Programming Assignment

Michael Frank
CSC 310
April 13, 2022

For this assignment, I was tasked with comparing the runtime efficiency of three different sorting algorithms on different size data sets, in varying states of pre-assortedness. I chose Python as the language of my implementation. The three algorithms used were *quick sort, merge sort,* and *heap sort*, and they each had to sort numbers across 270 different files. I had my input files organized into 3 large directories: *small* (10,000 elements), *medium* (100,000 elements), and *large* (1,000,000 elements). In each of those directories, there were three sub-directories: *unsorted* (randomly generated elements), *small to large* (randomly generated elements, but pre-sorted with Python's `sort()` function), and *large to small* (same as previous, but reversed). In each of these were 30 files organized according to the aforementioned descriptions. As each algorithm ran, it recorded the output in a separate output directory organized in a similar fashion, except it had another layer of sub-directories separating results by the algorithm that computed them. In tandem, each algorithm also recorded the resulting runtime of its calculation in a spreadsheet, which gave me the data I needed. This report details my findings from this experiment.

Theoretical Runtime

Algorithm	Average-Case Efficiency
Quick Sort	O(*n* log *n*)
Merge Sort	O(*n* log *n*)
Heap Sort	O(*n* log *n*)

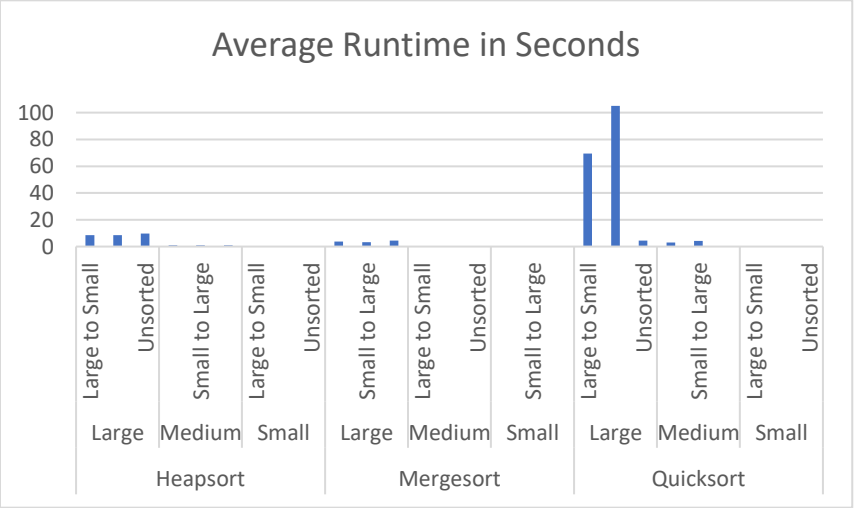
Figure 1: estimated basic operation and runtime for each algorithm

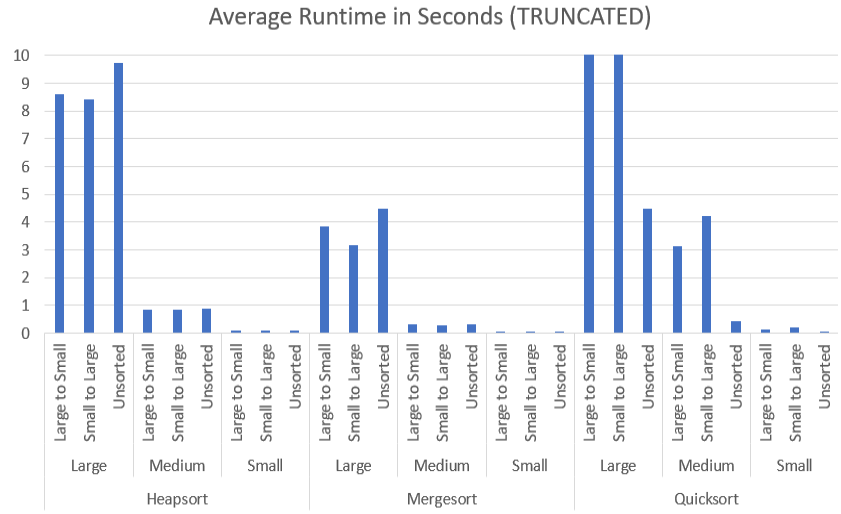
For each algorithm, the average-case efficiency is the same[fig. 1], but the actual runtime was quite different. On average, merge sort was the fastest, and heap sort was the slowest. The worst performance award would go to quick sort for the pre-sorted data, with an astonishing average time of about 80 seconds. I'm skeptical of these results, because the median-three pivot selection is supposed to mitigate the effect of this. I did implement it as described in the handout[fig. 2], but I'm not sure if I did it right, considering the results it gave me. But when switching the pivot so that its initial value is always the first element, it was about 10 seconds slower, so I must have implemented it at least partially correctly.

```
if len(A) < 101:
        p = A[lo]
    else:
        p = median_three(A[lo], A[(lo + hi) // 2], A[hi])
```

Figure 2

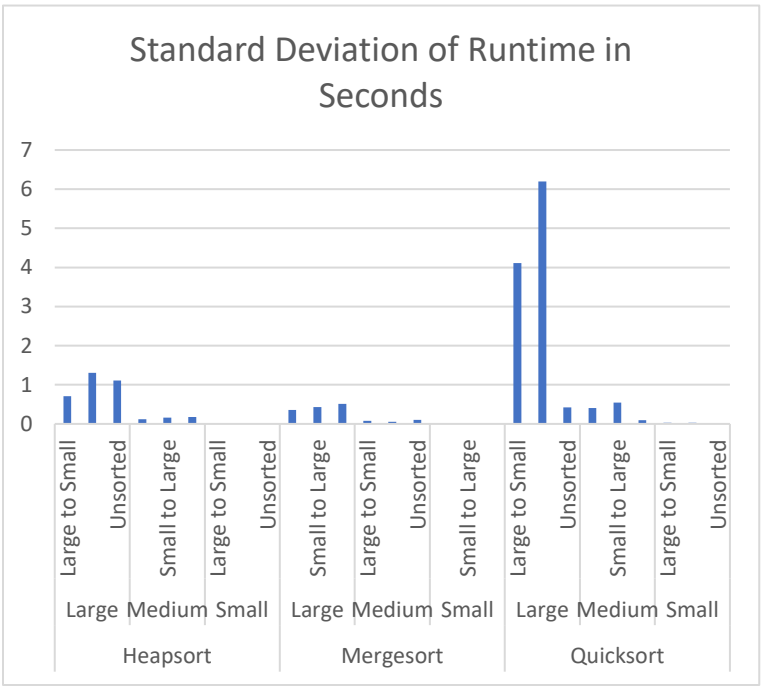
The order of growth is the same for all three in the average case, but quicksort has one of $O(n^2)$ in the worst case, which is what led to these massively skewed results.[fig. 3] I knew it would be slow, but I didn't know it would be that slow. I actually expected heap sort to outperform merge sort, but it looks like merge sort is the fastest for each of these instances. That is likely due to my implementation, as I think my heap sort implementation had more overhead than was necessary.





I also think some of the discrepancies come from the language I chose. Since python is an interpreted language, I'm not in control of what's actually happening under the hood. Python code that looks like a single operation is usually actually several C instructions under the hood. I believe that, had I chose to write this project in C, the results would have been more reliable. Python abstracts away most of what it's actually doing. This made the project much easier to code (I shudder when I hear "C" and "string" in the same sentence), but unfortunately led to a lack of control over exactly how many instructions are being run by the computer.

This is a more readable graph. Note that it crops out quicksort's worst-case results so that the rest is actually visible. The results are actually the same as the previous graph. I find it interesting how small to large seems to have a faster order of growth for quicksort than large to small does.



This graph shows the standard deviation. I think the instability of my implementation of quicksort is what led it to have such a high deviation compared to the rest, as well as the nature of the algorithm. It doesn't like when things are already sorted, and the results accurately reflect that, though I still wonder if they reflect it too much.