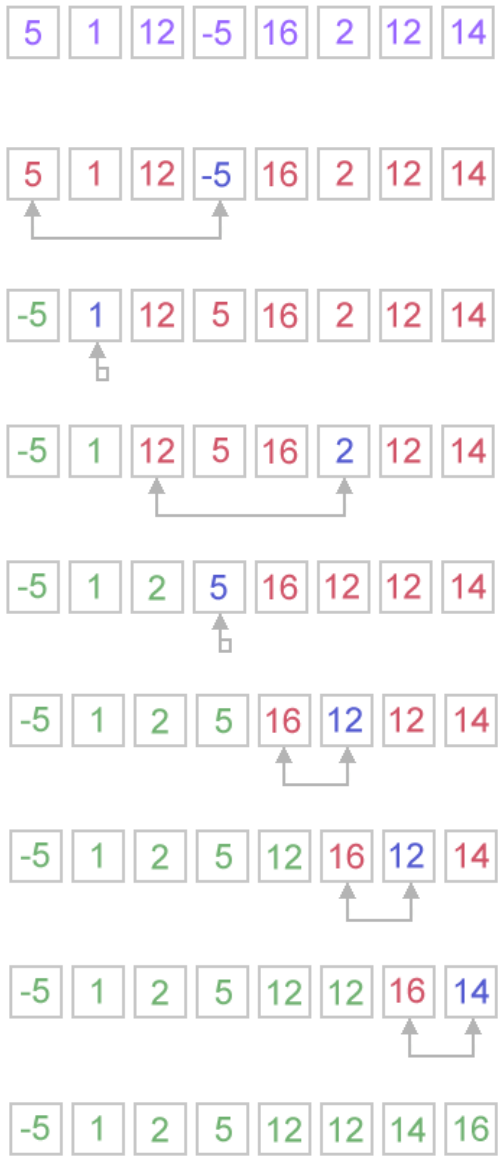
Los algoritmos más comunes para ordenamietno son:

*Selection sort in worst case O(n^2) :
The idea of algorithm is quite simple. Array is imaginary divided into two parts - sorted one and unsorted one. At the beginning, sorted part is empty, while unsorted one contains whole array. At every step, algorithm finds minimal element in the unsorted part and adds it to the end of the sorted one. When unsorted partbecomes empty, algorithm stops.

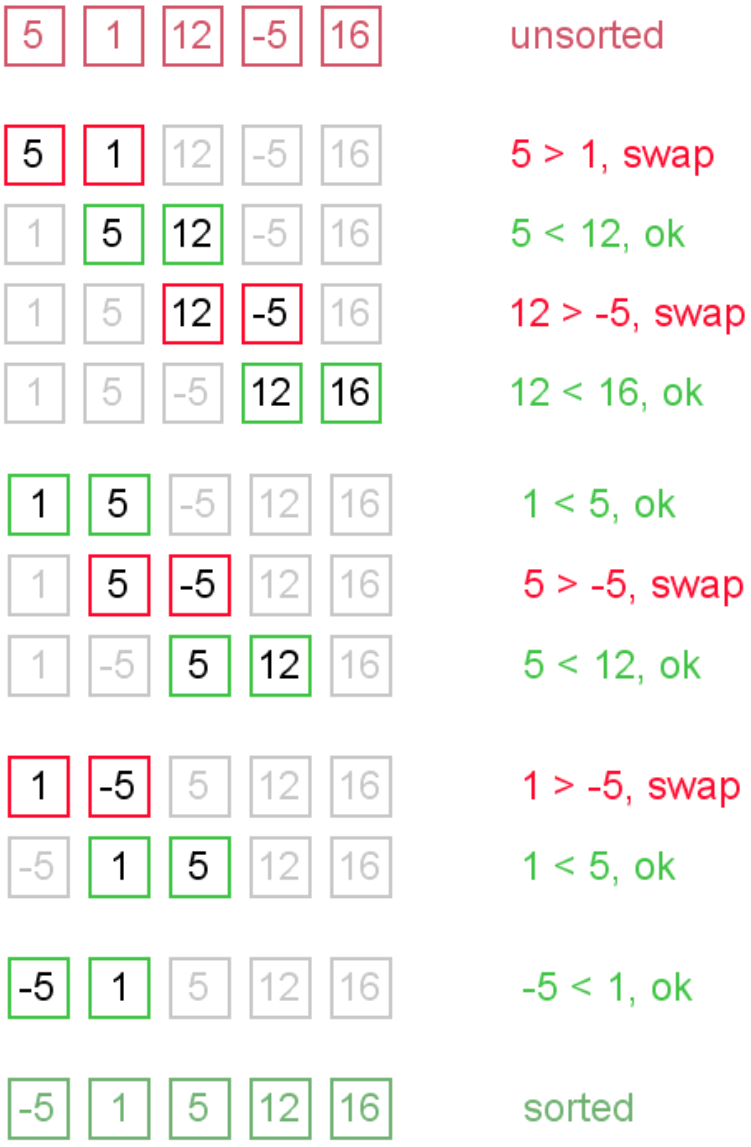
*Example.* Sort {5, 1, 12, -5, 16, 2, 12, 14} using selection sort.




```
public void selectionSort(int[] arr) {
      int i, j, minIndex, tmp;
      int n = arr.length;
      for (i = 0; i < n - 1; i++) {
            minIndex = i;
            for (j = i + 1; j < n; j++)
                  if (arr[j] < arr[minIndex])
                        minIndex = j;
            if (minIndex != i) {
                  tmp = arr[i];
                  arr[i] = arr[minIndex];
                  arr[minIndex] = tmp;
            }
```

```
        }
}
```

*Bubble sort in worst case O(n^2) :
In bubble sort, we start at the beginning of the array and swap the first two elements if the first is greater than the second. Then, we go to the next pair, and so on, continuously making sweeps of the array until it is sorted. In doing so, the smaller items slowly"bubble" up to the beginning of the list.

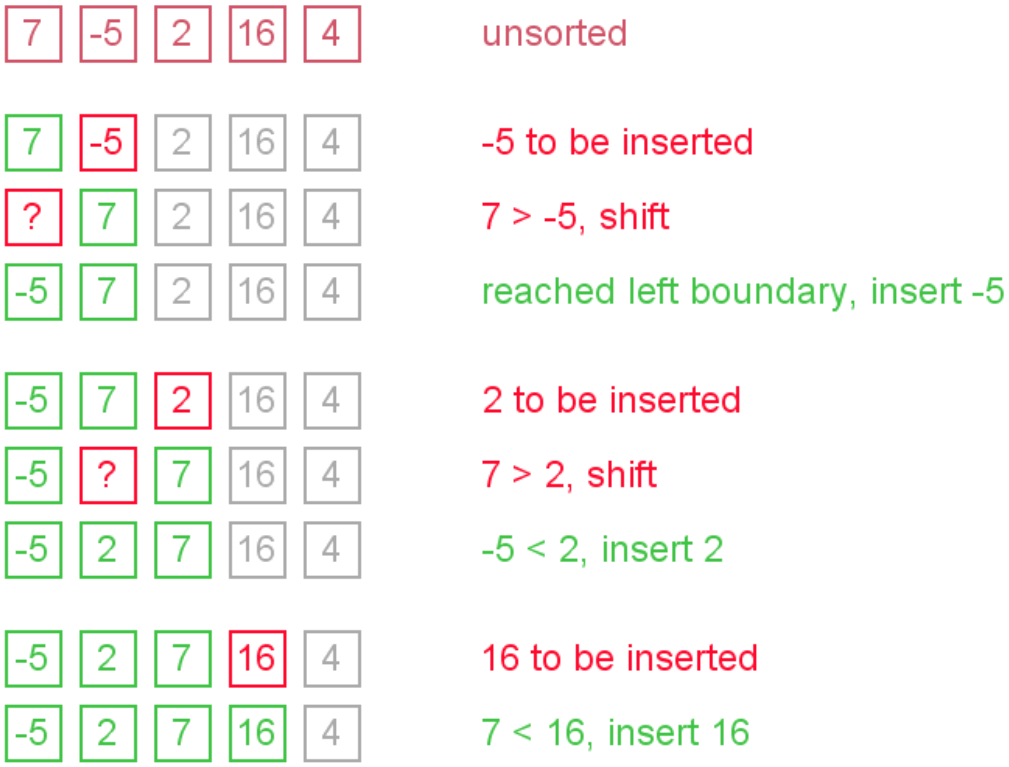


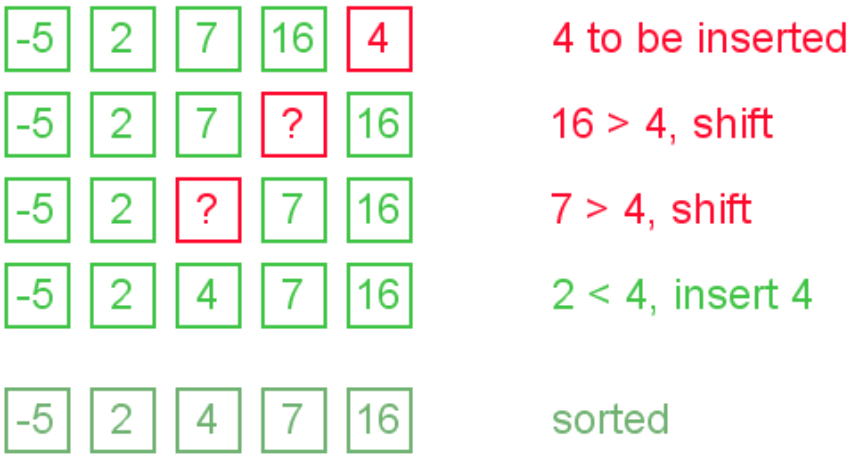

```
public class BubbleSort {

        public static int[] bubbleSort(int[] arr) {
            boolean swapped = true;
            int j = 0;
            int tmp;
            while (swapped) {
                swapped = false;
                j++;
                for (int i = 0; i < arr.length - j; i++) {
                    if (arr[i] > arr[i + 1]) {
                        tmp = arr[i];
```

```
                    arr[i] = arr[i + 1];
                    arr[i + 1] = tmp;
                    swapped = true;
                }
            }
        }
        return arr;
    }

    public static void main(String[] args) {

        int[] arr = {5,1,12,-5,16};
        int[] res = bubbleSort(arr);
        for(int i= 0; i< arr.length ; i++) {
            System.out.printf("%s,",res[i]);
        }
    }
}
```

*Insertion sort in worst case O(n^2)
Insertion sort algorithm somewhat resembles selection sort. Array is imaginary divided into two parts - sorted one and unsorted one. At the beginning, sorted part contains first element of the array and unsorted onecontains the rest. At every step, algorithm takes first element in the unsorted part and inserts it to the right place of the sorted one. When unsorted part becomes empty, algorithm stops.

*Example.* Sort {7, -5, 2, 16, 4} using insertion sort.






```
void insertionSort(int[] arr) {
      int i, j, newValue;
      for (i = 1; i < arr.length; i++) {
            newValue = arr[i];
            j = i;
            while (j > 0 && arr[j - 1] > newValue) {
                  arr[j] = arr[j - 1];
                  j--;
            }
            arr[j] = newValue;
      }
}
```

*Merge sort takes O(n log n) in worst case

Merge sort divides the array in half, sorts each of those halves, and then merges them back together. Each of those halves has the same sorting algorithm applied to it. Eventually, you are merging just two single element arrays. It is the "merge" part that does all the heavy lifting.
The merge method operates by copying all the elements from the target array segment into a helper array, keeping track of where the start of the left and right halves should be (helperLeft and helperRight).
We then iterate through helper, copying the smaller element from each half into the array. At the end, we copy any remaining elements into the target array.

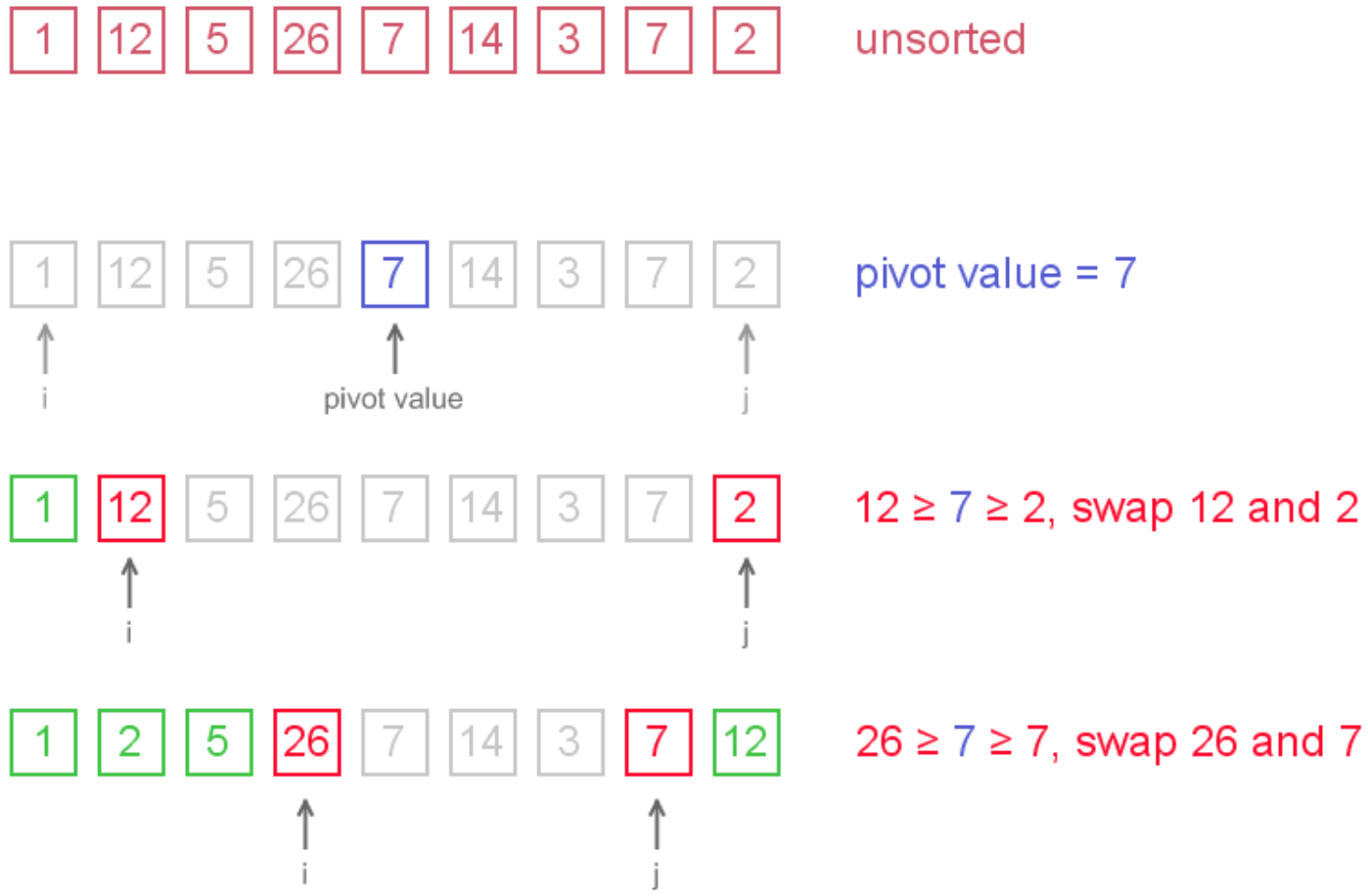
*Quick sort in avarage case O(n log n), in worst case O(n^2)
The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:

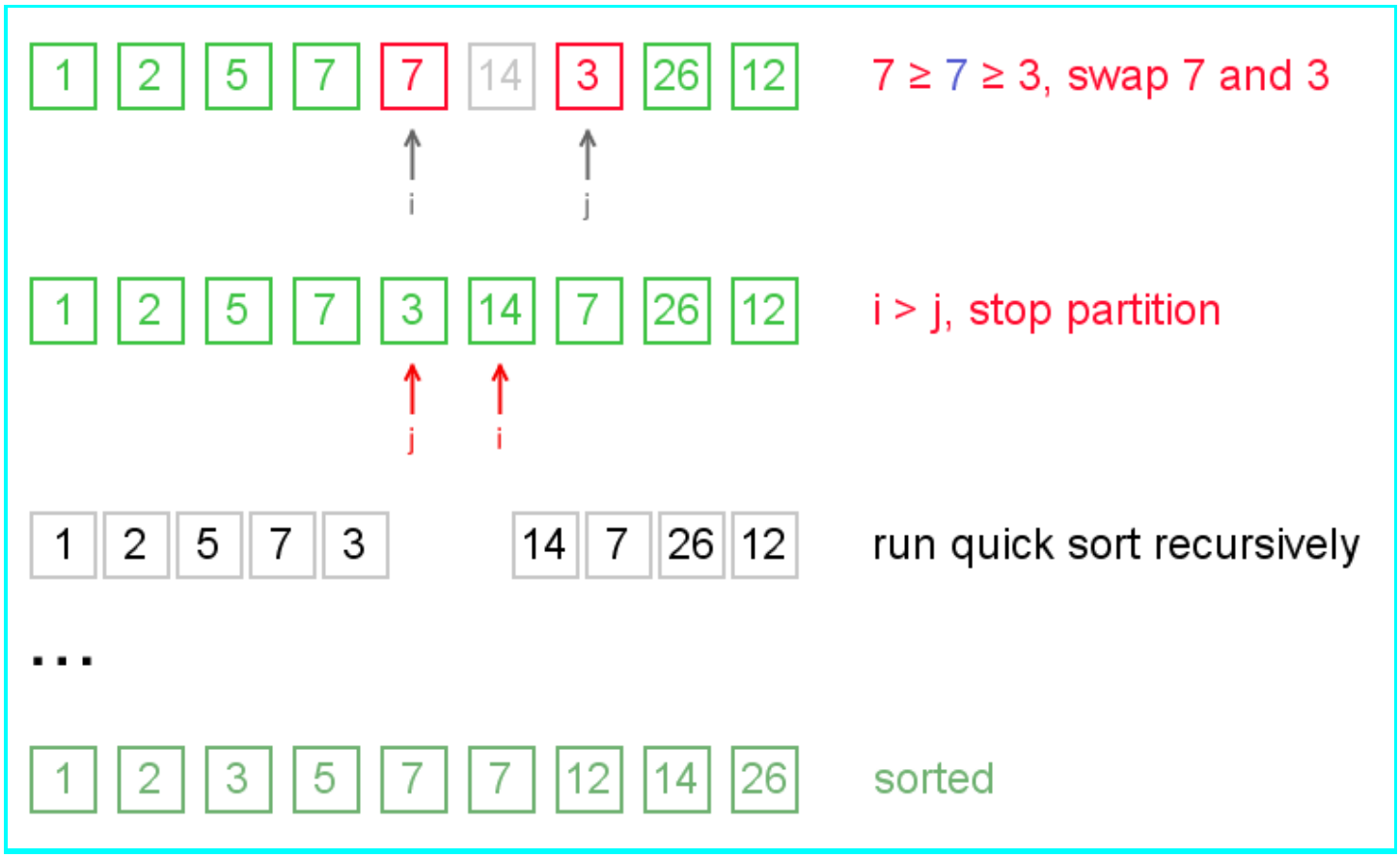
1. **Choose a pivot value.** We take the value of the middle element as pivot value, but it can be any value, which is in range of sorted values, even if it doesn't present in the array.
2. **Partition.** Rearrange elements in such a way, that all elements which are lesser than the pivot go to the left part of the array and all elements greater than the pivot, go to the right part of the array. Values equal to the pivot can stay in any part of the array. Notice, that array may be divided in non-equal parts.
3. **Sort both parts.** Apply quicksort algorithm recursively to the left and the right parts.

**Partition algorithm in detail**

There are two indices **i** and **j** and at the very beginning of the partition algorithm **i** points to the first element in the array and **j** points to the last one. Then algorithm moves **i** forward, until an element with value greater or equal to the pivot is found. Index **j** is moved backward, until an element with value lesser or equal to the pivot is found. If **i ≤ j** then they are swapped and i steps to the next position (**i + 1**), j steps to the previous one **(j - 1)**. Algorithm stops, when **i** becomes greater than **j**.
After partition, all values before **i-th** element are less or equal than the pivot and all values after **j-th** element are greater or equal to the pivot.

*Example.* Sort {1, 12, 5, 26, 7, 14, 3, 7, 2} using quicksort.






```
import java.util.Arrays;
import java.util.Random;
public class QuickSort  {

   private int[] numbers;
   private int number;
   private final static int SIZE = 7;
   private final static int MAX = 20;

   public void sort(int[] values) {
       // check for empty or null array
       if (values ==null || values.length==0){
           return;
       }
       this.numbers = values;
       number = values.length;
       quicksort(0, number - 1);
   }

   private void quicksort(int low, int high) {
       int i = low, j = high;
       // Get the pivot element from the middle of the list
       int pivot = numbers[low + (high-low)/2];

       // Divide into two lists
       while (i <= j) {
           // If the current value from the left list is smaller than the pivot
           // element then get the next element from the left list
           while (numbers[i] < pivot) {
               i++;
```

```
            }
            // If the current value from the right list is larger than the pivot
            // element then get the next element from the right list
            while (numbers[j] > pivot) {
                j--;
            }

            // If we have found a value in the left list which is larger than
            // the pivot element and if we have found a value in the right list
            // which is smaller than the pivot element then we exchange the
            // values.
            // As we are done we can increase i and j
            if (i <= j) {
                exchange(i, j);
                i++;
                j--;
            }
        }
        // Recursion
        if (low < j)
            quicksort(low, j);
        if (i < high)
            quicksort(i, high);
    }

    private void exchange(int i, int j) {
        int temp = numbers[i];
        numbers[i] = numbers[j];
        numbers[j] = temp;
    }

    public static void setUp(int[] num)  {
        Random generator = new Random();
        for (int i = 0; i < num.length ; i++) {
            num[i] = generator.nextInt(MAX);
        }
    }

    public static void main(String[] args) {

            QuickSort sorter = new QuickSort();
        //int[] test = { 5, 5, 6, 6, 4, 4, 5, 5, 4, 4, 6, 6, 5, 5 };
        int[] arr = new int[SIZE];
        setUp(arr);
        //sorter.sort(test);
        sorter.sort(arr);

        for(int i=0; i< arr.length ; i++) {
            System.out.printf("%s,", arr[i]);
        }
    }
}
```