Unit 8: Programming
Implementing Sorting Algorithms

Summary: In this assignment, you will work with sorting algorithms by modifying QuickSort, and reimplementing Mergesort.

1 Background

In order to practice sorting algorithms, we will undertake both their improvement (quicksort) and creation (mergesort). Your task is to first modify the version of QuickSort from the textbook to use a few way to pick a pivot. Second, you will re-implement mergesort from scratch to get a better understanding of the concepts behind Divide and Conquer (D&C) algorithms.

Divide and Conquer algorithms are conceptually similar to the recursive programming technique we saw in the last section. The idea, is to break apart a problem into multiple (large) pieces. For example, the half of an array, and then the second half of an array. In terms of recursion, we might save that the sub-problem is size $n/2$. This contrasts with the standard $n-1$ size of many recursive algorithms, where only a single piece of work is accomplished during each call. In general, D&C algorithms require multiple recursive calls while simple recursion, like summing or displaying a list, requires only a single call. D&C algorithms are often more complicated to write than simple recursive algorithms, but, the extra work pays off because D&C algorithms can end up with logarithmic Big-Oh factors instead of linear factors. This is why mergesort is an $O(nlogn)$ algorithm.

This document is separated into four sections: Background, Requirements, Testing, and Submission. You have almost finished reading the Background section already. In Requirements, we will discuss what is expected of you in this homework. In Testing, we give some basic suggestions on how the maze algorithm additions should be tested. Lastly, Submission discusses how your source code and writeup should be submitted on BlackBoard.

2 Requirements [35 points]

In this programming project you will practice the implementation of sorting algorithms. Download the attached base file for a starting place; it includes some very simple testing code. You may modify the base file. However, please retain the signatures of the two stub methods and make sure you keep all of your code in the file. Also attached is Sorting.java, which includes the textbook's sorting algorithm implementations for reference.

- [LC PP 9.4 modified]: Modify the quick sort method to choose the partition element using the middle-of-three technique described in the chapter. [10 points]
- Reimplement the mergesort algorithm to pass only arrays as parameters. This should include a helper method, public static Comparable[] mergesort(Comparable[] a), and a merge method, public static Comparable[] merge(Comparable[] a, Comparable[] b). Do not use global variables. (This approach is slower than the mergesort from the book. The goal is to better understand the mergesort concept.) [15 points]
- Lastly, write appropriate testing documentation - see next section. [10 points]

3 Testing

For this assignment, you will need to describe how you tested your code.Your aim is to demonstrate that you have anticipated things that might go wrong and done appropriate tests to verify that they do not go wrong. Before giving your specific tests, you should define the purpose of your testing methodology. If you used unit testing, or if you test using sample input and expected-output files, please include the contents of the files. You should include both a discussion of how you tested (e.g., which operations), as well as a screen shot of the output of running your program. You may organize this into subsections for different test cases if applicable. The writeup should be clear and well written. Quality over quantity.

When you set about writing your own tests, try to focus on testing the methods in terms of both the integrity of the input array and sorting the data within it. For example, you should test that elements don't disappear when an array is sorted, elements aren't duplicated when the array is sorted, that the resulting array really is sorted, and so on. If you compare the tests that you given, with the parts of your program that they actually use (the "code coverage"), you'll see that these tests only use a fraction of the conditionals that occur in your program. Consider writing tests that use the specific code paths that seem likely to hide a bug. You should also consider testing things that are not readily apparent from the interface specification. For example, write a test that prints your list starting at the head, and another that starts at the end. See if they give the same result.

4 Submission

The submission for this assignment has two parts: a testing write up and a source code submission. Both should be attached to the submission link on BlackBoard.

Writeup: Submit a PDF that discusses how you tested your program. Include a header that contains your name, the class, and the assignment. We not are expecting anything longer than a single page. (If it makes your writeup more clear, you may use additional pages.)

Source Code: Please zip all of your source code files together as "LastNameSorting.zip" (e.g. "AcunaSorting.zip"). It should contain only one file, the base source code file that you have renamed to "LastNameSorting.java" (e.g. "AcunaSorting.java"). Remember to change the class name as well. Be sure that you use the correct compression format - if you do not use the right format, we may be unable to your submission. Do not include your project files.