Selection Sort-

- Selection sort is one of the easiest approaches to sorting.
- It is inspired from the way in which we sort things out in day to day life.
- It is an in-place sorting algorithm because it uses no auxiliary data structures while sorting.

How Selection Sort Works?

Consider the following elements are to be sorted in ascending order using selection sort-

6, 2, 11, 7, 5

Selection sort works as-

- It finds the first smallest element (2).
- It swaps it with the first element of the unordered list.
- It finds the second smallest element (5).
- It swaps it with the second element of the unordered list.
- Similarly, it continues to sort the given elements.

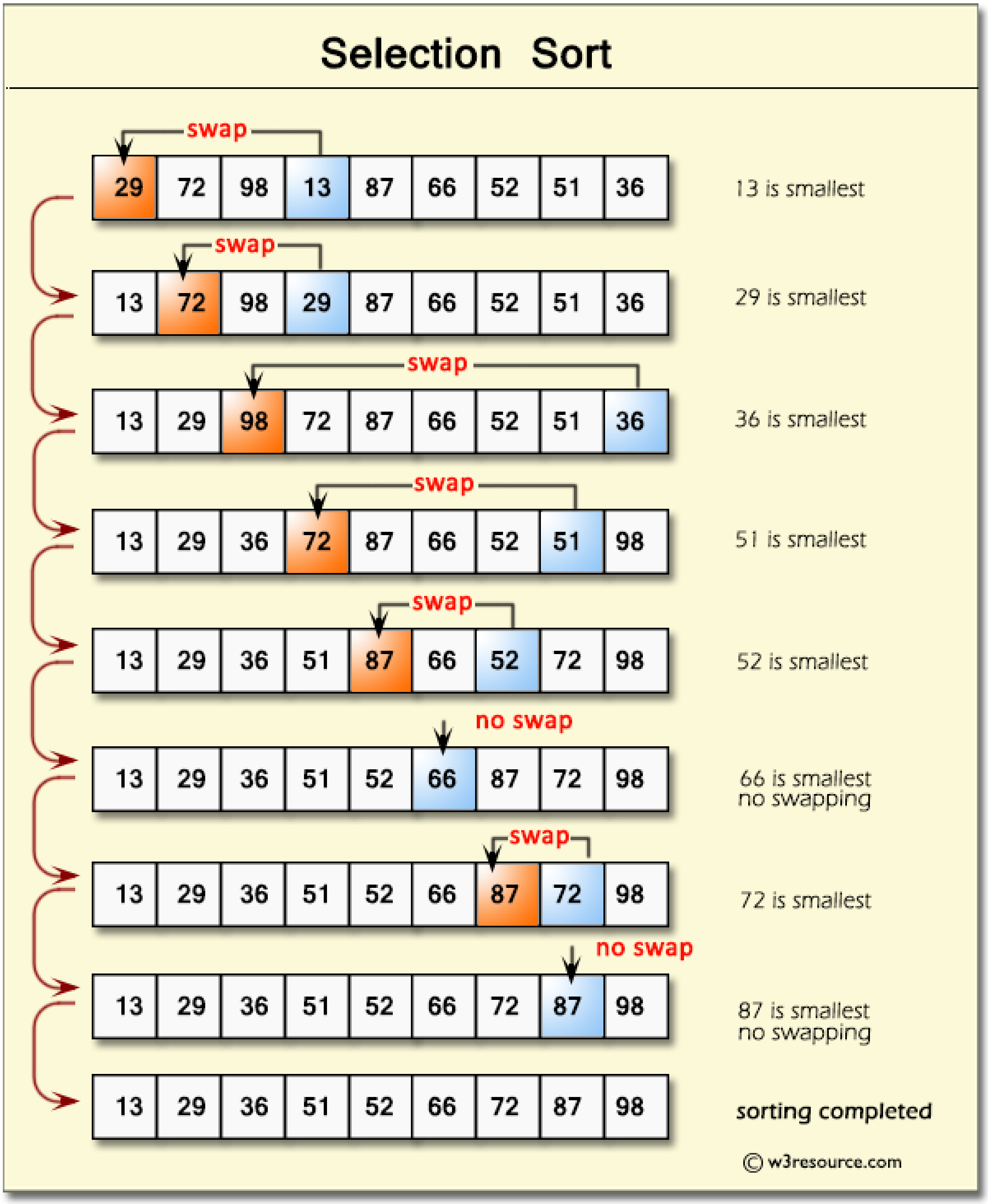
As a result, sorted elements in ascending order are-

2, 5, 6, 7, 11

Sorting ka matlab hota hai array ko ascending ya descending order m arrange karna

Generally hum ascending order me array ko arrande karte hai

Selection sort me har step me smallest element select karke usko proper position pe set kiya jata hai



Selection Sort

29	72	98	13	87	66	52	51	36	13 is smallest
13	72	98	29	87	66	52	51	36	29 is smallest
13	29	98	72	87	66	52	51	36	36 is smallest
13	29	36	72	87	66	52	51	98	51 is smallest
13	29	36	51	87	66	52	72	98	52 is smallest
13	29	36	51	52	66	87	72	98	66 is smallest no swapping
13	29	36	51	52	66	87	72	98	72 is smallest
13	29	36	51	52	66	72	87	98	87 is smallest no swapping
13	29	36	51	52	66	72	87	98	sorting completed

© w3resource.com

Jese ki image me show ho rha hai har step me smallest element search krke use proper position pe set kiya jaa rha hai

```
void selection_sort (int A[ ], int n) {
      // temporary variable to store the position of minimum element
//minimum naam ka variable har step me minimum variable ko store karega

           int minimum;

           // reduces the effective size of the array by one in  each iteration.

      for(int i = 0; i < n-1 ; i++)  {

        // assuming the first element to be the minimum of the unsorted array .
         minimum = i ;

       // gives the effective size of the unsorted  array .

         for(int j = i+1; j < n ; j++ ) {
            if(A[ j ] < A[ minimum ])  {             //finds the minimum element
            minimum = j ;
            }
         }
       // putting minimum element on its proper position.
       swap ( A[ minimum ], A[ i ]) ;
      }
  }
```

Bubble Sort

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

Example:
First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

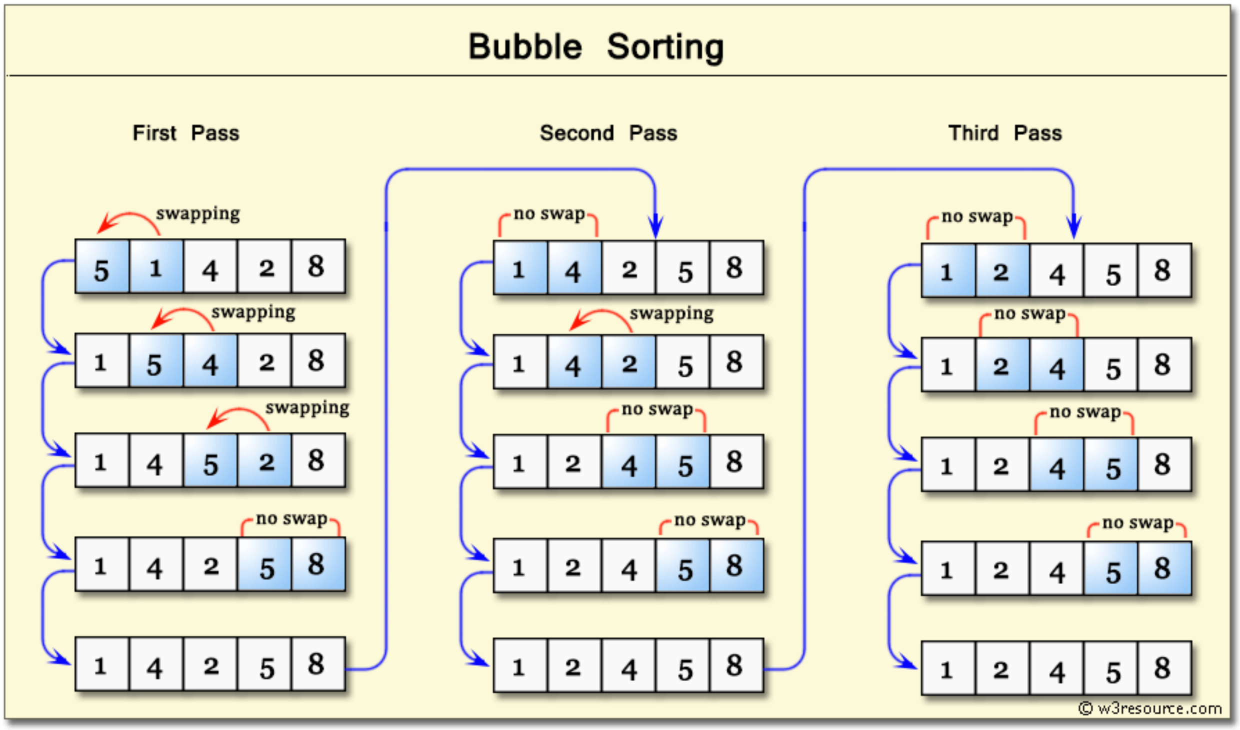
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

```
void bubble_sort( int A[ ], int n ) {
   int temp;
   for(int k = 0; k< n-1; k++) {
      // (n-k-1) is for ignoring comparisons of elements which have already been
compared in earlier iterations

      for(int i = 0; i < n-k-1; i++) {
         if(A[ i ] > A[ i+1] ) {
            // here swapping of positions is being done.
            temp = A[ i ];
            A[ i ] = A[ i+1 ];
            A[ i + 1] = temp;
         }
      }
   }
}
```

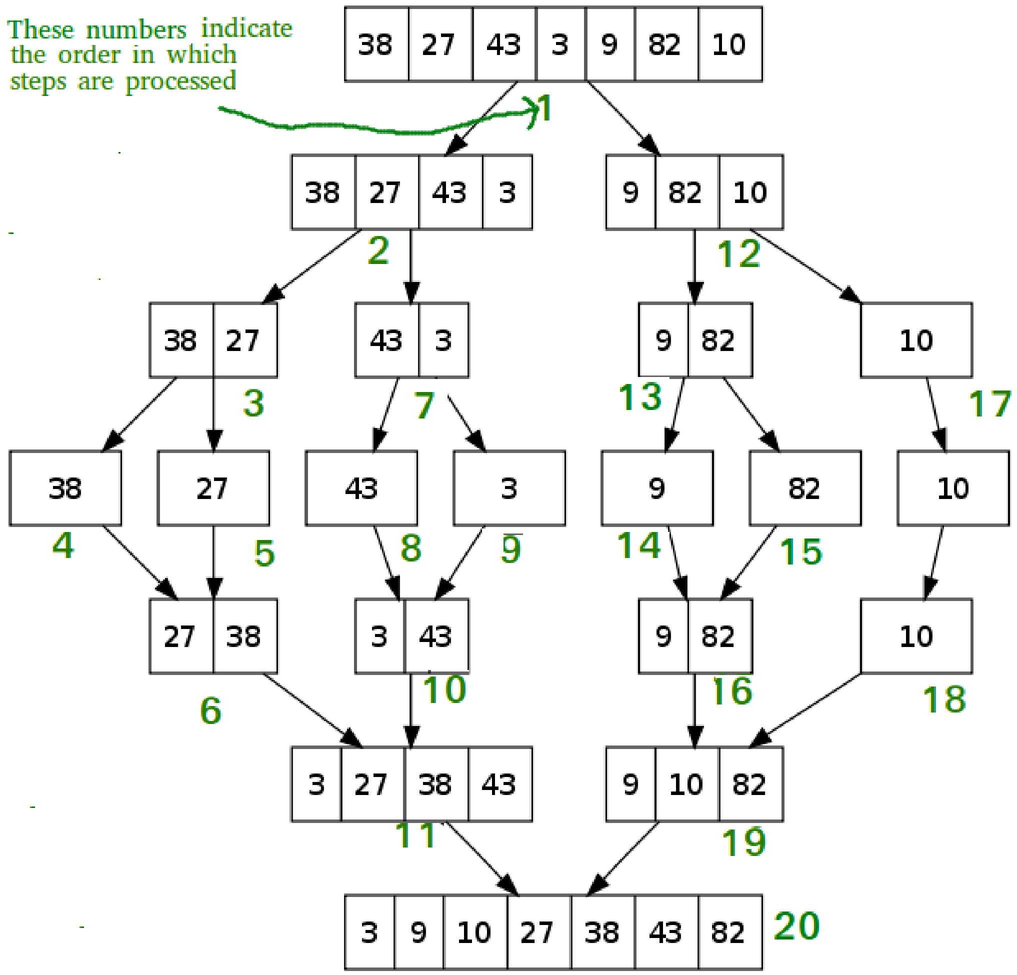


Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

```
MergeSort(arr[], l,  r)
If r > l
    1. Find the middle point to divide the array into two halves:
            middle m = (l+r)/2
    2. Call mergeSort for first half:
            Call mergeSort(arr, l, m)
    3. Call mergeSort for second half:
            Call mergeSort(arr, m+1, r)
    4. Merge the two halves sorted in step 2 and 3:
            Call merge(arr, l, m, r)
```

The following diagram from wikipedia shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



Tree Sort

Tree sort is a sorting algorithm that is based on Binary Search Tree data structure. It first creates a binary search tree from the elements of the input list or array and then performs an in-order traversal on the created binary search tree to get the elements in sorted order.

Algorithm:

Step 1: Take the elements input in an array.

Step 2: Create a Binary search tree by inserting data items from the array into the binary search tree.

Step 3: Perform in-order traversal on the tree to get the elements in sorted order.

For example follow the video link

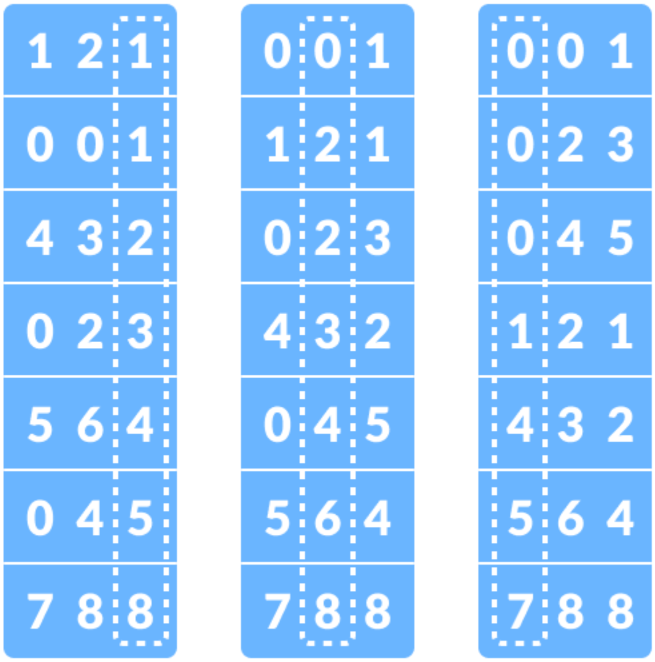
https://www.youtube.com/watch?v=n2MLjGeK7qA

Radix Sort

Radix sort is a sorting technique that sorts the elements by first grouping the individual digits of same **place value**. Then, sort the elements according to their increasing/decreasing order.

Suppose, we have an array of 8 elements. First, we will sort elements based on the value of the unit place. Then, we will sort elements based on the value of the tenth place. This process goes on until the last significant place.

Let the initial array be [121, 432, 564, 23, 1, 45, 788]. It is sorted according to radix sort as shown in the figure below.



sorting the integers according to units, tens and hundreds place digits

FOR EXAMPLE VISIT

https://www.youtube.com/watch?v=XiuSW_mEn7g

Quick Sort

It is also called **partition-exchange sort**. This algorithm divides the list into three main parts:

1. Elements less than the **Pivot** element
2. Pivot element(Central element)
3. Elements greater than the pivot element

Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this tutorial, we will take the rightmost element or the last element as **pivot**.

For example: In the array {52, 37, 63, 14, 17, 8, 6, 25}, we take 25 as **pivot**. So after the first pass, the list will be changed like this.

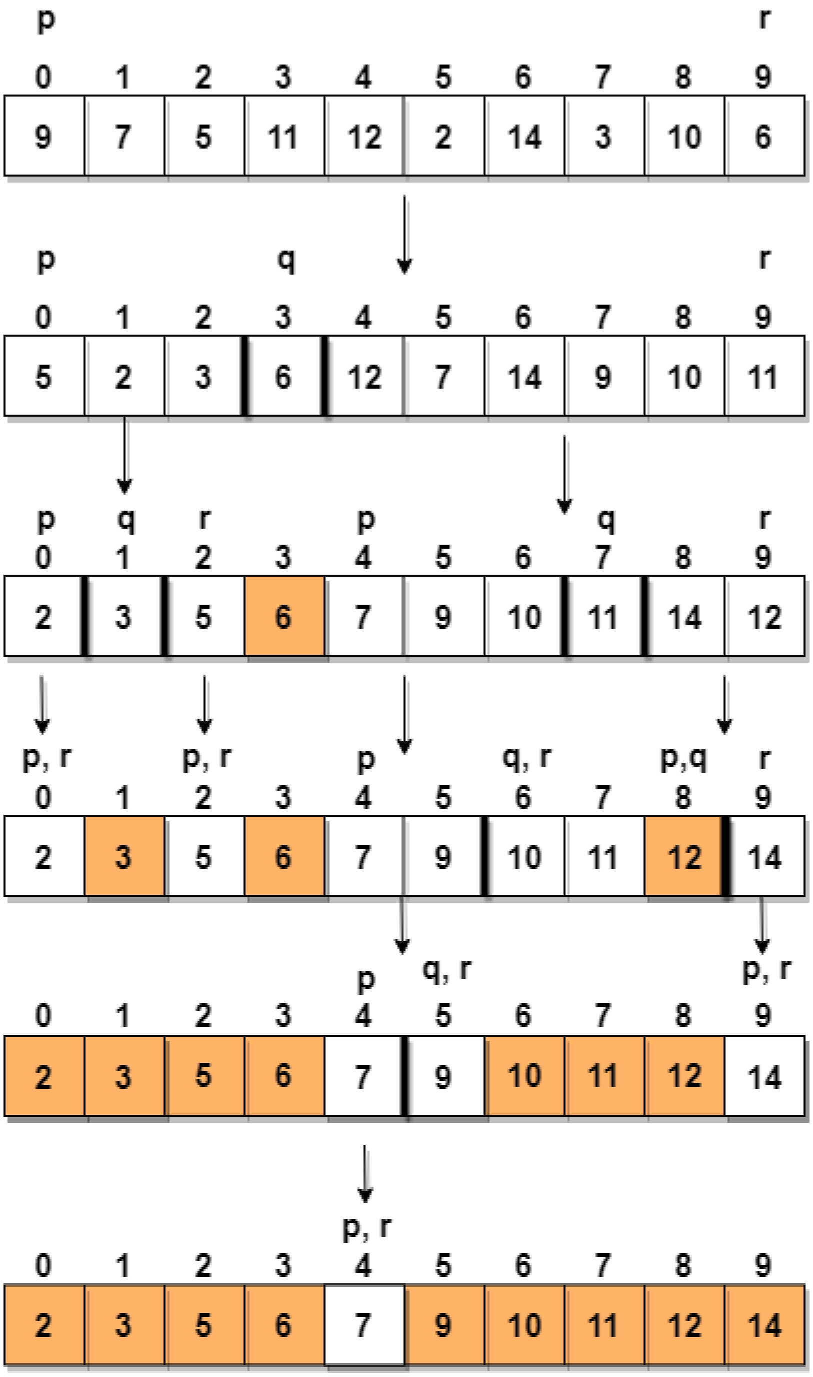
{6 8 17 14 **25** 63 37 52}

Hence after the first pass, pivot will be set at its position, with all the elements **smaller** to it on its left and all the elements **larger** than to its right. Now 6 8 17 14 and 63 37 52 are considered as two separate sunarrays, and same recursive logic will be applied on them, and we will keep doing this until the complete array is sorted.

Following are the steps involved in quick sort algorithm:

1. After selecting an element as **pivot**, which is the last index of the array in our case, we divide the array for the first time.
2. In quick sort, we call this **partitioning**. It is not simple breaking down of array into 2 subarrays, but in case of partitioning, the array elements are so positioned that all the elements smaller than the **pivot** will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of it.
3. And the **pivot** element will be at its final **sorted** position.
4. The elements to the left and right, may not be sorted.
5. Then we pick subarrays, elements on the left of **pivot** and elements on the right of **pivot**, and we perform **partitioning** on them by choosing a **pivot** in the subarrays.

> Let's consider an array with values {9, 7, 5, 11, 12, 2, 14, 3, 10, 6}
>
> Below, we have a pictorial representation of how quick sort will sort the given array.



In step 1, we select the last element as the **pivot**, which is 6 in this case, and call for partitioning, hence re-arranging the array in such a way that 6 will be placed in its final position and to its left will be all the elements less than it and to its right, we will have all the elements greater than it.

Then we pick the subarray on the left and the subarray on the right and select a **pivot** for them, in the above diagram, we chose `3` as pivot for the left subarray and `11` as pivot for the right subarray.

And we again call for `partitioning`.