Assignment 4

This fourth assignment focuses on implementation and comparison of the various sorting techniques considered so far.

Updating your repository

Most of the hard work was done in the last lab. All you need to do is fetch and merge the changes I made to the upstream repository.

Before we start anything, make sure that you have NO uncommitted changes. You must either commit what you have, if it is ready to commit, or stash the changes, allowing you to retrieve them later.

1. First make sure that your local master is the active branch. It should have a little checkbox next to it.
2. We should make sure to bring in the newest version of the upstream remote. Right-click on the "upstream" remote in the "Remotes" section on the left, and choose "fetch upstream".
3. Now right-click on the upstream/master branch within the "Remotes" section to bring up its context menu. One of the options will say "Merge upstream/master onto master". Click it.
4. You should now be ready to go. Whenever you push your changes to your origin, these commits from the upstream will go along with them. If you like, you can go ahead and push now.

The assignment

Introductory stuff

In this assignment there are two files you will be working on: `Sort.java`, inside `src/main/java/utils` is the main file that you will have to edit. You may also need to look into the `CompArray.java` file.

As in the previous assignment, you will find numerous TODO comments throughout the file. They mark the locations where you need to add something to the implementations. Some times it will be one or two lines, some times it may be more. If you use Eclipse, you can see blue rectangles on the right side of the code indicating the location of all TODOs, and you can click on them to go there. You can keep track of your progress that way, so make sure to remove the TODO from the methods you have already implemented.

A standard printout of a run of the various methods can be obtained by running the main method of the Sort class (so running the Sort class as an application). To do that, you will need to first build using the corresponding gradle task, and then run the runSort gradle task from the run group of tasks (you may need to refresh the gradle project and possibly close and reopen the gradle tasks to make it appear).

Alternatively, you can go to your terminal (or Eclipse's console), go to the main project folder and run:

```
java -cp build/libs/algorithms-assignments.jar utils.Sort
```

If you like to write more tests, you can create test files under /src/test/java. Ask for help if you are not sure how to start those tests. You can use the :test Gradle task to run the tests.

You should make a new *commit* in GitKraken after completing each self-contained part, typically after each method you complete. Do not try to commit everything at once.

When you are done with the assignment, in order to "submit" it, simply **push** your commits via GitKraken. These will now be posted on your account in GitLab and I will be able to follow them from there.

The structure of the files

There are two file of concern in this assignment. Sort.java and CompArray.java.

CompArray.java contains a wrapper around arrays. You will implement sorting algorithms on arrays but you are specifically asked to use the CompArray wrapper in the process. This is designed so that it counts how many comparisons and how many writes your algorithms do and prints those out.

Here are the main elements of the CompArray class, that you have at your disposal:

- The elements of the array are actually instances of the CompArray.CompElement class. You should take a look at its implementation within CompArray, around line 39.
  - It has a constructor CompElement() that generates a random element within a provided range, and a CompElement(key) constructor that generates a element with a specific value key. These are values are integers.
  - CompElement objects can be compared via a series of provided methods with straightforward names. For example, instead of doing $a < b$ you would be doing a.lessThan(b). Each time you do a comparison is counted.
- A new empty CompArray of a given size can be via the CompArray(size) constructor.
- A new array populated with random entries can be constructed via the makeRandom(size, bound) static function. See the main function in the Sort class for an example.
- You can interact with a CompArray instance array via a set of public instance methods:

- array.get(i) is the analog of doing A[i] for a normal array, and it will throw an error if you are out of bounds.
- array.put(i, v) is the analog of doing A[i] = v for a normal array, and will similarly throw an error if you are out of bounds.
- As an example of this, if you wanted to do the operation A[k] = A[i], then you would instead do something more like array.put(k, array.get(i)).
- array.swap(i, j) will swap the values in places i, j in the array.
- array.size() returns the array length.
- array.getNumComparisons() returns the number of times that array elements have been compared.
- array.getNumWrites() returns the number of times that array elements have been written, either via a call to put or via a call to swap (which does two writes).
- Before starting to test a sort method, you should reset these counts by calling CompArray.resetCounts(). The sample test in the main method of Sort does this.
- You can print the whole array by turning it into a string via its toString method. This also happens automatically if you use the array in a print call.

The Sort class is the class where you will implement a number of different sort methods.

All methods are expected to change the array in-place, rather than return anything.

The method insertionSort is implemented for you as an example. You should study it and compare it to the implementation in the book, to understand how the system works.

The actual tasks

Now on to the tasks you need to complete. Make sure you have read the previous section for an understanding of the structure of the classes you will be working with.

You will be implementing five sort methods. For each method the skeleton is provided for you, and you are asked to fill in the details.

1. The first method is selectionSort. It should implement the selection sort algorithm on page 99. In particular, your implementation should be doing a suitably small number of writes/swaps.
2. The next method you should implement is bubbleSort. This of course implements the bubble sort algorithm on page 100.
3. Next we have mergeSort, which of course implements the algorithm on page 172. The skeleton for mergeSort is provided for you.

- In the mergeSort method itself, the only two things you need to do is implement the two for loops that fill in the values in the arrays array1 and array2. These steps correspond to the two "copy" operations in the algorithm.
- The method then recursively calls itself on the two arrays, then calls the merge method. You do not need to do anything with that code.
- You DO need to fill in the parts of the merge method, which follows the mergeSort method. This implements the algorithm at the bottom of page 172. You will need to implement the body of the while loop as well as the two copy steps in that algorithm.

4. Next we have quickSort, which implements the QuickSort method with Hoare partitioning and without a special pivot mechanism.

    - The quickSort method itself calls the qSort helper with the array bounds as initial endpoints. You don't need to do anything there.
    - The qSort helper method is meant to sort the array between the two provided locations l and r, and it corresponds to the algorithm on page 176. This is also written for you, and you do not need to do anything.
    - You do need to implement the partition method that follows, which corresponds to the algorithm on page 178.
    - There is a possible extra-credit component to this problem. The default implementation as described in page 178 will occasionally throw an error, as it is possible that the index i may go out of bounds. This will happen on some of your runs, but not all. The extra-credit component is for suitably fixing your implementation of partition so that does not have this problem.

5. Lastly, we have heapSort. The heapSort method itself is done for you, and it takes care of the fact that arrays for heapSort need to start at index 1.

    - heapSort uses the heapBottomUp method of page 229 to heapify the array, and then does successive maximum element deletions to get the sorted array. This is done for you.
    - What you have to implement is the heapPercolateDown method, which corresponds to the contents of the for-loop in algorithm 229. This method is given an root index i, and it is supposed to push the element at that index i down to the correct spot.

And we're done!