Divide and Conquer and the Merge Concept

Summary: In this assignment, you will work with the divide and conquer algorithm design technique by creating several merge based algorithms, and reimplementing mergesort.

1 Background

In order to practice divide and conquer algorithms, we will apply the idea to sorting and other tasks. Your goal is to first create a method that uses the merge concept to combine two queue ADTs. Second, you will re-implement mergesort from scratch to get a better understanding of its divide and conquer (D&C) nature. Lastly, you will create an $O(nlogn)$ algorithm for shuffling an input array.

Divide and Conquer algorithms are conceptually similar to the recursive programming technique we saw earlier in the course. The idea is to break apart a problem into multiple (large) pieces. For example, the first half of an array, and then the second half of an array. In terms of recursion, we might say 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 taking a sum 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 code may be tested. Lastly, Submission discusses how your source code should be submitted on Canvas.

2 Requirements [36 points]

In this programming project you will practice the implementation of D&C algorithms. Download the attached base files for a starting place; they include some very simple testing code. You will modify the base file with your answers for all three questions. (This may feel a bit different than previous assignments since different problems will be in the same file.) Available on the course GitHub repository is Mergesort.java, which includes the textbook's mergesort algorithm implementation for reference. **The LinearNode, ListQueue, and Queue classes may not be modified. Do not use any class variables in your program.**

- (modified Sedgewick 2.2.14) Merge sorted queues. Develop a method that takes two queues of sorted items as arguments and returns a queue that results from merging the queues into sorted order. The method must run in linear time. Emptying the input queues is fine. [12 points]

- Reimplement the mergesort algorithm to pass only arrays as parameters. The starting point will be the method public static void sort(Comparable[] a), which will start the recursive mergesort process. Plan to include a recursive helper method, public static Comparable[] mergesort(Comparable[] a), and a merge method, public static Comparable[] merge(Comparable[] a, Comparable[] b). (Note that this approach is slower than the mergesort from the book. The goal is to better understand the mergesort concept.) [16 points]

- Implement a method, public static void shuffle(Object[] a), that shuffles an array in nlog(n) time using a recursive merging mechanism. Create and use helper methods as appropriate. Assume calls to the Random library happen in constant time. It should be possible for any element to re-position to any other position. [4+4points]

Required Files	Optional Files
CompletedMerging.java	(none)

Table 1: Submission ZIP file contents.

– Submit a short explanation of why your algorithm runs in nlogn time. The algorithm must use a recursive merging mechanism.

Sample Output

```
(tail) A E E L M O P R S T X
(tail) 1 4 5 12 12 13 15 16 17 18 20
A E E L M O P R S T X
R P M E L A O E X S T
S M A R X T L E E O P
```

Note that your shuffle algorithm will return a different result than what is shown above since it relies on random numbers.

2.1 Packages

Do not import any packages other than java.util.Random. (Do not use any star imports.)

3 Testing

When you set about writing tests for your mergesort, 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 specification.

4 Submission

The submission for this assignment has two parts: a short written document, and a source code submission.

Writeup: Submit a PDF that explains your shuffle algorithm. Include a header that contains your name, the class, and the assignment. It should be short: about half a page with the header.

Source Code: The source file must be named as "CompletedMerging.java", and then added to a ZIP file (which can be called anything). The class must be in the "edu.ser222.m02_02" package, as already done in the provided base file (do not change it!). You will submit the ZIP on Gradescope.

4.1 Gradescope

This assignment will be graded using the Gradescope platform. Gradescope enables cloud-based assessment of your programming assignments. Our implementation of Gradescope works by downloading your assignment to a virtual machine in the cloud, running a suite of test cases, and then computing a tentative grade for your assignment. A few key points:

- **Grades computed after uploading a submission to Gradescope are NOT FINAL.** We have final say over the grade, and may adjust it upwards or downwards.
- **Additional information on the test cases used for grading is not available, all the information you need (plus some commonsense and attention to detail) is provided in the assignment specification.** Note that each test case will show a small hint in its title about what it is testing that can help you to target what needs to be investigated.

If you have a hard time passing a test case in Gradescope, you should consider if you have made any additional assumptions during development that were not listed in the assignment, and then try to make your submission more general to remove those assumptions.

Protip: the Gradescope tests should be seen as a way to get immediate feedback on your program. This helps you both to make sure you are meeting the assignments requirements, and to check that you are applying your knowledge correctly. Food for thought: if you start on the assignment early, check against our suite often, and use its feedback, there's no reason why you can't both get full credit and know that you'll get full credit even before the deadline.

4.1.1 Standard Programming Deductions

Due to the nature of cloud grading, we use a different policy for the standard deductions:

- **If your submission does not compile, or is missing the file mentioned above, you will receive a zero grade.**
- **Following the file submission standards (e.g., the submission contains project files, lacks a proper header) is optional, and will not be enforced.** (We would appreciate if you at least included the header though!)