Practical Lab Session C:
Elementary Sorting Algorithms

Aim

The aim of this lab session is to improve your familiarity and practical experience with elementary sorting algorithms, and simple measures of their performance in terms of the number of comparisons made by an algorithm. Such understanding of the implementation of the algorithms and use of performance measurement will be instrumental in achieving the main deliverable for Assignment 1.

Introduction

Sorting is an important basic operation used by many applications. A lot of effort has been spent creating fast sorting algorithms. In this lab, three elementary sorting algorithms will be implemented. Initially, this will be done using an iterative approach to the algorithms. This will be followed by a recursive implementation of two of these algorithms. Then an example of a straightforward approach to measuring the performance of two of the algorithms will be outlined and implemented. Since better algorithms are known, these elementary algorithms are of somewhat limited use. Two advanced algorithms, Merge Sort and Quick Sort, will be examined in a later lab session.

Performance Statistics Overview

One method for dealing with the problem of measuring the performance of an algorithm is to find a core operation that the algorithm performs. The performance of the algorithm will then be determined by counting the number of times the selected operation is performed. This will provide a way of comparing two algorithms that is independent of the computer the algorithm runs on. For general purpose, elementary sorting algorithms, the standard measure is the **number of comparisons** that are made as the algorithm executes.

To get a fair view of the performance of a sorting algorithm, it is not enough to just try it on one array. Instead, its performance for all possible permutations of the data values in an array should be examined. For many sorting algorithms, the average performance can be computed mathematically. However, to do this experimentally is often impractical for all but the smallest of arrays as the number of permutations typically grows exponentially. In practice, we will test the performance on a number of randomly generated arrays: A_1, A_2, A_3, ... A_k. For each array, the number of comparisons made will be counted giving values: C_1, C_2, C_3, ... C_k. From these data values, statistics for the **average**, **minimum**, and **maximum** number of comparisons will be computed. The computed average number of comparisons will give an approximation to the true average number of comparisons over all array permutations for the algorithm. Assuming that the generation of the arrays is truly random, larger

values of k will lead to a closer approximation to the true average. The minimum and maximum number of comparisons give an indication of how consistent the performance of the algorithm is; they also give a rough indication of what the **best case** and **worst case** are, respectively. However, be aware that the number of possible permutations of an array of size n is n factorial (n!). If relatively few of the cases lead to best-case or worst-case behaviour, they are unlikely to show up in the randomly chosen test cases. In characterising the performance of an algorithm, usually only the worst case and average case are cited. The worst case allows you to guarantee the performance of an algorithm. The average gives the expected performance. We many even be willing to tolerate a bad worst case if the average is good, especially if the algorithm will be executed many times.

Part 1: Iterative Sorting Algorithms

During the first part of this lab session, you will complete the implementation of iterative versions of the Selection Sort, Insertion Sort and Shell Sort algorithms. Each of these will be separately implemented over Exercise 1- Exercise 3, along with a test program that will display the contents of an array before, during, and after each sorting algorithm is run. By displaying the output of the algorithm during each iteration of its operation, this will help facilitate your understanding of the operation of each algorithm. During the implementation of the algorithms and test program you will be creating **static classes** for each algorithm, in addition to the use of a static class containing some utility methods. As such, please keep in mind that static classes cannot be instantiated as normal, rather are called directly using the classname followed by the dot operator then the method name.

Exercise 1: Iterative Selection Sort

Create a new Java project (**ElementarySorts**) in IntelliJ, then download the files *SelectionSortArray.java*, *SortTest.java* and *ArrayUtil.java* from the **Exercise 1 Resources** folder within **Lab Session C** (in the **Lab Sessions** folder) on Blackboard, then copy the files into the **src** project folder on your computer. A skeleton of the Iterative Selection Sort algorithm is contained in the file *SelectionSortArray.java*. In addition, a skeleton of the program to test the implemented sorting algorithms is given in *SortTest.java* along with a set of utility methods in *ArrayUtil.java*. To complete the implementation of the Selection Sort algorithm and associated test program, complete the following steps:

1. Look at the skeleton test program in *SortTest.java* along with the methods in the *ArrayUtil.java* file. After the print statements in the SortTest class, use the appropriate ArrayUtil method to display the initial, unsorted array.

2. Invoke the Selection Sort algorithm with the generated data (*dataSelectionSort*) and associated array size (*arraySize*) as arguments for the parameters of the selectionSort() method.

3. Again, use the appropriate ArrayUtil method to display the final, sorted array.

Compile and run the program using the main() method in the SortTest class. The program will ask you for an array size. Enter 20. An array of 20 random values between 0 and 20 should be generated and displayed. Even though selection sort is applied to the array, the resulting array will not have been sorted (the values in the sorted array should be the same as those given for the initial array).

Next, look at the skeleton of the Iterative Selection Sort algorithm provided in the *SelectionSortArray.java* file. In the private selectionSort() method, complete the following steps:

4. Identify the index of the next smallest item in the array by assigning the predefined variable *indexOfNextSmallest* with the result from a call to the method

getIndexOfSmallest(). Ensure that you pass in the correct arguments to the parameters of the getIndexOfSmallest() method.

5. Invoke the swap() method to swap the item at the current index with the item at the index of the next smallest item. Again, ensure that you pass in the correct arguments to the parameters of the swap() method.

6. Display the current state of the array using an appropriate ArrayUtil method.

Run the main() method of the SortTest class. Again, the program will ask you for an array size. Enter 20. An array of 20 random values between 0 and 20 should be generated and displayed. Even though selection sort is now being applied to the initial array, the final array and the state of the array during each iteration of the sorting algorithm will still not have been changed!

7. In the getIndexOfSmallest() method, replace the conditional statement with an appropriate comparison. To do this, you need to use the compareTo() method from the Comparable interface[1] with the item at the current index in the array (i.e. *arr[index]*) and the current smallest value (i.e. *minVal*). The result of the compareTo() method should be used in a conditional statement by checking if the result is a "less than" comparison (please look at the documentation on the Comparable interface given in the footnote for more information). Subsequently, replace the statement *if (false)* with the new conditional statement that uses the compareTo() method.

8. Finally, complete the swap() method by swapping the two array entries (i.e. *arr[from]* and *arr[to]* – keep in mind the variable *temp* currently stores *arr[from]*).

Run the main() method of the SortTest class. Again, enter 20 when the program asks for an array size. You should now see the initial array, each iteration of the array during the Selection Sort algorithm and the final, sorted array. Observe the values of the array during the Selection Sort. Check the sorted array is correct. If not, please debug the previous steps and re-run the main() method until the Selection Sort works correctly.

Please show your working, fully tested Iterative Selection Sort to one of the demonstrators.

[1] https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

Exercise 2: Iterative Insertion Sort

You will now implement an iterative version of the Insertion Sort algorithm. The algorithm will be tested within the SortTest class again in order to provide a comparison of the operation of the Selection Sort and Insertion Sort algorithms. Please ensure you have fully completed Exercise 1, with a working implementation of both the Selection Sort algorithm and main() method of the SortTest class, before continuing.

Download the file *InsertionSortArray.java* from the **Exercise 2 Resources** folder within **Lab Session C** (in the **Lab Sessions** folder) on Blackboard, then copy the file into the **src** project folder on your computer. A skeleton of the Iterative Insertion Sort algorithm is contained in the file *InsertionSortArray.java*. In this exercise, you will expand upon the main() method in the SortTest class and complete the implementation of the Insertion Sort algorithm. Starting with the *SortTest.java* file complete the following steps:

1. Expand upon the source code in the main() method to create a duplicate of the original data set before it is sorted by the Selection Sort algorithm. Use the appropriate ArrayUtil method to create the duplicate array. Name the duplicate array *dataInsertionSort*.

2. After the Selection Sort algorithm has been invoked and final sorted array subsequently displayed, display the new (unsorted) duplicate array using an appropriate method from the ArrayUtil class.

Run the main() method of the SortTest class and enter 20 for the array size. The Selection Sort should run as expected, displaying the correct output, then the new duplicated array should be shown. Check it is the same as the original (unsorted) array used by the Selection Sort algorithm.

3. Invoke the Insertion Sort algorithm with the generated data (*dataInsertionSort*) and associated array size (*arraySize*) as arguments for the parameters of the insertionSort() method.

4. Use the appropriate ArrayUtil method to display the final, sorted array.

Again, run the main() method of the SortTest class and enter 20 for the array size. The Selection Sort should run as expected, displaying the correct output, then the duplicate array should be shown along with the final sorted version after the Insertion Sort. However, the final array will still be unsorted.

The skeleton of the Iterative Insertion Sort algorithm has been provided in the *InsertionSortArray.java* file. In the private insertionSort() method, complete the following steps:

5. Obtain the next item in the unsorted array by assigning the first unsorted entry in the array to the predefined variable *nextToInsert*.

6. Insert the next item into the unsorted part of the array between the *first* and *unsorted-1* elements. To do this invoke the insertInOrder() method with the next item (*nextToInsert*) along with the array, the first element in the array and the end of the unsorted array (*unsorted-1*).

7. Use an appropriate ArrayUtil method to display the current state of the array.

Run the main() method of the SortTest class. Again, the Selection Sort should run as expected, displaying the correct output, then the duplicated array should be shown, along with each iteration of the array during the Insertion Sort, and the final sorted array. However, both the iterations of the array and the final array will still be unsorted!

8. In the insertInOrder() method of the InsertionSortArray class, replace the *false* condition in the while loop with a comparison between *anEntry* and the current entry in the array at *index* (i.e. *arr[index]*). The comparison should make use of the compareTo() method from Comparable to form a conditional that tests *anEntry* is less than or equal to the current entry in the array at *index* (please refer to the documentation on the Comparable interface for more information).

Run the main() method of the SortTest class, entering 20 for the array size. Again, the Selection Sort should run as expected, displaying the correct output, then the duplicate array should be shown, along with the iterations of the array during the Insertion Sort, and the final sorted array after the Insertion Sort has been completed. This time, the iterations should show the array being sorted by the Insertion Sort and the final array should be sorted correctly. *Observe the values of the array during the Insertion Sort, compared with the values during the Selection Sort. Check the sorted array from Insertion Sort is correct. If not, please debug the previous steps and re-run the main() method until the Insertion Sort works correctly.*

Please show your working, fully tested Iterative Insertion Sort to one of the demonstrators.

Exercise 3: Shell Sort

As the final part of the comparison of elementary search algorithms, you will now implement the Shell Sort algorithm. The algorithm will again be tested within the SortTest class in order to provide further comparison of the operation of Shell Sort with the previously implemented Selection Sort and Insertion Sort algorithms. Please ensure you have fully completed Exercise 1 and Exercise 2, with working implementations of Selection Sort and Insertion Sort, along with a working implementation of the main() method of the SortTest class before continuing.

Download the file *ShellSortArray.java* from the **Exercise 3 Resources** folder within **Lab Session C** (in the **Lab Sessions** folder) on Blackboard, then copy the file into the **src** project folder on your computer. A skeleton of the Shell Sort algorithm is contained in the file *ShellSortArray.java*. Similar to the previous exercises, you will initially expand upon the main() method in the SortTest class, then complete the implementation of the Shell Sort algorithm. Starting with the *SortTest.java* file complete the following steps:

1. Expand upon the source code in the main() method to create another duplicate of the original data set using the appropriate ArrayUtil method. This time name the new duplicate array *dataShellSort*.

2. After the Insertion Sort algorithm has been invoked and the associated final sorted array displayed, display the new (unsorted) duplicate array using an appropriate method from the ArrayUtil class.

Run the main() method of the SortTest class and enter 20 for the array size. The previous algorithms should run as expected, displaying the correct outputs. Then the new duplicate array should be shown. Again, check it is the same as the original (unsorted) array used by the previous algorithms.

3. Invoke the Shell Sort algorithm with the generated data (*dataShellSort*) and associated array size (*arraySize*) as arguments for the parameters of the shellSort() method from the ShellSortArray class.

4. Use the appropriate ArrayUtil method to display the final, sorted array.

Again, run the main() method of the SortTest class and enter 20 for the array size. The previous algorithms should run as expected, displaying the correct outputs, then the unsorted duplicate array should be shown along with the final sorted version after the Shell Sort. As before, the final sorted array will still be unsorted.

The skeleton of the Shell Sort algorithm has been provided in the *ShellSortArray.java* file. In the private shellSort() method, complete the following steps:

5. Replace the value of the predefined variable *interval*. The value of *interval* should be half of the current size of the array (where the current size of the array is given as *n*).

6. Invoke the incrementalInsertionSort() method of the class with the appropriate set of arguments for the parameters of incrementalInsertionSort().

7. Use an appropriate ArrayUtil method to display the current state of the array.

8. Recalculate the value for *interval* by halving its current value (note: if the value for *interval* is not reduced the while loop will never complete!)

Run the main() method of the SortTest class. Again, the previous sorting algorithms should run as expected, then the new duplicate array should be shown, along with each iteration of the array during the Shell Sort, followed by the final sorted array, though both the iterations of the array and the final array will still be unsorted!

9. In the incrementalInsertionSort() method of the ShellSortArray class, replace the *false* condition in the while loop with a comparison between *nextToInsert* and the current entry in the array (i.e. *arr[index]*). The comparison should use the compareTo() method from Comparable. The resulting comparison should test that *nextToInsert* is less than the current entry (*arr[index]*) in the array (please refer to the documentation on the Comparable interface for more information).

Run the main() method of the SortTest class, entering 20 for the array size. Again, the previous algorithms should run as expected, displaying the correct output. The new duplicate array should then be shown, along with the iterations of the array during the Shell Sort, followed by the final sorted array after the Shell Sort. This time, the iterations should show the array being sorted by the Shell Sort algorithm and the final array should be sorted. Observe the values of the array during the Shell Sort, compared with the values during the Selection Sort and Insertion Sort. Check the sorted array from Shell Sort is correct. If not, please debug the previous steps and re-run the main() method until the Shell Sort works correctly.

Please show your working, fully tested Shell Sort to one of the demonstrators.

Part 2: Recursive Sorting Algorithms

In this part of the lab session, you will implement recursive variations of the Selection Sort and Insertion Sort algorithms. The aim of this part of the lab is to gain more familiarity and understanding of a recursive approach to solving a problem that may also be solved using an iterative approach. Please ensure you have completed Exercise 1- Exercise 3 before continuing with these recursive implementations.

Exercise 4: Recursive Selection Sort

The file *SelectionSortArray.java* will contain the skeleton for the recursive approach to the Selection Sort algorithm (given after the iterative approach). To complete and test the implementation of the recursive Selection Sort, carry out the following:

1. In the *SortTest.java* file replace the existing call to the Selection Sort with a call to the selectionSortRecursive() method. Make sure to use *dataSelectionSort* and *arraySize* as arguments to the method.

Examine the Recursive Selection Sort part of the *SelectionSortArray.java* file, then complete the following steps:

2. In the public selectionSortRecursive() method, invoke the private version of the selectionSortRecursive() method, passing the appropriate arguments for the *array*, *first* and *last* parameters of the method call. Keep in mind the argument *last* will be the index of the last element in the array (i.e. the size of array – 1). Note: this is not the recursive call, rather the actual call to start the sorting algorithm.

3. In the private selectionSortRecursive() method, determine the index of the next smallest item in the array (i.e. *indexOfNextSmallest*) using the getIndexOfSmallest() method. Ensure that you pass in the correct arguments to the parameters of the getIndexOfSmallest() method.

4. Then, invoke the swap() method to swap the current item with the item identified by *indexOfNextSmallest*.

5. Use an appropriate ArrayUtil method to display the current state of the array.

6. Finally, make the recursive call by invoking the private selectionSortRecursive() method with the smaller problem (note: the first item in the array is currently sorted; hence the smaller problem will be the unsorted remainder of the array).

Run the main() method of the SortTest class, entering 20 for the array size. The Selection Sort (Recursive) algorithm should now display the initial unsorted array, along with each iteration of the array during the sort, and the final, sorted array. Check the sorted array is correct. If not, please debug and re-run the main() method again.

Please show your working, fully tested Recursive Selection Sort to one of the demonstrators.

Exercise 5: Recursive Insertion Sort

The file *InsertionSortArray.java* will contain the skeleton for the recursive approach to the Insertion Sort algorithm (given after the iterative approach). However, unlike the recursive version of Selection Sort, to make Insertion Sort fully recursive, it will also utilize a recursive version of the insertInOrder() method to recursively insert a value into its correct location in the array. To complete and test the implementation of the Recursive Insertion Sort, carry out the following:

1. In the *SortTest.java* file replace the existing call to the Insertion Sort with a call to the insertionSortRecursive() method. Make sure to use *dataInsertionSort* and *arraySize* as arguments to the method.

Examine the Recursive Insertion Sort part of the *InsertionSortArray.java* file, then complete the following steps:

2. In the public insertionSortRecursive() method, invoke the private version of the insertionSortRecursive() method, passing the appropriate arguments for the *array*, *first* and *last* parameters of the method call. As before, keep in mind the argument *last* will be the index of the last element in the array (i.e. the size of array – 1).

3. In the private insertionSortRecursive() method, first make the recursive step by invoking the private insertionSortRecursive() method with the smaller problem (i.e. to sort all but the last entry in the array).

4. Then, invoke the insertInOrderRecursive() method to recursively insert the last entry (i.e. *arr[last])* into the remainder of the array (i.e. between *first* and *last-1*).

5. Use an appropriate ArrayUtil method to display the current state of the array.

Although for the Iterative Insertion Sort we defined the insertInOrder() method, a modified recursive version of the method, given as insertInOrderRecursive(), is used instead by insertionSortRecursive() – in Step 4 above. To complete the implementation, the insertInOrderRecursive() method now needs to be completed:

6. In the insertInOrderRecursive() method, replace the *true* condition at the start of the method with a comparison between *anEntry* and the last entry in the current array (i.e. *arr[end]*). The comparison should use the compareTo() method from Comparable. The resulting comparison should test that *anEntry* is greater than or equal to the last entry in the array (please refer to the documentation on the Comparable interface for more information).

7. If the entry (*anEntry*) is greater than or equal to the last entry in the array, then assign the entry to the array at the index after the last entry (i.e. at *arr[end + 1]*).

8. Finally, complete the insertInOrderRecursive() method by making a recursive call to insertInOrderRecursive() with the smaller problem in order to recursively insert the entry (i.e. *anEntry)* into the first part of the array (i.e. up to *end-1*).

Run the main() method of the SortTest class, entering 20 for the array size. The Insertion Sort (Recursive) algorithm should now display the initial unsorted array, along with each iteration of the array during the sort, and the final, sorted array. Check the sorted array from the Recursive Insertion Sort is correct. If not, please debug the previous steps and re-run the main() method until the recursive implementation works correctly.

Please show your working, fully tested Recursive Insertion Sort to one of the demonstrators.

Part 3: Instrumented Sorting Algorithms

This part of the lab session will now focus on obtaining performance statistics for two of the implemented algorithms. The statistics will be based on counting the number of comparison operations made during the execution of the sorting algorithms. The basis for the algorithms will be the Recursive Insertion Sort and the Shell Sort, so please ensure you have a working implementation of these two algorithms before proceeding with this part of the lab session. In the following exercises, you will work with a new file *SortArrayInstrumented.java* to provide an implementation of the class SortArrayInstrumented that will contain the chosen sorting algorithms, along with variables and methods to capture and generate comparison statistics. Another new file, *SortTestInstrumented.java*, will also be used to invoke the sorting algorithms and to specify the parameters for the tests that are run.

Download the *SortArrayInstrumented.java* file and the *SortTestInstrumented.java* file from the **Exercise 6 Resources** folder within **Lab Session C** (in the **Lab Sessions** folder) on Blackboard, then copy the files into the **src** project folder on your computer.

Exercise 6: Instrumented (Recursive) Insertion Sort

The *SortArrayInstrumented.java* file provides a skeleton of the SortArrayInstrumented class, which contains a set of private variables for capturing the number of comparisons made (*comparisons*, *totalComparisons*, *minComparisons*, *maxComparison*), along with an empty constructor for the class, and a placeholder for the methods belonging to the Recursive Insertion Sort. In this exercise, you will complete the implementation of the class by providing the accessor methods, methods to start and end data collection, and also incorporate the methods of the sorting algorithm with additional logging of the comparisons made during the sort. Similarly, the *SortTestInstrumented.java* file provides the skeleton of a test program that will be completed to run the sorting algorithm and display statistics on the number of comparisons that occurred. To complete the implementation of both files, complete the following steps:

1. In the constructor for the SortArrayInstrumented class, initialize the private member variables of the class (note: all private member variables should be initialized to zero, with the exception of *minComparison*, which should be set to the value *Long.MAX_VALUE*)

2. Copy the completed, working methods for the Recursive Insertion Sort from the InsertionSortArray class and paste these into the SortArrayInstrumented class (in the appropriate part of the file). Previously, the InsertionSortArray class was a static class, however, the Insertion Sort methods that have pasted into the SortArrayInstrumented class should no longer be static, so remove the *static* modifier from the method signature of both the insertionSortRecursive() method and the insertInOrderRecursive() method.

3. In the private insertionSortRecursive() method, comment out the call to display the output of the current state of the array, i.e. *ArrayUtil.displayArrayContent(arr);*

4. In the *SortTestInstrumented.java* file, instantiate an object of the SortArrayInstrumented class (the object should be called *sai*).

5. Within the for loop (iterating over the number of trials in the SortTestInstrumented class) invoke the Recursive Insertion Sort algorithm using the object (*sai*) created in the previous step. The arguments *data* and *arraySize* should be passed as arguments to the method call.

Run the main() method of the SortTestInstrumented class. Enter 10 for the size of the array and 3 for the number of trials. You should receive correct results for the three initial and sorted arrays. However, the statistics for total, average, minimum and maximum number of comparisons will not currently be producing values.

To produce statistics for the number of comparisons, you will now need to complete the associated methods in the SortArrayInstrumented class, including adding the statistics to the sort algorithm, and displaying the statistics in the SortTestInstrumented class. The following steps should be used:

6. In the SortArrayInstrumented class add a public accessor method called *getTotalComparisons()* to return the value of the private member variable *totalComparisons*.

7. Next, add a public accessor method called *getMinComparisons()* to return the value of the private member variable *minComparisons*.

8. Then, add a public accessor method called *getMaxComparisons()* to return the value of the private member variable *maxComparisons*.

To compute the minimum and maximum number of comparisons, code needs to be added at the beginning and end of the sort. While the needed code could be added directly to the sorts, it is better to encapsulate it in a couple new methods: *startStatistics()* and *endStatistics()*.

9. Add a private method called *startStatistics()* to the SortArrayInstrumented class. It should initialize the variable *comparisons* to zero.

10. Add a private method called *endStatistics()* to the SortArrayInstrumented class. It should add the *comparisons* to *totalComparisons*. It should also compare *comparisons* to *minComparisons* and set *minComparisons* to whichever is smaller. In addition, it should compare *comparisons* to *maxComparisons* and set *maxComparisons* to whichever is larger.

11. In the public insertionSortRecursive() method, call startStatistics() at the beginning of the method before the call to *insertionSortRecursive(arr, 0, n-1)*. Likewise, call endStatistics() after the call to *insertionSortRecursive(arr, 0, n-1)*.

12. Within the insertInOrderRecursive() method, add a statement to increment the value of *comparisons* when the compareTo() method is called (note: this should be done at the start of the method, prior to the comparison being made).

13. Finally, in the SortTestInstrumented class, modify the final set of *System.out.println()* method calls to incorporate calls to the appropriate SortArrayInstrumented methods in order to return and display the statistics after all trials have been performed (note: to determine the average comparisons made, remember to divide the total returned by the number of trials (*numTrials*)).

Run the main() method of the SortTestInstrumented class. Enter 10 for the size of the array and 3 for the number of trials. You should receive correct results for the three initial and sorted arrays, along with statistics for the total, average, minimum and maximum number of comparisons made. Repeat a number of times and observe the variation that occurs between runs. Likewise, repeat using an array size of 20 with 3 trials.

Once you are satisfied the sorting algorithm is working correctly and the statistics are being produced, comment out the lines in main() that display the initial and sorted arrays. Then run main() with an array size of 20 and 1000 trials. Observe the average number of comparisons for different numbers of trials.

Please show your working, fully tested instrumented Recursive Insertion Sort to one of the demonstrators.

Exercise 7: Instrumented Shell Sort

Now that the *SortArrayInstrumented.java* file provides a range of methods for generating statistics on the number of comparisons made by a sorting algorithm, it is reasonably straightforward to incorporate other sorting algorithms. In this exercise, you will add the methods for the Shell Sort algorithm, then make the modifications necessary to capture the number of comparisons made. Unlike the Recursive Insertion Sort, the Shell Sort requires a slight modification to correctly record the number of comparisons. To generate statistics for the Shell Sort algorithm, complete the following steps:

1. Copy the completed, working methods for the Shell Sort from the ShellSortArray class and paste these into the SortArrayInstrumented class (in the appropriate part of the file after the implementation of the Recursive Insertion Sort from Exercise 6). The Shell Sort methods pasted into the SortArrayInstrumented class should no longer be static, so remove the *static* modifier from the method signature of both the shellSort() method and the incrementalInsertionSort() method.

2. In the private shellSort() method, comment out the call to display the output of the current state of the array, i.e. *ArrayUtil.displayArrayContent(arr);*

3. Add calls to startStatistics() and endStatistics() to the public shellSort() method.

4. In the incrementalInsertionSort() method place code to increment the variable *comparisons* when the compareTo() method is invoked. Since the comparison is in the condition of a while loop, this is a bit trickier to handle than it was for the Recursive Insertion Sort. Certainly, the compareTo() method was invoked once for each time the body of the loop executed. The compareTo() method will also have been invoked one more time if the first clause in the condition (*index* ≥ *first*) was true and the result of the compareTo() method caused the loop to finish. Subsequently, you will also need to increment the variable *comparisons* after the while loop to handle this particular case (i.e. add a conditional statement after the while loop to capture this second possibility when the comparison was made, i.e. if (*index >= first*)).

5. In the *SortTestInstrumented.java* file, change the method call from the SortArrayInstrumented object (*sai*) that invokes insertionSortRecursive() to instead invoke the shellSort() method.

6. In addition, remove the comments around the print statements for displaying the initial and sorted arrays (from the end of Exercise 6).

Run the main() method of the SortTestInstrumented class. Enter 20 for the size of the array and 3 for the number of trials. You should receive correct results for the three initial and sorted arrays, along with statistics for total, average, minimum and maximum number of comparisons made. An average number of comparisons of

approximately 86 is likely – note: this may vary during runs depending on the set of values in the array. Repeat a number of times and observe the variation that occurs between runs. Ensure the Shell Sort is working correctly, as expected.

Once you are satisfied the algorithm is working correctly, comment out the lines in main() that display the initial and sorted arrays. Then run main() with an array size of 20 and 10000 trials. Observe the average number of comparisons for different numbers of trials. It may be potentially useful to record the statistics for a range of arrays sizes. As an example, the following table (or a similar table) may be used for this purpose:

ARRAY SIZE	MINIMUM COMPARISONS	AVERAGE COMPARISONS	MAXIMUM COMPARISONS
10			
50			
100			
200			
300			
400			
500			
750			
1000			

Please show your working, fully tested instrumented Shell Sort to one of the demonstrators.