COMP2521 Sort Detective Lab Report

by Nathan Ellis, Benjamin Ridler

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 tree:

- 100000 Numbers / 200000 Numbers?
    - With Duplicates / No Duplicates?
        - **Already Sorted Input (best case scenario)**
        - **Randomly Sorted Input**
        - **Reverse Sorted Input (worst case scenario)**

We used these kinds of input (as also indicated in Appendix (b) and Appendix (c)) because different types of sorts are more efficient than others given different orders. The inclusion of completely different input cases of varying sizes and of varying order helped us distinguish which types of sort we have been given, especially when compared to the official time complexities of the sorts which we have also researched (see Appendix (a)).

Each type of test conducted on our sort was repeated 50 times in order to reduce discrepancies between times on the same input. An average was then taken after running these tests to help determine the correct time results for the sorts in different situations. By including multiple instances of best case (being an already sorted list), average case (being a randomly sorted list) and worst case (being a reverse sorted list) scenarios, we were able to successfully explore the output data of our given sorts and hopefully distinguish both types correctly.

One such example of this is insertion sort, which is the best when the list is already sorted ($O(n)$) but in its worst case scenario it is $O(n^2)$. This can be directly compared to the time complexity of a mergesort, which has the same time $O(n\log n)$ regardless of the initial order of the list, in order to distinguish between the two sorts. This process was closely followed in order to distinguish each of the sorts from each other and form Appendix (a) with the aim of finding the types of sort we were given to solve.

Performance Analysis

In our performance analysis, we measured how each program's execution time varied as the size and initial sorted/random/unsorted sequences of the input varied.

To test for time complexities and the results of our given sorting algorithms on Unix/Linux, we were presented with a shell script called “runtests” which presented the time of each sort, split into “real”, “user” and “sys” as well as details regarding the ‘swaps’/movements undergone during the sorting process. We found it was necessary to repeat the same test multiple times in order to gain an average, in order to ensure the validity of our experiment, as there will always be discrepancies between repeated results due to the nature of the Unix terminal and the Linux environment when compared to other operating systems. The amount of system use also had an impact on the real-time time results gained and hence we resolved this by gaining a mean of our repeated results. We tested each sorting case we had already decided on, a total of 50 times each (on both sorts) in order to conclude the results we were getting were correct.

We decided to use very large test cases because:

(a) we had a data generator that could generate user-defined input sizes and could be repeated for multiple test runs;

(b) we had already demonstrated that the sorting program worked correctly, so there was no need to check our output, and;

(c) we had to clearly demonstrate differences in time and a large input size accentuated the difference in time results (see Appendix (b) and Appendix (c)).

We also investigated the stability of the sorting programs by checking for changes/movements in a specific list of {1,1,1,1,1} on both of our given sorts. This is because if a sorting algorithm is said to be “unstable”, then for any items that are of the same value (or rank the same), it is not guaranteed to stay the same with successive sorts of that collection. For a sort that is “stable”, the tied entries will not be moved and hence will always end up in the same order when sorted. If the sort was unstable, there would be no given output as there were no movements undergone. This was exactly what happened for both of our sorts (sortA and sortB) and hence our conclusion was that both of our sorts are “stable” and therefore, could not be ‘unmodified bubble sort’.

Experimental Results

Correctness Experiments

An example of one of our tests was when we gave sortA an input of 100000 randomly sorted numbers (without the presence of duplicates), the sorting algorithm took an average of 43.48 seconds.

To see a more detailed description of our test cases and their relevant time averages, please see Appendix (b) and (c).

Performance Experiments

For Program A, we observed that the sort was stable and correctly sorted the given output. Furthermore, we observed minimal difference in the time between our average (random sorted list) and the worst (reverse sorted list). However, the sorting time was exponentially reduced when the initial order of the input was sorted. This indicates the sort was stable and is in fact optimised as it seems to not undergo any unnecessary checks for already sorted input. Moreover, when we compared the results of sortA to sortB, it is clear that sortA is not as fast as sortB, and hence it appears the sort is either 'Bubble Sort with Early Exit' or 'Insertion Sort'.

Following this, it was necessary to add an additional test to distinguish between these two types in which we add one element at the very end of the input which was not sorted. In a bubblesort, this would only require one iteration to completely sort, whereas an insertion sort would require multiple iterations. Investigating this, I added an extra row to Appendix (b), which specifically treated this case and indeed, there was a miniscule difference between a sorted list input and one changed only slightly.

These observations (above) indicate that the algorithm underlying the program "sortA" is a 'bubble sort with an early exit'!*

**To see a more detailed bar graph and line graph of our data averages for sortA, please see Appendix (d) and (f).*

For Program B, we also observed that the sort was stable and correctly sorted the given output. Indeed, the best and worst case input cases resulted in almost identical time averages. However, puzzlingly, we viewed that an average, randomly sorted input resulted in very little time average. This confused us very much as none of the sorts could possibly have such a weird time difference for an average case, and after repeating different randomly sorted inputs with sortB, the result was consistently amazingly quick. We concluded that it was near impossible to distinguish why this would happen, and hence ignored this information.

Moreover, these were significantly quicker than that observed in sortA. This meant the algorithm given was consistent in its time complexity, when comparing the best and worst case, as well as needing lesser time than sortA as we had observed. The two options that this sort could be, we concluded, is 'Randomised Quick Sort' or 'Merge Sort'. Based on our research, however, we knew that a 'Randomised Quick Sort' would have more accentuated discrepancies due to the nature of its "randomised" choice of pivot values. This did not occur when we underwent our rigorous 50-repetition testing.

These observations (above) indicate that the algorithm underlying the program "sortB" is a 'Merge Sort'!*

**To see a more detailed bar graph and line graph of our data averages for sortB, please see Appendix (e) and (g).*

Conclusions

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

- ProgramA implements the *'Bubble Sort with an Early Exit'* sorting algorithm
- ProgramB implements the *'Merge Sort'* sorting algorithm

Appendix

Appendix (a):

Sorting Algorithm	***Time Complexity***			***Stability***
	Best	***Worst***	***Average***	
Pseudo-Bogo Sort	O(n)	O((n+1)!)	O((n+1)!)	Stable
Quick Sort Median of Three	O(n)	$O(n^2)$	O(n log n)	Stable
Bubble Sort with Early Exit	O(n)	$O(n^2)$	$O(n^2)$	Stable
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	Stable
Randomised Quick Sort	O(n log n)	O(n log n)	O(n log n)	Stable
Merge Sort	O(n log n)	O(n log n)	O(n log n)	Stable
Shell Sort (Sedgewick)	O(n log n)	$O(n^{4/3})$	???*	Stable
Shell Sort (Powers of Four)	O(n log n)	$O(n^2)$	???*	Stable
Vanilla Quick Sort	O(n log n)	$O(n^2)$	O(n log n)	Stable
Oblivious Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Unstable
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Stable

The results obtained above were collected from various websites provided in the Sort Detective lab exercise as well as expected complexities for the random, pseudo-bogo sort.

** Based on various articles, we obtained that the average complexity of shell sorts are in fact unable to be estimated for various reasons discussed above.*

Appendix (b):

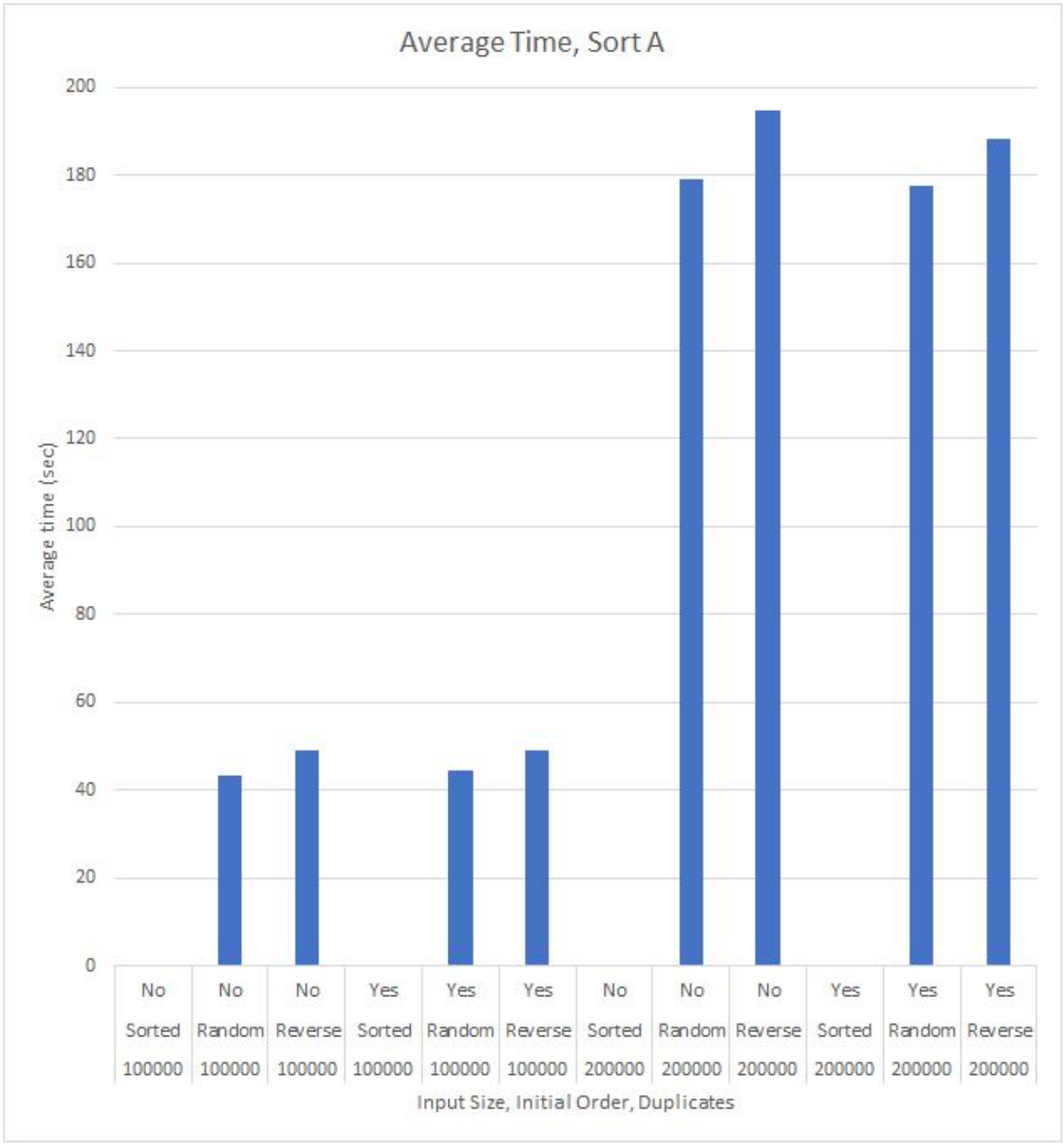
sortA				
Input Size	Initial Order	Has Duplicates	Number of Tests	Avg. Time (sec)
100000	Sorted	No	50	0.03
100000	Random	No	50	43.48
100000	Reverse	No	50	49.15
100000	Sorted	Yes	50	0.04
100000	Random	Yes	50	44.59
100000	Reverse	Yes	50	49.01
200000	Sorted	No	50	0.09
200000	Random	No	50	179.11
200000	Reverse	No	50	194.67
200000	Sorted	Yes	50	0.08
200000	Random	Yes	50	177.57
200000	Reverse	Yes	50	188.07

100000	Sorted, except one element!	No	50	0.56

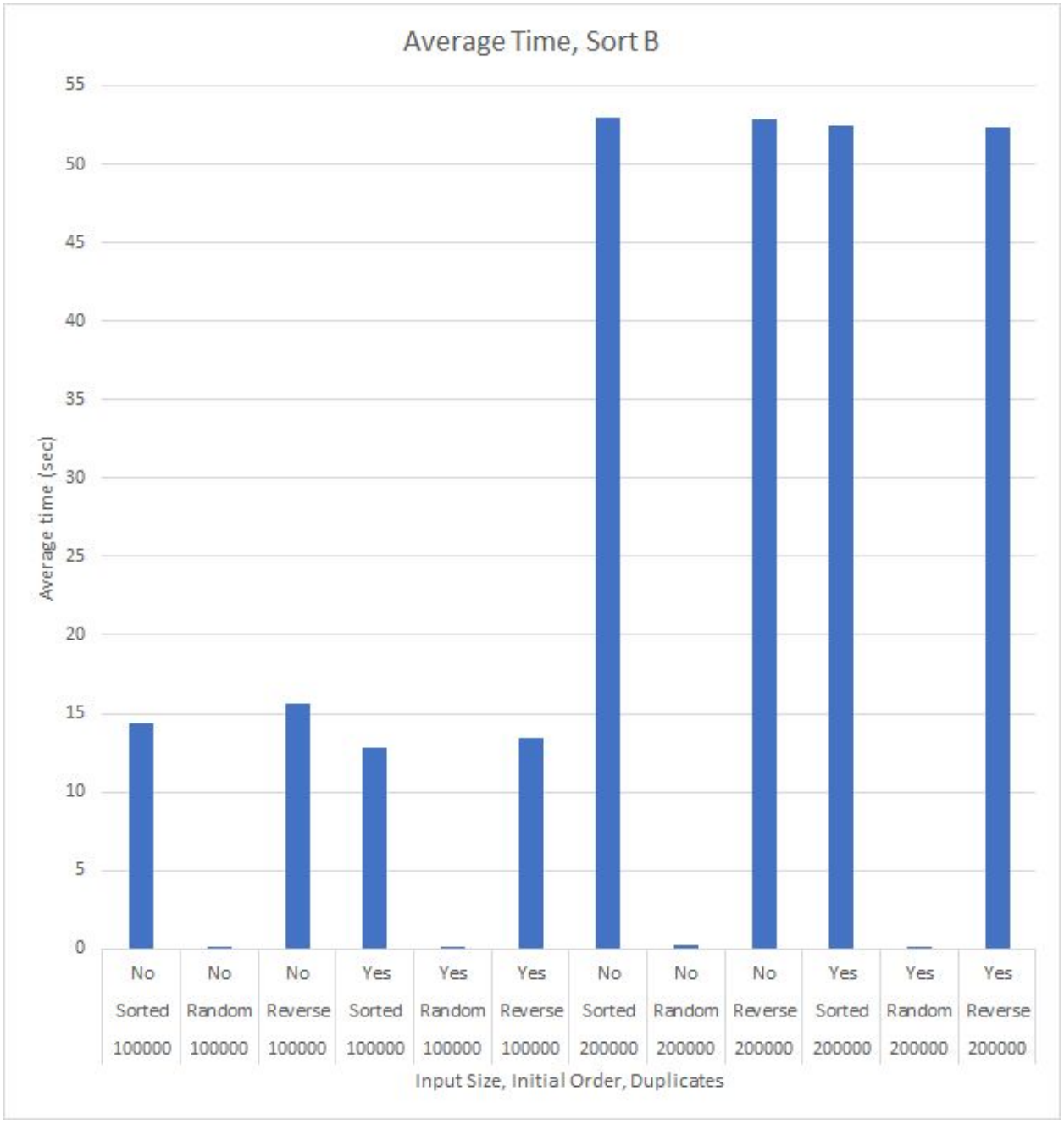
Appendix (c):

sortB				
Input Size	Initial Order	Has Duplicates	Number of Tests	Avg. Time (sec)
100000	Sorted	No	50	14.43
100000	Random	No	50	0.04
100000	Reverse	No	50	15.67
100000	Sorted	Yes	50	12.80
100000	Random	Yes	50	0.04
100000	Reverse	Yes	50	13.43
200000	Sorted	No	50	53.01
200000	Random	No	50	0.19
200000	Reverse	No	50	52.85
200000	Sorted	Yes	50	52.41
200000	Random	Yes	50	0.16
200000	Reverse	Yes	50	52.36

Appendix (d):


Average Time, Sort A
Average time (sec)
200
180
160
140
120
100
80
60
40
20
0
No Sorted 100000
No Random 100000
No Reverse 100000
Yes Sorted 100000
Yes Random 100000
Yes Reverse 100000
No Sorted 200000
No Random 200000
No Reverse 200000
Yes Sorted 200000
Yes Random 200000
Yes Reverse 200000
Input Size, Initial Order, Duplicates

Appendix (e):


Average Time, Sort B
Average time (sec)
55
50
45
40
35
30
25
20
15
10
5
0
No Sorted 100000
No Random 100000
No Reverse 100000
Yes Sorted 100000
Yes Random 100000
Yes Reverse 100000
No Sorted 200000
No Random 200000
No Reverse 200000
Yes Sorted 200000
Yes Random 200000
Yes Reverse 200000
Input Size, Initial Order, Duplicates

Appendix (f):

Appendix (g):


Average Time, Sort B
25
20
15
10
5
0
Sorted
Random
Reverse


Average Time, Sort B
25
20
15
10
5
0
Sorted
Reverse

Appendix (h):

INITIAL NOTES FROM FIRST TUTORIAL LESSON:

- Unmodified bubble sort is the only unstable sort
- Mergesort should be the quickest (nlog(n)) for descending series
- BubbleSort/SelectionSort should be the slowest (n^2) for ascending (sorted) series
- Quicksort/ShellSort1/ShellSort2 should be the quickest (nlog(n)) for ascending (sorted) series, but slowest for descending (not sorted) series.
- Psuedo-Bogo Sort will be completely random in times on the same set due to the randomness in the sorting algorithm.
- Unmodified Quicksort can help avoid worst case such as perfectly sorted sets (compared to Vanilla and median of Three)
- Shellsort Powers should be quicker than shellsort Sedgewick on smaller data sets