Assignment 6: Sorting Algorithm Analysis

Ariel Gutierrez

December 21, 2020

For this assignment, we had to implement 5 sorting algorithms in a C++ program, namely quick sort, merge sort, selection sort, insertion sort, and bubble sort. This report will go over the resulting durations of these algorithms on a list of 50,000 decimal values from "test.txt" and how the sorting algorithms compare with each other.

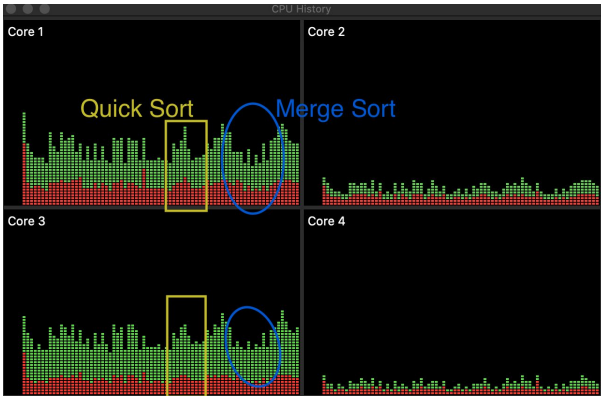

Figure 1: CPU History when using quick sort (yellow squares) and merge sort (blue circles) when using a 100,000 value data set.

I did not expect quick sort to be more than twice as fast as merge sort when looking at the duration of the algorithm. I also found the difference of runtime is between bubble sort and the other O(n^2) algorithms (selection sort and insertion sort) to be so drastic because they all have the same big-O runtime. I thought they would have roughly the same duration but bubble sort was very slow compared to the others when using a 50,000 value data set.

Quick sort was the fastest at sorting out the 50,000 decimal values from the text file "test.txt", and took 0.011394 seconds to run. The tradeoff with using quick sort is that it uses more CPU(Central Processing Unit). When running quicksort with the activity monitor open on my laptop, there was a spike in CPU usage as shown in Figure 1. The average and best case runtime for quick sort is O($n \log n$), which explains why it is so fast since it is a divide and conquer algorithm. However, if the pivot selected is the smallest or largest element of the data set, quick sort can reach a runtime of O(n^2), but there is a very low possibility of this occurring. Therefore, the best case to choose quick sort is when you are working with a large data set and you have a CPU that can handle it.

Merge sort was the second fastest to sort out the same data at 0.028799 seconds. It lagged behind quick sort by a total of 0.017405 seconds. The fast duration is expected since the big-O runtime of merge sort is O(*n log n*). The reason that the runtime is O(*n log n*) is because merge sort is a divide and conquer algorithm, like quick sort. Merge sort uses an extra temporary array to partition the data so it needs to use extra memory. When looking at the Activity Monitor, there wasn't a big spike in CPU usage as there was with quick sort, as shown in Figure 1. It would be best to use merge sort when you are working with a large data set and when you have access to extra memory.

Selection sort was the third fastest algorithm with a duration of 3.876394 seconds. Here, we start to see that the duration of the algorithms increases drastically from being under one second to taking over three seconds to sort data. The reason behind this is that selection sort has a big-O runtime of O(n^2). When tested with small data sets, such as sets with 100 items, selection sort would take less than a second to run. Selection sort makes a large number of comparisons, which is why it is slow with large data sets, so it is best to use it with a small number of items.

Insertion sort was slower than selection sort since it had a duration of 5.381302 seconds. Insert insert works well with partially sorted data; when using nearly sorted data, insertion sort's runtime can reach a runtime of O(n), however, for random data it has a runtime of O(n^2). In our empirical analysis, the data said that we used was not a partially sorted list, therefore, insertion sort did not work as well and took longer time to sort. If you are working with a data set that has partially or nearly sorted data and the data set is not very large then you should use insertion sort.

Bubble sort was the slowest algorithm to sort data since it took 11.230715 seconds to sort 50,000 data values. The bubble sort algorithm is relatively easy to code because all you do is compare two numbers and swap them if the right element is larger than the left element. Though it is easy to code it is very slow, with runtime O(n^2). Therefore, bubble sort should be used when working with a relatively small data set.

The choice of the programming language affects the results because it allows us to use recursive sorting algorithms. C++ allows recursion however there exists programming languages that do not allow recursion, so we would not be able to get full use over all the possible sorting algorithms. Furthermore, C++ is a compiled language, so it is faster to run whereas an interpreted language such as Python would take longer to run the sorting algorithms because it takes time to interpret the code to actually run it.

Some of the shortcomings of this empirical analysis is that it was dependent on many variables, such as hardware and compilers. This is shown in how quick sort may not be as effective if a computer with a not as good CPU would run it. Moreover, algorithms such as insertion sort may underperform because the type of input used matters; for a fast runtime, a partially sorted dataset must be used. Therefore, empirical analysis needs to have multiple data sets, platforms/hardware, and compilers/linkers to test to get the full picture of the efficiency of these algorithms.