UNIVERSITY OF WASHINGTON, TACOMA
TCSS 343: Design and Analysis of Algorithms
Homework #4, Programming Project, individual: ***k*-way Mergesort**
Group Problem: **Recurrence equations**
Due: Wednesday, October 29, 2014, 9:30am **(Section A)** or 1:30pm **(Section B)**

Autumn 2014 October 15, 2014

Written homework is due at the *beginning* of class on the day specified. Any homework turned in after the deadline will be considered late. **This is an individual assignment. The Gilligan's Island Rule applies. Late homework policy: You may turn this programming project late (one lecture) at a penalty of 20%.**

## Learning Objectives

- Empirical analysis of algorithmic alternatives
- Greater familiarity with recursion
- Use of heaps
- More practice with reading other people's code
- More practice with debugging and constructing good test cases

## The Idea

Mergesort is a good algorithm. It runs in O($n$ log $n$) time in the worst case. But let's say we want to use it in a real application, which means we want to make it run as fast as possible. One idea for how to improve the performance of mergesort is to break the recursion into $k$ parts instead of just 2. The recursion tree would be shallower, and so *maybe* the algorithm would run faster.

Imagine $k$ to be a medium-sized number (imagine it to be 20, but it can be anything). If we just take the regular mergesort algorithm and modify it so that it breaks up the array into 20 equal parts and then merges the 20 sorted subarrays, we discover that it takes 19 comparisons to determine the minimum when we do the merge. Then we have to do 19 more comparisons to find the next minimum. It would take O($k$ $n$) time to do a merge and so the algorithm would run in O($k$ $n$ log $n$) total time. Very slow.

An improvement is to use a Priority Queue ADT (implemented as a binary heap) to make that merge process require less work. We put each of the 20 numbers in a binary heap, and when we need a number to output, we do a deleteMin operation, put that number in the output array, and then insert the next number from the subarray (if there is one) into the heap. So, to do the merge, it would take O($n$ log $k$) time. (Why?) If you work through the analysis, you will discover that $k$-way mergesort takes O($n$ log $n$) time. But that still leaves the question: which value of $k$ leads to the fastest implementation of $k$-way mergesort?

To find out, you will code $k$-way mergesort and run timing experiments to decide what value of $k$ is best. You will need to implement the merge method using a binary heap. After you have implemented a correct version of $k$-way mergesort, you will then run experiments on different values of $k$ and different input sizes.

The code for mergesort and doing the timings are provided for you. You need only write the merge method and the binary heap code to implement it. For the experiments, choose values of $k$ = 2, 3, 5, 10, 20, 50 and input sizes $n$ = 200000, 400000, 800000, 1600000, and 3200000. Run it three times (warm start) for each size and average the results.

For each value of $k$, plot your results on a graph ($x$-axis is the input size; $y$-axis is the time in milliseconds it takes to sort), connecting the dots for each value of $k$. **Make sure that the $x$-axis and $y$-axis values are drawn to scale.** Also, create a single graph with all of your results.

**Starter Code and Some Test Files**

Here is the java code to start with:
Sort.java BinaryHeap.java MergesortHeapNode.java EmptyHeapException.java

**Software Engineering**

To make sure your implementation is correct, test your code on small values of $k$ and input size $n$ and see how the heap changes and verify that the output to the merge method is correct. Does your code work correctly when one or more of your subarrays has no more elements?

**What to Turn In**

- Electronically turn in your modified Sort.java file. You should *not* have to modify the mergesort method. You should only modify the merge method and any code to help you do the timings or test for correctness. (The modifications to merge are where there is the comment "`do a k-way merge on the k sorted subarrays.`")
- Also, turn in a hard copy of your code.
- Turn in (hard copy) a set of graphs that indicate how much time is spent for different values of $k$ and different sizes of input. How do you explain the experimental results? What can you conclude about $k$-way mergesort? How does it compare to "regular" mergesort?
- Finally, turn in a report (at most one or two pages, hard copy) that described how you approached the problem, what troubles you had, and what you learned.

**Bonus Exercise**

This assignment will be graded on a scale of 30 points. Here is a more challenging exercise.

1. (5 points) Suppose instead of using a binary heap to implement the $k$-way merge, we use a $d$-heap, where $d = k$. (A $d$-heap is a complete $d$-ary tree and can be implemented using an array similar to the way binary heaps are implemented.) What is the asymptotic running time of $k$-way mergesort using this new merge method? Show your analysis.

**Group Problem**

1. Solve the following recurrence equations. Do **not** use the Master Theorem (although you should use the Master Theorem to verify that your answer might be correct). Use the repeated substitution method. Show your work.
   a. $T(n) = 4\ T(n/2) + n^2$ , $T(1) = 1$
   b. $T(n) = 5\ T(n/3) + n$ , $T(1) = 1$
   c. $T(n) = 8\ T(n/4) + n^3$ , $T(1) = 1$

* * *

**Suggested problems** (highly recommended, but not to be turned in):

1. 4.4.5.
2. Read and understand the proof of the Master Theorem (pages 490-491).
3. 5.1.3, 5.1., 5.1.9.
4. 5.2.8, 5.2.9.