COMP 2140 Assignment 1: Sorting and Recursion

Cuneyt Akcora

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". Click the link and follow the instructions. Please note the following:

- Submit ONE .java file only. The .java file must contain all the source code that you wrote and the small report (in the comments at the end of the file). The .java file must be named `A1<your last name><your first name>.java` (e.g., `A1AkcoraCuneyt.java`).
- Please do not submit anything else.
- We only accept homework submissions via UM Learn. Please DO NOT try to email your homework to the instructors or TAs 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 not be accepted. Please submit early versions to avoid mishaps.

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 add a meaningful phrase after that to the 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. We will also post questions related to the assignment that we receive by email, and answer them there. Please do not post solutions, not even snippets of solutions there, or anywhere else. **We expect you to read the assignment discussion group.**

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 three sorting algorithms (radix sort, merge sort, quick sort) 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, in the comments of your program; see the end of this document.

Code you can use: You are permitted to use and modify the code your instructor gave you in class for the various sorting algorithms, and code from Lab 1. Of course, you must use the code we provided to you below. You are NOT permitted to use code from any other source, You are NOT permitted to 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 recursive merge sort algorithm:** Implement the algorithm exactly as discussed in class. These are the methods you need to write:

   - The `public` driver `mergeSort` method: It simply calls the `private` recursive method with the array and the extra parameters (the `start` and `end` indices and the extra array `temp`) that the recursive method needs. This method must check the validity (null? empty?) of the array to be sorted.
   - 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. This is the method that should have the new base case added to it.
   - The non-recursive helper `merge` method: It merges two sorted sublists (defined by three indices `start`, `mid`, and `end`) into one sorted list, using the extra array `temp`. Make sure that you use the indices to define the sublists in a way that is consistent with how sublists are defined by indices in all other parts of the code: from some index up to, but not including, some other index.

   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.

2. **A recursive merge sort algorithm that does not use a shared temp array:** Implement the algorithm exactly as discussed in class, however do not send a shared temporary array to the private helper method. There are three methods you need to write:

   - The `public` driver `mergeSortInefficient` method: It simply calls the `private` recursive method with the array and the extra parameters (the `start` and `end` indices) that the recursive method needs. This method must check the validity (null? empty?) of the array to be sorted.
   - The `private` recursive helper `mergeSortInefficient` method: It does the recursive merge sort. It receives the array, indices `start` and `end`, **but does not receive a temporary array as a parameter**. Calls will be in the form `mergeSortInefficient(array,start,end)`. 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 (defined by three indices `start`, `mid`, and `end`) into one sorted list and return a sorted array. This function will explicitly return a merged sorted array. Make sure that you use the indices to define the sublists in a way that is consistent with how sublists are defined by indices in all other parts of the code: from some index up to, but not including, some other index.
   - Please note that not using a temporary array will require copying many values over and over again.

3. **A recursive quick sort algorithm:** Implement the algorithm exactly as discussed in class. If there are just two items to be sorted, swap them if necessary. You need to write 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. Consider the case when the array does not have three values to compute the pivot.
   - 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. It should use one simple `for`-loop.

4. **A radix sort algorithm that sorts a positive integer array by starting from the biggest digits of numbers:** In the class example where the biggest value was in hundreds, radix sort used 1s first, then 10s and finally 100s to sort values. In this assignment, radix sort should start from the biggest digit to sort the array. For example, if the biggest number is 3289, it should first consider 3 to assign the number to a bucket.

5. **A method that verifies that an array is in sorted order:** It should be passed an array and it must check that each item is no bigger than the next item in the array. It should return `true` if the array is in sorted order and `false` if it is not.

6. **A method to fill an array with values to sort:** It should be passed a non null int array `array`. It should fill the array with the numbers 0 to `array.length-1`, in order.

7. **A method to randomize an array that is filled with numbers:** It should be passed an array `array` and a number n. It should 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.

8. **A testing method that allows you to compare (and to verify) these sorting methods using a simple statistic:**

   Time each of the sorting algorithms 100 times working on an array of arraySize items (use a randomization parameter numSwaps to make randomized swaps on the array). Make sure you randomize 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 — print a useful error message (including the name of the sorting algorithm that sorted the array) if the array is not sorted.

   Store the timings of each sorting algorithm in another array (which needs to be of type `long`). You can find out time costs by using System.nanoTime() before and after the call, and recording the elapsed time.

   Then use the arrays of timings and the method that we have provided below for you to report the arithmetic mean of the timings for each of the sorting algorithms. The following example shows how to use it:

   ```
   double quickSortMean = arithmeticMean( quickSortTimings );
   ```

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

9. **The method that computes the arithmetic mean of timings:**

```
/**************************************************************
 * arithmeticMean
 *
 * Purpose: Compute the average of long values.
 *          To avoid long overflow, use type double in the computation.
 *
 **************************************************************/
public static double arithmeticMean( long data[] ) {
    double sum = 0;
    for (int i = 0; i < data.length; i++)
        sum += (double)data[i];
    return sum / (double)data.length;
} // end arithmeticMean
```

10. Of course, you need to write a `main` method to call the above testing method (number 8) for a given array size and number of swaps, which in turn calls the sorting algorithms, and the methods that fill the array and verify that it is sorted.

Concerns. Output of your Java file must contain a report of the experimental set-up. What is the array size, what is the number of swaps? Print out all relevant parameters. Please avoid magical numbers in your code. There should be only ONE return per method. See the programming standards (under content/course documents) file on UMLearn, and follow the suggestions. Assignments will be graded by considering these standards. Your code must not give any warnings or errors when compiled and run.

Report

Write a small report in comments at the end of your program.

Answer the following questions (use short sentences, ideally only one sentence):

1. In the main function, increase array size from arraySize= $5K, \ldots, 60K$ by 5K at every step and record sorting times (keep the randomization at 25%, i.e., $numSwaps = 0.25 \times arraySize$) for each algorithm. In your report, crate a table (in text, columns separated by the | character) of times versus array size for each algorithm. The columns of the table will be 1) alg name, 2) arraySize, 3) time (ms).

2. Was merge sort (the efficient version) faster than quick sort? Why or why not? Answer the question for arraySize= 10000, number of swaps=2500.

3. What is the cost of using the inefficient merge sort vs the merge sort? Answer the question for arraySize= 10000, number of swaps=2500.

4. Was radix sort faster than merge sort (the efficient version)? Answer the question for arraySize= 10000, number of swaps=2500.

5. Compared to comparison based sort functions, what are the operations that slow down radix sort?

6. **Optional, not graded**: Try to fit a curve to the sort times you will record in item number 1. Report the curve values for algorithms. See `https://commons.apache.org/proper/commons-math/userguide/fitting.html` for a solution.