Center for it and electronic Informatics

Technical University of Denmark DTU

Lautrupvang 15, 2750 Ballerup

# Basic Sorting Algorithms in JAVA

*Development and efficiency test of Insertion and Selection sort methods.*

**Supervisor:** **Roger Munck-Fairwood**



| Student name | Student number | Signature |
|---|---|---|
| James Testmann | S071954 | |
| Shicheng Dai | S133342 | |
| Sudhir chaurasiya | S137239 | |

## Problem and Program Briefly

The problem given is to implement and test sorting algorithms in JAVA. And to compare the result set with the theoretical O-notation of each algorithm along with a built in variety of QuickSort found in java.utils.arrays.sort().

Furthermore, it is required that the solution and result is properly documented such that a developer must be able to use our solution.

# Sorting Algorithms

The purpose of these algorithms is to sort an int array in ascending order.

There are a number of sorting algorithms developed to handle this problem which each have an efficiency between $O(n * \log(n))$ (such as quicksort, which is used as reference in this assignment) to $O(n^2)$ (such as the developed insertion and selection sort methods). Which is an average efficiency when sorting random arrays.

This efficiency O-notation is a way of indicating nested operations in an algorithm. So an algorithm that does n things n times is said to have an O-notation of $O(n^2)$.

## Selection Sort

One method for sorting arrays is selection sorting. This algorithm is always $O(n^2)$ and has no particular "best case" or "worst case" array when it comes to the speed to sort it. All equal size arrays will require the same amount of operations.

The operations is visualized by an illustration from a Google search in fig 1.

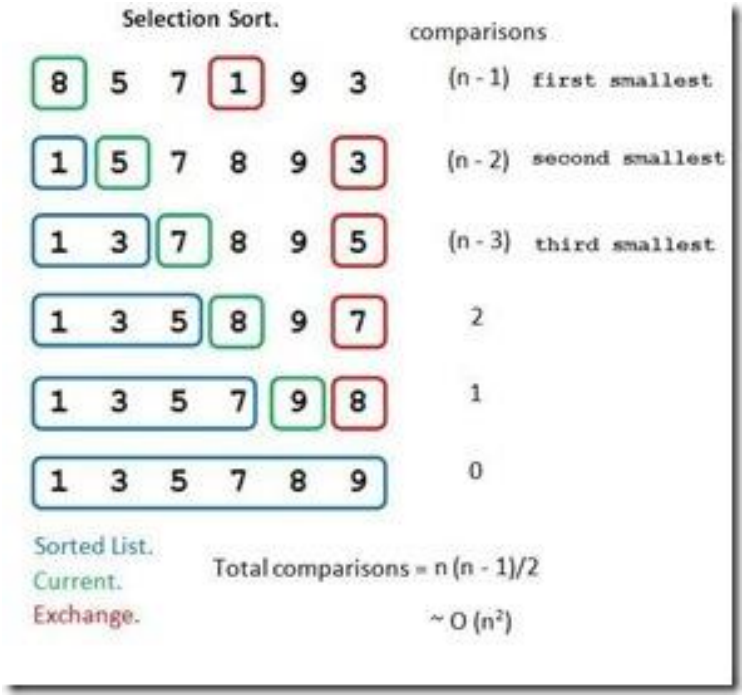


*fig 1*

The process is as follows:

1. The entire array is searched through for the smallest element (marked in red above).
2. The found element is swapped out with the first element (so the smallest is first).
3. The "search from" index is incremented (marked in blue above).
4. The next smallest element is searched for, etc.

Here it's obvious that no matter which array we sort, we will have to search the same number of elements. One optimization could be to only swap IF the smallest element is not in index "search from".

## Insertion Sort

Another method for sorting is insertion sorting. This algorithm is usually $O(n^2)$ but this does vary depending on the "tidiness" of the array to be sorted. An already sorted array will have an efficiency of $O(n)$ (being best case) and a descending array will be slower than an average random array.

The operations is visualized by an illustration from a Google search in fig 2.

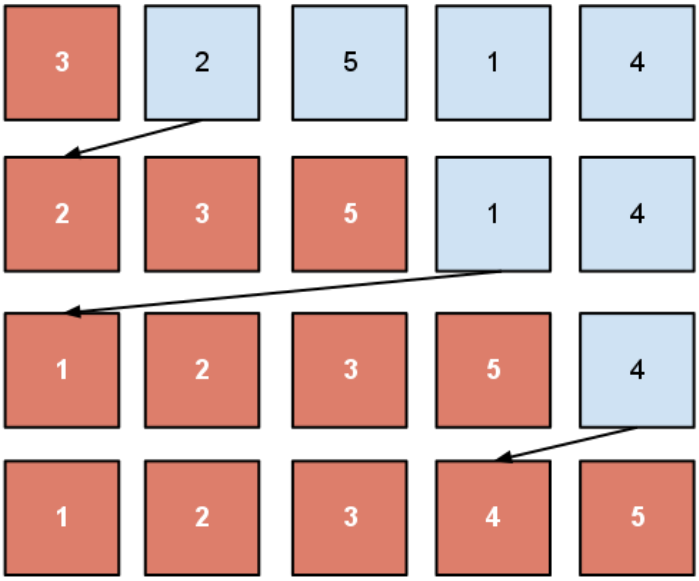


*fig 2*

The process is as follows:

1. The second element is compared with the element before it (first).
2. If it is bigger it will remain where it is, otherwise this element swaps places with the previous element. (Here we see the element swapping since 2 < 3).
3. IF the element swapped places with the previous element, it is then compared with the element before it (if such exist).
4. When the comparison tests true (meaning it IS bigger than the element before it) or it is the last element in the array, the "to sort" index is incremented.
5. The third element is compared with the element before it (second), etc.

Here we can see that only the "compare to" step is executed if the element fits, which makes an already sorted array very fast. On the other hand, a descending array will have to "swap and compare" through the entire array.

# Generate Arrays to Test

## Random Arrays

The random arrays are generated with Java's Math.random() method multiplied by an integer and then typecast to int. This gives a floored value from "0" to "n", where n is one less than the multiplied integer.

## Nearly Sorted Arrays

A nearly sorted array is interesting in the case of the Insertion sort algorithm, in that element swapping only occurs when the element being sorted is misplaced. This means that nearly sorted arrays are much faster with insertion sort than selection sort – which has to search all elements regardless of their tidiness.

Nearly sorted arrays start out as perfectly sorted arrays with elements going from "0" to "n" where n is the number of elements.

The array is then divided into chunks of 4 elements (variable) and these elements are then randomly swapped amongst themselves.

This creates a pseudo-random array where an element is never more than 3 spaces from its sorted index. As shown in fig 3.

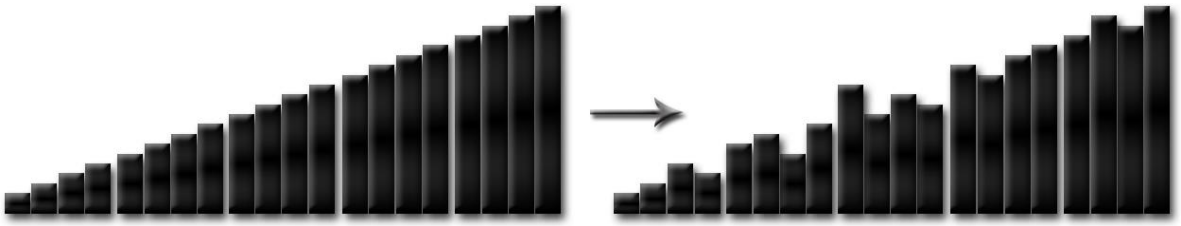

*fig 3*

## Descending (reverse order) Arrays

Descending arrays is a worst case scenario for Insertion sorting. Every element in the array is moved the maximum number of places backwards to sort the array.

A descending array is made with a for loop, giving an incrementing int i that is subtracted from the length of the array. In effect:

```
descendingArray[i] = arraySize - i;
```

# Noise and Result Resolution

The fewer elements you have in the arrays to be sorted, the less consistent any timing results become.

Our developed Selection and Insertion sort algorithms have average efficiency notation of $O(n^2)$.

In actuality the expression can be written as: $an^2 + bn + c$ with the squared part dominating the result as long as "n" is large. If "n" is small, then the result is too uncertain to properly compare. It's fair to speculate that other background processes take up too much CPU time compared to the sorting algorithms.

Therefore, a certain amount of noise is expected from “small” arrays in our test runs. The more elements in the arrays the closer we should expect the test result to match the theoretical efficiency.

Furthermore, we must expect that too high resolution in the test speed results will reveal only noise – even if taken as an average over a large sample size, this will be average noise. This is again because computer processes will fight over system resources and slightly randomize the time taken to sort each array.

During experimentations no consistency between runs was found with resolutions higher than 0.001 ms which is set as the cap. Results will be rounded off to that resolution.

As a note for time capture during tests; we used the java nanosecond counter to get good resolution with small arrays.

## Test Results

All tests were run a number of times as detailed in the Test Results below to get an average completion time.

This is done to minimize variation that might derive from process switching and array differences. The latter of which can be most impactful considering the possibility of a neatly sorted “random” test array using the Math.random() method.

| Array Sorting | | N1 = 100 elements<br>T(N1) run 5.000.000 times | N3 = 1000 elements<br>T(N2) run 50.000 times | N4 = 10000 elements<br>T(N3) run 500 times | N5 = 100000 elements<br>T(N4) run 50 times | N5 = 200000 elements<br>T(N5) run 20 times | **Timerelation for N2 / N1** | O-notation for N2 / N1 | **Timerelation for N3 / N2** | O-notation for N3 / N2 | **Timerelation for N4 / N3** | O-notation for N4 / N3 | **Timerelation for N5 / N4** | O-notation for N5 / N4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Random Array | Selection Sort ($O(n^2)$) | 0.008ms | 0.515ms | 44.210ms | 4373.447ms | 17660.703ms | **63.91** | 100 | **85.85** | 100 | **98.92** | 100 | **4.04** | 4 |
| | Insertion Sort ($O(n^2)$) | 0.005ms | 0.401ms | 23.900ms | 2394.452ms | 9666.058ms | **81.33** | 100 | **59.65** | 100 | **100.18** | 100 | **4.04** | 4 |
| | Insertion Sort 2 ($O(n^2)$) | 0.003ms | 0.218ms | 20.441ms | 2030.723ms | 7990.015ms | **67.94** | 100 | **93.55** | 100 | **99.34** | 100 | **3.93** | 4 |
| | Quick Sort (O(n * log(n))) | 0.003ms | 0.037ms | 0.485ms | 6.293ms | 14.085ms | **14.25** | 15 | **12.97** | 13.33 | **12.98** | 12.5 | **2.24** | 2.12 |
| Nearly Sorted Array | Selection Sort ($O(n^2)$) | 0.006ms | 0.482ms | 40.111ms | 4002.991ms | 16202.244ms | **78.95** | 100 | **83.24** | 100 | **99.8** | 100 | **4.05** | 4 |
| | Insertion Sort ($O(n^2)$) | 0.001ms | 0.007ms | 0.069ms | 0.674ms | 1.322ms | **10.2** | 100 | **10.57** | 100 | **9.76** | 100 | **1.96** | 4 |
| | Insertion Sort 2 ($O(n^2)$) | 0.001ms | 0.007ms | 0.065ms | 0.624ms | 1.237ms | **9.83** | 100 | **9.5** | 100 | **9.57** | 100 | **1.98** | 4 |
| | Quick Sort (O(n * log(n))) | 0.001ms | 0.013ms | 0.145ms | 1.698ms | 3.547ms | **12.9** | 15 | **10.82** | 13.33 | **11.74** | 12.5 | **2.09** | 2.12 |
| Descending Array | Selection Sort ($O(n^2)$) | 0.005ms | 0.436ms | 43.739ms | 4367.336ms | 17591.862ms | **92.77** | 100 | **100.25** | 100 | **99.85** | 100 | **4.03** | 4 |
| | Insertion Sort ($O(n^2)$) | 0.007ms | 0.788ms | 47.703ms | 4825.143ms | 19441.902ms | **105.42** | 100 | **60.55** | 100 | **101.15** | 100 | **4.03** | 4 |
| | Insertion Sort 2 ($O(n^2)$) | 0.005ms | 0.433ms | 40.733ms | 4055.208ms | 15998.765ms | **91.42** | 100 | **94.16** | 100 | **99.56** | 100 | **3.95** | 4 |
| | Quick Sort (O(n * log(n))) | 0.001ms | 0.001ms | 0.013ms | 0.133ms | 0.353ms | **2.61** | 15 | **9.14** | 13.33 | **10.51** | 12.5 | **2.65** | 2.12 |

## Test Results

The time relation is calculated as: $O(n^2) = \frac{{N_2}^2}{{N_1}^2}$ for the insertion and selection sort, and $O(n * \log(n)) = \frac{N_2 * \log(N_2)}{N_1 * \log(N_1)}$ for the quicksort method. This is done to show the increase in processing time over an increase in array size as a ratio.

As it was expected, the measured time relation only matches the theoretical efficiency when run with a large number of elements! The less elements in the arrays, the less the $n^2$ part of our efficiency equation $an^2 + bn + c$ matters, and the more unpredictable the results become.

However, the tests with large number or elements matches the theoretical number closely. The small discrepancies are explained by the lingering bn + c part of the equation not being dominated enough by the squared part. And the simplistic nature of the equation in lieu of the executed code.

It is furthermore noticeable how Insertion sorting is very efficient with nearly sorted arrays and how descending arrays are slower than random. This is expected.

NOTE: The second Insertion sort method is made to simplify the code of the initially developed method, and is faster too.

# Class Descriptions

Here is a brief description of how to use the developed java classes. Only important methods and tests are discussed.

## RunTests.class

***Main() method***

The *main()* method implements a number of **arrayProperties** objects which is a value object containing size of test arrays, number of sorting iterations, and the difference span n of integers possible (from 0 to n-1).

***executeTestsAndPrintAverageTime*(*arrayProperties arrayProperties, Sortable method)***

This method runs the sorting algorithm with arrays as specified in the method arguments. The sorting algorithm class must implement the **Sortable interface** which has one signature: "sort(Integer[] array)".

The method then prints the average time to sort arrays in the System.out log window.

## Tester.class

***Tester constructor***

The constructor takes an arrayProperties object to initialize the array data.

***run(Sortable sortMethod)***

This method has a for loop to run the sorting algorithm the number of times given by the arrayProperties object. The loop has a simple life of:

- Create random, nearlySorted, and descending arrays.
- Start nanosecond timer.
- Sort arrays.
- Stop nanosecond timer and increment elapsedTimes field.

This method will return the total elapsed time for all array sorts.

The interface argument is used to call the doSort() method which handles the timers and runs the sort algorithm "sortMethod".

## SelectionSort.class

***sort(Integer[] arrayToSort)***

The method loads an array as an argument and with a for loop it does the following two things and then increments i:

- Find smallest element in the array, from element "i" (0 to start)
- Swap found element with element "i".

Last element in the array is not searched, as this must always be in the right position.

When done the method returns the sorted array.

## InsertionSort.class

***sort(Integer[] arrayToSort)***
The method loads an array as an argument and with a for loop that calls method shiftElements(). When done the method returns the sorted array.

***shiftElements()***
This method first checks if the second object in the array is smaller than the previous element and saves the result to a boolean. Using this boolean it swaps the second element with the previous one and does the check again IF there are any elements left to check.

When the element is placed, the for loop in sort() is incremented and the next element is shifted back.

## InsertionSortFSM.class

This was designed as a more readable version of InsertionSort and turned out to be faster too as it fixed an issue with the "swap and compare" algorithm in the first iteration of this class.

## BuiltInQuickSort.class

***sort(Integer[] arrayToSort)***
The method runs java.util.Arrays.sort(array) on the input array and returns the sorted array.

## UnitTest Classes

The **tester class** has a test designed to create and sort one of each type of array using the developed algorithms and then compare that result with the same arrays sorted using Java's built in method.

A separate test method was written in Tester.class to perform this operation because the sorted arrays needed to be returned back to the test class.

This is not ideal since it relies on the stability and correctness of Java's built in sort method – but it was the best solution found within the scope of this assignment.

Common for all **algorithm test** classes is that they each have a random array of 100 elements from 0 to 500 with a maximum of 3 repeated elements (element value '119'). And a sorted array of the same 100 elements.

The sort method is called and the resulting array is compared to the expected array.

**A test suite is written to run all test classes at once.**

The written code as of publication is attached as a printed appendix.

# *Appendix 1*

*Here follows the two execution classes.*

*These are extensive and show no part of the actual algorithm.*

*For the algorithms move ahead to the next appendix.*

```java
/**
 * @author James Testmann, S071954
 */
import Helpers.*;
import Sorting.*;

public class RunTests
{

    public static void main(String[] args)
    {
        arrayProperties veryShortTest = new arrayProperties();
        veryShortTest.sizeOfSortArray = 100;
        veryShortTest.numberOfTimesToRun = 5000000; //5.000.000
        veryShortTest.testFromZeroTo = 500;

        arrayProperties shortTest = new arrayProperties();
        shortTest.sizeOfSortArray = 1000;
        shortTest.numberOfTimesToRun = 50000; //50.000
        shortTest.testFromZeroTo = 5000;

        arrayProperties longTest = new arrayProperties();
        longTest.sizeOfSortArray = 10000;
        longTest.numberOfTimesToRun = 500; //500
        longTest.testFromZeroTo = 50000;

        arrayProperties veryLongTest = new arrayProperties();
        veryLongTest.sizeOfSortArray = 100000;
        veryLongTest.numberOfTimesToRun = 50; //50
        veryLongTest.testFromZeroTo = 500000;

        arrayProperties crazyLongTest = new arrayProperties();
        crazyLongTest.sizeOfSortArray = 200000;
        crazyLongTest.numberOfTimesToRun = 20; //20
        crazyLongTest.testFromZeroTo = 1000000;

        System.out.println("Selection Sorting Method:");
        System.out.println("");

        executeTestsAndPrintAverageTime(veryShortTest, new SelectionSort());
        executeTestsAndPrintAverageTime(shortTest, new SelectionSort());
        executeTestsAndPrintAverageTime(longTest, new SelectionSort());
        executeTestsAndPrintAverageTime(veryLongTest, new SelectionSort());
        executeTestsAndPrintAverageTime(crazyLongTest, new SelectionSort());

        System.out.println("");
        System.out.println("Insertion Sorting Method:");
        System.out.println("");

        executeTestsAndPrintAverageTime(veryShortTest, new InsertionSort());
        executeTestsAndPrintAverageTime(shortTest, new InsertionSort());
        executeTestsAndPrintAverageTime(longTest, new InsertionSort());
        executeTestsAndPrintAverageTime(veryLongTest, new InsertionSort());
        executeTestsAndPrintAverageTime(crazyLongTest, new InsertionSort());

        System.out.println("");
        System.out.println("Builtin Quicksort Sorting Method:");
        System.out.println("");

        executeTestsAndPrintAverageTime(veryShortTest, new BuiltInQuickSort());
        executeTestsAndPrintAverageTime(shortTest, new BuiltInQuickSort());
        executeTestsAndPrintAverageTime(longTest, new BuiltInQuickSort());
        executeTestsAndPrintAverageTime(veryLongTest, new BuiltInQuickSort());
        executeTestsAndPrintAverageTime(crazyLongTest, new BuiltInQuickSort());

        System.out.println("");
        System.out.println("Second Insertion Sorting Method:");
        System.out.println("");

        executeTestsAndPrintAverageTime(veryShortTest, new InsertionSortFSM());
        executeTestsAndPrintAverageTime(shortTest, new InsertionSortFSM());
```

```
        executeTestsAndPrintAverageTime(longTest, new InsertionSortFSM());
        executeTestsAndPrintAverageTime(veryLongTest, new InsertionSortFSM());
        executeTestsAndPrintAverageTime(crazyLongTest, new InsertionSortFSM());
    }

    public static void executeTestsAndPrintAverageTime(arrayProperties arrayProperties, Sortable method)
    {
        long[] executionTimes = new Tester(arrayProperties).run(method);

        long averageTime = executionTimes[0] / arrayProperties.numberOfTimesToRun;
        double averageTimeMiliseconds = ((double) averageTime) / 1000000;
        System.out.println("Scrambled arrays: Average Runtime of " + averageTimeMiliseconds + "ms, average over " +
arrayProperties.numberOfTimesToRun + " runs with " + arrayProperties.sizeOfSortArray + " elements in each array.");

        averageTime = executionTimes[1] / arrayProperties.numberOfTimesToRun;
        averageTimeMiliseconds = ((double) averageTime) / 1000000;
        System.out.println("Nearly Sorted arrays: Average Runtime of " + averageTimeMiliseconds + "ms, average over " +
arrayProperties.numberOfTimesToRun + " runs with " + arrayProperties.sizeOfSortArray + " elements in each array.");

        averageTime = executionTimes[2] / arrayProperties.numberOfTimesToRun;
        averageTimeMiliseconds = ((double) averageTime) / 1000000;
        System.out.println("Descending arrays: Average Runtime of " + averageTimeMiliseconds + "ms, average over " +
arrayProperties.numberOfTimesToRun + " runs with " + arrayProperties.sizeOfSortArray + " elements in each array.");
        System.out.println("------------");
    }
}
```

---

```
/**
 * @author James Testmann, S071954
 */
import Helpers.*;
import Sorting.*;
import java.util.ArrayList;

public class Tester
{

    private final int numberOfTimesToRun;
    private final int arraySize;
    private final int testFromZeroTo;

    private int[] scrambledArray;
    private int[] nearlySortedArray;
    private int[] descendingArray;

    private volatile long startTime;
    private volatile long[] elapsedTimes;

    private Sortable sortMethod;

    public Tester(arrayProperties arrayProperties)
    {
        elapsedTimes = new long[3];

        numberOfTimesToRun = arrayProperties.numberOfTimesToRun;
        arraySize = arrayProperties.sizeOfSortArray;
        testFromZeroTo = arrayProperties.testFromZeroTo;
    }

    public long[] run(Sortable sortMethod)
    {
        this.sortMethod = sortMethod;

        for (int i = 0; i < numberOfTimesToRun; ++i)
        {
            createRandomArray();
            createNearlySortedArray();
            createDescendingArray();
```

```
            doSort(Method.scrambledArray, scrambledArray);
            doSort(Method.nearlySortedArray, nearlySortedArray);
            doSort(Method.descendingArray, descendingArray);
        }
        return elapsedTimes;
    }

    private void createRandomArray()
    {
        scrambledArray = new int[arraySize];
        for (int i = 0; i < arraySize; ++i)
        {
            scrambledArray[i] = (int) (Math.random() * testFromZeroTo);
        }
    }

    private void createNearlySortedArray()
    {
        nearlySortedArray = new int[arraySize];
        for (int i = 0; i < arraySize; ++i)
        {
            nearlySortedArray[i] = i;
        }
        int chunkSize = 4;
        for (int i = 0; i < arraySize; i += chunkSize)
        {
            int cap = Math.min(chunkSize + i, arraySize);
            for(int j = i; j < (cap - 1); ++j)
            {
                int swapTarget = (int) (Math.random() * (cap - j)) + j;
                swapElements(nearlySortedArray, j, swapTarget);
            }
        }
    }

    private void swapElements(int[] array, int from, int to)
    {
        int placeholder = array[to];
        array[to] = array[from];
        array[from] = placeholder;
    }

    private void createDescendingArray()
    {
        descendingArray = new int[arraySize];
        for (int i = 0; i < arraySize; ++i)
        {
            descendingArray[i] = arraySize - i;
        }
    }

    private void doSort(Method method, int[] arrayToSort)
    {
        startTimer();
        sortMethod.sort(arrayToSort);
        stopTimerAndSaveElapsedTime(method);
    }

    private void startTimer()
    {
        startTime = System.nanoTime();
    }

    private void stopTimerAndSaveElapsedTime(Method method)
    {
        switch (method)
        {
            case scrambledArray:
                elapsedTimes[0] += System.nanoTime() - startTime;
                break;
```

```
            case nearlySortedArray:
                elapsedTimes[1] += System.nanoTime() - startTime;
                break;
            case descendingArray:
                elapsedTimes[2] += System.nanoTime() - startTime;
                break;
        }
    }

    public ArrayList<int[]> returnSortedArraysForTesting(Sortable sortMethod)
    {
        this.sortMethod = sortMethod;
        ArrayList<int[]> returnArrays = new ArrayList<>();

        createRandomArray();
        createNearlySortedArray();
        createDescendingArray();

        doSort(Method.scrambledArray, scrambledArray);
        doSort(Method.nearlySortedArray, nearlySortedArray);
        doSort(Method.descendingArray, descendingArray);

        returnArrays.add(scrambledArray);
        returnArrays.add(nearlySortedArray);
        returnArrays.add(descendingArray);

        return returnArrays;
    }

    public ArrayList<int[]> returnComparisonArraysForTesting(ArrayList<int[]> arrays)
    {
        ArrayList<int[]> returnArrays = new ArrayList<>();

        java.util.Arrays.sort(arrays.get(0));
        java.util.Arrays.sort(arrays.get(1));
        java.util.Arrays.sort(arrays.get(2));

        returnArrays.add(arrays.get(0));
        returnArrays.add(arrays.get(1));
        returnArrays.add(arrays.get(2));

        return returnArrays;
    }

    private void printArray(int[] array)
    {
        for (int element : array)
        {
            System.out.println(Integer.toString(element));
        }
    }
}

enum Method
{
    scrambledArray, nearlySortedArray, descendingArray
}
```

# *Appendix 2*

*Here follows the three algorithm classes.*

*These are the sorting algorithms developed as a solution to the problem given.*

```java
/**
 * @author James Testmann, S071954
 */
public class SelectionSort implements Sortable
{
    private int[] arrayToSearch;
    private int currentSortedIndex;
    private int currentSmallestElementID;

    @Override
    public int[] sort(int[] arrayToSort)
    {
        this.arrayToSearch = arrayToSort;
        currentSortedIndex = 0;
        for (; currentSortedIndex < arrayToSearch.length - 1; ++currentSortedIndex)
        {
            searchSmallestElementID();
            swapElements();
        }
        return arrayToSearch;
    }

    private void searchSmallestElementID()
    {
        currentSmallestElementID = currentSortedIndex;
        for (int i = currentSortedIndex; i < arrayToSearch.length; ++i)
        {
            testIfSmallerAndStoreIndex(arrayToSearch[currentSmallestElementID], arrayToSearch[i], i);
        }
    }

    private void testIfSmallerAndStoreIndex(int storedElement, int testElement, int testIndex)
    {
        if (testElement <= storedElement)
        {
            currentSmallestElementID = testIndex;
        }
    }

    private void swapElements()
    {
        int elementPlaceHolder = arrayToSearch[currentSmallestElementID];
        arrayToSearch[currentSmallestElementID] = arrayToSearch[currentSortedIndex];
        arrayToSearch[currentSortedIndex] = elementPlaceHolder;
    }
}
```

---

```java
/**
 * @author James Testmann, S071954
 */
public class InsertionSort implements Sortable
{

    private int[] arrayToSearch;
    private int currentSortIndex;
    private int currentInsertElementID;

    @Override
    public int[] sort(int[] array)
    {
        arrayToSearch = array;
        currentSortIndex = 1;
        for (; currentSortIndex < arrayToSearch.length; ++currentSortIndex)
        {
            shiftElements();
        }
        return arrayToSearch;
    }
```

```
    private void shiftElements()
    {
        currentInsertElementID = currentSortIndex;
        boolean keepShifting = isCurrentElementSmallerThanPrevious();

        while (keepShifting && currentInsertElementID > 0)
        {
            keepShifting = isCurrentElementSmallerThanPrevious();

            if (keepShifting)
            {
                shiftOneElementDown();
                --currentInsertElementID;
            }
        }
    }

    private boolean isCurrentElementSmallerThanPrevious()
    {
        return arrayToSearch[currentInsertElementID] < arrayToSearch[currentInsertElementID - 1];
    }

    private void shiftOneElementDown()
    {
        int smallElementPlaceHolder = arrayToSearch[currentInsertElementID];
        arrayToSearch[currentInsertElementID] = arrayToSearch[currentInsertElementID - 1];
        arrayToSearch[currentInsertElementID - 1] = smallElementPlaceHolder;
    }
}
```

---

```
/**
 * @author James Testmann, S071954
 */
public class InsertionSortFSM implements Sortable
{

    private int[] arrayToSearch;
    private int currentSortIndex;
    private int currentExtractedElement;
    private int openSpaceIndex;

    @Override
    public int[] sort(int[] array)
    {
        arrayToSearch = array;
        currentSortIndex = 1;
        for (; currentSortIndex < arrayToSearch.length; ++currentSortIndex)
        {
            extractElement();
            if (!testElement())
            {
                shiftTestContinous();
            }
            insertElementToOpen();
        }
        return arrayToSearch;
    }

    private void extractElement()
    {
        currentExtractedElement = arrayToSearch[currentSortIndex];
        openSpaceIndex = currentSortIndex;
    }

    private boolean testElement()
    {
        if (openSpaceIndex > 0)
        {
```

```
            return currentExtractedElement >= arrayToSearch[openSpaceIndex - 1];
        }
        return true;
    }

    private void shiftTestContinous()
    {
        do
        {
            shiftElement();
        } while (!testElement() && openSpaceIndex > 0);
    }

    private void shiftElement()
    {
        arrayToSearch[openSpaceIndex] = arrayToSearch[openSpaceIndex - 1];
        --openSpaceIndex;
    }

    private void insertElementToOpen()
    {
        arrayToSearch[openSpaceIndex] = currentExtractedElement;
    }

}
```

# *Appendix 3*

*Here follows the test classes.*

*These are the test classes for the tester class and the four sorting methods. Since the individual algorithm tests are so similar, only one is given here.*

```java
/**
 * @author James Testmann, S071954
 */
import Helpers.arrayProperties;
import Sorting.InsertionSort;
import Sorting.InsertionSortFSM;
import Sorting.SelectionSort;
import java.util.ArrayList;
import org.junit.Test;
import static org.junit.Assert.*;

public class TesterTest
{
    @Test
    public void SelectionSortTest()
    {
        arrayProperties shortTest = new arrayProperties();
        Tester instance = new Tester(shortTest);
        shortTest.sizeOfSortArray = 1000;
        shortTest.numberOfTimesToRun = 1;
        shortTest.testFromZeroTo = 5000;

        ArrayList<int[]> result = instance.returnSortedArraysForTesting(new SelectionSort());
        ArrayList<int[]> expResult = instance.returnComparisonArraysForTesting(result);

        assertArrayEquals(expResult.get(0), result.get(0));
        assertArrayEquals(expResult.get(1), result.get(1));
        assertArrayEquals(expResult.get(2), result.get(2));
    }

    @Test
    public void InsertionSortTest()
    {
        arrayProperties shortTest = new arrayProperties();
        Tester instance = new Tester(shortTest);
        shortTest.sizeOfSortArray = 1000;
        shortTest.numberOfTimesToRun = 1;
        shortTest.testFromZeroTo = 5000;

        ArrayList<int[]> result = instance.returnSortedArraysForTesting(new InsertionSort());
        ArrayList<int[]> expResult = instance.returnComparisonArraysForTesting(result);

        assertArrayEquals(expResult.get(0), result.get(0));
        assertArrayEquals(expResult.get(1), result.get(1));
        assertArrayEquals(expResult.get(2), result.get(2));
    }

    @Test
    public void InsertionSortFSMTest()
    {
        arrayProperties shortTest = new arrayProperties();
        Tester instance = new Tester(shortTest);
        shortTest.sizeOfSortArray = 1000;
        shortTest.numberOfTimesToRun = 1;
        shortTest.testFromZeroTo = 5000;

        ArrayList<int[]> result = instance.returnSortedArraysForTesting(new InsertionSortFSM());
        ArrayList<int[]> expResult = instance.returnComparisonArraysForTesting(result);

        assertArrayEquals(expResult.get(0), result.get(0));
        assertArrayEquals(expResult.get(1), result.get(1));
        assertArrayEquals(expResult.get(2), result.get(2));
    }

}
```

```java
/**
 * @author James Testmann, S071954
 */
import org.junit.Test;
import static org.junit.Assert.*;

public class SelectionSortTest
{

    @Test
    public void testSelectionSort()
    {
        System.out.println("Selection Sort Test");
        int[] arrayToSort = {100, 474, 233, 443, 449, 452, 128, 362, 207, 420, 482, 260, 284, 436, 66, 485, 317,
143, 411, 61, 315, 306, 79, 464, 362, 459, 63, 229, 444, 61, 180, 333, 199, 456, 189, 228, 338, 119, 290, 300, 391,
269, 173, 58, 405, 175, 267, 158, 91, 4, 174, 470, 31, 489, 336, 370, 117, 365, 221, 73, 428, 181, 185, 281, 410,
372, 186, 204, 183, 18, 165, 109, 405, 119, 213, 329, 407, 309, 199, 218, 416, 482, 62, 134, 354, 109, 119, 191,
320, 257, 389, 181, 280, 237, 331, 174, 217, 372, 258, 474};
        SelectionSort instance = new SelectionSort();
        int[] expResult = {4, 18, 31, 58, 61, 61, 62, 63, 66, 73, 79, 91, 100, 109, 109, 117, 119, 119, 119, 128,
134, 143, 158, 165, 173, 174, 174, 175, 180, 181, 181, 183, 185, 186, 189, 191, 199, 199, 204, 207, 213, 217, 218,
221, 228, 229, 233, 237, 257, 258, 260, 267, 269, 280, 281, 284, 290, 300, 306, 309, 315, 317, 320, 329, 331, 333,
336, 338, 354, 362, 362, 365, 370, 372, 372, 389, 391, 405, 405, 407, 410, 411, 416, 420, 428, 436, 443, 444, 449,
452, 456, 459, 464, 470, 474, 474, 482, 482, 485, 489};
        int[] result = instance.sort(arrayToSort);
        assertArrayEquals(expResult, result);
    }
}
```

NB: All individual algorithm test classes follow this setup. Only one is shown.