Computer Science 228
Project 4
Non-recursive Merge Sort

Important Notes

- This project is worth 150 points.
- The due date is 11:59 pm, Friday, April 12, 2013. As usual, we recommend that you start the assignment as soon as possible and get your questions answered early.
- Read through this specification completely before you start.
- Some aspects of this specification are subject to change in response to issues detected by students or the course staff. ***Check Blackboard regularly for updates and clarifications.***
- ***This assignment is to be done on your own.*** If you need help, see one of the instructors or the TAs. Please make sure you understand the "Academic Dishonesty Policy" section of the syllabus.

First posted: April 3

Goals

In this assignment, you will

- become more familiar with stack manipulation and
- develop a deeper understanding of how recursion is implemented.

1 Introduction

Any recursive program can be converted into a non-recursive one by using a stack. In class, we illustrated this by showing how to make quicksort non-recursive. In this assignment, you will explore this idea further, by developing a stack-based non-recursive version of merge sort. You already encountered merge sort in class and in Project 2; we briefly review it here[1].

[1]The description is adapted from Chapter 2 of *Introduction to Algorithms* (3rd ed.), by T. H. Cormen et al., MIT Press and McGraw-Hill, 2009.

Merge sort is based on a method MERGE(A, p, q, r), which takes as arguments an array A and indices p, q, and r into A such that $p \leq q < r$. The method assumes that subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ are in sorted order. It *merges* them to form a single sorted subarray that replaces $A[p\,..\,r]$. MERGE(A, p, q, r) can be implemented to run in $O(n)$ time, where $n = r - p + 1$.

We can use the MERGE method to sort a subarray $A[p\,..\,r]$ as follows. If $p \geq r$, the subarray contains at most one element, so it is already sorted and there is nothing to do. Otherwise, we first divide $A[p\,..\,r]$ into two roughly equal sides $A[p\,..\,q]$ and $A[q+1\,..\,r]$, by choosing $q = (p+r)/2$. Then, we recursively sort each half. Finally, we combine the two sorted halves into a sorted whole using MERGE. The pseudocode is given next.

```
MERGESORT(A, p, r)
    if (p < r)
        q = (p + r)/2
        MERGESORT(A, p, q)
        MERGESORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

To sort the entire array $A = (A[0], A[1], \ldots, A[n-1])$, where $n = A.\textit{length}$, we simply invoke MERGESORT$(A, 0, n-1)$. The running time is $O(n \log n)$.

There is a significant difference between quicksort and merge sort. In quicksort, much of the work is done *before* the recursive calls, by the partitioning method: when the two recursive calls to quicksort are complete, the array is sorted. In contrast, much of the work in merge sort is done *after* the recursive calls, by the merging method: the sort is not complete until the merge is. Thus, the strategy to make merge sort non-recursive must be different the one we used for quicksort.

2 Your Task

Your job is to write the code for the following class.

```
public class NonRecursiveMergeSort
```

`NonRecursiveMergeSort` should contain two generic versions of merge sort for objects of type `T`. In the first version, `T` is arbitrary, but the caller must supply a `Comparator`. The signature of this method is

```
public static <T> void mergeSort(T[] arr, Comparator<?  super T> comp)
```

The second version of merge sort imposes bounds on `T` to guarantee that we can call the `compareTo()` method on objects of type `T`. The signature of this method is

```
public static <T extends Comparable<?  super T>> void mergeSort(T[] arr)
```

Your implementations must be non-recursive and stack-based, and must run in $O(n \log n)$ ***time. No credit will be given for recursive implementations.***

3 Getting Started

This assignment will require considerably fewer lines of code than the previous two. The key is to work out the algorithmic strategy. To get started, it can help to visualize the recursion tree of the original *recursive* algorithm. At any given point in the execution, the recursive algorithm is either

- going *down* the recursion tree, making further recursive calls on smaller subproblems, or
- going *up* the recursion tree, when the recursive calls are complete; the completion of recursive calls should trigger a merge.

The *non-recursive* version would simulate the recursion using a stack to keep track of where in the recursion tree the algorithm currently is. A stack element would have three parts:

(i) an index p to the left endpoint of the current subarray,
(ii) an index r to the right endpoint of the current subarray, and
(iii) a boolean flag signaling whether that subarray is sorted or not; this flag indicates whether we are going out of or into the recursion.

The challenge is to figure out how manage the stack so as to maintain the right state of execution at all times. We will discuss this further in class.

You are free to use whichever stack implementation you find most convenient. We recommend that you use either the `PureStack` interface posted on Blackboard or Java's `Deque` interface.

4 Submission

Turn in a zip file named `Firstname_Lastname_HW4.zip` containing all your source code for the `NonRecursiveMergeSort` class. The package name for the project must be `edu.iastate.cs228.hw4`. Include the Javadoc tag `@author` in each class source file. ***Do not turn in*** `.class` ***files.***

You are strongly encouraged to write unit tests as you develop your solution. However, you do not have to turn in any test code, so you may share your tests on Blackboard.

All submissions must be done through Blackboard Learn. Please follow the guidelines posted under "Documents & Links".

5 Grading

A significant portion of your grade will be based on running our unit tests on your classes. We will also inspect your code to verify that your methods are implemented as specified. Documentation and style will count for roughly 15% of the total points.

Late Penalties

Assignments may be submitted up to 24 hours after the the deadline with a 25% penalty (not counting weekends or holidays). No credit will be given for assignments submitted after that time.