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 integrate 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 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.

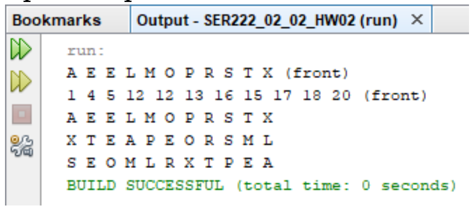
This document is separated into four sections: Background, Requirements, Testing, and Submission. You have almost finished reading the Background section already. In Requirements, we will discuss what is expected of you in this homework. In Testing, we give some basic suggestions on how the code may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

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 may modify the base file with your answers for all three questions. However, please retain the signatures of the two stub methods and make sure you keep all of your code in the base file. Also attached is Sorting.java, which includes the textbook's sorting algorithm implementations for reference. The LinearNode, ListQueue, and Queue classes should not be modified.

- (Sedgewick and Wayne: 2.2.14) Merging sorted queues. Develop a static method that takes two queues of sorted items as arguments and returns a queue that results from merging the queues into sorted order. Emptying the input queues is fine. If you need to make any assumptions about the typing of the contents of the queues, please state them in a comment. [14 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). Do not use global or class variables. (Note that this approach is slower than the mergesort from the book. The goal is to better understand the mergesort concept.) [14 points]
- Implement a method, public static void shuffle(Object[] a), that shuffles an array in nlog(n) time using a recursive merging mechanism. Assume calls to the Random library happen in constant time. It should be possible for any element to re-position to any other position. Include a comment explaining why your algorithm runs in nlogn time. [8 points]

Sample Output



Bookmarks | Output - SER222_02_02_HW02 (run)

```
run:
A E E L M O P R S T X (front)
1 4 5 12 12 13 16 15 17 18 20 (front)
A E E L M O P R S T X
X T E A P E O R S M L
S E O M L R X T P E A
BUILD SUCCESSFUL (total time: 0 seconds)
```

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

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 one part: a source code submission. The file should be attached to the homework submission link on BlackBoard.

Writeup: For this assignment, no write up is required.

Source Code: Please name your main class as "LastNameMerging.java" (e.g. "AcunaMerging.java"). The class must be in the default package.