```
/*
 * A small set of sorting algorithms, written in Java and C++
* Note  that they are written by a C++ beginner, may contain mistakes
* Or bad habits that have to be avoided
 * @author Kadir Can Çelik
 */
Sorting.java
public class Sorting {
   // A helper for the sorting algorithms
   // Changes the places of two numbers in an array
   public static void swap (int[] nums, int index1, int index2) {
      int temp;
      temp = nums [index2];
      nums [index2] = nums [index1];
      nums [index1] = temp;
   }

   // BubbleSort algorihm
   // Worst case performance   : O(n^2)
   // Best case performance    : O(n)
   // Checks all the numbers two by two and changes their order if needed
   // Does this repeatedly until all the numbers are sorted
   public static void bubbleSort (int[] nums) {
      boolean swappedThatTurn;
      swappedThatTurn = true;
      for (int outerCounter = nums.length - 1; outerCounter > 0 && swappedThatTurn; outerCounter --
) {
         swappedThatTurn = false;
```

```
            for (int innerCounter = 0; innerCounter < outerCounter; innerCounter ++) {

                if (nums [innerCounter] > nums [innerCounter + 1] ) {

                    swap (nums, innerCounter, innerCounter + 1);

                    swappedThatTurn = true;

                }

            }

        }

    }


    // SelectionSort algorihm

    // Worst case performance   : O(n^2)

    // Best case performance    : O(n^2)

    // Finds the smallest number in the range nums[0 .. outerCounter] and places the smallest at the
beginning of that range

    // Does this repeatedly until all the numbers are sorted

    // In this algorithm, we are sure that the first outerCounter numbers are sorted

    public static void selectionSort (int[] nums) {

        int outerCounter;

        int innerCounter;

        int smallestIndex;

        outerCounter = 0;

        while ( outerCounter < nums.length - 1) {

            innerCounter = outerCounter;

            smallestIndex = innerCounter;

            while ( innerCounter < nums.length ) {

                if ( nums [innerCounter] < nums [smallestIndex] ) {

                    smallestIndex = innerCounter;

                }
```

```
            innerCounter = innerCounter + 1;
        }
        swap (nums, outerCounter, smallestIndex);
        outerCounter = outerCounter + 1;
    }
}

// InsertionSort algorihm
// Worst case performance   : O(n^2)
// Best case performance    : O(n)
// Just like sorting cards, finding the correct place for a card at a time
// Does this repeatedly until all the numbers are sorted
public static void insertionSort (int[] nums) {
    int outerCounter;
    int innerCounter;
    int temp;
    outerCounter = 1;
    while (outerCounter < nums.length ) {
        innerCounter = outerCounter;
        temp = nums [innerCounter];
        while ( innerCounter > 0 && temp < nums[innerCounter - 1] ) {
            nums[innerCounter] = nums [innerCounter - 1];
            innerCounter = innerCounter - 1;
        }
        nums [innerCounter] = temp;
        outerCounter = outerCounter + 1;
    }
```

```java
}

// Partitioning method for quicksort

public static int partition (int[] nums, int lower, int higher) {

   int pivot;

   int outerCounter;

   int innerCounter;

   pivot = nums [higher];

   outerCounter = lower - 1;

   innerCounter = lower;

   while ( innerCounter < higher) {

      if ( nums [innerCounter] <= pivot ) {

         outerCounter = outerCounter + 1;

         swap (nums, outerCounter, innerCounter);

      }

      innerCounter = innerCounter + 1;

   }

   outerCounter = outerCounter + 1;

   swap (nums, outerCounter, higher);

   return outerCounter;

}


// QuickSort algorithm

// Worst case performance   : O(n^2)

// Best case performance    : O(n logn)

// Divides the array into smaller pieces and sorts them instead of sorting the full array

// Does this repeatedly until all the numbers are sorted

// In this algorithm we are sure that the elements before the pivot are smaller than the pivot
```

```
    private static void quickSort (int[] nums, int lower, int higher) {
        if ( lower < higher ) {
            int wall;
            wall = partition (nums, lower, higher);
            quickSort (nums, lower, wall - 1);
            quickSort (nums, wall, higher);
        }
    }
    // Just a wrapper for quicksort, in order to be able to call it with just an array
    public static void quickSort (int[] nums) {
        quickSort (nums, 0, nums.length - 1);
    }
}
```

sorting.h:

```
#ifndef SORTING_H
#define SORTING_H
void swap (int *x, int *y);
void bubbleSort (int[], int);
void selectionSort (int[], int);
void insertionSort (int[], int);
#endif
```

sorting.cpp:

```
void swap (int *x, int *y) {
        int temp;
        temp = *y;
        *y = *x;
        *x = temp;
```

```
}

   void bubbleSort (int[] nums, int size) {
      bool swappedThatTurn;
      swappedThatTurn = true;
      for (int outerCounter = size - 1; outerCounter > 0 && swappedThatTurn; outerCounter --) {
         swappedThatTurn = false;
         for (int innerCounter = 0; innerCounter < outerCounter; innerCounter ++) {
            if (nums [innerCounter] > nums [innerCounter + 1] ) {
               swap (&nums[innerCounter], &nums[innerCounter + 1]);
               swappedThatTurn = true;
            }
         }
      }
   }

void selectionSort (int nums[], int size) {
           int outerCounter;
           int innerCounter;
           int minThatTurn;
           outerCounter = 0;
           while ( outerCounter < size - 1 ) {
                      innerCounter = outerCounter;
                      minThatTurn = innerCounter;
                      while ( innerCounter < size ) {
                                 if ( nums[innerCounter] < nums [minThatTurn] ) {
                                    minThatTurn = innerCounter;
```

```
                }
                innerCounter = innerCounter + 1;
            }
            swap (&nums[outerCounter], &nums[minThatTurn]);
            outerCounter = outerCounter + 1;
        }
}
void insertionSort (int nums[], int size) {
        int outerCounter;
        int innerCounter;
        int temp;
        outerCounter = 1;
        while ( outerCounter < size ) {
          innerCounter = outerCounter;
          temp = nums[innerCounter];
          while ( innerCounter > 0 && temp < nums [innerCounter - 1] ) {
                nums [innerCounter] = nums[innerCounter - 1];
                innerCounter = innerCounter - 1;
            }
            nums [innerCounter] = temp;
            outerCounter = outerCounter + 1;
        }
}
```