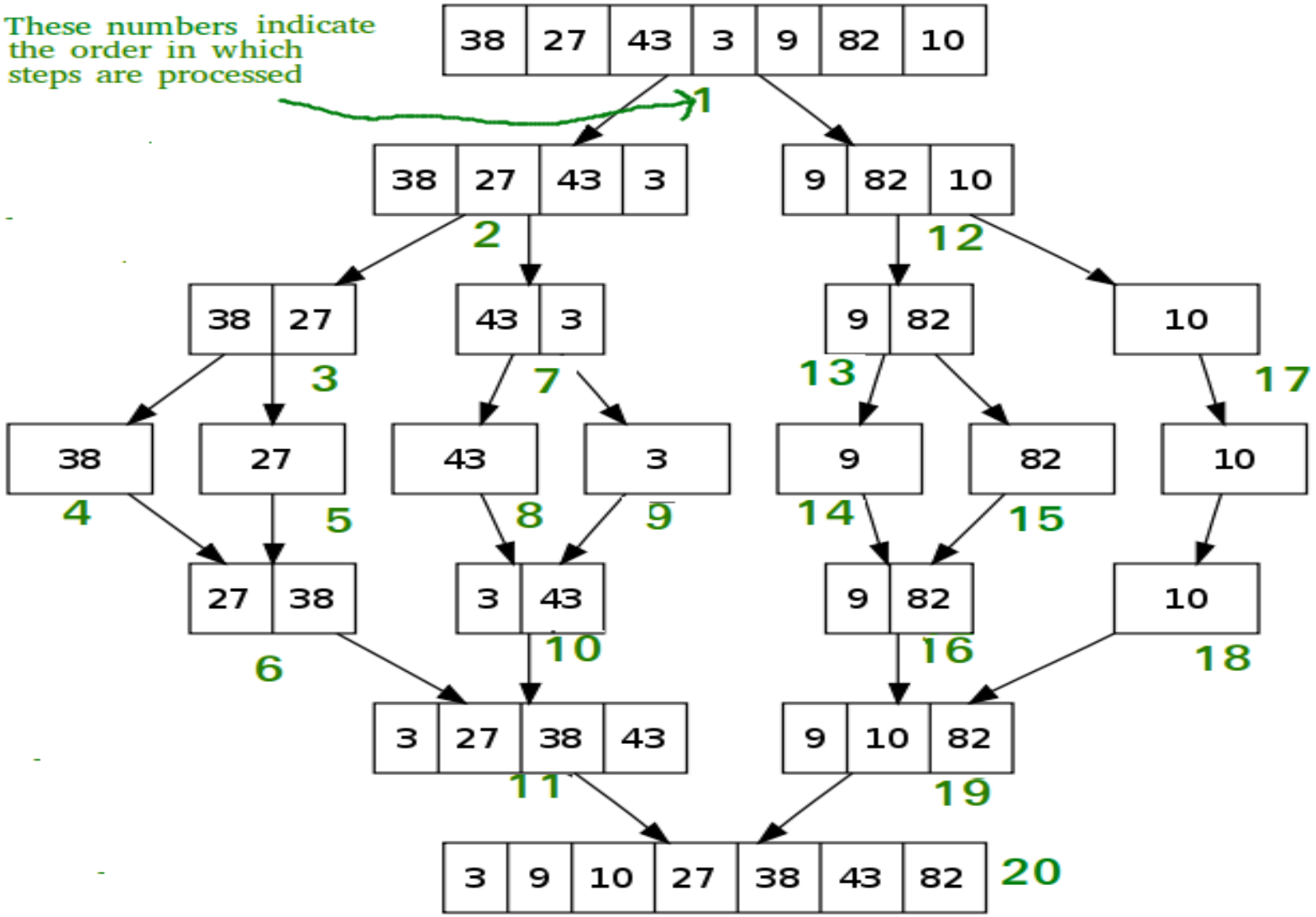
3. Call mergeSort for second half:

   Call mergeSort (array, m+1, right)

4. Merge the two halves sorted in step 2 and 3:

   Call merge (array, left, middle, right)

MERGE SORT


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
2
9 82 10
12
38 27
3
43 3
7
9 82
13
10
17
38
4
27
5
43
8
3
9
9
14
82
15
10
27 38
6
3 43
10
9 82
16
10
18
3 27 38 43
11
9 10 82
19
3 9 10 27 38 43 82
20

MERGE SORT

- Divide & Conquer
- Not Adaptive
- Not In Place
- Stable
- Worst Case T.C. : O(nlogn)
- Best Case T.C. : O(nlogn)
- S.C. : O(n)

QUICK SORT

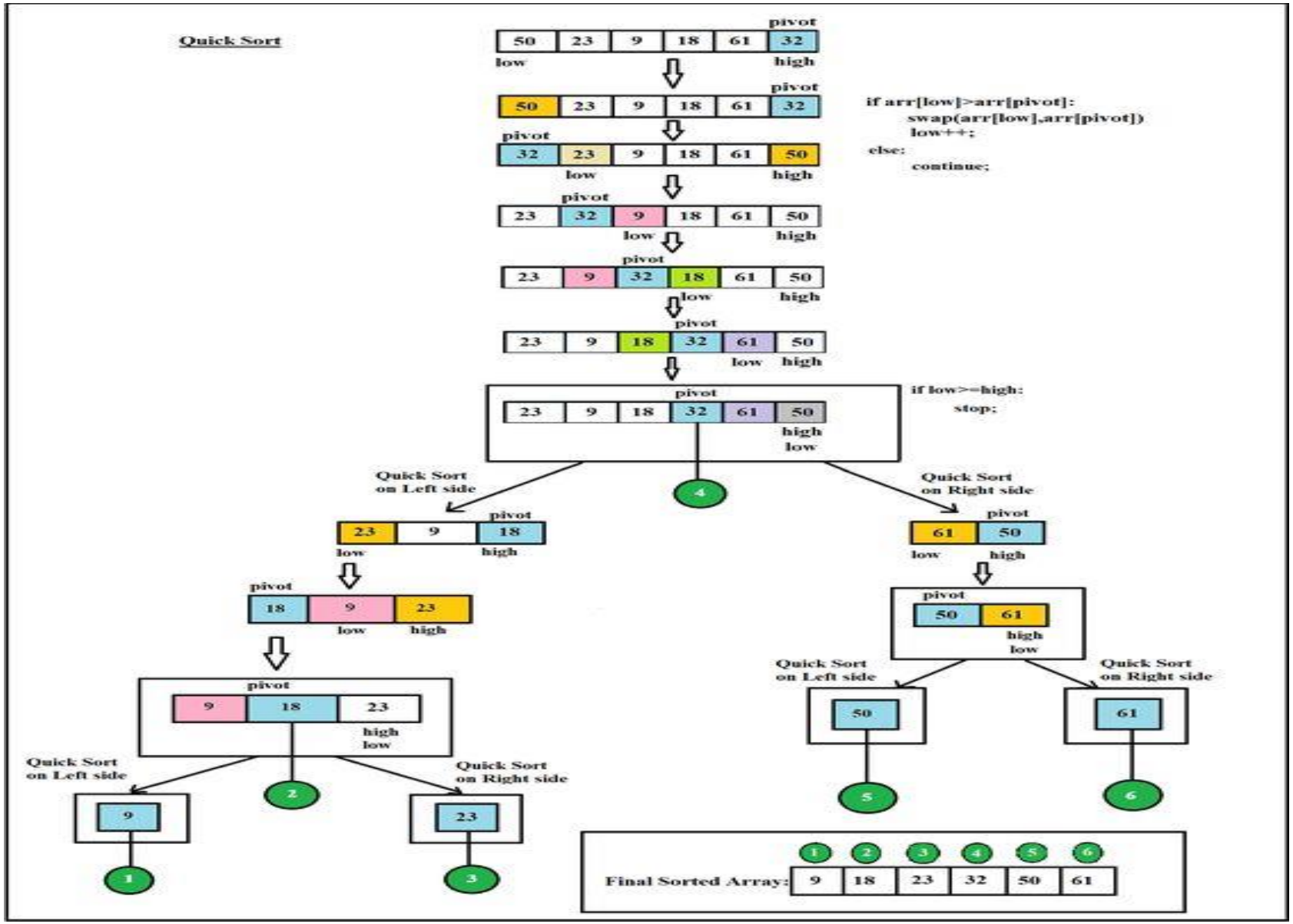
PIVOT CHOICE

- Always pick first element as pivot
- Always pick last element as pivot
- Pick a random element as pivot
- Pick median as pivot

ALGORITHM

- Make the right most index value pivot
- Partion the array using pivot value
- Quicksort left partition recursively
- Quicksort right partition recursively

QUICK SORT


Quick Sort
pivot
50 23 9 18 61 32
low
high
pivot
50 23 9 18 61 32
pivot
32 23 9 18 61 50
low
high
pivot
23 32 9 18 61 50
low
high
pivot
23 9 32 18 61 50
low
high
pivot
23 9 18 32 61 50
low
high
pivot
23 9 18 32 61 50
high
low
if arr[low]>arr[pivot]:
swap(arr[low],arr[pivot])
low++;
else:
continue;
if low>=high:
stop;
Quick Sort
on Left side
Quick Sort
on Right side
pivot
23 9 18
low
high
pivot
18 9 23
low
high
pivot
9 18 23
high
low
Quick Sort
on Left side
Quick Sort
on Right side
9
23
1
2
3
4
pivot
61 50
low
high
pivot
50 61
high
low
Quick Sort
on Left side
Quick Sort
on Right side
50
61
5
6
Final Sorted Array:
1 2 3 4 5 6
9 18 23 32 50 61

QUICK SORT

- Divide & Conquer
- Not Adaptive
- In Place
- Not Stable
- Worst Case T.C. : O(n^2) (already sorted) (if the partion in the end)
- Best Case T.C. : O(nlogn) (if the partion in the middle)
- S.C. : O(logn)