Agenda:

1. Bubble Sort
2. Insertion Sort
3. Selection Sort → Idea
4. Inversion count [Merge Sort]

Bubble Sort

Basic Idea: Sorting done by swapping adjacent value.

arr:

8	2	4	-1*	6	7	5	10	-1
0	1	2	3	4	5	6	7	8

Iteration 1:

1. Compare 8 and 2, Swap 8 and 2, $8 > 2$

   array: [2, 8, 4, -1*, 6, 7, 5, 10, -1]

2. Compare 8 and 4, Swap 8 and 4, $8 > 4$

   array: [2, 4, 8, -1*, 6, 7, 5, 10, -1]

2. Compare 8 and -1*, Swap 8 and -1*, $8 > -1^*$

   array: [2, 4, -1*, 8, 6, 7, 5, 10, -1]

4. Compare 8 and 6, Swap 8 and 6, $8 > 6$

   array: [2, 4, -1*, 6, 8, 7, 5, 10, -1]

5. Compare 8 and 7, Swap 8 and 7, $8 > 7$

   array: [2, 4, -1*, 6, 7, 8, 5, 10, -1]

6. Compare 8 and 5, Swap 8 and 5, $8 > 5$

   array: [2, 4, -1*, 6, 7, 5, 8, 10, -1]

7. Compare 8 and 10, no Swap 8<10

Array: [2, 4, -1*, 6, 7, 5, 8, 10, -1]

8. Compare 10 and -1, Swap 10 and -1., 10>-1

Array: [2, 4, -1*, 6, 7, 5, 8, -1, 10]

Similar Iteration:

Iteration 2:

[2, 4, -1*, 6, 7, 5, 8, -1, 10]

[2, -1*, 4, 6, 7, 5, 8, -1, 10]

[2, -1*, 4, 6, 5, 7, 8, -1, 10]

[2, -1*, 4, 6, 5, 7, -1, 8, 10]

Iteration 3:

Swap Swap Swap

[2, -1*, 4, 6, 5, 7, -1, 8, 10]

-1* 2 5 6 -1 7

[-1*, 2, 4, 5, 6, -1, 7, 8, 10]

Iteration 4:

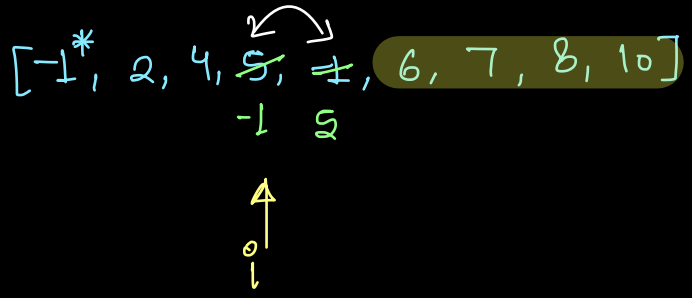
[-1*, 2, 4, 5, 6, -1, 7, 8, 10]

-1 6

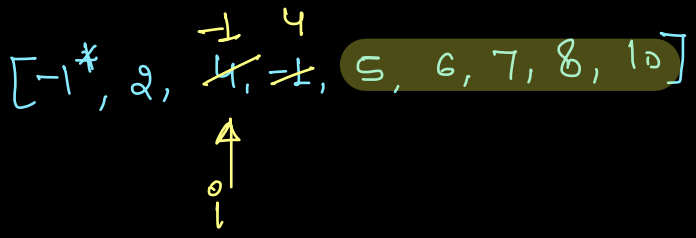
i

[-1*, 2, 4, 5, -1, 6, 7, 8, 10]

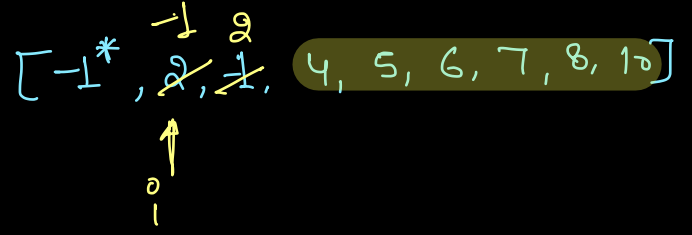
Iteration 5:



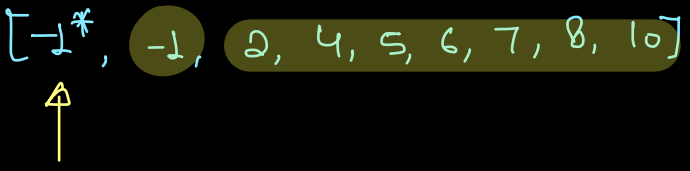
Iteration 6:



Iteration 7:



Iteration 8:



final answer: [-1*, -1, 2, 4, 5, 6, 7, 8, 10]

Bubble sort
- → Stable → maintains the original order of same value
- → inplace → changing in given array itself.

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

T.C : $O(n^2)$

S.C : $O(1)$

Dry Run:

arr[] → [3, 1, 6, 10, 8]

i = 0 Iteration 1 → [~~3~~ 1 3, ~~1~~, 6, ~~10~~ 8 10, ~~8~~] → Swapping count = 2

j ↑

i = 1, Iterations → [1, 3, 6, 8, 10]

j ↑

NOTE: Swapping count is 0, No further swapping is required, data is already sorted now.

Optimisation of Bubble Sort:

→ If in complete iteration, there is no swapping done that means data is already sorted.

→ If data is already sorted, no need to perform more iteration.

```
void bubbleSort(int[] arr){
    int n = arr.length;
    for(int i=0; i<n; i++){
        int swaps = 0;
        for(int j=0; j< n-1; j++){
            if(arr[j] > arr[j+1]){
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                swaps++;
            }
        }
        if(swaps == 0){
            // Data is already sorted
            break;
        }
    }
}
```

Time Complexity :-

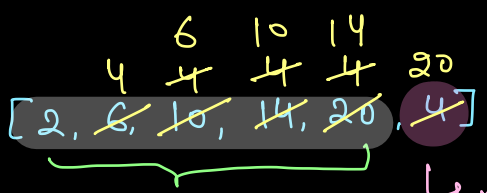
worst case : $O(n^2)$

best case : $O(n)$

space complexity : $O(1)$

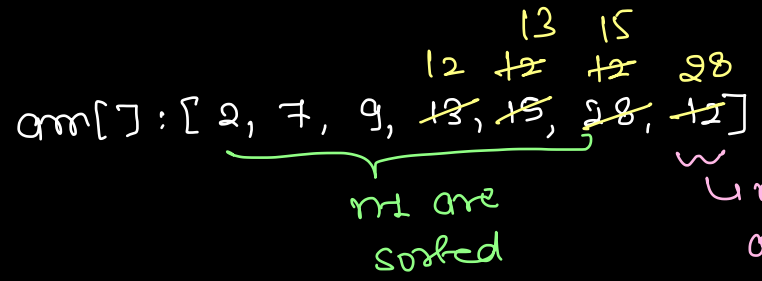
Insertion Sort:

Insertion Step:



array: [2, 4, 6, 10, 14, 20]

Example:



arr[]: [2, 7, 9, 12, 13, 15, 28]

Assume all elements are sorted, except for last one

write code to sort it.

```
void InsertionStep(int[] arr){
    int n= arr.length;
    // first n-1 elemt is sorted, last elemt is not
    for(int j=n-2; j>=0; j--){
        if(arr[j] > arr[j+1]){
            int temp= arr[j];
            arr[j]= arr[j+1];
            arr[j+1] = temp;
        }else{
            break;
        }
    }
}
```

T.C: O(n) → worst case.
S.C: O(1)

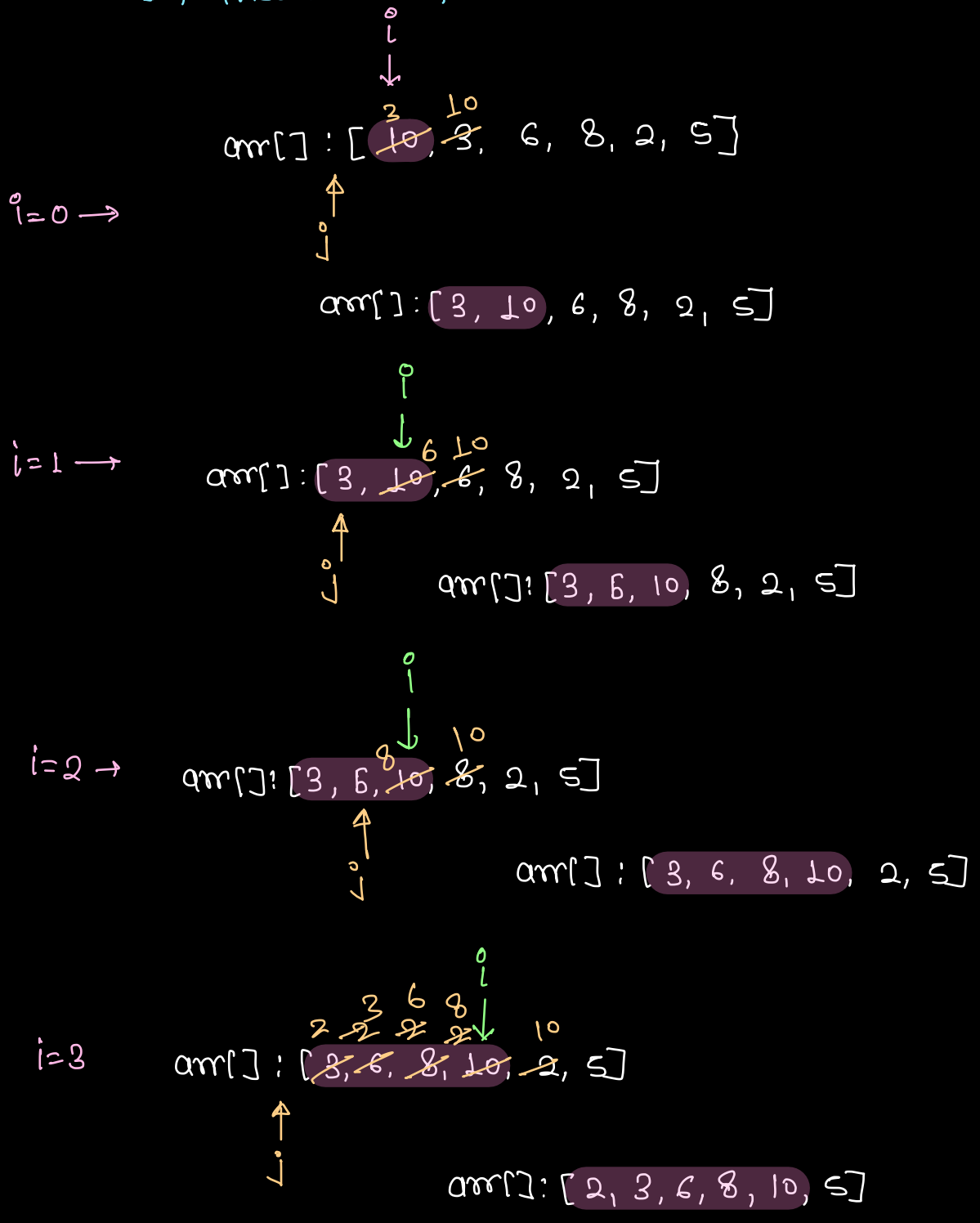
Insertion Sort:

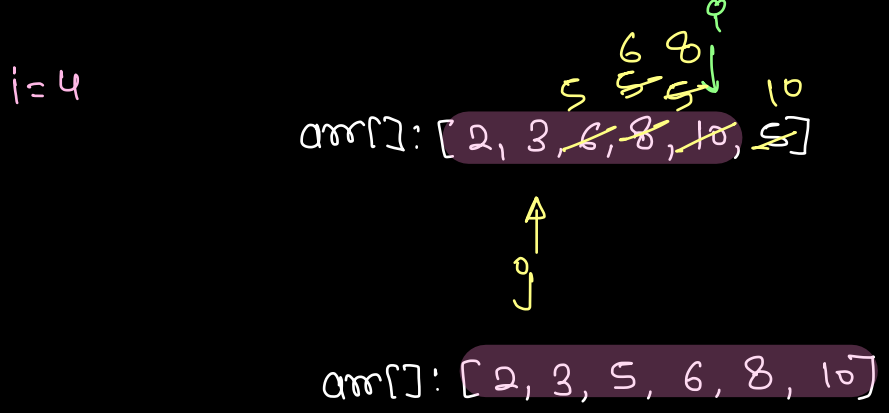
arr[] : [10, 3, 6, 8, 2, 5]

① Assume that array is sorted from 0 to i.

② we have to insert (i+1) element in correct position after sorting it.

→ Insertion Step.






```
void InsertionSort(int[] arr){
    int n = arr.length;
    for(int i=0; i< n-1; i++){
        for(int j=i; j>=0; j--){
            if(arr[j] > arr[j+1]){
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }else{
                break;
            }
        }
    }
}
```

T.C: O(n²)
S.C: O(1)

Best Case : Array is Sorted

T.C: O(n)

NOTE: * Stable Sort
* Inplace Sort

10:37 - 10:50

Break

Selection Sort:

i

↓

arr:	29	10	14	37	13
	0	1	2	3	4

i=0

Steps:

Repeat

* Start from i = 0.
* Find min Element index from 'i' to n-1
* Swap i & minIndex elements.

arr:	29	10	14	37	13
	0	1	2	3	4

arr[]: [~~29~~ (10), ~~10~~ (29), 14, 37, 13]

0 1 2 3 4

i=0 → minIndex, [b/w i to n-1]

minElementIndex = 1

Swap (arr[i], arr[minElementIndex]);

i

↓

i=1 → arr[]: [10, ~~29~~ (13), 14, 37, ~~13~~ (29)]

0 1 2 3 4

minIndex = 4, Swap(arr[i], arr[minIndex]);

i

↓

i=2 → arr[]: [10, 13, 14, 37, 29]

0 1 2 3 4

minIndex = 2, Swap (arr[i] arr[minIndex])

i=3, arr[]: [10, 13, 14, 37, 29]
0 1 2 3 4

minIndx = 4, Swap(arr[i], arr[minIndx]);

i=4] → No need to solve.

```
void selectionSort(int[] arr){
    int n = arr.length;
    for(int i=0; i<n-1; i++){
        // find index of min element
        from 'i' to n-1
        int minIndx = i;
        for(int j=i+1; j<n; j++){
            if(arr[j] < arr[minIndx]){
                minIndx = j;
            }
        }
        // swap i and minIndx
        int temp = arr[i];
        arr[i] = arr[minIndx];
        arr[minIndx] = temp;
    }
}
```

T.C: $O(n^2)$

S.C: $O(1)$

[2A, 3B, 2C, 1D] → after sorting [1D 2A 2C 3B]

Stable ✓

Selection Sort - [2A, 3B, 2C, 1D]
0 1 2 3

minIndx = 3, Swap(arr[i], arr[minIndx])

[1D, 3B, 2C, 2A]
2C 3B

→ [1D 2C 3B 2A] → [1D 2C 2A 3B]

not stable

Inversion Count

Given an arr[n], calculate no of pairs [i,j] such that i<j && arr[i]>arr[j], i and j are index of array?

arr[]: [10, 3, 8, 15, 6]
0 1 2 3 4

i<j && arr[i] > arr[j]

i	j	arr[i] > arr[j]
0 < 1 →		10 > 3
0 < 2 →		10 > 8
0 < 4 →		10 > 6
2 < 4 →		8 > 6
3 < 4 →		15 > 6

Count of pair = 5

Question: arr: [5, 2, 6, 1]
0 1 2 3

All possible pair, i<j →

arr[i] > arr[j]

- 0 1 ✓
- 0 2 ✗
- 0 3 ✓
- 1 2 ✗
- 1 3 ✓
- 2 3 ✓

4 pairs.

Question:

[5	3	1	4	2]
0	1	2	3	4

All pairs:

	j=0	j=1	j=2	j=3	j=4
i=0	0,0	0,1 ✓	0,2 ✓	0,3 ✓	0,4 ✓
i=1	1,0	1,1	1,2 ✓	1,3 ✗	1,4 ✓
i=2	2,0	2,1	2,2	2,3 ✗	2,4 ✗
i=3	3,0	3,1	3,2	3,3	3,4 ✓
i=4	4,0	4,1	4,2	4,3	4,4

i<j → Relevant pairs.

arr[i] > arr[j]

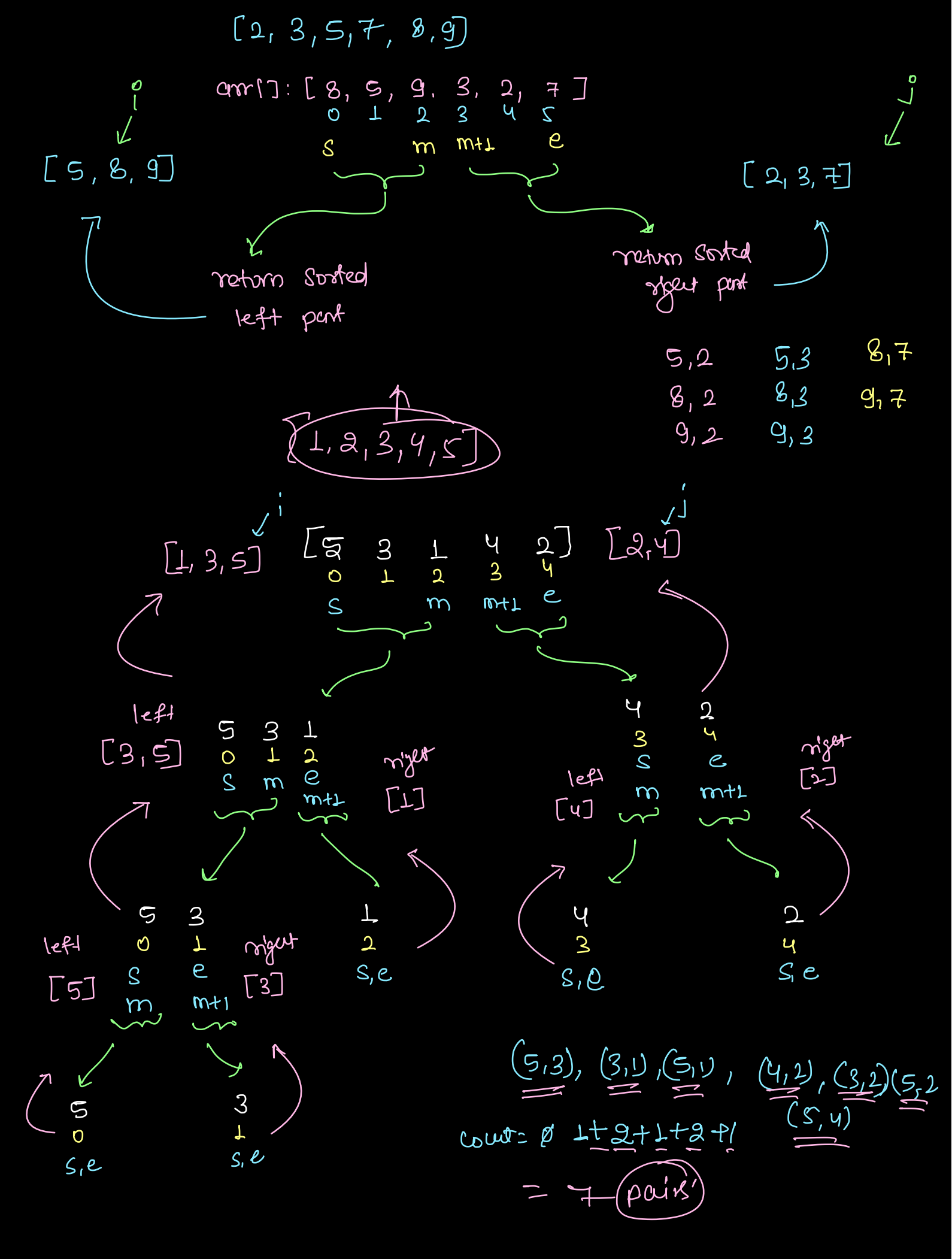
ans = 7 pairs.

```
int InversionCount(int[] arr){
    int n = arr.length;
    int count = 0;
    for(int i=0; i<n-1; i++){
        for(int j=i+1; j<n; j++){
            // All valid i & j pair here,
            if(arr[i] > arr[j]){
                count++;
            }
        }
    }
    return Count;
}
```

T.C: O(n²)

S.C: O(1)

Optimisation of Inversion count:



Merge two Sorted array!

left [5, 6, 9, 15, 16]
0 1 2 3 4

i

[2, 4, 8, 10]
0 1 2 3

j

Part of merge two Sorted array.

```
if( left[i] > right[j] ){
    // valid inversion pair
    count = count + (n-i);
    res[k] = right[j];
    j++;
    k++;
}
else {
    res[k] = left[i];
    i++;
    k++;
}
```

this code in mergeSort code + make count global variable.

Elements: [i n-1]

[a b]

⇒ b-a+1

⇒ (n-1) - i + 1

= (n-i)

Time Complexity of inversion = nlogn

Space Complexity = O(n)

$n^2 \Rightarrow (100)^2 \Rightarrow 10000$

$a_1\ a_2\ a_3\ a_4 \cdots a_{50}$ $\qquad$ $a_{51} \cdots a_{100}$

$a_1\ a_2 \cdots a_{25}\ a_{26} \cdots a_{50}$ $\qquad$ $a_{51}\ a_{52} \cdots a_{75}\ a_{76} \cdots a_{100}$

size = 50 $\qquad$ size = 50

$\Rightarrow n^2 = 50 \times 50 = 2500$ $\qquad$ $n^2 = 50 \times 50 = 2500$

left $\Rightarrow$ 2500

right $\Rightarrow$ 2500

mid $\Rightarrow$ 100

total ghent = $2500 + 2500 + 100$

$= 5100$

$625 + 625 + 50 + 625 + 625 + 50 + 100 \Rightarrow 2700$

$a_1\ a_2 \cdots a_{50}$ $\qquad$ $a_{51} \cdots a_{100}$

$625 + 625 + 50$ $\qquad$ $625 + 65 + 50$

$a_1\ a_2 \cdots a_{25} \cdots a_{50}$ $\qquad$ $a_{51}\ a_{52} \cdots a_{75} \cdots a_{100}$

$a_1 \cdots a_{25}$ $\qquad$ $a_{26} \cdots a_{50}$ $\qquad$ $a_{51} \cdots a_{75}$ $\qquad$ $a_{76} \cdots a_{100}$

$s = 25$ $\qquad$ 625 $\qquad$ 625 $\qquad$ 625

$n^2 = 625$