COMP 2140 Assignment 1: Sorting and Recursion

Helen Cameron and Robert Guderian

Due: Friday, October 12, 2018 at 4:30 p.m.

Important Announcement

Until you agree to the honesty declaration (see the instructions in the next paragraph), you cannot hand in any assignments — that is, until you agree to the honesty declaration, you will not be able to see the assignment folders in UM Learn.

This blanket honesty declaration covers ALL your work in the course. To agree to the honesty declaration:

- Go to the COMP 2140 homepage on UM Learn, look at the main header (the navigation bar) at the top. Along with "Resources", "Communication", and "Assessments", you will see "Checklist" — click on "Checklist".
- If you can see a blue "Honesty Declaration" above the red "Note" near the top (and you canNOT see the "Save" option at the bottom), click on the blue "Honesty Declaration" at the top.
- Read the honesty declaration, check off "Agreement" (verifying that you have read the honesty declaration and agree to it), and finally, click "Save". Only after this task is completed will you be able to see any assignment folders (under "Assessments" in the navigation bar at the top, click on "Assignments").

You only have to agree to the honesty declaration once, because it covers all your work in the course.

Hand-in Instructions

Go to COMP2140 in UM Learn; then, under "Assessments" in the navigation bar at the top, click on "Assignments". You will find an assignment folder called "Assignment 1". (If you cannot see the assignment folder, follow the directions under "Honesty Declaration" above.) Click the link and follow the instructions. Please note the following:

- Submit ONE .java file and ONE plain-text .txt file only. The .java file must contain all the source code that you wrote. (Do not include the `Stats.java` file we provided for you.) The .java file must be named `A1<your last name><your first name>.java` (e.g., `A1CameronHelen.java`). The .txt file must contain a report on the results you created, as specified below. It must be a plain-text file, not a Word (`.docx`) or a Pages (`.pages`) file. The .txt file must be named `A1<your last name><your first name>.txt` (e.g., `A1GuderianRob.txt`).
- Please do not submit anything else.
- We only accept homework submissions via UM Learn. Please DO NOT try to email your homework to the instructor or TA or markers — it will not be accepted.
- We reserve the right to refuse to grade the homework or to deduct marks if you do not follow instructions.
- **Assignments become late immediately after the posted due date and time.** Late assignments will be accepted up to 49 hours after that time, at a penalty of 2% per hour or portion thereof. After 49 hours, an assignment is worth 0 marks and will no longer be accepted.

How to Get Help

Your course instructor is helpful: For our office hours and email addresses, see the course website on UM Learn (on the right side of the front page).

For email, please remember to put "[COMP2140]" in the subject and use a meaningful subject, and to send from your UofM email account.

Course discussion groups on UM Learn: A discussion group for this assignment is available in the COMP 2140 course site on UM Learn (click on "Discussions" under "Communications"). Post questions and comments related to Assignment 1 there, and we will respond. Please do not post solutions, not even snippets of solutions there, or anywhere else. We strongly suggest that you read the assignment discussion group for helpful information.

Computer science help centre: The staff in the help centre can help you (but not give you assignment solutions!). See their website at `http://www.cs.umanitoba.ca/undergraduate/computer-science-help-centre.php` for location and hours. You can also email them at `helpctr@cs.umanitoba.ca`.

Programming Standards

When writing code for this course, follow the programming standards, available on this course's website on UM Learn. Failure to do so will result in the loss of marks.

Question

Remember: You will need to read this assignment many times to understand all the details of the program you need to write.

Goal: The purpose of this assignment is to write a Java program that times the execution of four sorting algorithms (insertion sort, merge sort, quick sort, and a hybrid quick sort (details below)) and reports on the run times and whether the algorithms actually sorted the list (using a testing method you will also write). Then you will write a brief report describing your results.

Code you can use: You are permitted to use and modify the code your instructor gave you in class for the various sorting algorithms. Of course, you must use the `Stats.java` file that we provided to you. You are NOT permitted to use code from any other source, nor should you show or give your code to any other student. Any other code you need for this assignment, you must write for yourself. We encourage you to write ALL the code for this assignment yourself, based on your understanding of the algorithms — you will learn the material much better if you write the code yourself.

What you should implement: Implement the following algorithms and methods (you can add any necessary `private` helper methods):

1. **A (non-recursive) insertion sort algorithm:** Use the following header:

   ```
   private static void insertionSort ( int[] nums, int start, int end )
   ```

   Take the insertion sort algorithm from class and modify it so that it sorts only positions `nums[start]` to (and including) `nums[end-1]` in array `nums`, not touching any other position in the array.

   Also, write a `public` driver method with the following header:

   ```
   public static void insertionSort ( int[] nums )
   ```

   Its task is to simply call the above `private` method, passing the correct values to the `private` method's parameters so that it sorts the entire array.

   Hint: Test the `private` method's ability to sort only part of the array without touching any other part of the array. Many students get this wrong, and then can't figure out why their hybrid quick sort (see below) doesn't work.

2. **A recursive merge sort algorithm:** Take the algorithm exactly as discussed in class:

   - The `public` driver `mergeSort` method: It simply calls the `private` recursive method with the extra parameters (the `start` and `end` indices and the extra array `temp`) that the recursive method needs.
   - The `private` recursive helper `mergeSort` method: It does the recursive merge sort. It receives the array, indices `start` and `end`, and the temporary array as parameters. Its task is to merge sort positions `start` to `end-1` (inclusive) in the array — and it must not touch any other positions in the array.
   - The non-recursive helper `merge` method: It merges two sorted sublists into one sorted list.

   Reminder: Only the `public` driver method should create an array. All the other methods used by merge sort must use the arrays and positions (indices) that they are passed in their parameters without creating any other arrays.

3. **A recursive quick sort algorithm:** Take the algorithm exactly as discussed in class, which uses the following methods:

   - The `public` driver `quickSort` method: It simply calls the `private` recursive method, passing it the extra parameters (the `start` and `end` indices) the recursive method needs.
   - The `private` recursive helper `quickSort` method: It is the recursive quick sort method, which receives the array, and indices `start` and `end` as parameters. Its task is to quick sort positions `start` to `end-1` (inclusive) in the array — and it must not touch any other positions in the array.
   - The `private` non-recursive median-of-three method: It chooses a pivot from the items in positions `start` to `end-1` (inclusive) in the array using the median-of-three method, and swaps the chosen pivot into position `start` in the array.
   - The `private` non-recursive partition method: It partitions the items in positions `start` to `end-1` (inclusive) in the array using the chosen pivot (which it assumes is already in position `start`), and returns the final position of the pivot after the partition is complete.

   None of these methods should create an array.

4. **A hybrid recursive quick sort algorithm that uses a breakpoint:** Use the following header:

   ```
   private static void hybridQuickSort( int[] nums, int start, int end )
   ```

   This method is similar to the recursive `quickSort` algorithm above, except it has a different base case:

   - If `nums[start]` to (and including) `nums[end-1]` is fewer than `BREAKPOINT` items, then call the `private insertionSort` method above to sort `nums[start]` to (and including) `nums[end-1]` (and not touch any other position in the array).

   If `nums[start]` to (and including) `nums[end-1]` consists of at least `BREAKPOINT` items, then do the usual quick sort steps (choose a pivot using the median-of-three technique, partition the items using the chosen pivot, and finally recursively call `hybridQuckSort` twice to sort each of the smalls and the bigs.

   In order for this algorithm to work, you need to define a global constant `BREAKPOINT` in the class containing `hybridQuickSort`. Set this constant to 50.

   (Make sure the recursive calls in `hybridQuickSort` are to `hybridQuickSort`, NOT to `quickSort`.)

   Also, write a `public` driver method with the following header:

   ```
   public static void hybridQuickSort ( int[] nums )
   ```

Its task is to simply call the above `private hybridQuickSort` method, passing the correct values to the `private` method's parameters so that it sorts the entire array.

None of the hybrid quick sort methods should create an array.

5. **A method that verifies that an array is in sorted order:** It must check that each item is no bigger than the next item in the array. It should print a useful error message (including the name of the sorting algorithm that sorted the array) if the array is not sorted.

6. **A method to fill an array with values to sort:** It should be passed an array `nums` and a number n. It should fill the array with the numbers 0 to `nums.length-1` and then perform n random swaps in the array. (To perform a random swap, randomly choose two positions in the array and then swap the contents of those two positions.)

7. **A testing method that allows you to compare (and to verify) these sorting methods using some simple statistics:** Time each of the sorting algorithms 100 times working on an array of 10,000 items (perform 2,500 randomized swaps on the array). Make sure you refill the array every time you sort it, so you are not sorting a sorted array. Also, make sure that you verify that the sorting method works each time!

   Store the timings of each sorting algorithm in an array (which needs to be of type `long`). To see how to do the timing, check out the `main` method of Lab 1 (which timed the Rolen methods you wrote).

   Then use the arrays of timings and the methods in the `Stats` class that we have provided for you to report the mean and standard deviation of the timings for each of the sorting algorithms. The following example shows how to use them:

   ```
   double quickSortMean = Stats.mean( quickSortTimings );
   double quickSortStdDev = Stats.standardDeviation( quickSortTimings );
   ```

   Also, report the results of a Z-test between each pair of sorting algorithms. The purpose of the Z-test is to determine if two samples are the same or different. The following example shows how to use it:

   ```
   double zStat = Stats.zTest( quickSortTimings, insertionSortTimings );
   ```

   The provided `Stats.java` file needs to be in the same directory as your code.

   (Note: We expect that your method will contain repeated code — Java does not make it easy to avoid in this case. So do not worry about it!)

Of course, you need to write a `main` method to call the above testing method, which in turn calls the sorting algorithms, and the methods that fill the array and verify that it is sorted.

Report

If a Z-test result is greater than 2.5, or less than -2.5, the results can be considered to be statistically significant — that is, the two samples can be considered different.

Write a small report in a plain-text file (`.txt`). Include your name, student ID, course number, section number and instructor's name, and the date at the top of your report. Then compare the run times of your sorting algorithms. Answer the following questions:

1. Was the quick sort provably faster than the insertion sort? Why or why not?

2. Was the hybrid quick sort provably faster than the quick sort? Why or why not?