ITCS-6114 Project 1 - Report

Ankur Huralikoppi
800782885
ahuralik@uncc.edu

Abstract:

This report provides findings of the project carried out to modify quicksort to stop calling itself recursively and call insertion sort when the size of the sub array hits a defined threshold.

Both the algorithms were modified to accommodate the change mentioned above and tests were run for different sizes of sub arrays with arrays of random numbers. The whole algorithm was run 1000 times on different random arrays to ensure that it has run on a fair mix of random numbers.

Series of values of the insertion sort threshold were taken and average number of comparisons and assignments were calculated. Then, the normal quick sort was run on the same arrays and its counts were also averaged. Graphs were plotted to see how assignments and comparisons varied in the modified quick sort and the trivial quick sort.

Approach:

The algorithm runs on 1000 arrays, which contain random numbers, all less than a value 'k'. Once the arrays are fed with random numbers, each of them is sorted once with each value of insertion sort threshold: 'm'. An array is sorted once with each value of 'm'. The counts of assignments and comparisons are saved for computing averages. The below lines depict the approach:

For each k = 100, 1000, 10000, 100000 do

- Get 1000 arrays with random numbers less than K
- For each 'm' selected do
  - Sort each of the 1000 arrays with 'm'
  - Save the comparisons and assignments counts to calculate average.
- Sort each of the arrays with normal quick sort and save the comparison and assignment counts.

Modified Algorithms:

Modified Quick Sort algorithm:

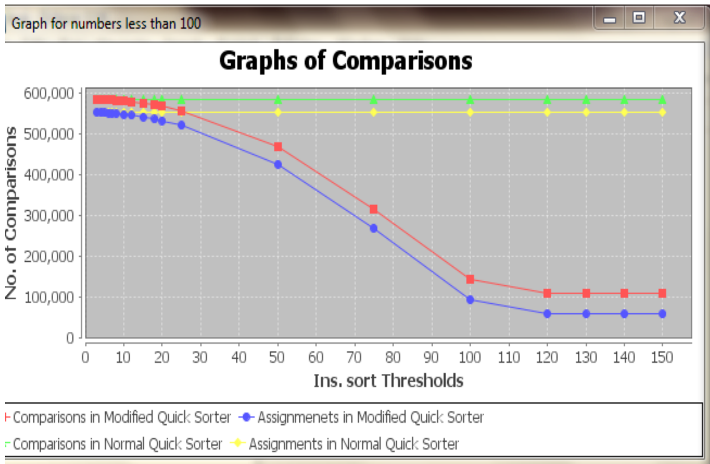
```
MODIFIED_QUICKSORT (a, start, end, m)
  lengthOfArray = start – end + 1
  if start < end
    if lengthOfArray <= m
        MODIFIED_INS_SORT(a, start, end)
    else
        pivot = PARTITION(a, start, end)
        MODIFIFED_QUICKSORT(a, start, pivot - 1)

MODIFIED_INS_SORT(a, start, end)
  for j = start + 1 to end do
    key = arr[j]
    i = j – 1;
    while arr[i] > key && i >= start do
        a[i + 1] = a[i]
        i = i – 1
    a[i + 1] = key
```

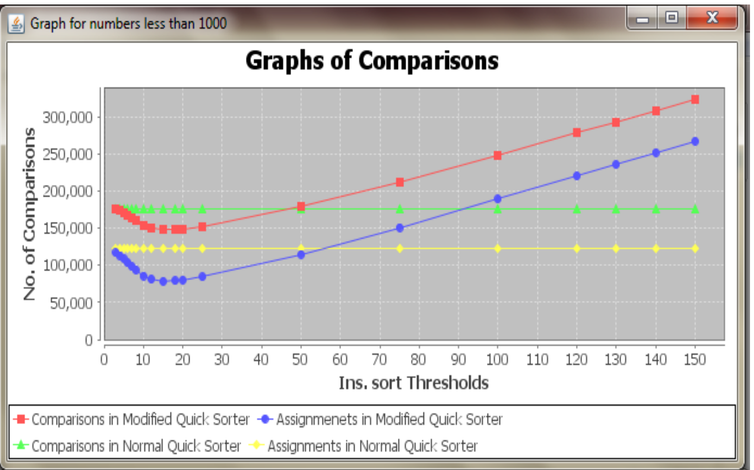
Findings:

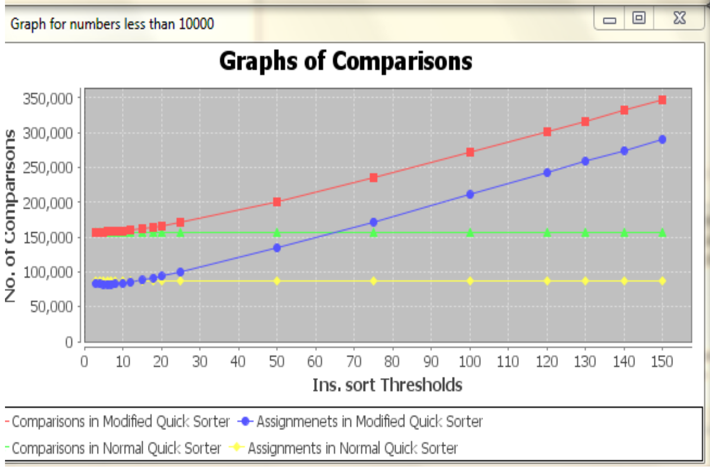
The starting values selected for 'm' were 3, 5, 10 & 20. Tests were run and several other values to 'm' were added after studying the trend in the numbers of comparisons and assignments. The algorithm was then run on 19 values of 'm' to determine the best possible value for it for k=100, 1000, 10000, 100000.

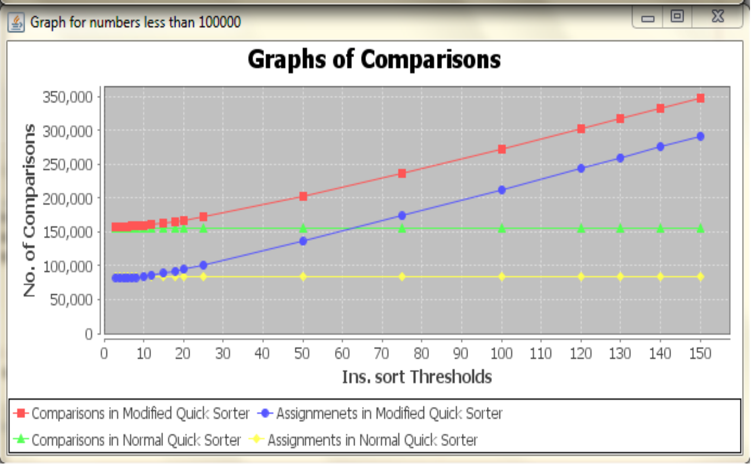
The following table provides details about the minimum most comparisons got for various ‘k’ values:

K	100		1000		10000		100000	
Counter Type	C	A	C	A	C	A	C	A
‘m’ for minimum most values of counts.	140	140	15	15	3	6	3	5
	140	140	15	15	3	6	3	5
	140	140	15	15	3	6	3	5
	140	140	15	15	3	6	3	5
	140	140	15	15	3	6	3	5
	140	140	15	15	3	6	3	5









The above table shows the values of ‘m’ for which the minimum most comparisons and assignments were done for different values of ‘k’. The graphs show the relative growth in the numbers of comparisons and assignments of the modified sorter with respect to the number of comparisons and assignments done with the normal sorter on the same data.

Analysis:

From the tests, we could see that the minimum number of comparisons were got at values 150, 15, 3 & 3 for k values of 100, 1000, 10000, 100000. When K = 100, the probability of a number repeating itself again is very high. If a number has equal probability of occurring again, then a number should occur at least 10 times in an array of size 10000. Thus, when we have insertion sort called for sub-arrays of relatively greater sizes, the number of comparisons and assignments greatly reduce since insertion sort assigns only on a number being lesser than another. But, in the case of quick sort, a duplicate number is either assigned to belong to the part of array which is smaller than pivot, or to the part of array which has all greater elements than the pivot.

When K increases, the probability of getting duplicates decreases. Hence, the probability that a sub array is generated in such a way that the elements are relatively sorted decreases. Now, the threshold to call for insertion sort is crucial. Knowing the complexities of the algorithms, insertion sort performs better for a smaller array of sizes of the order of 8-12. For K = 100000, there is a fair chance that a random number generator gives all distinct numbers. At this point, the insertion sort performance depends solely on how the values in the sub array provided to it are sorted relatively.

From the experiments run, for K = 100000, quick sort performs lesser number of comparisons than the modified quick sort. But however, when assignments are considered, the modified quick sorter performs better, but not by a considerable margin.

Conclusion:

The modified quick sorter doesn't always guarantee better performance. The performance metrics solely depend upon two factors in the input:

1. How much of the elements are relatively sorted in a sub array of size n/2. When the quick sort runs for the first time, it places the pivot in its appropriate position. The other two sub-arrays will be considered in the further iterations. When the point where insertion sort has to be called arrives, it will perform fewer assignments if the sub file has elements relatively sorted.
   And with respect to quick sort, if there are very few elements, greater than the pivot, the assignments will again be very less.
2. The probability of duplicates. When the probability of duplicate numbers to occur is very high, it's a good idea to call insertion sort at a relatively bigger sub array size. If the number of duplicates is high, the number of assignments

Of the above mentioned conditions, the first one holds good for the normal quick sort as well. Also, it is not possible to determine how much of numbers are relatively sorted with each other. But the latter could be calculated if we know the nature of input, or the input is generated based on rules, like the use of the number 'k' in our test.

If we know the probability of duplicate numbers, we can use the modified sorter. If the probability of duplicates is very high, about 80% of the numbers are duplicates, we can set the insertion sort threshold to a higher value in the range of 135-150. If the number of duplicates is in the range of 30-40%, the threshold should be set in the range of 8-15. And when the probability is just about 2-5%, the threshold can be set to 3-5. But when we know that there is a good chance that the input can have no duplicates, it will be a better idea to choose the normal quick sort algorithm.

Thus, when we have knowledge about the above mentioned factors of the input, we could choose to use the modified quick sorter. Otherwise, the normal quick sort algorithm would perform better.

CODE LISTING:

```
----------------------------------------------------------------------------------------------------------------------------
//main.java
package uncc.edu.main;

import edu.uncc.helpers.StatsHelper;
import edu.uncc.helpers.TestHelper;

public class Main {
    public static void main(String[] args) {
        TestHelper testHelper = new TestHelper();

        if(args.length > 0){
            for(String s : args){
                if(s.equals("test_correctness"))
                    testHelper.testCorrectnessOfAlgorithms(20, 8, 4);
                else if (s.equals("plot_graphs"))
                    StatsHelper.setPlotGraphs(true);
            }
        }

        //Do the tests.
        testHelper.testAlgorithms();

    }
}

----------------------------------------------------------------------------------------------------------------------------
//constants.java
package edu.uncc.constants;

public interface Constants {
        int MAX_ITER_FOR_AVERAGE = 1000;
        int MAX_ARRAY_SIZE = 10000;

        int INITIAL_SEED = 10;

        int MAX_RAND_NO_FOR_SEEDS = 500;
}

----------------------------------------------------------------------------------------------------------------------------
//AbstractQuickSorters.java
package edu.uncc.sorter;

import edu.uncc.helpers.ArrayHelper;

public abstract class AbstractQuickSorters {

        protected long comparisonCounter;
        protected long assignmentCounter;

        public AbstractQuickSorters(){
                resetComparisonCounter();
                resetAssignmentCounter();
        }

        public long getAssignmentCounter() {
                return assignmentCounter;
        }

        public void setAssignmentCounter(int assignmentCounter) {
                this.assignmentCounter = assignmentCounter;
        }

        protected int partition(int [] arr, int start, int end){
                int pivot = arr[end];
                int i = start - 1;

                for(int j = start; j < end ; j ++)
                {
                        comparisonCounter++;
                        if(arr[j] <= pivot)
                        {
                                i++;
                                int temp = arr[j];
                                arr[j] = arr[i];
```

```java
                                arr[i] = temp;
                                assignmentCounter++;
                        }
                }

                arr[end] = arr[i + 1];
                arr[i + 1] = pivot;
                assignmentCounter++;

                return i + 1;
        }

        public void resetComparisonCounter(){
                setComparisonCounter(0);
        }

        public void resetAssignmentCounter(){
                setAssignmentCounter(0);
        }

        public long getComparisonCounter() {
                return comparisonCounter;
        }

        public void setComparisonCounter(int comparisonCounter) {
                this.comparisonCounter = comparisonCounter;
        }

        public void sortAndPrintArray(int [] arr, int start, int end, String msgToPrint){
                sort(arr, start, end);
                ArrayHelper.printArray(arr, msgToPrint);
                System.out.println("Comparisons: " + getComparisonCounter());
        }

        public abstract void sortAndPrint(int [] arr, int start, int end, String msg);

        public abstract void sort(int [] arr, int start, int end);
}

-------------------------------------------------------------------------------------------------
//QuickSort.java
package edu.uncc.sorter.impl;

import edu.uncc.sorter.AbstractQuickSorters;

public class QuickSort extends AbstractQuickSorters {

        public QuickSort(){
                super();
        }

        @Override
        public void sort(int [] arr, int start, int end){
                if(start < end)
                {
                        int pivot = partition(arr, start, end);
                        sort(arr, start, pivot - 1);
                        sort(arr, pivot + 1, end);
                }
        }

        @Override
        public void sortAndPrint(int[] arr, int start, int end, String msg) {
                super.sortAndPrintArray(arr, start, end, msg);
        }

}

-------------------------------------------------------------------------------------------------
//ModifiedQuickSort.java
package edu.uncc.sorter.impl;

import edu.uncc.sorter.AbstractQuickSorters;

public class ModifiedQuickSorter extends AbstractQuickSorters {
	private Integer insertionSortThreshold;

	public ModifiedQuickSorter(){
		super();
		insertionSortThreshold = 5; //set 5 by default
	}

	public ModifiedQuickSorter(int i) {
		insertionSortThreshold = i;
	}

	private void modifiedInsertionSort(int [] arr, int start, int end){

		int length = (end - start + 1);

		for(int j = start + 1; j < length + start ; j ++)
		{
			int key = arr[j];
			int i = j - 1;

			while(true){
				comparisonCounter++;
				if(arr[i] > key){
					if(i >= start){
						assignmentCounter++;
						int temp = arr[i+1];
						arr[i+1] = arr[i];
						arr[i] = temp;
						i--;
					}
					if(i < 0)
						break;
				} else
					break;
			}
			arr[i+1] = key;
		}
	}

	public void setInsertionSortThreshold(Integer threshold){
		insertionSortThreshold = threshold;
	}

	@Override
	public void sort(int [] arr, int start, int end){
		int lengthOfArray = end - start + 1;

		if(start < end){
			if(lengthOfArray <= insertionSortThreshold)
				modifiedInsertionSort(arr, start, end);

			else {
					int pivot = partition(arr, start, end);

					sort(arr, start, pivot - 1);
					sort(arr, pivot + 1, end);
			}
		}
	}

	@Override
	public void sortAndPrint(int[] arr, int start, int end, String msg) {
		super.sortAndPrintArray(arr, start, end, msg);
		System.out.println("Insertion sort threshold: " + insertionSortThreshold);
	}
}

------------------------------------------------------------------------------------------------------------
//TestHelper.java
package edu.uncc.helpers;

import edu.uncc.constants.Constants;
import edu.uncc.sorter.impl.ModifiedQuickSorter;
import edu.uncc.sorter.impl.QuickSort;

public class TestHelper {
        public void testCorrectnessOfAlgorithms(int arraySize, int maxRandNo, int insertionSortThreshold){

                //declare arrays for the normal and the modified quick sorters.
                int [] arrayForModifiedQuickSorter = new int[arraySize] ;
                int [] arrayForNormalQuickSorter = new int[arraySize] ;

                //Initialize the arrays with the same set of random numbers.
                RandomNumberHelper.resetSeedsCounter();
                RandomNumberHelper.fillArrayWithRandomNumbers(arrayForNormalQuickSorter, maxRandNo, arraySize, 0);

                RandomNumberHelper.resetSeedsCounter();
                RandomNumberHelper.fillArrayWithRandomNumbers(arrayForModifiedQuickSorter, maxRandNo, arraySize, 0);

                ModifiedQuickSorter modifiedQuickSorter = new ModifiedQuickSorter(insertionSortThreshold);
                QuickSort quickSorter = new QuickSort();

                ArrayHelper.printArray(arrayForNormalQuickSorter, "Array for normal   quick sorter: ");
                ArrayHelper.printArray(arrayForModifiedQuickSorter, "Array for modified quick sorter: ");

                modifiedQuickSorter.sortAndPrint(arrayForModifiedQuickSorter, 0, arraySize - 1, "Sorted by modified quick 
sort: ");
                quickSorter.sortAndPrint(arrayForNormalQuickSorter, 0, arraySize - 1, "Sorted by normal quick sort: ");
        }

        public void testAlgorithms(){

                //Declare and initialize the sorter objects.
                ModifiedQuickSorter modifiedQuickSorter = new ModifiedQuickSorter();
                QuickSort quickSorter = new QuickSort();

                //Declare arrays for holding the testing parameters.
                Integer [] maxRandNos = {100, 1000, 10000, 100000};
                Integer [] insSortThresholds = { 3, 4, 5, 6, 7, 8, 10, 12, 15, 18, 20, 25, 50, 75, 100, 120, 130, 140, 
150};

                //Initialize the StatsHelper
                StatsHelper.initializeStatsMaps(insSortThresholds);

                //Declare an array to hold values for sorting
                int [] arrayToSort = new int[Constants.MAX_ARRAY_SIZE];

                for(Integer maxRandNo : maxRandNos){

                        //Below loop for calculating the counts of modified quick sorter.
                        //Run CONSTANTS.MAX_ITER_FOR_AVERAGE times to get an average.
                        for(int i = 0 ; i < Constants.MAX_ITER_FOR_AVERAGE ; i ++){

                                //Run for each integer threshold.
                                for(Integer insSortThreshold : insSortThresholds){
                                        //Get an array.
                                        RandomNumberHelper.fillArrayWithRandomNumbers(arrayToSort, maxRandNo, i);

                                        //set the threshold and sort the array.
                                        modifiedQuickSorter.setInsertionSortThreshold(insSortThreshold);
                                        modifiedQuickSorter.sort(arrayToSort, 0, Constants.MAX_ARRAY_SIZE - 1);

                                        //save the comparisons and assignments
                                        StatsHelper.addComparisonValueAgainstKey(insSortThreshold, 
modifiedQuickSorter.getComparisonCounter());
                                        StatsHelper.addAssignmentValueAgainstKey(insSortThreshold, 
modifiedQuickSorter.getAssignmentCounter());

                                        //reset the comparisons, since we use the same sorter object.
                                        modifiedQuickSorter.resetComparisonCounter();
                                        modifiedQuickSorter.resetAssignmentCounter();
                                }

                                //Below loop for calculating counts of the normal quick sorter.
                                //Get the same array.
                                RandomNumberHelper.fillArrayWithRandomNumbers(arrayToSort, maxRandNo, i);

                                //sort and save counts.
                                quickSorter.sort(arrayToSort, 0, Constants.MAX_ARRAY_SIZE - 1);

                                //Save the count
                                StatsHelper.addValueInNormalSorterComparisonList(quickSorter.getComparisonCounter());
                                StatsHelper.addValueInNormalSorterAssignmentList(quickSorter.getAssignmentCounter());

                                //reset the sorter counter.
                                quickSorter.resetComparisonCounter();
                                quickSorter.resetAssignmentCounter();
                        }

                        StatsHelper.computeAndPrintStats(maxRandNo);
                }
        }
}


//GraphPlotter.java
package edu.uncc.graphs;

import java.awt.Color;
import java.util.HashMap;
import java.util.List;

import javax.swing.JFrame;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;
import org.jfree.data.xy.XYDataset;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;
import org.jfree.ui.RefineryUtilities;

@SuppressWarnings("serial")
public class GraphPlotter extends JFrame {

    public GraphPlotter(HashMap<Integer,List<Long>> comparisonsStats, HashMap<Integer, List<Long>> assignmentStats,
                List<Long> normalSorterComparisonList, List<Long> normalSorterAssignmentList, final String title) {
        super(title);

        final  XYDataset  dataset  =  createDataChartWith(comparisonsStats,  assignmentStats,  normalSorterComparisonList,
normalSorterAssignmentList);
        final JFreeChart chart = createChart(dataset);
        final ChartPanel chartPanel = new ChartPanel(chart);
        chartPanel.setPreferredSize(new java.awt.Dimension(500, 270));
        setContentPane(chartPanel);

    }

    public XYDataset createDataChartWith(HashMap<Integer, List<Long>> comparisonsStats,
                HashMap<Integer,   List<Long>>   assignmentStats,   List<Long>   normalSorterComparisonList,   List<Long>
normalSorterAssignmentList) {

        final XYSeriesCollection dataSet = new XYSeriesCollection();

        //Compute the comparisons for normal quick sort.
        long normalComparisonSum = 0, normalAssignmentSum = 0;
        for(Long i : normalSorterComparisonList)
        normalComparisonSum += i;
        for(Long i : normalSorterAssignmentList)
        normalAssignmentSum += i;

        double avgForNormalSortComparisons = normalComparisonSum / (normalSorterComparisonList.size());
        double avgForNormalSortAssignments = normalAssignmentSum / (normalSorterAssignmentList.size());

        XYSeries comparisonCountSeries = new XYSeries("Comparisons in Modified Quick Sorter");
        XYSeries assignmentCountSeries = new XYSeries("Assignmenets in Modified Quick Sorter");
        XYSeries normalSorterComparisonSeries = new XYSeries("Comparisons in Normal Quick Sorter");
        XYSeries normalSorterAssignmentSeries = new XYSeries("Assignments in Normal Quick Sorter");

        long comparisonSum = 0, assignmentSum = 0;

        for(Integer key : comparisonsStats.keySet()){
        comparisonSum = 0;
```

```java
        assignmentSum = 0;

        for(Long count : comparisonsStats.get(key))
                comparisonSum += count;

        for(Long count : assignmentStats.get(key))
                assignmentSum += count;

        comparisonCountSeries.add((double)key, (comparisonSum / (comparisonsStats.get(key).size())));
        assignmentCountSeries.add((double)key, (assignmentSum / (assignmentStats.get(key).size())));
        normalSorterComparisonSeries.add((double)key, avgForNormalSortComparisons);
        normalSorterAssignmentSeries.add((double)key, avgForNormalSortAssignments);
        }

        dataSet.addSeries(comparisonCountSeries);
        dataSet.addSeries(assignmentCountSeries);
        dataSet.addSeries(normalSorterComparisonSeries);
        dataSet.addSeries(normalSorterAssignmentSeries);

        return dataSet;

    }

    private JFreeChart createChart(XYDataset dataset) {

        final JFreeChart chart = ChartFactory.createXYLineChart("Graphs of Comparisons", "Ins. sort Thresholds",
                "No. of Comparisons", dataset, PlotOrientation.VERTICAL, true, true, false
        );

        chart.setBackgroundPaint(Color.white);

        final XYPlot plot = chart.getXYPlot();

        plot.setDomainGridlinePaint(Color.white);
        plot.setRangeGridlinePaint(Color.white);

        XYLineAndShapeRenderer renderer = new XYLineAndShapeRenderer();
        plot.setRenderer(renderer);

        final NumberAxis rangeAxis = (NumberAxis) plot.getRangeAxis();
        rangeAxis.setStandardTickUnits(NumberAxis.createIntegerTickUnits());

        return chart;

    }

    public   static   void   plotGraph(HashMap<Integer,   List<Long>>   comparisonsStats,   HashMap<Integer,   List<Long>>
assignmentStats,
                List<Long> normalSortComparisonList,  List<Long> normalSorterAssignmentList, String title) {

        GraphPlotter   graph   =   new   GraphPlotter(comparisonsStats,   assignmentStats,   normalSortComparisonList,
normalSorterAssignmentList, title);
        graph.pack();
        RefineryUtilities.centerFrameOnScreen(graph);
        graph.setVisible(true);
        graph.setDefaultCloseOperation(DISPOSE_ON_CLOSE);

    }
}

//ArrayHelper.java
package edu.uncc.helpers;

public class ArrayHelper {

        public static void printArray(int [] arr, String msg){
                System.out.print(msg);

                for(int i = 0 ; i < arr.length ; i ++)
                        System.out.print(arr[i] + " ");

                System.out.println();
        }
}

//RandomNumberHelper.java
```

```
package edu.uncc.helpers;

import java.util.Random;

import edu.uncc.constants.Constants;

public class RandomNumberHelper {

        private static int [] seeds;
        private static int seedCounter;
        private static Random randGenerator;

        static {
                randGenerator = new Random();
                randGenerator.setSeed(Constants.INITIAL_SEED);
                seeds = new int [Constants.MAX_ITER_FOR_AVERAGE];

                for(int i = 0 ; i < Constants.MAX_ITER_FOR_AVERAGE ; i ++)
                        seeds[i] = randGenerator.nextInt(Constants.MAX_RAND_NO_FOR_SEEDS);

                seedCounter = 0;
        }

        public static void fillArrayWithRandomNumbers(int [] arr, int maxRandNo, int seedIndex){
                fillArrayWithRandomNumbers(arr,maxRandNo, 0 , seedIndex);
        }

        public static void fillArrayWithRandomNumbers(int [] arr, int maxRandNo){
                fillArrayWithRandomNumbers(arr, maxRandNo, 0, -1);
        }

        public static void fillArrayWithRandomNumbers(int [] arr, int maxRandNo, int noOfElements, int seedIndex){

                if(seedIndex == -1)
                        randGenerator.setSeed(seeds[seedCounter++]);
                else
                        randGenerator.setSeed(seeds[seedIndex]);

                if(noOfElements != 0){
                        for(int i = 0 ; i < noOfElements ; i ++)
                                arr[i] = randGenerator.nextInt(maxRandNo);
                } else {
                        for(int i = 0 ; i < Constants.MAX_ARRAY_SIZE ; i ++)
                                arr[i] = randGenerator.nextInt(maxRandNo);
                }
        }

        public static void resetSeedsCounter(){
                seedCounter = 0;
        }
}

//StatsHelper.java
package edu.uncc.helpers;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;

import edu.uncc.graphs.GraphPlotter;

public class StatsHelper {

        private static HashMap<Integer, List<Long>> comparisonStatsMap;
        private static HashMap<Integer, List<Long>> assignmentStatsMap;

        private static List<Long> normalSorterComparisonsList;
        private static List<Long> normalSorterAssignmentList;

        private static MinValue minComparisons;
        private static MinValue minAssignments;

        private static boolean plotGraphs;

        static {
```

```
		comparisonStatsMap = new HashMap<Integer, List<Long>>();
		assignmentStatsMap = new HashMap<Integer, List<Long>>();

		normalSorterComparisonsList = new ArrayList<Long>();
		normalSorterAssignmentList = new ArrayList<Long>();

		minComparisons = new MinValue();
		minAssignments = new MinValue();

		//don't plot graphs by default.
		plotGraphs = false;
	}

	public static void setPlotGraphs(boolean plotGraphs) {
		StatsHelper.plotGraphs = plotGraphs;
	}

	public static void initializeStatsMaps(Integer [] keys){
		for(Integer key : keys){
			comparisonStatsMap.put(key, new ArrayList<Long>());
			assignmentStatsMap.put(key, new ArrayList<Long>());
		}
	}

	public static void addComparisonValueAgainstKey(Integer key, Long value){
		comparisonStatsMap.get(key).add(value);
	}

	public static void addAssignmentValueAgainstKey(Integer key, Long value){
		assignmentStatsMap.get(key).add(value);
	}

	public static void addValueInNormalSorterComparisonList(Long l){
		normalSorterComparisonsList.add(l);
	}

	public static void addValueInNormalSorterAssignmentList(Long l){
		normalSorterAssignmentList.add(l);
	}

	public static void computeAndPrintStats(Integer maxRandNo) {

		System.out.println("----------------------------------------------------------------------------------");
		System.out.println("Stats for random numbers less than " + maxRandNo);
		System.out.println("----------------------------------------------------------------------------------");

		Object[] insSortThresholds = comparisonStatsMap.keySet().toArray();
		Arrays.sort(insSortThresholds);

		for(Object key : insSortThresholds){
			long comparisonAverage = 0;
			long assignmentAverage = 0;

			for(Long count : comparisonStatsMap.get(key))
				comparisonAverage += count;

			for(Long count : assignmentStatsMap.get(key))
				assignmentAverage += count;

			comparisonAverage /= comparisonStatsMap.get(key).size();
			assignmentAverage /= assignmentStatsMap.get(key).size();

			if(minComparisons.getValue() == 0 || comparisonAverage < minComparisons.getValue()){
				minComparisons.setValue(comparisonAverage);
				minComparisons.setKey((Integer) key);
			}

			if(minAssignments.getValue() == 0 || assignmentAverage < minAssignments.getValue()){
				minAssignments.setValue(assignmentAverage);
				minAssignments.setKey((Integer) key);
			}

			System.out.printf("Avg. (comparisons, assignments) with ins. sort threshold %3d: (%7d, %7d )\n",
key, comparisonAverage, assignmentAverage);
		}
		long averageComparisons = 0;
		long averageAssignments = 0;

		for(Long i : normalSorterComparisonsList)
			averageComparisons += i;

		for(Long i : normalSorterAssignmentList)
			averageAssignments += i;

		System.out.println("\nAverage comparisons with normal quick sort: " +
(averageComparisons/normalSorterComparisonsList.size()));
		System.out.println("Average assignments with normal quick sort: " +
(averageAssignments/normalSorterComparisonsList.size()) + "\n");

		System.out.println("Minimum most comparisons of " + minComparisons.getValue() + " were done with insertion
sort threshold " + minComparisons.getKey());
		System.out.println("Minimum most assignments of " + minAssignments.getValue() + " were done with insertion
sort threshold " + minAssignments.getKey());

		if(plotGraphs)
			GraphPlotter.plotGraph(comparisonStatsMap, assignmentStatsMap, normalSorterComparisonsList,
normalSorterAssignmentList, "Graph for numbers less than " + maxRandNo);

		//clear the data structures.
		resetDataStructures();

	}

	private static void resetDataStructures() {

		for(Integer i : comparisonStatsMap.keySet()){
			comparisonStatsMap.get(i).clear();
			assignmentStatsMap.get(i).clear();
		}

		normalSorterComparisonsList.clear();
		normalSorterAssignmentList.clear();

		minComparisons.reset();
		minAssignments.reset();

	}
}

class MinValue {
	Integer key;
	long value;

	public MinValue(){
		key = 0;
		value = 0;
	}

	public Integer getKey() {
		return key;
	}

	public void setKey(Integer insSortThreshold) {
		this.key = insSortThreshold;
	}

	public long getValue() {
		return value;
	}

	public void setValue(long avgComparisons) {
		this.value = avgComparisons;
	}

	public void reset(){
		setKey(0);
		setValue(0);
	}
}
```