CS2040S: Data Structures and Algorithms

Problem Set 3

Collaboration Policy. You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). You can do so in Coursemology by adding a Comment. Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

Problem 1. (The Sorting Detectives)

After being forced to sort arrays for your CS2040S assignments one too many times, you have finally decided to outsource the sorting process to one of the many *sorting vendors* around. These vendors each have their own trademark method of sorting, and you, being a good student who paid attention during the lectures, decided that **Mr. QuickSort** will give you the best bang for your buck. After a little bit of research, you managed to narrow the search for **Mr. QuickSort** down to five choices, but every single one of them are claiming to be the one you are looking for, and only one of them is telling the truth. Three of the other four are just harmless imitators, **Ms. SelectionSort**, **Mr. InsertionSort**, and **Ms. MergeSort**. Beware, however, one of the impostors is *not a sorting algorithm*! **Dr. Evil** maliciously returns unsorted arrays! And he won't be easy to catch. He will try to trick you by often returning correctly sorted arrays. Especially on easy instances, he's not going to slip up.

Your job is to investigate, and identify who is who. Attached to this problem set, you will find five sorter implementations: (i) `SorterA.class`, (ii) `SorterB.class`, (iii) `SorterC.class`, (iv) `SorterD.class`, and (v) `SorterE.class`.

These are provided in a single JAR file: `Sorters.jar`. Each of these class files contains a class that implements the `ISort` interface which supports the following method:

```
    public long sort(KeyValuePair[] array);
```

Everytime you call the Sorter.sort method, it will return an integer, representing how much the Sorter is charging you to sort this array. The amount they charge scales with the amount of effort needed to sort this array, i.e. the amount of comparisons/time taken needed to sort the array. Furthermore, as all the sorters are from foreign countries, and each of them will charge you in their own currency. With no knowledge of exchange rate, *you won't be able to directly compare the amount they are charging across the Sorters. You will only be able to compare the amount the same Sorter charges for different arrays.*

Note: When calling the method `public long sort(KeyValuePair[] array)`, ensure that the size of your array is at most $\approx 30{,}000$. This is not a hard limit (indeed, we do not enforce this in any way) but instead a guideline so that the returned `cost` does not run into any weird overflow issues.

You can find the code for the `KeyValuePair` class attached as well. It is a simple container that holds two integers: a key and a value. The sort routines will sort the array of objects by key.

You can test these sorting routines in the normal way: create an array, create a sorter object, and sort. See the example file `SortTestExample.java`.

Each sorting algorithm has some inputs on which it performs better, and some inputs on which it performs worse. Some sorting algorithms are stable, while others are not. Using these properties, you can figure out the real identities of the sorters, and also identify Dr. Evil.

IntelliJ tips: Refer to `setup.mp4` on Coursemology for instructions on setting up PS3 in IntelliJ.

Note: You are only allowed to modify `SortingTester.java` in your submission. Any modifications made to the other files provided (i.e., `ISort.java`, and `KeyValuePair.java`) will not be accepted.

0.1 Sorting Stability

One property that will be helpful for you to help you discern between sorting algorithms is the notion of **stability**. Informally, stability talks about the **relative ordering** between two identically valued items and whether it is preserved after the sorting algorithm is done.

For example, given the following array:

$$[1_a, 2, 1_b]$$

There are two copies of 1, they have been subscripted with a and b to mark which order they appear in the array. Sorting algorithms may opt to output either:

$$[1_a, 1_b, 2]$$

or

$$[1_b, 1_a, 2]$$

because in terms of value comparisons, both copies of the value 1 now appear before 2 in the final array.

However, if an algorithm *always guarantees* that the **initial relative ordering of identically valued items** is **always preserved**, then the algorithm is considered **stable**. If no such guarantees hold, then the algorithm is considered **unstable**.

So, stable sorting algorithms are only allowed to ever output the first array, and not the second.

0.1.1 Stability of the sorting algorithms in CS2040S

As mentioned, this problem set is about distinguishing between the 4 known sorting algorithms in CS2040S. Here are a few tips to get you started on reasoning about the stability of the 4 algorithms:

1. Insertion sort will iteratively move elements towards the left **as long as** the next element (on the left) is **strictly larger than it**. You should think about what that means on an array where there are duplicates. What about an array like $[1_a, 1_b]$. Will the relative ordering ever be changed?

2. If in the merge step of merge sort, the 2 elements being compared in the left half vs. the right half are the same, as long as we always opt to take the left element, then the algorithm is stable.

3. The crux of quicksort lies in the partitioning algoirthm. Based on the in-place partitioning algorithm given in the slides, can you guarantee that the relative ordering is always preserved when the partition step is run?

4. Similarly, the crux of selection sort lies in what we are deciding to swap. Can you guarantee that the relative ordering is always preserved? (Even if we always swap based on the first minimum found, consider an array like $[2_a, 2_b, 1]$. What will selection sort do?)

Problem 1.a. Write a routine `boolean checkSort(ISort sorter, int size)` that runs a test on an array of the specified size to determine if the specified sorting algorithm sorts correctly.

Problem 1.b. Write a routine `boolean isStable(ISort sorter, int size)` that runs a test on an array of the specified size to determine if the specified sorting algorithm is stable.
Note: Any sort performed on an array of size 0 or 1 is vacuously stable.

Problem 1.c. Write whatever additional code you need in order to test the sorters to determine which is which. All evidence you give below must rely on properties of the sorting algorithms, along with data from your tests that supports your claim. **You are strictly prohibited from analyzing or decompiling the provided files to identify the sorters as it defeats the main purpose of this problem set.**

Problem 1.d. What is the true identity of `SorterA`? Give the evidence that proves your claim.

Problem 1.e. What is the true identity of `SorterB`? Give the evidence that proves your claim.

Problem 1.f. What is the true identity of `SorterC`? Give the evidence that proves your claim.

Problem 1.g. What is the true identity of `SorterD`? Give the evidence that proves your claim.

Problem 1.h. What is the true identity of `SorterE`? Give the evidence that proves your claim.