COMP2521 Sort Detective Lab Report

By z5421641

The purpose of the following lab report is to analyse the performance of two unknown sorting programs, in an attempt to identify the sorting algorithms being used.

Experimental Design

In order to identify the algorithm we are given in SortA and SortB, it is best if we first differentiate each sorting algorithm based on stability, best and worst case time complexity. As such refer to the table below:

	Best Case Time Complexity	Worst Case Time Complexity	Stability
Bubble sort	O(n)	O(n^2)	Yes
Insertion sort	O(n)	O(n^2)	Yes
Selection sort	O(n^2)	O(n^2)	No
Merge sort	O(nlog(n))	O(nlog(n))	Yes
Naive Quicksort	O(nlog(n))	O(n^2)	No
Randomised Quicksort	O(nlog(n))	O(n^2)	No
Median-of-three Quicksort	O(nlog(n))	O(n^2)	No
Bogosort	O(n)	O(infinity)	No

Figure 1: The following table compares the various sorting algorithms based on time complexity and stability and was adapted from the David and Chelsea week 8 lab.

To successfully identify which algorithm is which, we must first set up some test cases which consists of observing the behaviour of sorting on duplicates inputs, analysing the time taken with inputs of different sizes and finally observing the time taken to sort:

1. Input that is already sorted
2. Input that is randomised
3. Input that is reversed

As a result, we will first analyse the time taken for each algorithm to sort a sorted, randomised and reversed set of data which starts from 10000 and doubles up each time until 320000. Finally, we will measure the stability of each algorithm by utilising duplicate alpha-numeric characters to identify if the algorithm is stable.

Time Complexity Analysis

We will first begin by analysing the time taken for the programs to sort in direct comparison to an increasing number of inputs. To further increase the validity of the following experiment we will perform these sorts on various types of inputs that have been sorted, randomised and reversed and represent the results in tabular form. As such we will perform the tests in the following order:

1. 10000 inputs which are sorted, randomised and reversed
2. 20000 inputs which are sorted, randomised and reversed
3. 40000 inputs which are sorted, randomised and reversed
4. 80000 inputs which are sorted, randomised and reversed
5. 160000 inputs which are sorted, randomised and reversed

In order to ensure that the tests are reliable these tests will be repeated 5 times in order to narrow any inconsistencies as a result of Unix/Linux time disparities, where I will take the average of all the 5 runs. By testing on sorted, randomised and reversed cases, this will enable us to effectively test the worst, best and average case time complexity.

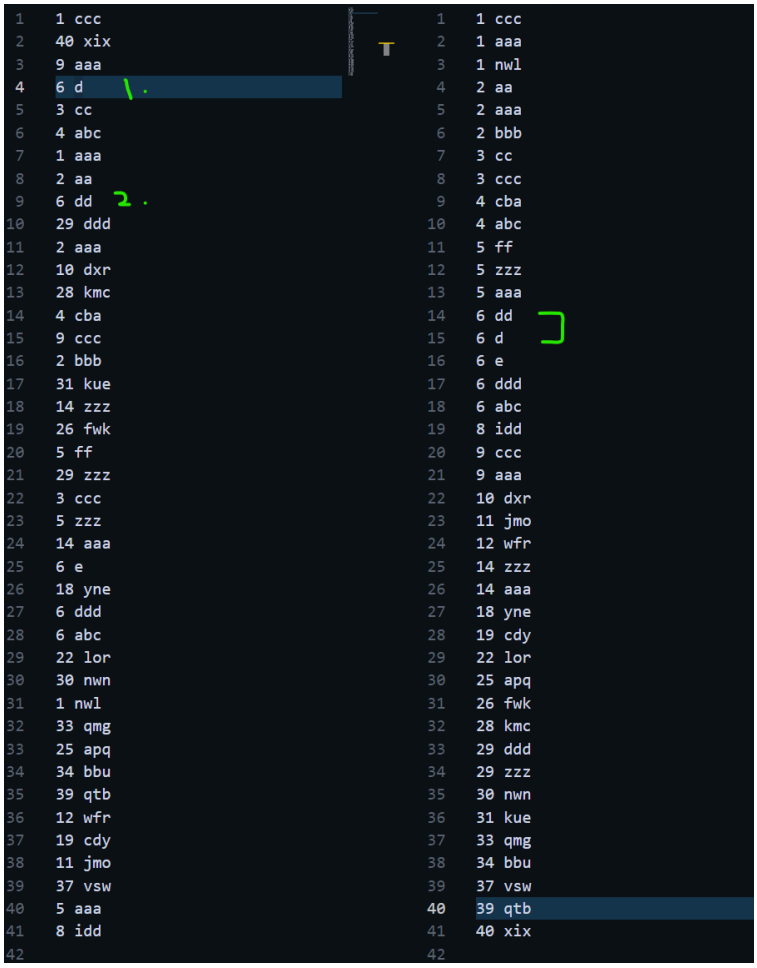
Stability Analysis

To further confirm that we have found the correct algorithm, we will then compare the stability of each sorting algorithm by creating a random range of numbers which consist of duplicate numbers which have alphabets to keep track of their order. It is best if we first understand what stability refers to, a stable algorithm is one that preserves their order when dealing with duplicate keys, these include bubble sort, insertion sort and merge sort. An unstable algorithm on the other hand doesn't preserve their order when dealing with duplicate keys, these include selection sort, quick sort and bogo sort. As a result, with both time complexity and stability analysis we can find the sorting algorithm.

Experimental Results

Program A

For Program A, through analysing the stability, it can be seen that the sortA isn't stable. This being due to the fact that in line 4 of the output, "6 d" appears before line 9, "6 dd". When observing the sorted result, in line 14, "6 dd" is placed before line 15 "6 d". As a result, this confirms that the sort doesn't maintain the order of duplicates when performing the sort hence being unstable. This narrows down our search to selection sort, naive quick sort, median-of-three quicksort, randomised quicksort and bogosort.

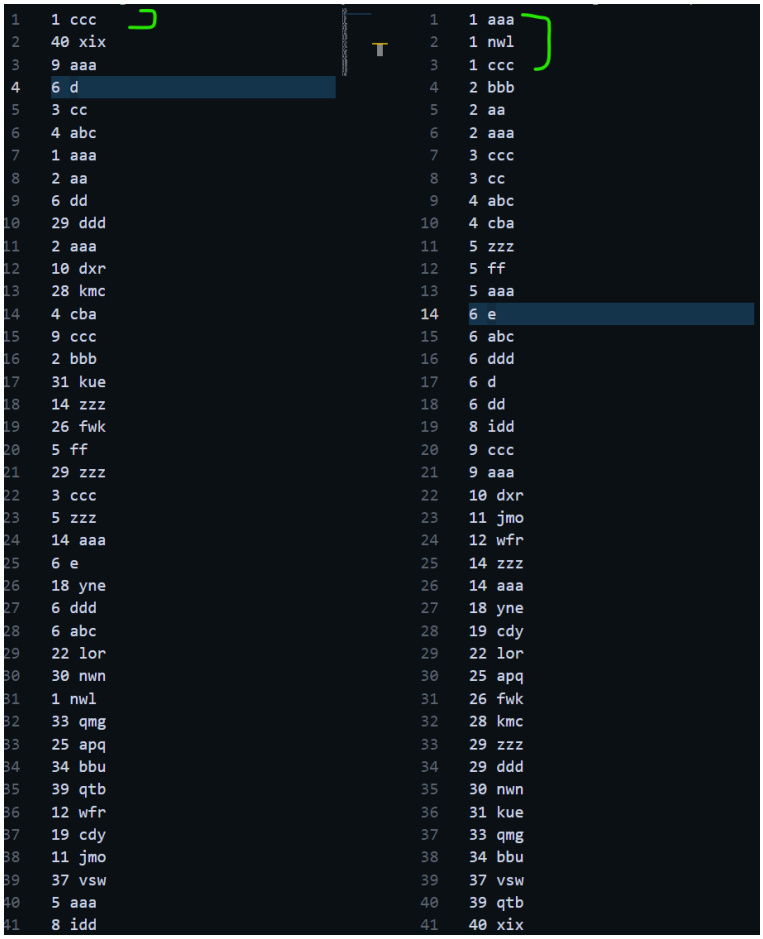


Shifting our attention to the time complexity analysis in figure 2, it can be seen that the differences in time taken between inputs which were sorted, randomised and reversed are similar. As a result this allows us to understand that the best and worst case time complexities must be very similar. This concept is further consolidated by the graph in figure 4, where all the three lines are incredibly close meaning that the best and worst

time complexities are very similar and the graph resembles the shape of an O(n^2) time complexity graph. As a result we can come to a conclusion that this graph models the selection sort algorithm due to having an O(n^2) time complexity curve and being unstable.

Program B

For Program B, through analysing the stability, it can be seen that sortB is unstable. This being due to in line 1 of the input file, it can be seen that “1 ccc” should appear first in the output file, however line 1 of the output file shows “1 aaa” to appear first. This as a result confirms that as sortB attempts to sort duplicates, it fails to maintain its order. This in turn narrows our search down to selection sort, naive quick sort, median-of-three quicksort, randomised quicksort and bogosort.



1 ccc	1 aaa
40 xix	1 nwl
9 aaa	1 ccc
6 d	2 bbb
3 cc	2 aa
4 abc	2 aaa
1 aaa	3 ccc
2 aa	3 cc
6 dd	4 abc
29 ddd	4 cba
2 aaa	5 zzz
10 dxr	5 ff
28 kmc	5 aaa
4 cba	6 e
9 ccc	6 abc
2 bbb	6 ddd
31 kue	6 d
14 zzz	6 dd
26 fwk	8 idd
5 ff	9 ccc
29 zzz	9 aaa
3 ccc	10 dxr
5 zzz	11 jmo
14 aaa	12 wfr
6 e	14 zzz
18 yne	14 aaa
6 ddd	18 yne
6 abc	19 cdy
22 lor	22 lor
30 nwn	25 apq
1 nwl	26 fwk
33 qmg	28 kmc
25 apq	29 zzz
34 bbu	29 ddd
39 qtb	30 nwn
12 wfr	31 kue
19 cdy	33 qmg
11 jmo	34 bbu
37 vsw	37 vsw
5 aaa	39 qtb
8 idd	40 xix

Shifting our attention to the time complexity analysis of sortB in figure 3, it can be seen that sorting a set of sorted and reversed inputs took relatively similar time compared to sorting a randomised set of numbers, which took significantly less time. This in turn tells us that there exists a huge difference between the best and worst case time complexity of the following algorithm. This concept is further consolidated by figure 5, where the graph seems to produce

an O(n^2) time complexity curve for the time taken to sort a sorted and reversed set of inputs, alluding to the fact that the worst case time complexity may be O(n^2). This even further narrows our search to the naive quick sort, median-of-three quicksort and randomised quicksort. Since a randomised set of numbers is incredibly fast to solve, it hints to the fact that the algorithm we are dealing with is indeed a randomised quicksort. To make sure this is the case, observe figure 6, where the following graph zooms in the randomised input plot. Here it can be seen that the following curve produced looks like the O(nlog(n)). Since the best case time complexity is around O(nlog(n)), seen in randomised inputs, the worst case time complexity being O(n^2) and the algorithm being unstable, clearly confirms that this is a randomised quicksort algorithm.

Conclusions

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

- sortA implements the *selection* sorting algorithm
- sortB implements the *randomised quick* sorting algorithm

Appendix

Size of Input / Type of input	Sorted (seconds nearest to 3dp)	Randomised (seconds nearest to 3dp)	Reversed (seconds nearest to 3dp)
10000	0.140	0.146	0.132
20000	0.570	0.570	0.548
40000	2.288	2.298	2.210
80000	9.140	9.182	8.782
160000	36.862	37.058	35.39
320000	146.388	146.442	140.156

Figure 2: The following table compares the the time taken to sort 10000 - 320000 inputs which are sorted, randomised and reversed, using the sortA sorting algorithm.

Size of Input / Type of input	Sorted (seconds nearest to 3dp)	Randomised (seconds nearest to 3dp)	Reversed (seconds nearest to 3dp)
10000	0.120	0.000	0.130
20000	0.494	0.000	0.534
40000	1.992	0.010	2.136
80000	7.966	0.020	8.556
160000	31.812	0.040	34.204
320000	100.956	0.102	108.630

Figure 3: The following table compares the the time taken to sort 10000 - 320000 inputs which are sorted, randomised and reversed, using the sortB sorting algorithm.

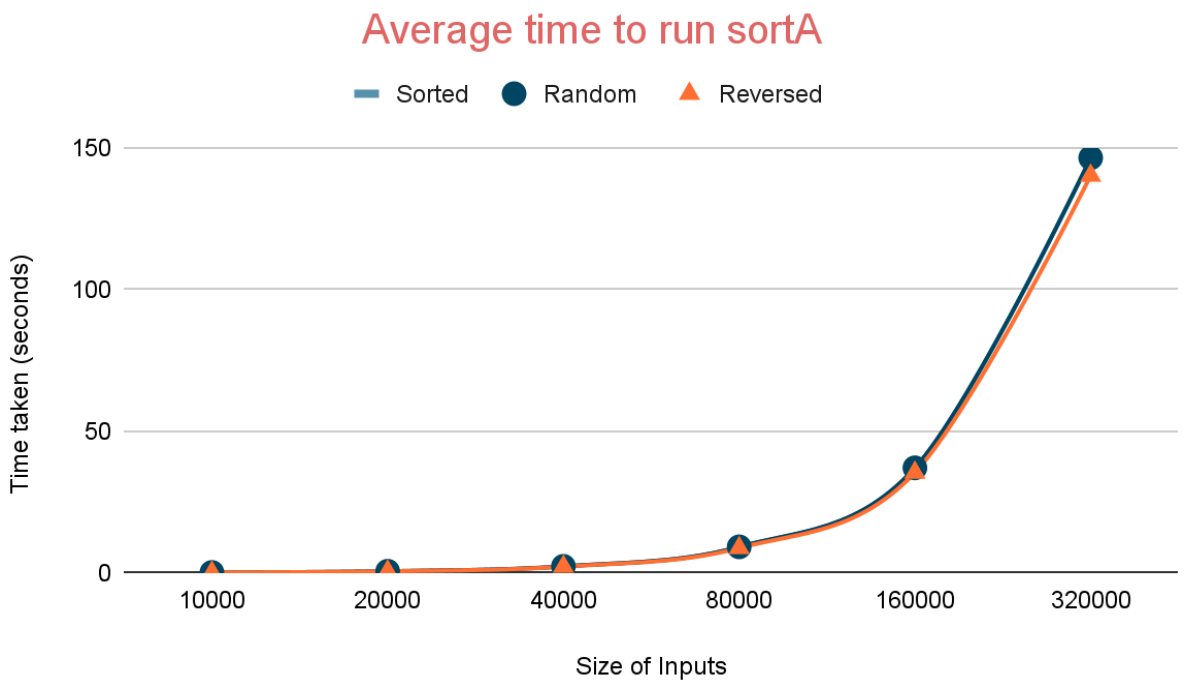


Figure 4: The following graph compares and visually defines the the time taken to sort 10000 - 320000 inputs which are sorted, randomised and reversed, using the sortA sorting algorithm.

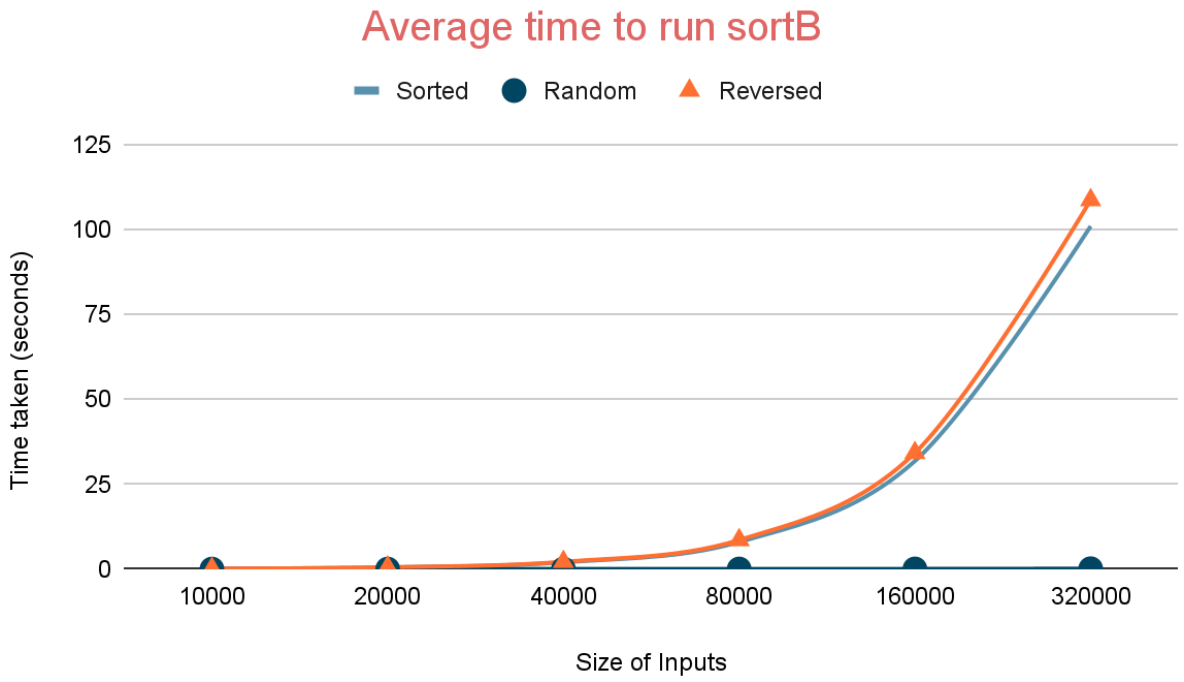


Figure 5: The following graph compares and visually defines the the time taken to sort 10000 - 320000 inputs which are sorted, randomised and reversed, using the sortB sorting algorithm.

Figure 6: The following graph compares and visually defines the the time taken to sort 10000 - 320000 inputs which randomised, using the sortB sorting algorithm.