Assignment 6: Sorting Algorithms

Josh Taylor

December 11 2019

1 Introduction

For this assignment, I implemented 5 sorting algorithms in C++: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and Quick Sort. I tested each sorting algorithm with 3 different array sizes: 10,000, 100,000, and 1,000,000 double values.

2 Bubble Sort

Bubble sort deemed to be the slowest sorting algorithm out of the ones that I tested. At 10,000 values, bubble sort took approximately 0.25 seconds. At 100,000 values, this sort took 31 seconds. And at 1 million values, this sort took over an hour.

While bubble sort showed the slowest results for sorting doubles, it was the easiest and quickest to build. Implementing a bubble sort would be best utilized when you need quick results from a small population size.

3 Selection Sort

Selection sort was slightly faster than bubble sort, but also slightly more difficult to implement in C++. The results for selection sort were 0.15 seconds, 12.3 seconds, and 21.2 minutes for 10,000, 100,000, and 1 million values respectively. Selection sort improved by slowly decreasing the size/length of the inner loop every time we iterated through the outer loop, but run-time still hovered close to O(n squared).

4 Insertion Sort

Insertion sort's results were 0.10 seconds, 7.5 seconds, and 11.9 minutes for 10,000, 100,000, and 1 million double values respectively. In terms of implementation, it was similar in difficulty to selection sort.

Run-time landed between O(n) and O(n squared) because the randomly generated values caused the array to already be partially sorted.

5 Merge and Quick Sort

These two algorithms were the most difficult to build, for they were implemented via recursion. Because of their divide-and-conquer design, their runtimes decreased *significantly* to O(nlogn). For sizes 10,000, 100,000, and 1 million, merge sort resulted in 0.002 seconds, 0.020 seconds, and 0.236 seconds respectively. Similarly, quick sort resulted in 0.002 seconds, 0.018 seconds, and 0.183 seconds respectively.

While I knew ahead of time that these algorithms would perform quicker sorts, I was still quite surprised by their results, and its direct comparison to algorithms like bubble and selection sort showed me very clearly the power and efficiency of these divide-and-conquer algorithms.

6 System Usage

Looking at task manager when running the experiment, each sorting algorithm caused a spike in CPU. However, it was difficult to view changes in CPU and RAM for the divide-and-conquer algorithms (merge sort and quick sort) because they were completed quite quickly.

7 C++ and Shortcomings

Using C++ allowed me to dynamically allocate memory. Since the sorting file included the size in the first line, my program was able to allocate the exact/correct amount of memory prior to running the sorting algorithms. Other programming languages like java might have difficulties in specific memory allocation and would cause more overhead in run-time. C++ allowed me to avoid this overhead. This experiment has some shortcomings due to the fact that it was only performed on my personal machine. Also, my computer's condition was not consistent throughout my trials. Additional open applications and computer temperature are just two examples of factors that were not accounted for that may have affected the results of my experiment.