TCSS 343: Design and Analysis of Algorithms
Spring 2018, Homework #2
Programming project: empirical analysis
Due: Thursday, April 12

For this assignment you are asked to (1) electronically turn in modified source code on the course Canvas webpage, and (2) submit a report (hard copy) in class. Details are given below. Make sure you have acknowledged all persons with whom you worked. Even though you are encouraged to work together on problems, the work you turn in is expected to be your own. When in doubt, invoke the Gilligan's Island rule (see the syllabus) or ask the instructor.

All assignments (written or programming) are due at the beginning of the lecture on the due date. You may turn 1 assignment in up to 1 lecture late. No assignments will be accepted *after* the beginning of the lecture *after* the due date. The reason for this is that in the lecture after the due date, I might already return graded homeworks to students who submitted on time.

Learning objectives

Most of the algorithm analysis we do in TCSS343 is analytical. This homework, on the other hand, requires you to perform an *empirical analysis* of algorithmic alternatives. Furthermore this homework is also intended to give you greater familiarity with *recursion* (*divide-and-conquer*), the *use of heaps*, and finally, more practice with *debugging and constructing good test cases.*

The idea

If you are not fully familiar with mergesort yet, read section 5.1 in the textbook first!

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. This version of mergesort is called k-way mergesort.

Imagine k to be a medium-sized number (imagine it to be 20). 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 $O(kn)$ time to do a merge and so the algorithm would run in $O(kn \log_k n)$ 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. 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 binary heap code, and the operations that you need to use it as a priority queue, such as insert and deleteMin, are provided for you. You do not need to make modifications to the BinaryHeap class, just use it as it is. The overall code for mergesort and doing the timings are also provided for you. You only need to adapt the *merge method* in Sort.java so that it does a k-way merge using a binary heap.

For the experiments, choose values of $k = 2, 3, 5, 10, 20, 50$ and input sizes $n =$ 200000, 400000, 800000, 1600000, and 3200000. Run the algorithm three times for each size and average the results. In other words, you will need to run the algorithm 90 times in total. 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. Create a single figure (graph) with all of your results, rather than making individual figures for every value of k.

Starter code

The java code to start with is available on the course Canvas webpage: Sort.java, BinaryHeap.java, MergesortHeapNode.java, EmptyHeapException.java.

Located within the merge method (line 76):
// divide the array into k subarrays and do a k-way merge
This comment means the following. The subrange from `low` to `high` in the `data` array contains k subarrays. Each of these k subarrays is sorted. The else block is supposed to extract these k subarrays from `data` and do a k-way merge of them, sorting them in the process.

What to turn in

- Electronically turn in your modified Sort.java file on the course Canvas webpage. 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.
- Submit a report (hard copy) in class. This report should contain:
  - A hard copy of your code.
  - A graph that indicates how much time is spent for different values of k and different sizes of input (see above).
  - An analysis of the results: how do you explain the experimental results? What can you conclude about k-way mergesort?
  - A description of (at most) one page on how you approached the problem, what troubles you had, and what you learned.

Grading

This assignment will be graded on a scale of 30 points[1] (15 points for the code, 10 points for the graphs and the result analysis, 5 points for the description of how you approached the problem).

The minimal requirement is that your code is correct, i.e. that your implementation correctly sorts inputs for different values of n and k. If your solution does not satisfy this basic requirement, it will become very hard to give you credit for the other parts such as the graph and the result analysis too.

Another requirement is that your code implements k-way mergesort and not another sorting algorithm like heapsort. Note that you are expected to use a heap of size k to facilitate a merge of k sorted subarrays; this is different from putting all elements of these k sorted subarrays on a heap simultaneously and extracting them one by one. In particular, the heap in your implementation should not contain more than k elements at once.

Test your code on small values of k and input size n. See how the heap changes. 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?

Bonus exercise

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

[1]Your homeworks in TCSS343 are graded on the following scales: HW1 (20 points), HW2 (30 points), HW3 (25 points), HW4 (25 points), HW5 (25 points). This makes a total of 125 points, which will be converted to 25% of your final grade.