CS 171: Introduction to Computer Science II
Assignment #4: Comparing Sorting Algorithms

Due: Tuesday Nov 15 at 11:59PM sharp [Late policy: see Canvas].

Early submission bonus (+3 points) if submitted early, at or before Friday Nov 11 at 11:59PM

Goals: For this assignment, you will put on a scientist hat! You are given several sorting algorithms that are already implemented, but you will implement a few optimizations and variations of them. Then, you will compare the average running times of these various sorting algorithms on array inputs of different sizes. You will also visualize the results on a chart.

Step 1: Implement these sorting algorithms

The starter code file `Sorting.java` contains the implementation of several sorting algorithms already. Your first task is to complete the implementation of the algorithms described below.

[10 points] Selection Sort: Fill in the method `SelectionSort(long[] a)` with code that implements the standard Selection Sort algorithm discussed in class.

[25 points] Optimized QuickSort: The classical quicksort algorithm is provided in the starter code. Implement an optimized version of quicksort in `QuickSortOptimized(long[] a)` which also uses recursion to apply quicksort to an array, but should include two specific optimizations:

- When the number of elements being sorted is below a certain cutoff limit of your choice (e.g. 10 elements), your quicksort method should switch to using *Insertion Sort* instead, which incurs a smaller memory overhead and performs less swaps.
- To protect against quicksort's worst-case performance of $O(N^2)$, your optimized method should randomly shuffle the elements in the array first before sorting them. You will need to implement the helper method `shuffle(long[] a)`. Notice that shuffling the array can incur a small run-time overhead, but the performance benefits should outweigh the cost in scenarios where the array is already sorted, semi-sorted, reversed, etc.

Note that you may utilize the `partition()` method already implemented in the starter code to accomplish quicksort partitioning, and you may also add additional methods if needed.

[30 points] Non-Recursive MergeSort: Fill in the method `MergeSortNonRec(long[] a)` with code that implements a non-recursive version of mergesort. In a non-recursive mergesort, we start by a pass of 1-by-1 merges (merge every two adjacent elements, subarrays of size 1, to make subarrays of size 2), then a pass of 2-by-2 merges (merge subarrays of size 2 to make subarrays of size 4), then 4-by-4 merges and so forth, until we do a merge that encompasses the entire array. You may call the `merge()` that is already implemented in the starter code to accomplish merging. Similar to recursive merge sort, you need to create auxiliary arrays in order to call this `merge()` method. In addition, you need to correctly calculate the `lo`, `mid`, and `hi` indices and pass them to the `merge()` method. Alternatively, you may implement your own version of the `merge()` method if you wish.

You can assume that the size of the input is always a power of 2. This reduces the complexity of the solution. If your implementation is correct, it should in fact run slightly faster than the recursive

mergesort for very large arrays, because there is no overhead of recursive method calls (although the performance improvement may not be very notable in relatively small arrays).

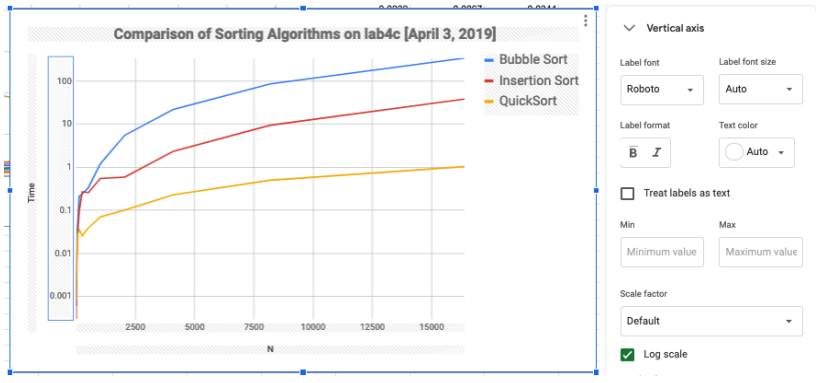
Step 2: Empirical Comparison and Analysis of Sorting Algorithms

The `main` method in `Sorting.java` compares the performance of 7 variations of sorting algorithms, including the ones you implemented above. For each algorithm, it tests different input sizes (default is 14 which tests the first 14 powers of 2), and the input data are randomly generated. It performs each test multiple times (default is 5) in order to get an average running time. Afterward, it will print the results as a table: each row reports the input size (N), and the average running time for each algorithm in milliseconds. Familiarize yourself with the code in the `main` method and make sure you understand how the experiments are carried out.

Note that the starter code checks after each sorting algorithm whether the sorting result is correct. If your sorting implementation produces incorrect sorting results, a message 'The sorting is INCORRECT!' will be printed out. Watch for such messages. The autograder will deduct points if your sorting result is incorrect.

Run the `main` method and examine the table to compare the run times for the various algorithms. By default, the input data is **randomly** generated. When you're done running the code in default mode and making sure your algorithms all work correctly, change the variable `useOrderedArray` in the main method to `true` to run all sorting algorithms on an **already sorted** array. You will report the final results from both the average case and the ordered, corner case.

Plotting the Running Times: UPDATE CHART TO REFLECT INSTRUCTIONS! After the program prints out the table, plot the results in graphs. You can use Google SpreadSheet, Microsoft Excel, OpenOffice, or other tools. The following are the steps using Google Spreadsheet. If you use Microsoft Excel, or OpenOffice Calc, the steps should be similar. Just make sure that you select "Line chart" as your chart type. 1. For each experiment, first copy the running times reported to a spreadsheet. 2. Click on "Insert" → "Chart", and select "Use row 1 as headers" and "Use column A as labels." Select "Line" chart.



This chart will help you visualize the difference of the running times of all the sorting algorithms, and also the trend when N grows large (the example shown here only plots 3 algorithms). You must label both axes (i.e., N and Time for the x and y axis respectively) and ensure there are appropriate tick values so that we can tell what the axis range is. You *must* set your vertical axis scale to *Log* scale (as shown in the figure) to visualize the results clearly. Also be sure that your legend is on the right hand side.

The title of the chart should be "Comparison of Sorting Algorithms on Random Array". Finally, click on the drop down menu from the chart and select "Save image" to save the image to a disk file, and name it **Sorting.png** . It is important to save it as a PNG file as this is a lossless compression so it doesn't lose detail and quality after you save it!

For the second experiment where you enable the `useOrderedArray` flag, name the corresponding plot **SortingOrdered.png**. Again, make sure that you label and mark both axes, the vertical axis scale is on the *Log* scale, and that the legend is on the right hand side. The title of the chart should be "Comparison of Sorting Algorithms on Ordered Array".

Getting started: Step by Step

1. Download the starter code `Sorting.java` and `SortHelper.java`.
2. Complete the required methods in the class. Test them incrementally and gradually!
3. Run `Sorting.java` to compare the running times for different sorting algorithms.
4. Plot the performance chart (File name: Sorting.png).
5. Run `Sorting.java` again with `useOrderedArray` set to `true` and compare the running times again.
6. Plot the performance chart (File name: SortingOrdered.png).

Submission Instructions

You must submit all 3 files: **Sorting.java**, **Sorting.png**, and **SortingOrdered.png**.

Grading

- Correctness and robustness: your algorithms work correctly for any input array (that has a size of a power of 2) (65 points)
- Efficiency: the relative running time of the algorithms matches their expected behavior; e.g., your optimized quicksort is more efficient than basic quicksort (particularly when the input array is ordered) (15 points)
- Correctness of the plots: your running time charts are correctly plotted and match your code results (10 points)
- Coding style (10 points)

Honor Code

The assignment is governed by the College Honor Code and Departmental Policy. Please remember to have the following comment included at the top of the file.

```
/*
THIS CODE WAS MY OWN WORK, IT WAS WRITTEN WITHOUT CONSULTING
CODE WRITTEN BY OTHER STUDENTS OR ONLINE SOURCES. _Student_Name_Here_
*/
```

Submission Checklist: We've created this checklist to help you make sure you don't miss anything important. Note that completing all these items does not guarantee full points, but at least assures you are unlikely to get a zero.

- ☐ Did your file compile on the command line using JDK 11 or above (e.g. JDK 17)?
- ☐ Did your submission on Gradescope successfully compile and pass at least one autograder test case?

☐ Have you included the honor code on top of the file?

☐ Did you remove the `TODO` prefix from the methods you needed to implement?

☐ Did you give your variables meaningful names (i.e., no `foo` or `bar` variables)?

☐ Did you add meaningful comments to the code when appropriate?

☐ Did you submit the two sorting plots and make sure they are appropriately named?