COMP2521 Sort Detective Lab Report

by Dinh Minh Nhat Nguyen (z5428797)

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

Experimental Design

There are two aspects to our analysis:

- Determine that the sorted programs are correct.
- Measure their performance over a range of inputs.

Correctness Analysis:

To determine correctness, both programs will be compared with the Linux's inbuilt sorting function through the outputs, increasing by 10,000, between 10,000 to 100,000. We chose numerical data to analyse as it was relatively easy to use the diff command in Linux and check the correctness of the sorting functions at the different outputs.

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:

Sorted A: 10,000 – 100,000

Sorted B: 10,000 – 100,000

As well as inputs of random, sorted, reversed cases will be provided as to showcase different time complexities.

Because of the way timing works on Unix/Linux, it was necessary to repeat the same test multiple times. The command works by sampling and producing different results for the same program run multiple times.

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.

We chose these inputs because they were sufficiently large to showcase significant numbers in terms of timing. The large range of numbers also allowed it to plot the time complexity on a graph (see appendix).

At each size, we had lists of sorted, reversed and random data to determine if the sort was adaptable (I.e.: Faster if the data is already sorted). The data had two variables, first were the numbers on which the data was sorted and next were a bunch of characters. This allowed us to check for stability.

Experimental Results

For Program A, we observed that sortA keeps the original list order. This means sortA must be bubble sort, insertion sort or merge sort due to its stability.

From the timings, sortA timing consistently increases proportionally with increases in list length. Sort A cannot be a insertion sort with early exit as this would mean that sorted times are the fastest in all cases. Moreover, the data shows that random and reversed have a time complexity is O(n^2) so it cannot be merge sort.

Therefore, sort A is bubble sort.

For Program B, we observed that sortB does not preserve the original list order. This means sortB must be selection sort, Naive quicksort, Median-of-three quicksort, Randomised quicksort, Bogosort.

From the timings, we can see that sortB is much faster than sortA so we can conclude that it is not Bongo sort since Bongo is most likely the slowest of all the algorithms. Moreover, the random sort has the fastest time in all cases and has time complexity is O(n^2).

Therefore, sort B is Selection Sort.

Conclusions

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

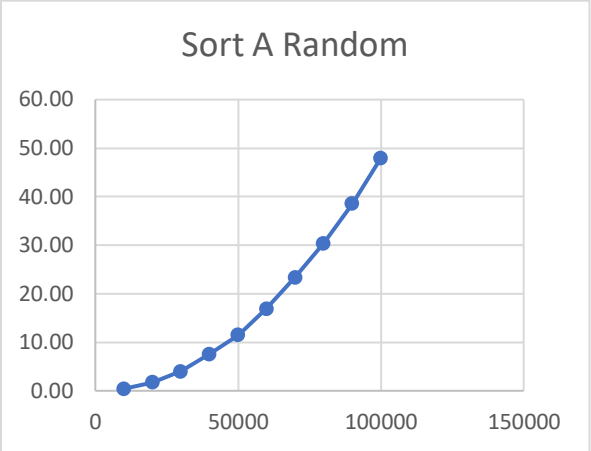
- sortA implements the *bubble* sorting algorithm
- sortB implements the *Selection Sort* sorting algorithm

Appendix

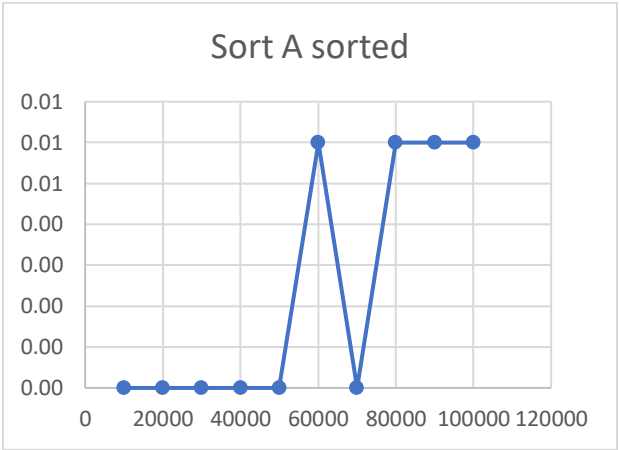
Any large tables of data that you want to present ...

Sort A data:

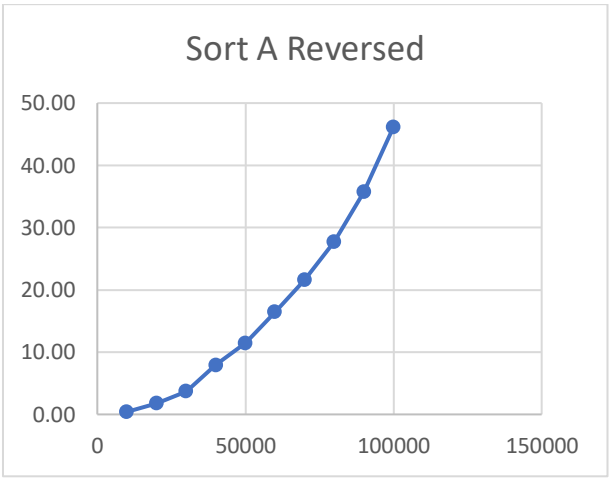
Random		aver
	10000	0.40
	20000	1.70
	30000	4.01
	40000	7.59
	50000	11.52
	60000	16.90
	70000	23.35
	80000	30.36
	90000	38.54
	100000	47.95



Sorted		aver
	10000	0.00
	20000	0.00
	30000	0.00
	40000	0.00
	50000	0.00
	60000	0.01
	70000	0.00
	80000	0.01
	90000	0.01
	100000	0.01

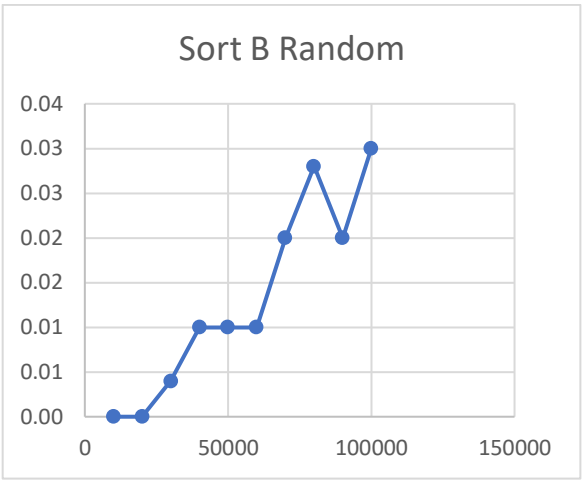


Reversed		aver
	10000	0.41
	20000	1.80
	30000	3.73
	40000	7.93
	50000	11.46
	60000	16.45
	70000	21.62
	80000	27.77
	90000	35.71
	100000	46.18

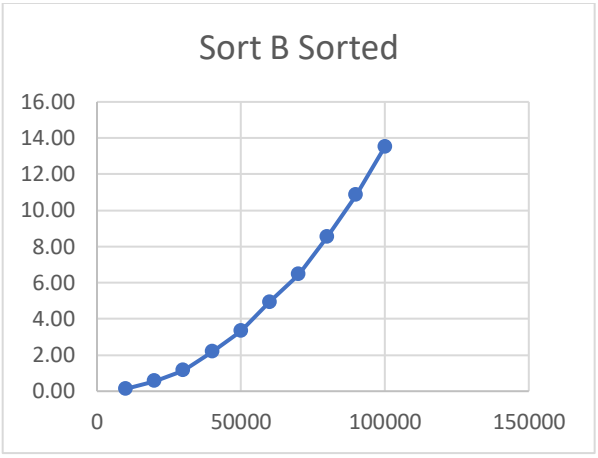


Sort B data:

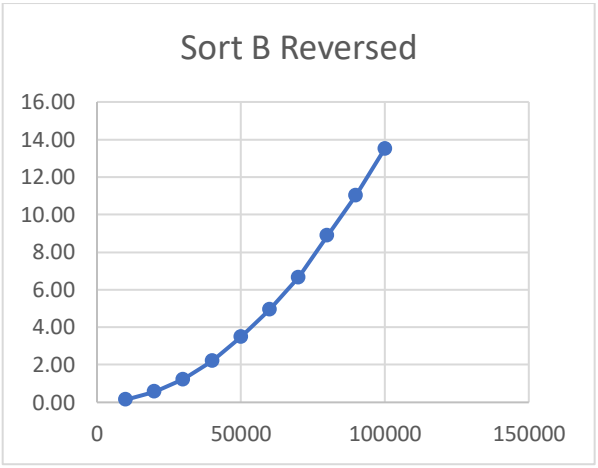
Random		aver
	10000	0.00
	20000	0.00
	30000	0.00
	40000	0.01
	50000	0.01
	60000	0.01
	70000	0.02
	80000	0.03
	90000	0.02
	100000	0.03



Sorted		aver
	10000	0.13
	20000	0.55
	30000	1.14
	40000	2.18
	50000	3.32
	60000	4.92
	70000	6.45
	80000	8.54
	90000	10.85
	100000	13.52



Reversed		aver
	10000	0.14
	20000	0.57
	30000	1.22
	40000	2.21
	50000	3.48
	60000	4.93
	70000	6.66
	80000	8.88
	90000	11.03
	100000	13.52



Testing stability:

data1

OG	SortedA	SortedB
1 dxr	1 dxr	1 qsc
4 hcd	1 qsc	1 dxr
5 arz	2 mqb	2 mqb
7 kyh	2 rbb	2 rbb
4 nwl	4 hcd	4 hcd
8 idd	4 nwl	4 nwl
5 owk	5 arz	5 arz
1 qsc	5 owk	5 owk
2 mqb	7 kyh	7 kyh
2 rbb	8 idd	8 idd

data2

OG	SortedA	SortedB
1 nwl	1 nwl	1 qsc
2 rbb	1 qsc	1 dxr
2 mqb	1 dxr	1 nwl
3 hcd	2 rbb	2 kyh
5 arz	2 mqb	2 mqb
2 owk	2 owk	2 owk
2 kyh	2 kyh	2 rbb
8 idd	3 hcd	3 hcd
1 qsc	5 arz	5 arz
1 dxr	8 idd	8 idd

data3

OG	SortedA	SortedB
5 dxr	1 arz	1 arz
5 qsc	1 hcd	1 hcd
2 idd	1 nwl	1 nwl
4 kyh	2 idd	2 idd
2 owk	2 owk	2 owk
1 arz	4 kyh	4 mqb
1 hcd	4 mqb	4 rbb
4 mqb	4 rbb	4 kyh
4 rbb	5 dxr	5 dxr
1 nwl	5 qsc	5 qsc