CS102 – Algorithms and Programming II
Lab Programming Assignment 6
Spring 2022

ATTENTION:

- Feel free to ask questions on Moodle on the Lab Assignment Forum.
- Compress all of the Java program source files (.java) files into a single zip file.
- The name of the zip file should follow the below convention:
  CS102_SecX_Asgn6_YourSurname_YourName.zip
- Replace the variables "YourSurname" and "YourName" with your actual surname and name and X with your Section id (1, 2 or 3).
- Upload the above zip file to Moodle by April 26, 23:59 with at least Part 1 completed. Otherwise, significant points will be reduced. You will get a chance to update and improve your solution (Part 1 and Part 2) by consulting the TA during the lab. You will resubmit your code (Part 1 and Part 2 together) once you demo your work to the TA.

The work must be done individually. Codesharing and copying code from any source are strictly forbidden. We are using sophisticated tools to check the code similarities. We can even compare your code against online sources. The Honor Code specifies what you can and cannot do. Breaking the rules will result in a disciplinary action.

This lab is about Sorting. Throughout the lab, you can only use arrays. Other data structures such as Lists and ArrayLists are not allowed. Plus, any usage of built-in/external sorting functions (e.g., Collections.sort()) is forbidden.

Part 1

Question 1

Let's begin with a simple task. Create an interface called **ArrayGenerator** with the following method:

- public int[] **generate**(int n) => implementer of this interface should generate an array of size n in the range [1, n].

Then, create the following classes implementing the **generate** method of **ArrayGenerator** interface:

- **RandomArrayGenerator:** generate a random array of size n. (eg. [1, 8, 6, 4, 5, 2, 7, 3] for n = 8)
- **DecreasingArrayGenerator:** generate an array of size n whose elements are in decreasing order. (eg. [8, 7, 6, 5, 4, 3, 2, 1] for n = 8)
- **IncreasingArrayGenerator:** generate an array of size n whose elements are in increasing order. (eg. [1, 2, 3, 4, 5, 6, 7, 8] for n = 8)

Question 2

In this question, you'll take a deep look at the runtime characteristics of the MergeSort algorithm. That is, you will investigate how many comparisons MergeSort does. The number of comparisons is crucial as it is connected to the runtime complexity of the sorting algorithm. Now, create an abstract **SortAnalyzer** class, which has the following variables:

1. private int **numberOfComparisons** => you can define a getter for this variable.
2. protected int **k** => Don't worry about the **k** yet; it will make sense in the next question.

and at least following methods:

1. protected int **compare**(Comparable o1, Comparable o2) => compares two objects. This method should also update the number of comparisons.
2. public boolean **isSorted**(Comparable[] arr) => checks if the array is sorted. You cannot use Collections.sort() etc., even inside this method.
3. public abstract Comparable[] **sort**(Comparable[] arr) => sorting function that is being analyzed.

Now, create a concrete (not abstract) **MergeSortAnalyzer** class that extends **SortAnalyzer** class. Specifically, override the abstract sort method by implementing the MergeSort algorithm recursively.

Generate test arrays using the classes you implemented in Question 1 and ensure that they are sorted using your **isSorted** method. Print out the **numberOfComparisons** for each case. Which case seems to require the fewest comparisons?

Question 3

As you know, MergeSort is a divide-and-conquer algorithm that divides the problem into two halves and merges them after sorting each half. In this question, we want you to divide the problem into **k** instead of 2. So, you will develop a MergeSort algorithm that sorts the given array by dividing it into **k** sub-problems at each recursive step and merges them. Now, create a **GeneralizedMergeSortAnalyzer** class that extends **SortAnalyzer** class and implement the sort method recursively. Note that you will now use the **k** variable of the **SortAnalyzer** class. You cannot call the MergeSort that you developed in Question 2. That is, the sort method of this class should only call itself not another sort method.

Generate test arrays using the classes you implemented in Question 1 and ensure that they are sorted using your **isSorted** method. For each test case, experiment with different **k** values. Print out the **numberOfComparisons** for each case. Which **k** value seems to require the fewest comparisons?

Part 2

In this part, you will do the same for QuickSort.

Question 4

Create a concrete (not abstract) **QuickSortAnalyzer** class that extends **SortAnalyzer** class. Override the abstract sort method by implementing the QuickSort algorithm recursively.

Generate test arrays using the classes you implemented in Question 1 and ensure that they are sorted using your **isSorted** method. Print out the **numberOfComparisons** for each case. Which case seems to require the fewest comparisons?

Question 5

Now, create a **GeneralizedQuickSortAnalyzer** class that extends **SortAnalyzer** class and implement the sort method recursively. Again, you need to divide the QuickSort problem into **k** sub-problems at each step instead of 2. You cannot call the QuickSort that you developed in Question 4. That is, the sort method of this class should only call itself not another sort method.

Generate test arrays using the classes you implemented in Question 1 and ensure that they are sorted using your **isSorted** method. For each test case, experiment with different **k** values. Print out the **numberOfComparisons** for each case. Which **k** value seems to require the fewest comparisons?

IMPORTANT NOTES:

1. Please comment your code according to the documentation and commenting conventions used in the textbook.