Evaluation of Sorting Algorithms

Nicholas Mirchandani
Computer Science
Chapman University
Orange, USA
nmirchandani@chapman.edu

***Abstract*—This document details experimental data found from comparing implementations of various sorting algorithms. The algorithms discussed herein are as follows: bubble sort, selection sort, insertion sort, and quick sort.**

***Index Terms*—sorting, algorithms, analysis**

I. Implementation

Before the run-time efficiencies of the various algorithms could be empirically compared, they had to be implemented. Bubble sort, selection sort, and insertion sort were implemented in class utilizing professor Rene German's guidance while quick sort was implemented separately. User input was implemented as a text file; the first line of the text file denoted the number of elements in the file, of which there were one per line. The input was properly parsed and stored into various identical arrays for testing utilizing the various sorting algorithms.

II. Methods

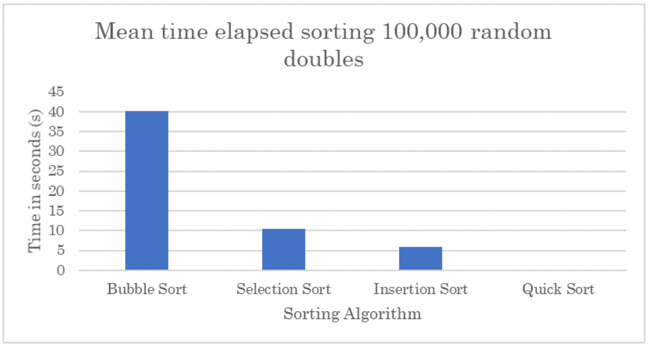
A. Generating the Random Input Data

To generate the random input data with which to test the algorithms, a custom snippet of C++ code was utilized to generate random double values by concatenating 2 random integer values with a decimal point in between; while this may not be the greatest random number generation for doubles, it ensured numbers that were random distances apart due to the overwhelming weight of the first random integer generated while still ensuring that the numbers were in fact doubles with many digits of precision beyond the decimal place. Utilizing this pseudo-random double generator, a text file was created containing 100,000 doubles. For each trial, the 100,000 doubles were regenerated seeded based on the epoch time.

B. Measuring Time Elapsed

Time elapsed was measured utilizing the high resolution clock residing in the <chrono> header of the standard C++ implementation. Time just before sorting began and time just after sorting finished were identified and subtracted to result in the total time elapsed for each algorithm. Each algorithm sorted a duplicate copy of the same array, so there was no variable factors influencing the results except for the sorting implementations themselves.

III. Results



As shown above, bubble sort was by far the least efficient sorting algorithm, taking a whopping 40 seconds on average. By contrast, quick sort was the most efficient sorting algorithm, taking only a measly 0.015 seconds on average. The data was collected from 10 identical trials, and the coefficient of variation for all sorting algorithms was less than 3%, dictating that the trials were incredibly uniform and was not very much influenced at all by outlying factors.

IV. Discussion

Overall, the difference in time elapsed between the various sorting algorithms is astounding. If you ever had to pick between these four sorting algorithms, and performance was your only concern, quick sort would most certainly be the way to go every time. However, since quick sort requires recursion, when using a language that does not easily support recursion, it may not be the best solution.

V. Limitations

The limitations of this empirical analysis that are quite obvious, but should be pointed out regardless. This analysis only took 4 sorting algorithms and compared them, as opposed to taking all sorting algorithms and comparing them. Additionally, this analysis only compared performance, but did not compare setup/run-time costs of these algorithms, so the best algorithm in a practical environment may vary.

Acknowledgment

Nicholas Mirchandani would like to thank Rene German for his assistance in producing this assignment and for being an amazing professor throughout the entire course of CPSC 350: Data Structures and Algorithms as well as the preparatory course CPSC 298: C++ Programming.