Empirical Analysis – Sorting

Problem Overview

The focus of this assignment is on experimentation, analysis, critical thinking, and applying the scientific method. The provided resources for this assignment are the following.

- `resources.jar` — A jar file containing various classes needed for this assignment. One class, `SortingLab`, has five methods for which you must experimentally determine the sorting algorithms implemented.
- `SortingLabClient.java` — Source code of a class that illustrates basic calls to public methods of `SortingLab`. *This class is for illustration only.* You can modify and use it for your own purposes or you can use it as an example to create a different but similar class of your own.
- `sample-report.pdf` — A sample report that you are required to use as a style, structure, and formatting guide for the deliverables that you submit for this assignment.

Experimental Identification of Sorting Algorithms

You must develop and perform a repeatable experimental procedure that will allow you to empirically discover the sorting algorithms implemented by the five methods of the `SortingLab` class — `sort1`, `sort2`, `sort3`, `sort4`, `sort5`. The five sorting algorithms implemented are merge sort, randomized quicksort, non-randomized quicksort, selection sort, and insertion sort.

Insertion sort, selection sort, and merge sort are implemented exactly as described in lecture and the associated note set. Quicksort has two implementations, randomized and non-randomized. Both quicksort implementations select the pivot in the same way: the left-most element (i.e., `a[left]`) of the current partition is always used as the pivot. Both implementations also use the same algorithm for the partitioning operation, although it varies slightly from the one presented in lecture. The two quicksort implementations are different, however, in the following way. The randomized quicksort implementation makes the worst case probabilistically unlikely by randomly permuting the the array elements before the quicksort algorithm begins. The non-randomized quicksort exposes the algorithm's worst case by never shuffling the array elements.

You must apply what we know about the *time complexity* of each sorting algorithm, along with *stability* in the case of merge sort and quicksort, to match each sorting method with the sorting algorithm that it implements. You may find jGRASP viewers, debugger information, and other data useful in confirming your findings, but your conclusions must be fully supported by timing data and stability data that you collect.

To generate timing data you are required to use `System.nanoTime()`, as illustrated in `SortingLabClient`. Time values must be expressed in seconds. Using time data to empirically discover the big-O time complexity of various methods will allow you to identify all sorting algorithms, but it likely won't be sufficient to distinguish merge sort from randomized quicksort. Since both algorithms have $O(N \log N)$ time complexity, you should use *stability* to distinguish the two. Recall that the merge sort implementation is stable while the randomized quicksort implementation is not.

To document your work you must write a lab report that fully describes the experiments used to discover the sorting algorithm associated with each of the five sort methods. The lab report must discuss the experimental procedure (what you did), data collection (all the data you gathered), data analysis (what you did with the data), and interpretation of the data (what conclusions you drew from the data). The lab report must be well written, it must exhibit a high degree of professionalism, and it must be structured like the provided sample. Your experiment must be described in enough detail that it could be reproduced by someone else.

Using the provided `jar` file

To compile and run the provided source files and any of your own that might use for this assignment, you will need to include `resources.jar` on the classpath. You can do this from the command line or from within your IDE.

From the command line. In the following example, `>` represents the command line prompt and it is assumed that the current working directory contains both the source code and the jar file.

```
> javac -cp .:resources.jar SortingLabClient.java
> java -cp .:resources.jar SortingLabClient
```

From the jGRASP IDE.

1. Create a project for the assignment (suppose you call it SortingProject), and include `SortingLabClient.java`.
2. Click on *Settings* → *PATH/CLASSPATH* → *Project* → *<SortingProject>*.
3. Click on the *CLASSPATHS* button.
4. Click *New*.
5. In the "Path or JAR File" text box, use the *Browse* button to navigate to `resources.jar` and select it.
6. Click on *OK*.
7. Click on *Apply*.
8. Click on *OK*.

The constructor `SortingLab(int key)` will create a SortingLab object whose sort method implementations are tailored specifically to the given key value. You will use your Banner ID number as the key required by the constructor. For example, one student might invoke the constructor `SortingLab(903111111)` and another student might invoke the constructor `SortingLab(903222222)`. This will create two distinct objects with (very likely) sorting algorithm to sort method mappings. Note that you **must** use your own Banner ID since grading will be based on Banner ID rather than student name.

Increasing the JVM stack size

It may be necessary for you to increase the amount of memory allocated to the JVM's runtime stack. The `Xss` flag is used to request an amount of memory in bytes. For example, the following command runs the `SortingLabClient` class with four gigabytes of stack memory.

```
java -Xss4G -cp .:resources.jar SortingLabClient
```

You can set the `Xss` flag in jGRASP at *Settings* → *Compiler Settings* → *Workspace* → *Flags/Args* → *FLAGS2 or ARGS2*.

Assignment Submission

The only deliverable for this assignment is an electronic copy of your lab report in either PDF or Microsoft Word format. The lab report must be uploaded to Canvas no later than the date and time specified for full credit. Late submissions will be accepted up to three days late with a 10% penalty per day.