

CS113 - Program Design and Algorithms

Lab Assignment #5

Summer 2002

The purpose of this exercise is to observe the performance of the sorting algorithms we studied in class, using a technique called *program instrumentation*. We will also make modifications to several of the sorting algorithms, and observe any performance improvements we obtain as a result.

Instrumenting a program involves the addition of code (counters, summations, etc.) necessary to observe the operation of the algorithm. In the case of sorts, we have been characterizing the algorithms by the number of *key comparisons* and *key moves* each requires. Counting the number of these operations should give us some insight into the efficiency of the various algorithms.

For this exercise, you are to do two main tasks:

1. For the sorting algorithms we discussed in class, verify that their run-time behavior is consistent with our class discussions - i.e., that the order of each algorithm, in terms of key comparisons and key moves, matches the analysis we did on each algorithm. Run each algorithm on several datasets - include both small and large datasets, and also on totally random as well as partially (or completely) sorted ones. Draw one or more graphs to display your results.
2. Make modifications to the algorithms listed below, and compare the performance of the new algorithm to the original. Use the same datasets you used for part 1.
 - (a) For the *bubble sort*, modify the algorithm so that for *every other* pass, it starts at the *end* of the array and works its way toward the beginning. Recall that this should cause the smaller elements to move more quickly into position. Use the "terminate if sorted" version of the bubble sort as a starting point.
 - (b) Note that the purpose of the inner loop of the *insertion sort* is to find the place within the already sorted portion of the list where the next element should be placed. Since this involves searching an already sorted list, the linear search that the original algorithm uses can be replaced with a binary search. Make this change, and observe the results.
 - (c) The *quicksort* algorithm spends most of its time sorting very small pieces of the entire array. The running time has been found to improve by switching to a different (usually selection) sort after the list has been split into small pieces. The exact point at which to switch is system and compiler dependent, but is usually around 10 elements. Make this modification, and compare the results to the original quicksort algorithm.

You should hand in copies of your modified codes, as well as a short report (one or two pages, plus graphs) that summarizes your findings.