CAS CS 112 – Spring 2008, Assignment 2

Problems 1-3 due at 10:00 pm on Thursday, February 14
Problem 4 due in class on Thursday, February 14

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 which takes **N** and **t** as input, generates **t** sequences of **N** random integers, sorts each one in **increasing** order using each of the three algorithms, and prints the average running time for each algorithm. You may use code provided in the textbook or in class, but if you do, please cite your references via comments in your code. We will give guidance on how to implement random number generation and routines to time your code in lab during the week of February 4th.

2. For each power of 10, e.g. $N = 10, 100, 1000, \ldots$, run your program until sorting takes longer than 1 minute for a single run of one of the algorithms. Use reasonable values of t (large enough for results to be significant, but small enough so that you are not waiting for hours for the program to finish, especially as N gets large). Tabulate the results (and plot them, if you have familiarity with a graphing program).

3. Repeat part 2 for a strictly increasing sequence of random integers of size N. Do you see a difference in performance between the three algorithms? How do you explain it? Please include your answer in the written part of the homework. To generate a strictly increasing sequence of random integers, you can generate your array of random integers as before, and then use the sort method of the Arrays class.

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: Iterative Mergesort

(30 points) Mergesort does not need to be written recursively. An alternative approach performs merges of larger and larger size iteratively using nested loops. The first pass of iterative mergesort sets a **mergesize** of 2, and sorts $\lceil N/2 \rceil$ neighboring pairs of values. The second pass sets the **mergesize** to 4, and merges previously sorted pairs. The **mergesize** continues to double until the final pass does one top-level merge of size N.

Here is how iterative mergesort would sort an array of 8 characters in three passes (spaces added for readability):

```
o r g a n i z e
or  ag  in  ez
agor    einz
aeigornz
```

Write an iterative mergesort that sorts an array of integers. Make sure your code works correctly when N is not an exact power of two. Submit your code in a file called mergesort.java.

Problem 3: Quicksort

1. (5 points) Assume that the pivot element in quicksort is always chosen to be the middle element in the array. What is the worst case asymptotic running time of quicksort with this pivot choice? Give an example with 15 elements that results in worst case behavior.

2. (5 points) A deterministic improvement to quicksort (which the book's implementation uses) 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 running time of quicksort with this pivot choice? Give an example with 15 elements that results in worst case behavior.

3. (10 points) 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). It would be easiest to read in **base**, N and the number of iterations of the sort as input of your program. 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.

Problem 4: Stable Sorting

(20 points) A stable sort is one which preserves the relative order of equal elements. For example, if A equals B and A appears before B in the original list, then A will appear before B in the sorted list. This is relevant if, for example, we are sorting a list of **Employee** records, and each employee has a **name** and an **id** field. If we have a stable sort, we can sort employees by name and id by first sorting by name, and then sorting by id.

Of the four sorts discussed in class (bubble, insertion, mergesort, and quicksort), which ones are stable? Please give an explanation for each case. For mergesort and quicksort, consider the implementation given in the book.