COMP2521 Sort Detective Lab Report

by Anthony and Zac

In this lab, the aim is to measure the performance of two sorting programs, without access to the code, and determine which sort algorithm each program implements.

Experimental Design

There are two aspects to our analysis:

- determine that the sort programs are actually correct
- measure their performance over a range of inputs

Correctness Analysis

To determine correctness, we tested each program on the following kinds of input

- small and large amounts of numbers
- random order
- already sorted
- reversed order

We will test our two given programs with the sort command already implemented in linux. Using the *diff* command, we expect it to have no output, meaning that there are no differences in the output of the sort, to show that our programs are sorting the input correctly and not removing values, etc

Also some programs will either randomly sort duplicate values (unstable) or sort them in the order they were inputted (stable). We will test this by creating data in a text file and sending it into the program using the “<” command. For example, when we input values such as *1 abc* then *1 fhg* we expect it to output it in the same order if it is stable (after multiple tests) and randomly every time if unstable.

Performance Analysis

In our performance analysis, we measured how each program's execution time varied as the size and initial sortedness of the input varied. We used the following kinds of input;

- small and large amounts of numbers
- random order

- already sorted
- reversed order

We used these test cases because

- small and large amounts of numbers: Some sorts will have a more linear time scale as the size increases, whereas others will grow experientially.
- random order: how efficient it is in swapping the numbers, different algorithms will have a different number of swaps.
- already sorted: Used as a baseline to see how quickly algorithms moves through numbers
- reversed order: this will show the difference between some algorithms particularly clearly, like selection sort.

Because of the way timing works on Unix/Linux, it was necessary to repeat the same test multiple times to get more consistent data, as timing data is different every time.

We were able to use up to quite large test cases without storage overhead because (a) we had a data generator that could generate consistent inputs to be used for multiple test runs, (b) we had already demonstrated that the program worked correctly, so there was no need to check the output.

Experimental Results

Correctness Experiments

We tested both algorithms against the linux sort command using initial inputs of random, sorted and reverse sequences of lengths ranging from 100 to 1000000. We then tested the difference of the outputs from the two using the *diff* command. It produced no output meaning that there were no differences between our programs and the linux sort. Therefore we can say that the algorithms produce correct results.

For stability test, we created text files with data with unique labels for and identical values (eg. *1 abc* and *1 gfh*) and then inputted them into the programs.

For sortA, we observed that for every test we did, it would output the sorted values with duplicates in the exact order they were initially in. From these results we can conclude that sortA is stable.

For sortB, we observed that the output order of duplicates would randomly change every single time we ran tests. We can conclude that sortB is an unstable program.

Performance Experiments

For Program A, we observed that for the three types of tests (random, sorted and reverse order), the times were all very similar and this was true for all tested sizes. We saw that as the size increased, the program took increasingly more time to sort it properly for all cases. Another thing we observed was that the results was noticeably smaller for when the initial input was in reverse order but because the difference was very small and will not have that much of an effect on the final results.

These observations indicate that the algorithm underlying the program has a constant time complexity no matter the initial input order. We tested each size with each input order three times and used that data to find the average for each size at a particular input order. Fortunately, because all our times for each input order were so similar, we were able to then find an average time for each input size which we then graphed. We found that the graph for the sortA program followed a x^2 curve (results followed 1^2, 2^2 all the way to 12^2). Also, because all the times for each input order were very similar, the sortA program's worst case is also it's best case and also it's average case.

For Program B, we observed that for each tested size, the three inputs (reverse, sorted and random order) produced very similar times. Something notable was that the results grew incredibly slowly even when incrementing the size by 200000 each time, rising only by 0.1 every time.

These observations indicate that the algorithm underlying the program also has a constant time complexity no matter the initial input order. After finding all the averages for each input size and input order, we graphed the results which displayed a logarithmic curve, more precisely $n \log n$. Similar to sortA, sortB's worst case is also it's best and average case because the times are so close to each other.

Conclusions

On the basis of our experiments and our analysis above, we believe that

- ProgramA implements the *Selection* sorting algorithm. Selection sort is quite inefficient because it has to go through the whole list looking for the smallest value each time. This will take a long amount of time for the algorithm to sort it, which is represented in our results and graph showing that it follows the curve of n^2. The data shows that it has very similar times for all three test (reverse, ordered and random) meaning that it has the same best, worst and average case. This would make sense as is has to loop through the whole list everytime, making the time complexity constant. Our results also show that ProgramA is stable which would be correct as selection sort finds a minimum value and looks for and that are smaller than it (not smaller than or equal to).

- ProgramB implements the *Quicksort Median of Three* sorting algorithm. Quicksort in general is quite quick and efficient as it is a recursive function picks the last element as the pivot, reorders values less than pivot to the left and values more than pivot to the right. Because of this, quicksort has an average and best case of $n \log n$ and a worst case of n^2. However, the results show that the program is similar for all three cases which leads us to believe that it is quicksort median of three. It is the exact same except that it uses the median of the two endpoints and the current middle value as the pivot. Our experiment and results also show that programB is unstable which is in accordance with quicksort median of three as it will switch the last duplicate with the pivot.

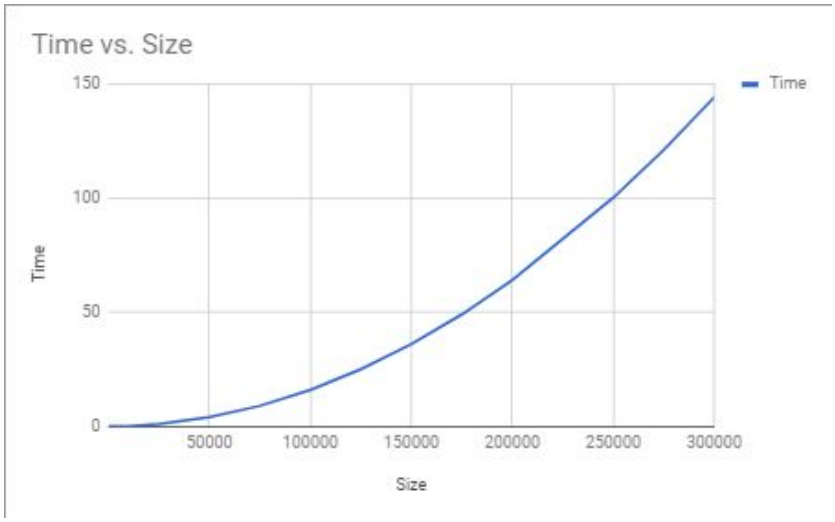
Appendix

Average time of all three cases for sortA

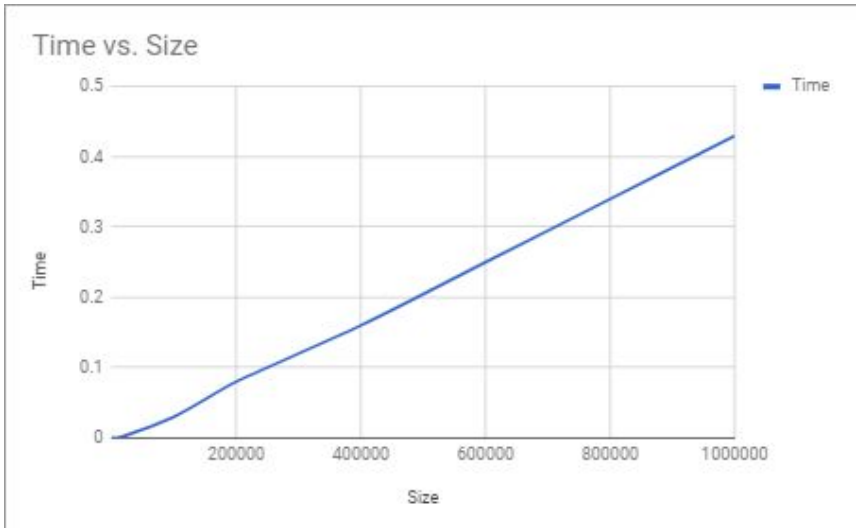
Size	Time (s)
100	0
1000	0
10000	0.16
25000	1.02
50000	4.03
75000	9.05
100000	16.06
125000	25.09
150000	36.11
175000	49.14
200000	64.21
250000	100.31
275000	121.33
300000	144.36

Average time of all three cases for sortB

Size	Time (s)
100	0
1000	0
10000	0
100000	0.03
200000	0.08
400000	0.16
600000	0.25
800000	0.34
1000000	0.43



Graph for sortA



Graph for sortB