DATA STRUCTURES (ITPC-203)

Sorting Techniques – Part I

Mrs. Sanga G. Chaki

Department of Information Technology

Dr. B. R. Ambedkar National Institute of Technology, Jalandhar

Contents

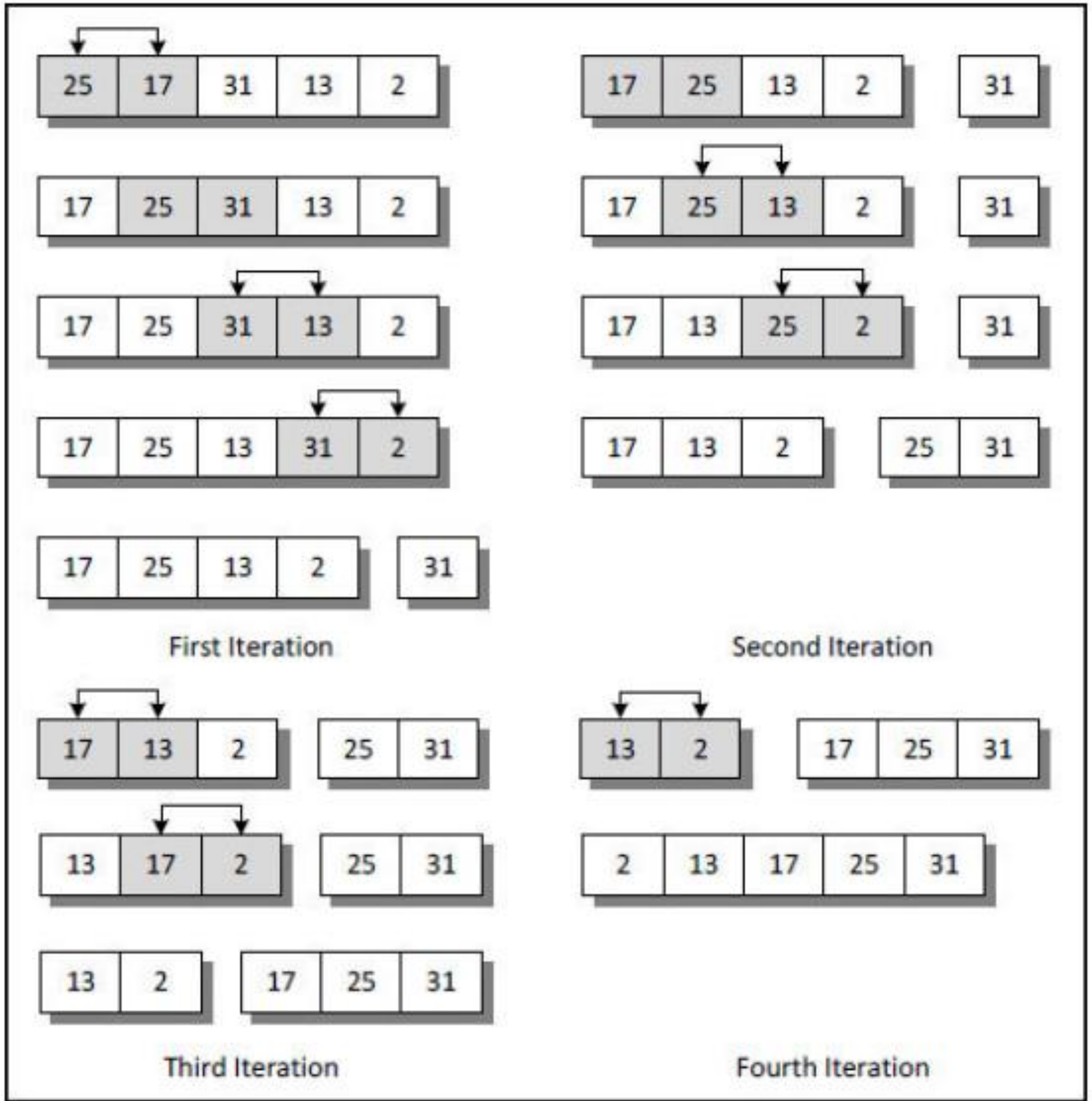
Sorting

1. Sorting refers to arranging elements of a list in some order.
2. Order: ascending or descending
3. There are different methods that are used to sort the data
4. These methods can be divided into two categories. They are as follows:
   - Internal Sorting: If all the data to be sorted can be accommodated at a time in memory then internal sorting methods are used.
   - External sorting: When the data to be sorted is so large that some of the data is present in the memory and some is kept in auxiliary memory (hard disk, tape, etc.), then external sorting methods are used.
5. We will primarily focus on internal sorting techniques.

Bubble Sort

1. The sorting process proceeds in several passes/iteration.
2. In every pass, we go on comparing neighboring pairs, and swap them if out of order.
3. If we are sorting in ascending order, in every pass, the largest of the elements under consideration will bubble to the top (i.e., the right).
4. Number of comparisons: n(n-1)/2, if there are n elements in the array.
5. When will bubble sort stop?

Bubble Sort

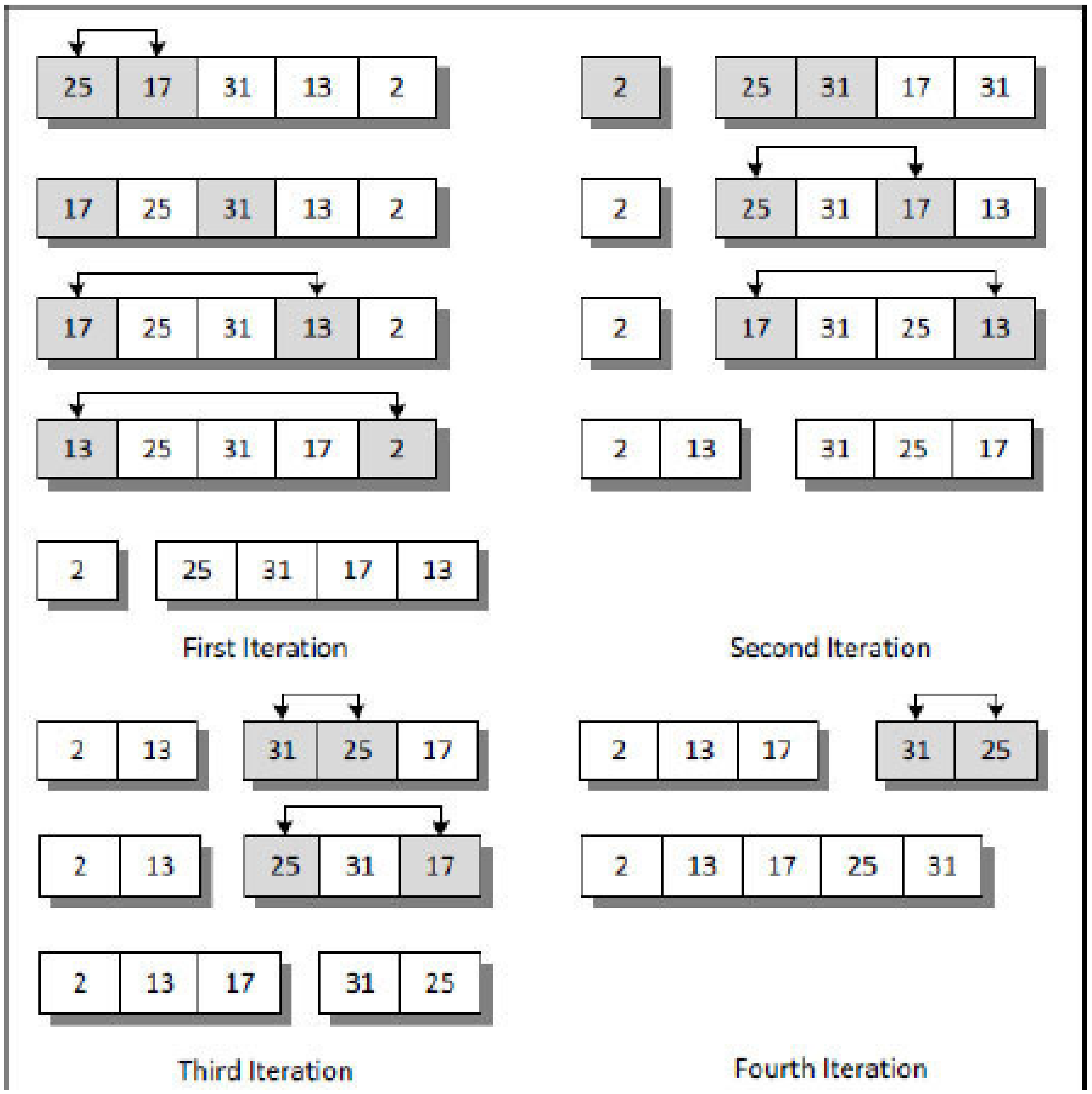


25 17 31 13 2
17 25 31 13 2
17 25 31 13 2
17 25 13 31 2
17 25 13 2 31
First Iteration
17 25 13 2 31
17 25 13 2 31
17 13 25 2 31
17 13 2 25 31
Second Iteration
17 13 2 25 31
13 17 2 25 31
13 2 17 25 31
Third Iteration
13 2 17 25 31
2 13 17 25 31
Fourth Iteration

Selection Sort

1. In this method, to sort the data in ascending order, in the first step, the first element is compared with all other elements.
2. If the first element is found to be greater than the compared element then they are interchanged.
3. So after the first iteration/pass the smallest element gets placed at the first position
4. In the second iteration, the second element is compared to all other elements, and the same procedure is followed.

Selection Sort



Selection Sort - Code

```
void selectionsort (int a[ ], int size){
int i, j, temp;
for (i = 0; i < size - 1; i++){
for (j = i + 1; j < size; j++){
if (a[ i ] > a[ j ]){ //swap for ascending
temp = a[ i ];
a[ i ] = a[ j ];
a[ j ] = temp;}}}}
```

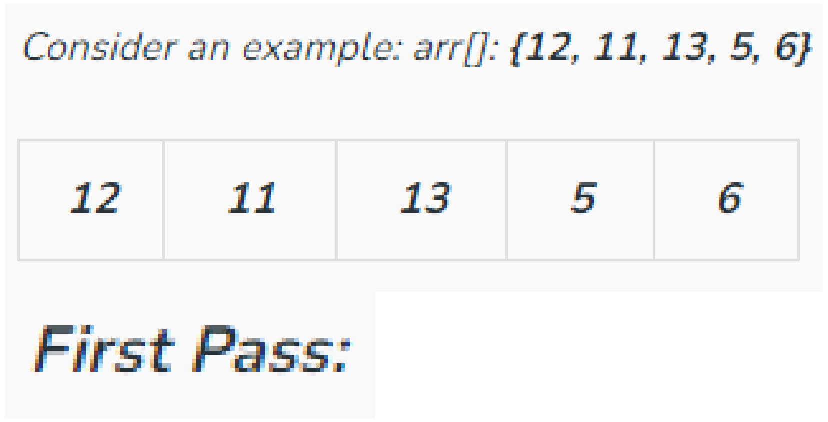
Insertion Sort

1. Insertion sort works similar to the way you sort playing cards in your hands.
2. The array is virtually split into a sorted and an unsorted part.
3. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Insertion Sort

1. To sort an array of size N in ascending order, we iterate over the array and compare the current element (key) to its predecessor
2. If the key element is smaller than its predecessor, compare it to the elements before → to find the correct place to insert this key in the array
3. Move the greater elements one position up to make space for the swapped element.

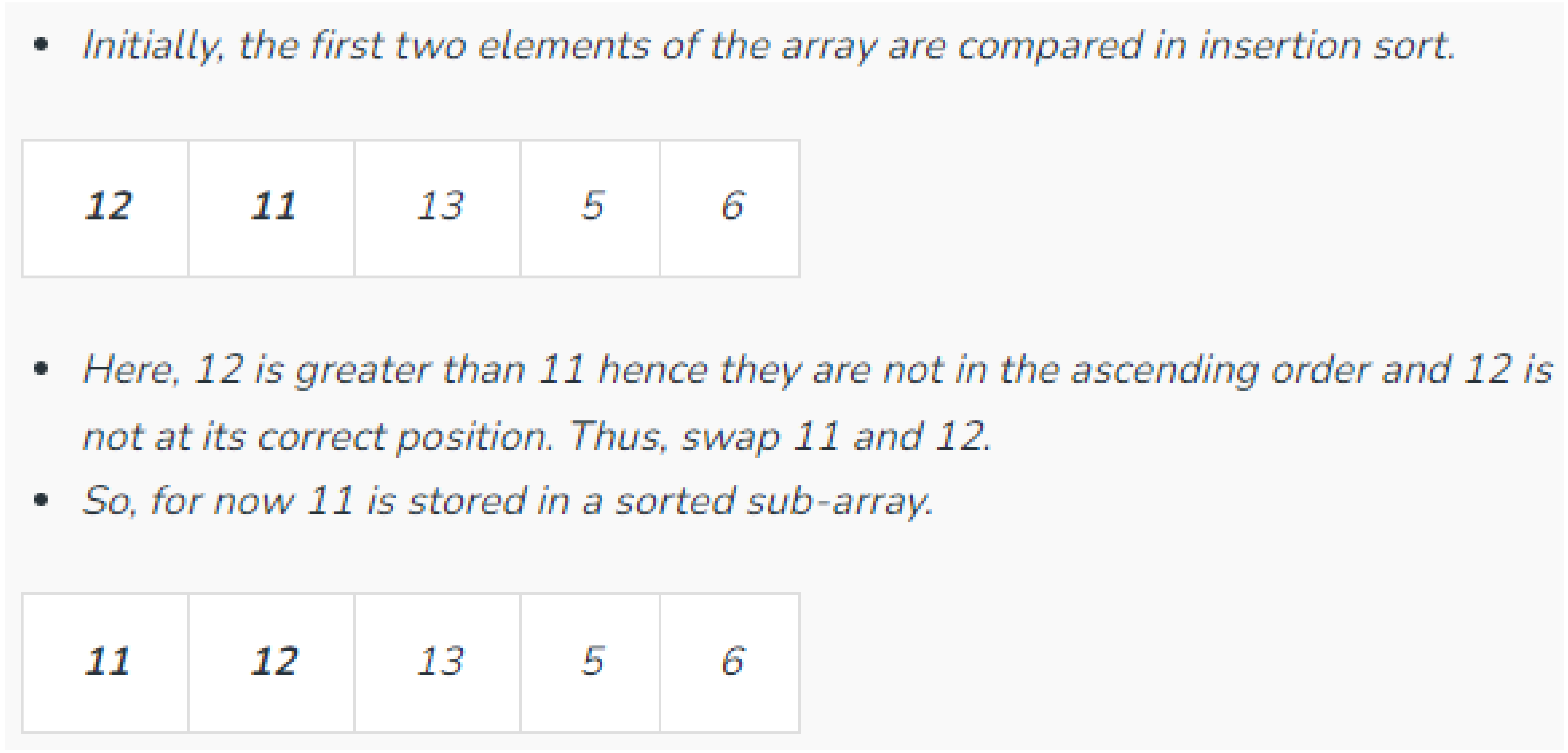
Insertion Sort



Consider an example: arr[]: ***{12, 11, 13, 5, 6}***

12	**11**	**13**	**5**	**6**

First Pass:



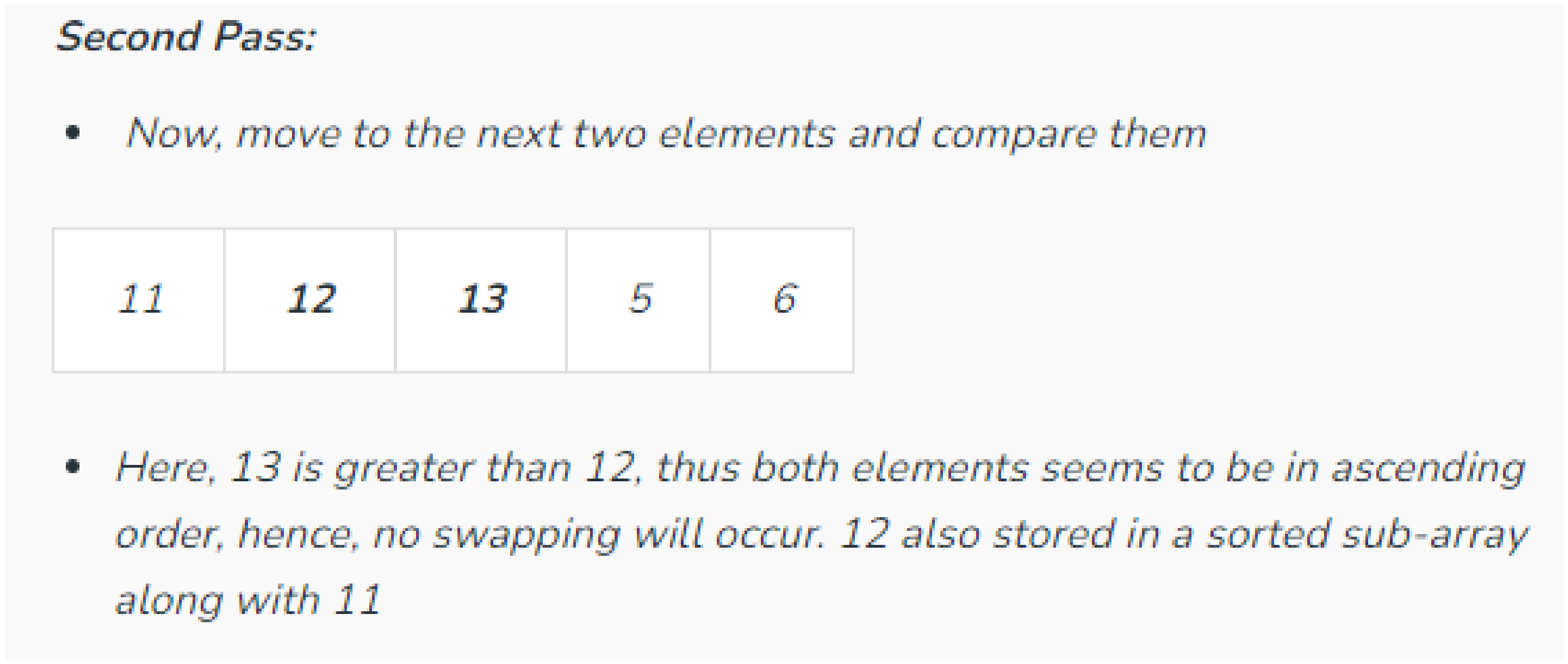
- *Initially, the first two elements of the array are compared in insertion sort.*

12	**11**	13	5	6

- *Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.*
- *So, for now 11 is stored in a sorted sub-array.*

11	**12**	13	5	6

Insertion Sort



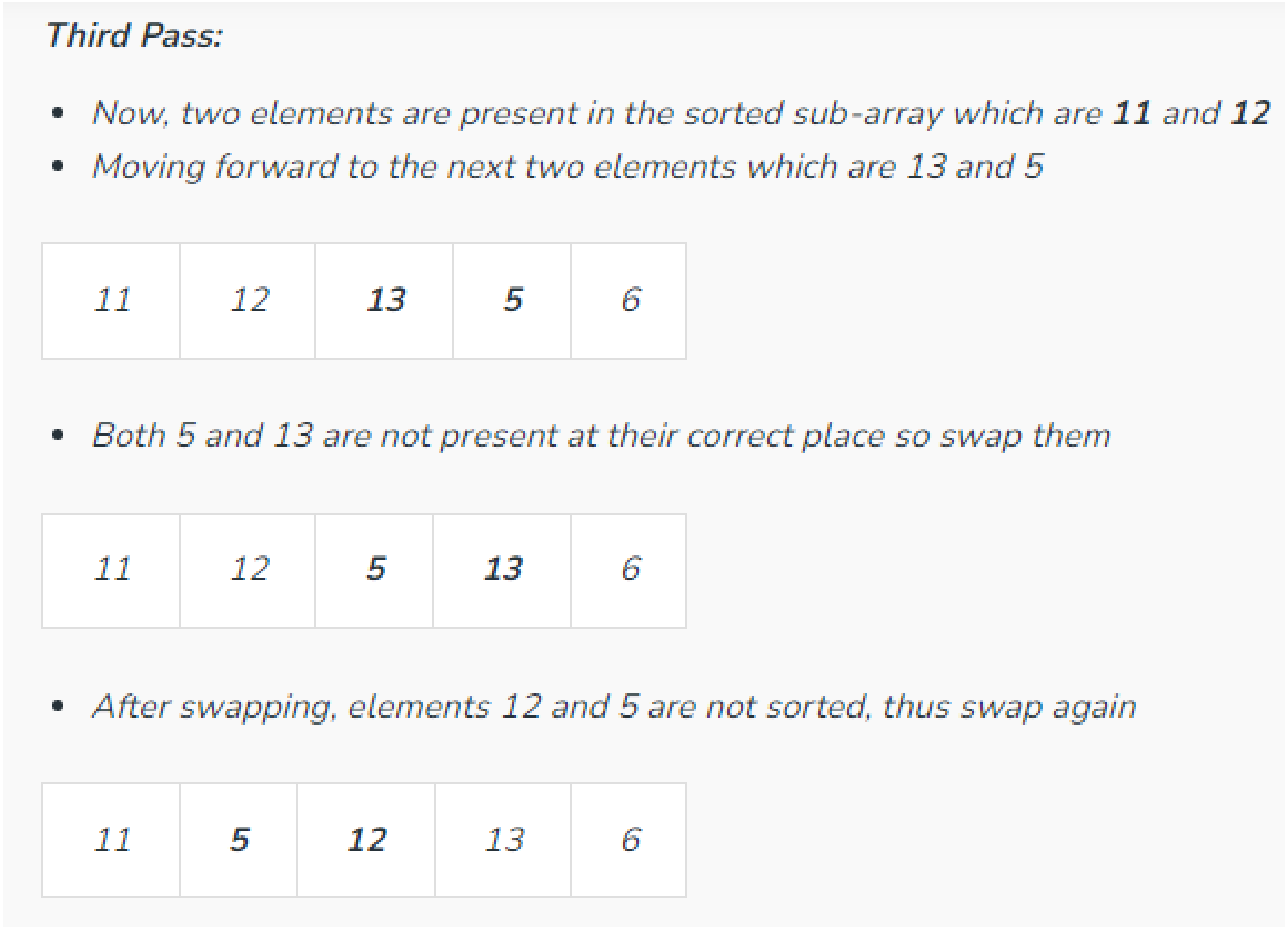
Second Pass:

- *Now, move to the next two elements and compare them*

11	***12***	***13***	*5*	*6*

- *Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11*

Insertion Sort



Third Pass:

- *Now, two elements are present in the sorted sub-array which are **11** and **12***
- *Moving forward to the next two elements which are 13 and 5*

11	*12*	***13***	***5***	*6*

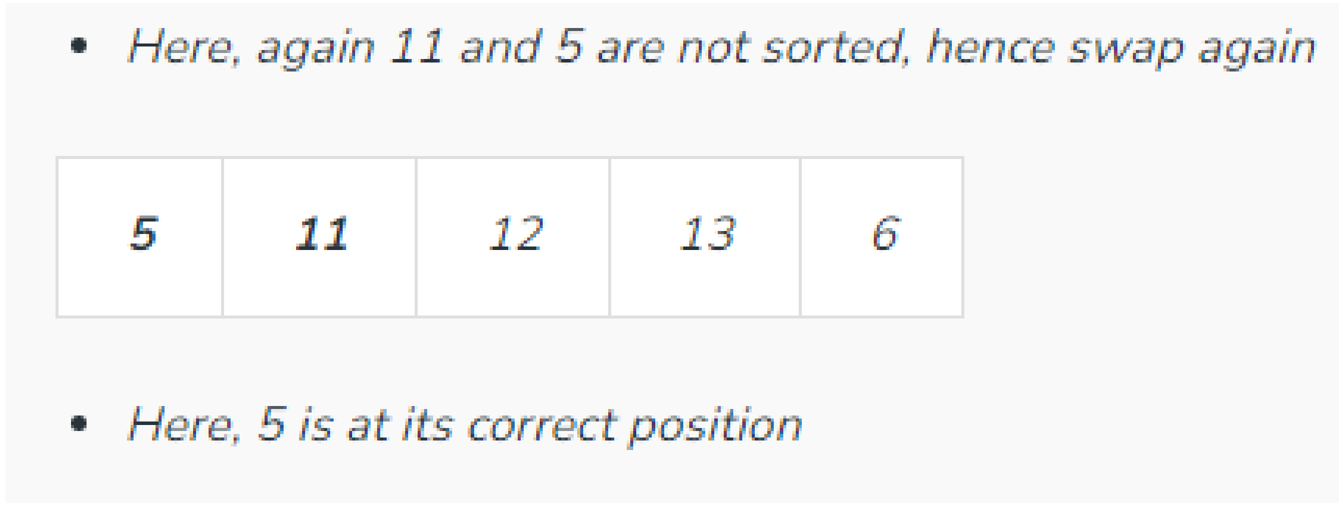
- *Both 5 and 13 are not present at their correct place so swap them*

11	*12*	***5***	***13***	*6*

- *After swapping, elements 12 and 5 are not sorted, thus swap again*

11	***5***	***12***	*13*	*6*

Insertion Sort

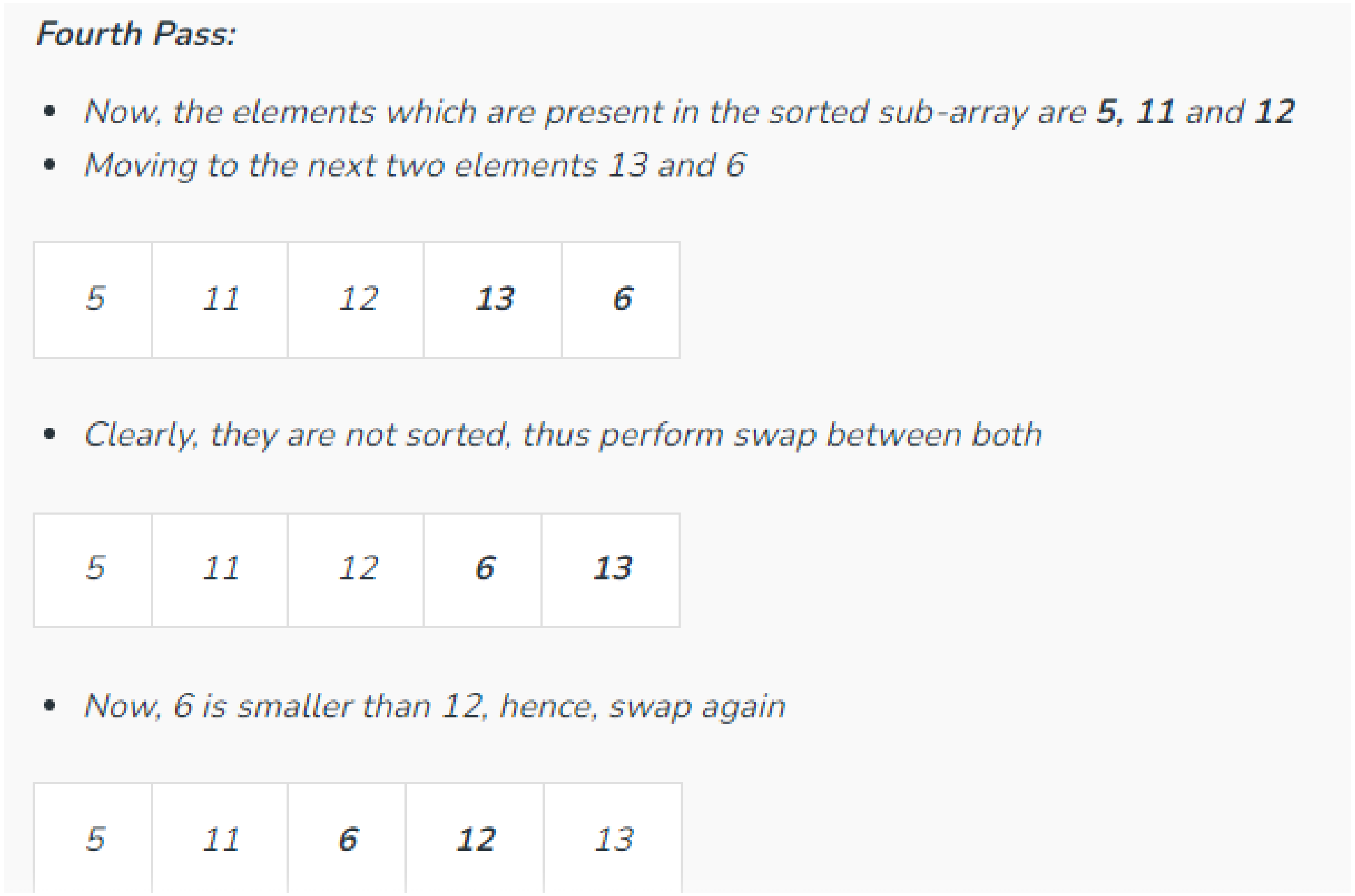


- *Here, again 11 and 5 are not sorted, hence swap again*

5	**11**	12	13	6

- *Here, 5 is at its correct position*

Insertion Sort



Fourth Pass:

- *Now, the elements which are present in the sorted sub-array are **5, 11** and **12***
- *Moving to the next two elements 13 and 6*

5	*11*	*12*	***13***	***6***

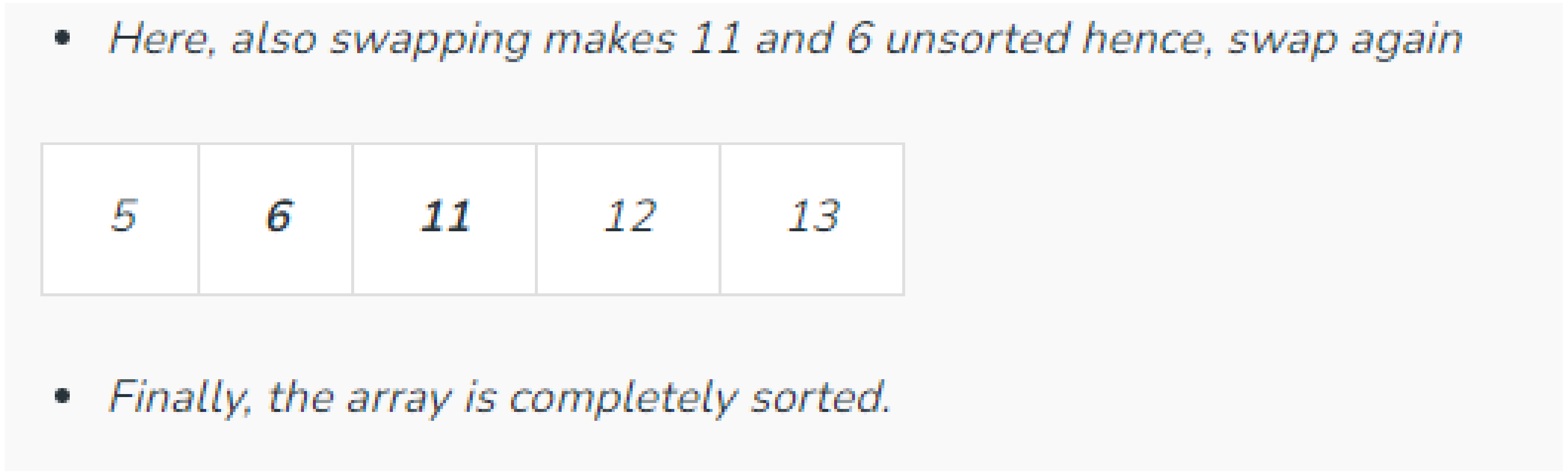
- *Clearly, they are not sorted, thus perform swap between both*

5	*11*	*12*	***6***	***13***

- *Now, 6 is smaller than 12, hence, swap again*

5	*11*	***6***	***12***	*13*

Insertion Sort



- *Here, also swapping makes 11 and 6 unsorted hence, swap again*

5	***6***	***11***	*12*	*13*

- *Finally, the array is completely sorted.*

Insertion Sort

```
void insertionSort(int arr[], int n){
        int i, key, j;
        for (i = 1; i < n; i++) {
                key = arr[i];
                j = i - 1;
/* Move elements of arr[0..i-1], that are greater than key, to one position
ahead of their current position */
                while (j >= 0 && arr[j] > key) {
                        arr[j + 1] = arr[j];
                        j = j - 1;
                }
                arr[j + 1] = key;
        }
}
```

Quick Sort

1. QuickSort is a sorting algorithm based on the Divide and Conquer algorithm.
2. It picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

Quick Sort

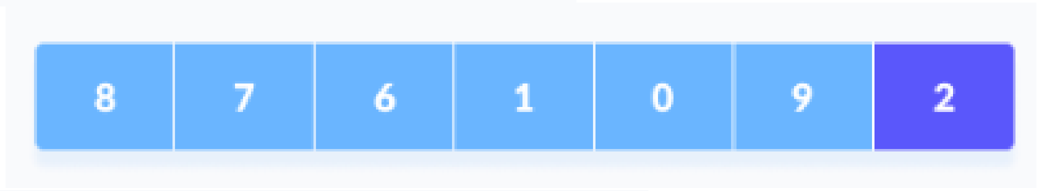
1. The key process in quickSort is a partition().
2. The target of partitions is to:
   - place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and
   - Put all smaller elements to the left of the pivot,
   - and all greater elements to the right of the pivot.
3. Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

Quick Sort

1. How to choose the pivot? Many options
    - Always pick the first element as a pivot.
    - **Always pick the last element as a pivot**
    - Pick a random element as a pivot.
    - Pick the middle as the pivot.

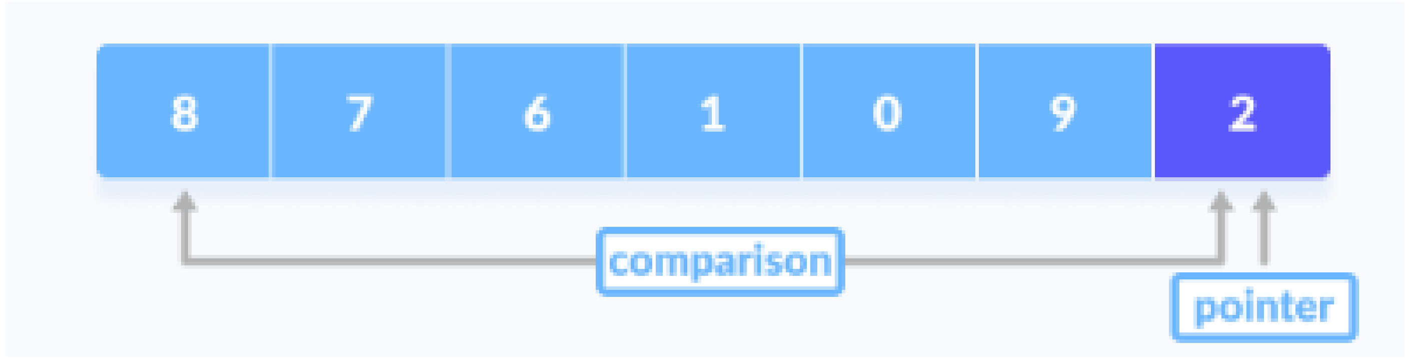
Quick Sort - Example

1. Select the Pivot Element



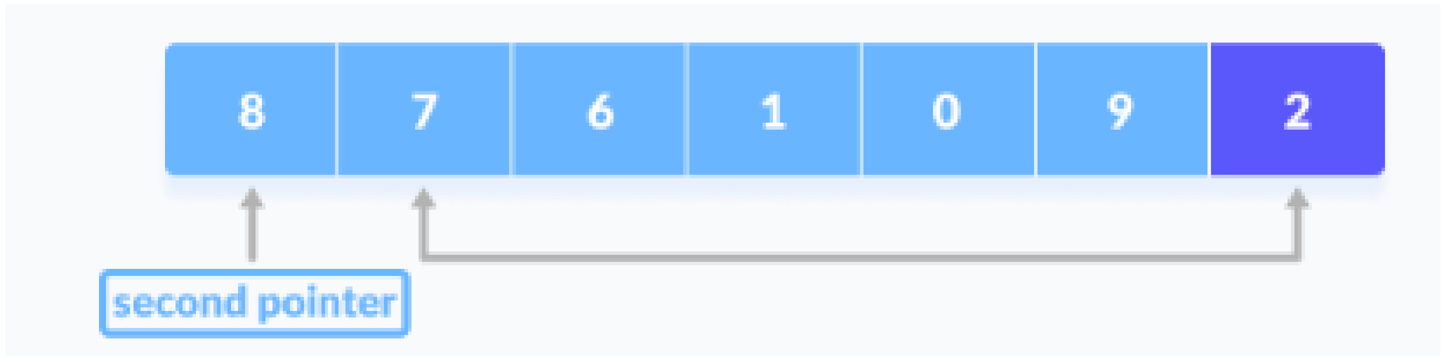
2. Rearrange the Array

A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.



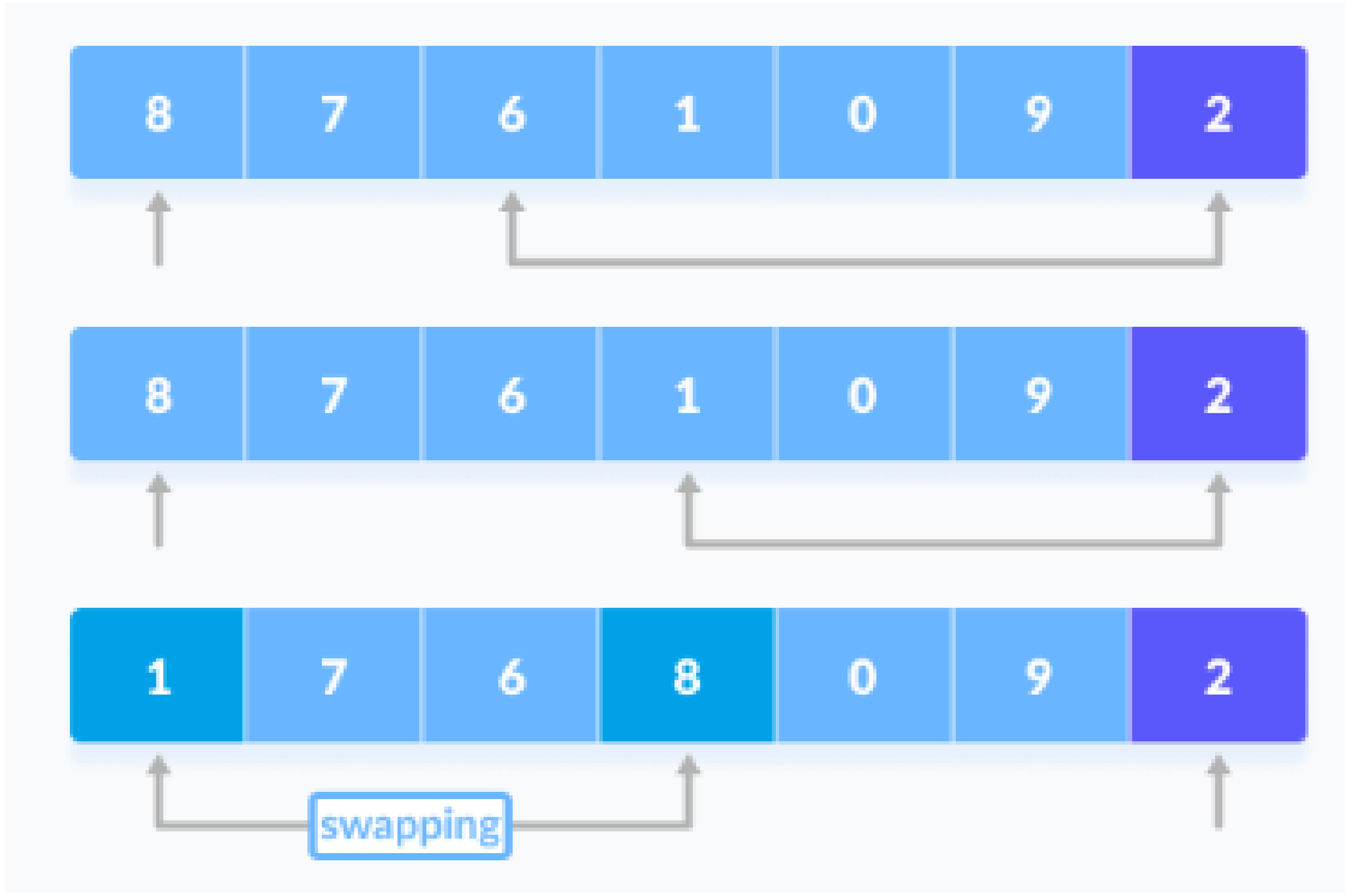
Quick Sort - Example

If the element is greater than the pivot element, a second pointer is set for that element.



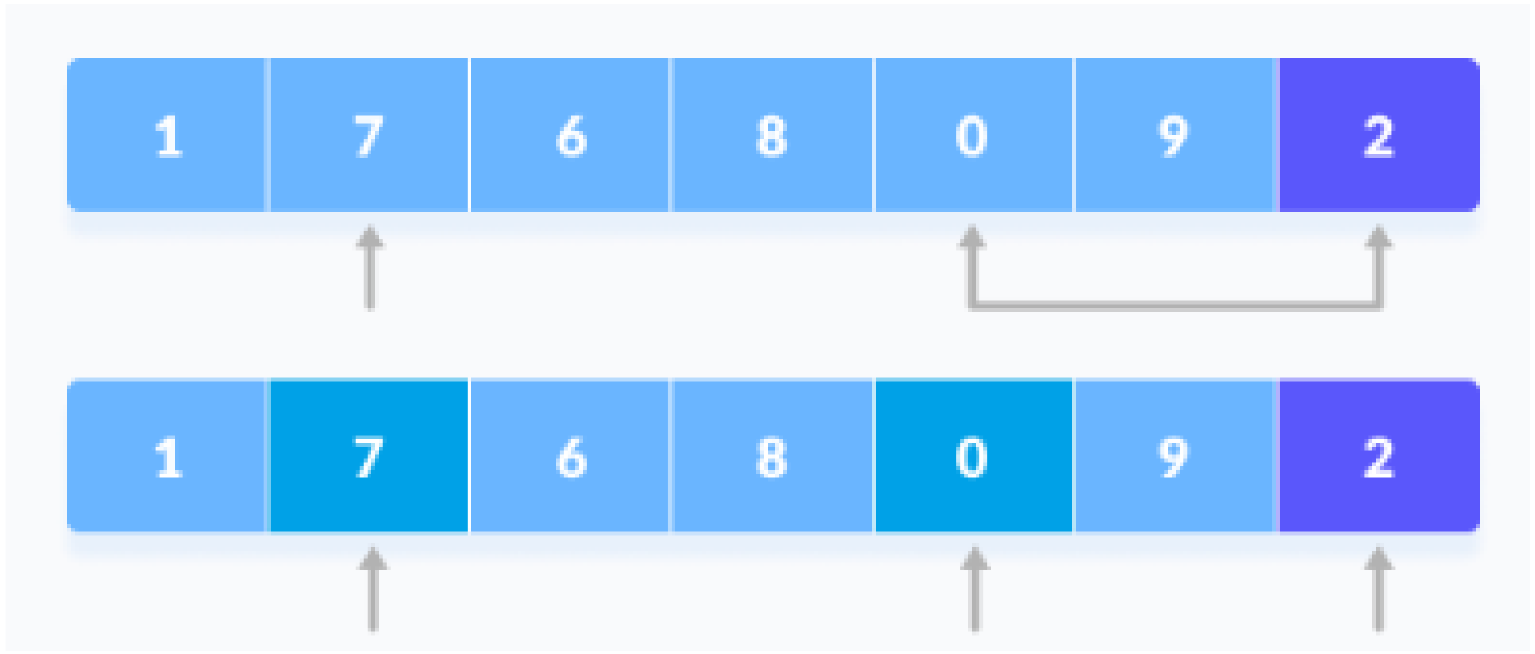
Quick Sort - Example

1. Now, pivot is compared with other elements.
2. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.



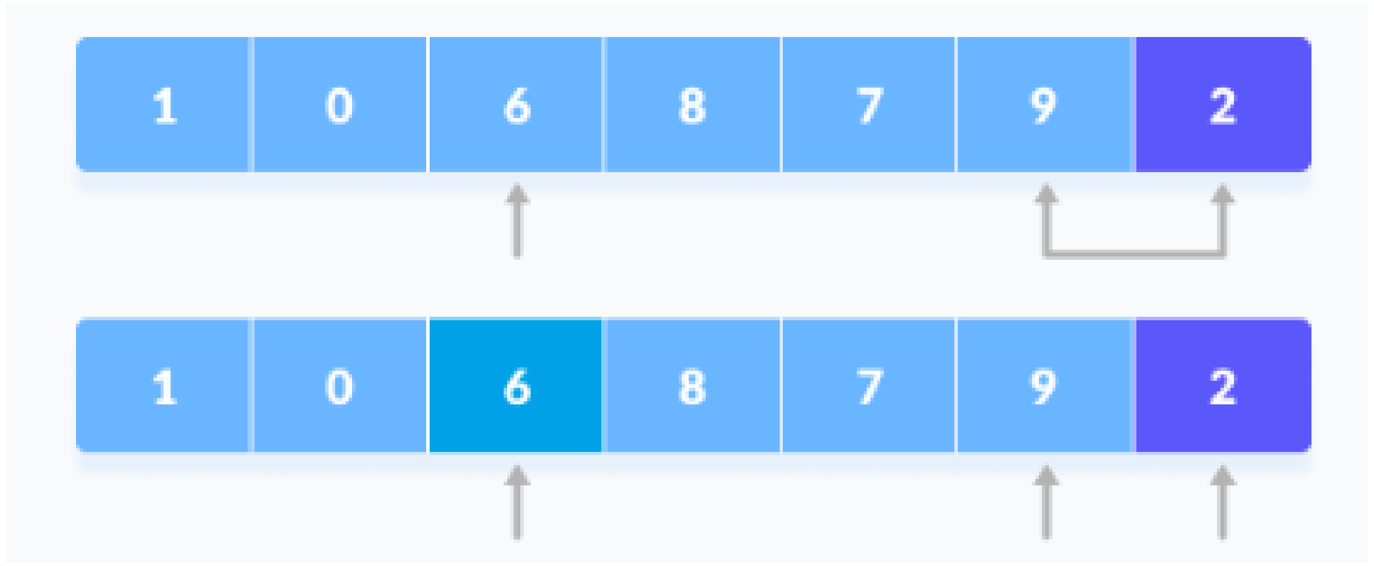
Quick Sort - Example

1. Again, the process is repeated to set the next greater element as the second pointer.
2. And, swap it with another smaller element.



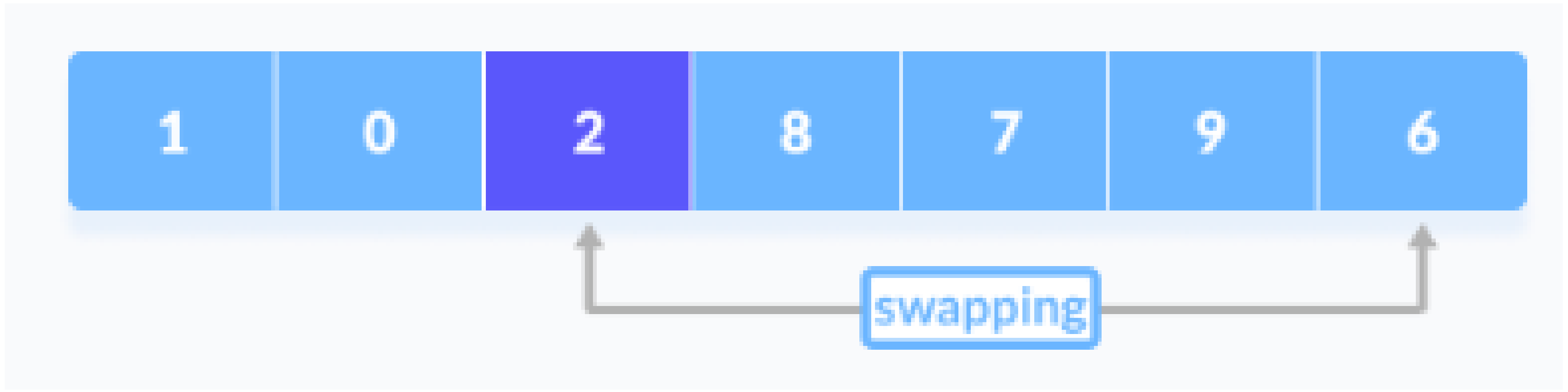
Quick Sort - Example

1. The process goes on until the second last element is reached



Quick Sort - Example

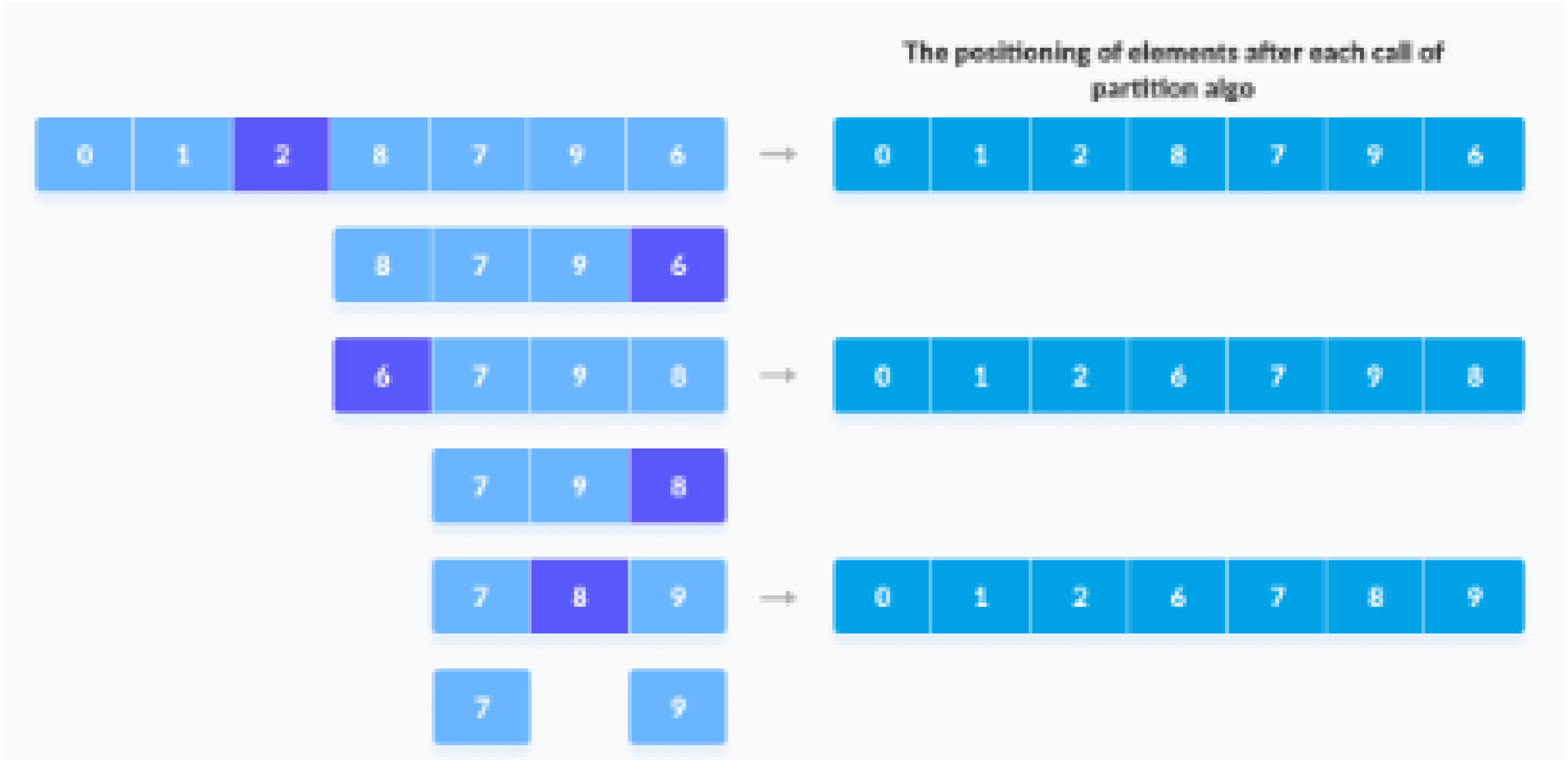
1. Finally, the pivot element is swapped with the second pointer



- So, the pivot has divided the array into two sub-arrays.
- The array is not sorted yet.
- New pivots are chosen for these two sub-arrays.

Quick Sort - Example

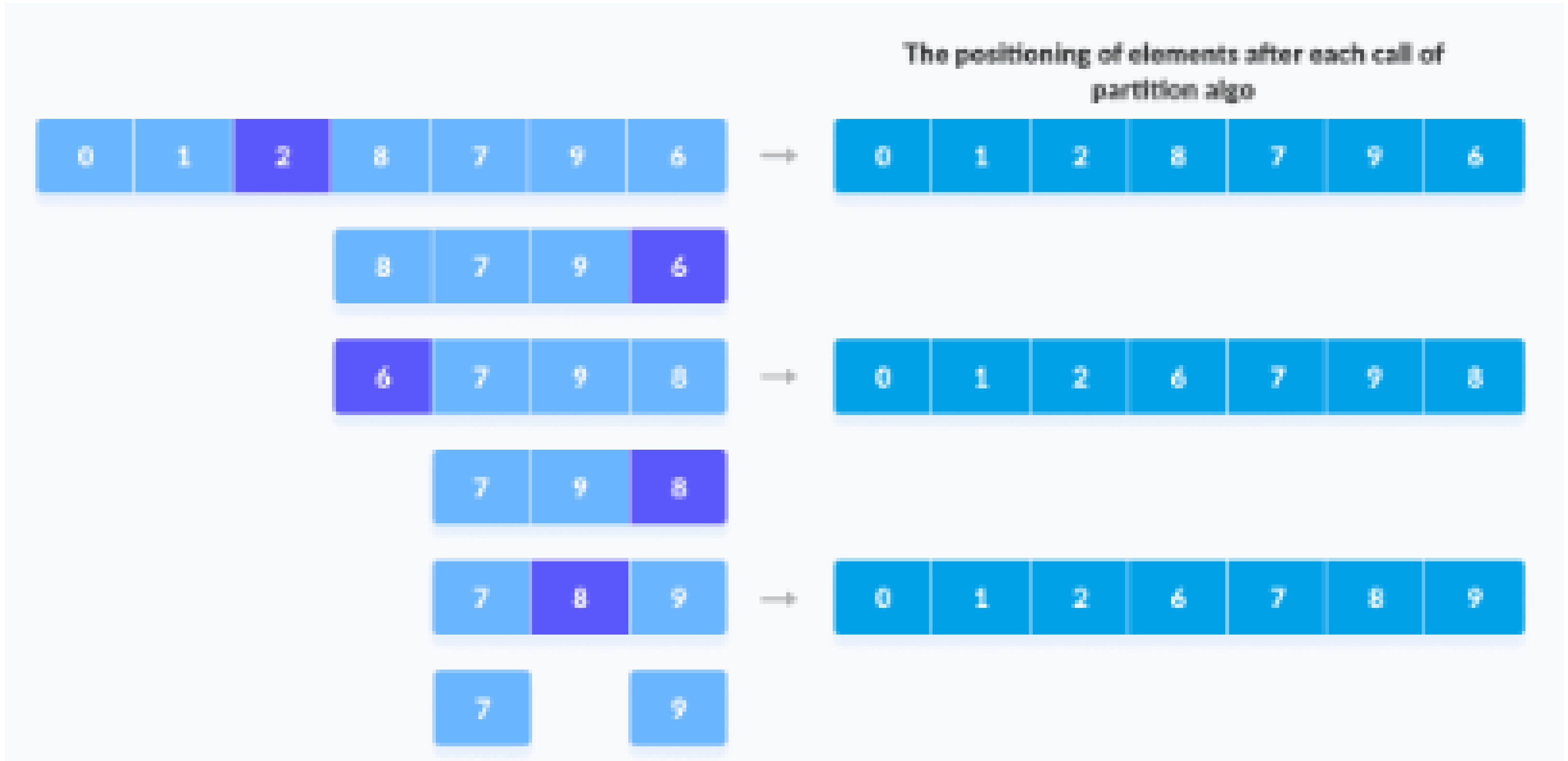
1. Pivot elements are again chosen for the left and the right sub-parts separately.
2. And, entire step 2 is repeated.
3. The subarrays are divided until each subarray is formed of a single element. At this point, the array is already sorted.



Quick Sort - Example

3. Divide Subarrays

1. Pivot elements are again chosen for the left and the right sub-parts separately.
2. And, entire step 2 is repeated.



The positioning of elements after each call of partition algo
0 1 2 8 7 9 6 → 0 1 2 8 7 9 6
8 7 9 6
6 7 9 8 → 0 1 2 6 7 9 8
7 9 8
7 8 9 → 0 1 2 6 7 8 9
7 9