CAS CS 112 A1– Spring 2012, Programming Assignment 3

Problems due at 10:00 pm on Thursday, March 1

Problem 1: Comparing Sorting Algorithms (30 points)

Compare the performance of insertion sort, merge sort and quicksort by doing the following.

1. Write a program capable of repeatedly generating a sequence of N random integers, sorting them in decreasing order using each of the three algorithms, and printing the average running time for each of the algorithms. You may use code provided in the textbook or in class, but if you do, please cite your references via comments in your code.

2. For each power of 10, e.g. $N = 10, 100, 1000, \ldots$, run your program until sorting takes longer than 2 minutes elapsed time for one of the algorithms. Tabulate the results (and plot them, if you have familiarity with a graphing program).

3. Add a routine to your program in (1) to generate a strictly increasing sequence of integers (instead of a random sequence). Then repeat (2) for a strictly increasing sequence of N integers.

Submit your code in a file called threesorts.java, and your tables in another appropriately named file such as threesorts.txt or threesorts.pdf.

Problem 2: Linked-list Mergesort (30 points)

Mergesort is not just for arrays. In fact, to sort a linked list, mergesort turns out to be a natural candidate. Implement a Merge class that implements a sorting routine to sort a linked list of integers in ascending order: `void sort (List l)` . Follow the design pattern for the Merge class on arrays in our textbook starting on p. 273. Your routine can (and should) destructively modify the list that is passed in as the parameter `l`, i.e., that list's pointers can be rearranged. A List should be defined as a class storing a first Node and an integer length. A Node should be defined as a class storing a next Node and an integer item. (You may re-use the linked list implementation of Bag on p. 155; feel free to add methods like constructors and print methods, but don't worry about generics). Note that your code should handle the case when the original list contains duplicates. You may implement either top-down recursive mergesort, or bottom-up iterative mergesort. Submit your Node, List, and Merge classes, as well as a client that tests your Merge class. We will write our own client to test your class.

Problem 3: Improvements to Quicksort (30 points)

1. (6 pts) Assume that the pivot element in quicksort is always chosen to be the middle element in the array. What is the worst case (asymptotic) number of comparisons performed by quicksort with this pivot choice? Give an example with 15 elements that results in a worst case number of comparisons.

2. (6 pts) A deterministic improvement to quicksort (mentioned on p. 296 of our text) is as follows: to choose the pivot we pick three possible candidates. The first one, the last one and the middle one in the array, then we set the pivot to the median of these three elements and then partition. What is the worst case (asymptotic) number of comparisons performed by quicksort with this pivot choice? Give an example with 15 elements that results in a worst case number of comparisons.

3. (18 pts) Because of the overhead of recursive calls, insertion sort is faster than quicksort for sufficiently small array sizes. Thus, to speed up quicksort, it makes sense to stop recursing when the array gets small enough and to use insertion sort instead. In such an implementation, the base case of quicksort is some value **base** > 1. Experiment with various settings of **base** to see, roughly, what the optimal setting is. In your experiments, use a large array filled with random integers (likely on the order of 1,000,000 elements, but you will have to see what value of N produces meaningful information not obscured by noise and system clock measurement resolution). In the comments of your code, provide a table that shows the array size you used, present the running times it took with different values of **base**, describe your experiments, and the ultimate value of **base** that you determined to be optimal.

Submit a single file quicksort.java that includes the quicksort with insertion sort base-case code, as well as the code you used to time and find your optimal value of **base**. Provide the answers to the first two short parts of this question as comments within your quicksort.java file.