

CAS CS 112 – Spring 2008, Assignment 2

Problems due at 10:00 pm on Thursday, February 14

Problem 1: Comparing Sorting Algorithms

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. 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. $N = 10, 100, 1000, \dots$, run your program until sorting takes longer than 10 minutes 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 3sorts.java, and your tables in another appropriately named file such as 3sorts.txt or 3sorts.pdf.

Problem 2: Iterative Mergesort

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 merge-sort.java.

Problem 3: Quicksort

1. 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. 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. 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.