

Lab Week 9

Kerelos Tawfik

Jake Edwards z5165158

Experiment Design

- **Bubble sort - standard bubble sort (as shown in lectures)**
 - Passes through an array.
 - Compares adjacent elements and swaps until they are at the right position.
 - Repeats until no swaps are done during one pass.
- **Insertion sort - standard insertion sort**
 - Treat first element as sorted array
 - Takes next element then inserts it into sorted array in the correct order
 - Repeats until array is sorted
- **Selection sort - standard unstable selection sort**
 - Finds smallest element, puts it in first array slot.
 - Finds second smallest puts it into second array slot.
 - Repeat until all elements are in correct position.
- **Merge sort - standard merge sort**
 - Split the array into two equally sized partitions.
 - Recursively sort each of the partitions.
 - Merge the two partitions into a new sorted array.
 - Copy back to original array.
- **Naive quicksort - uses the leftmost element as the pivot**
 - Chooses an element to be a pivot
 - In this case the leftmost element
 - Partitions the array so that elements to the left of the pivot are less than the pivot and elements to the right of the pivot are greater than the pivot
 - Recursively sorts each partition
- **Median-of-three quicksort - uses the median of the first, middle and last elements as the pivot**
 - Looks at the first middle and last elements of the array.
 - Chooses the median of those three elements as the pivot.
- **Randomised quicksort - uses a randomly chosen element as the pivot**
 - Same as Naive quicksort with random pivot
 - Worst case smallest or largest element chosen.
- **Bogosort - repeatedly generates permutations of its input until one is found that is sorted**
 - The algorithm successively generates permutations of its input until it finds one that is ordered.
-

Draw.io figure

Is

<https://stackoverflow.com/questions/5001602/does-quicksort-with-randomized-median-of-three-do-appreciably-better-than-random>

Sorting Algorithm Properties

	Bubble Sort	Insertion Sort	Selection Sort	Merge Sort	Naive Quicksort	Median-of-three Quicksort	Randomised Quicksort	Bogosort
Is adaptive	yes	Yes (seems to stop when position is found)	no	no	yes	yes	yes	yes
Is stable	yes	yes	no	no	no	no	no	yes
Best Case	$O(n)$	$O(n)$	$O(n^2)$	$n \log(n)$	$n \log(n)$	$n \log(n)$	$n \log(n)$	$n - 1$
Worst Case	$O(n^2)$ $n^2/2$	$O(n^2)$ $n^2/4$	$O(n^2)$	$n \log(n)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	unbounded
					Slowest Quicksort Average Speed	Fastest Quicksort on average	Middle Quicksort Average Speed	

COMP2521 Sort Detective Lab Report

by Jake Edwards (z5165158), Kerelos Tawfik (z5217519)

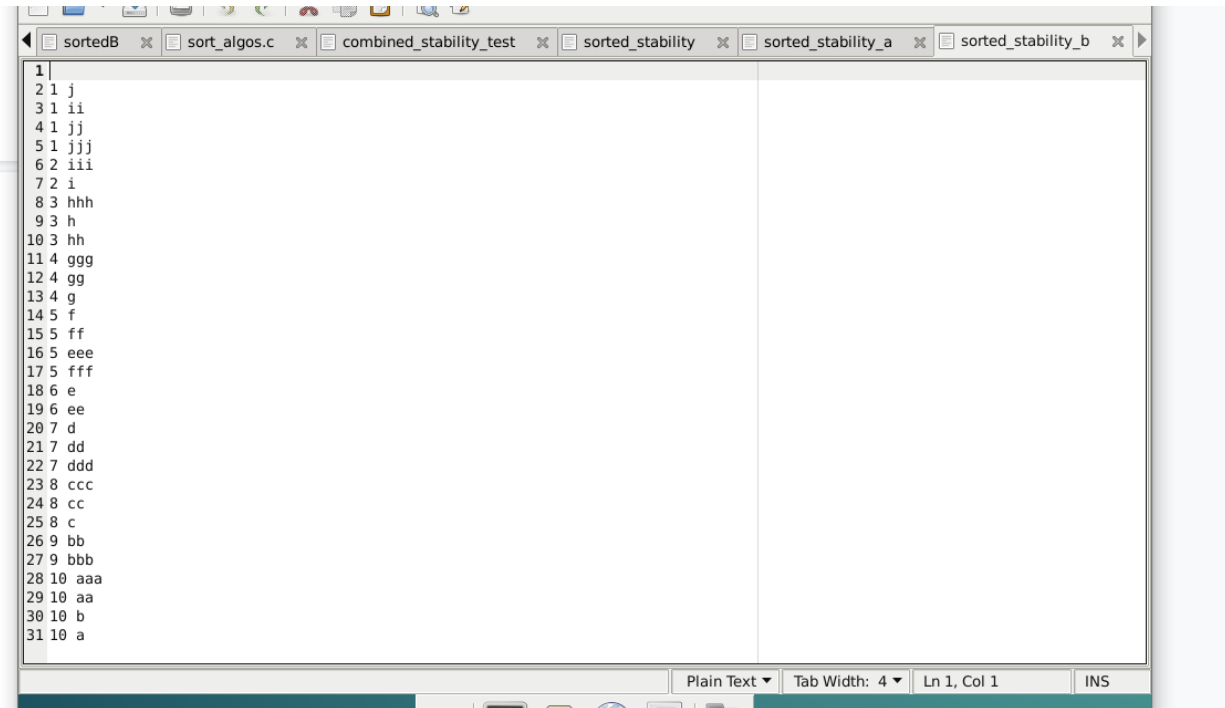
In this lab, the aim is to measure the performance of two sorting programs, without access to the code, and determine which sorting algorithm each program uses.

Stability

The following examples shows that Sort A is a stable algorithm and SortB is unstable.

```
1 10 a
2 10 b
3 8 c
4 7 d
5 6 e
6 5 f
7 4 g
8 3 h
9 2 i
10 1 j
11 10 aa
12 9 bb
13 8 cc
14 7 dd
15 6 ee
16 5 ff
17 4 gg
18 3 hh
19 1 ii
20 1 jj
21 10 aaa
22 9 bbb
23 8 ccc
24 7 ddd
25 5 eee
26 5 fff
27 4 ggg
28 3 hhh
29 2 iii
30 1 jjj
31
```

```
1
2 1 j
3 1 ii
4 1 jj
5 1 jjj
6 2 i
7 2 iii
8 3 h
9 3 hh
10 3 hhh
11 4 g
12 4 gg
13 4 ggg
14 5 f
15 5 ff
16 5 eee
17 5 fff
18 6 e
19 6 ee
20 7 d
21 7 dd
22 7 ddd
23 8 c
24 8 cc
25 8 ccc
26 9 bb
27 9 bbb
28 10 a
29 10 b
30 10 aa
31 10 aaa
```



```
1 |
2 | 1 j
3 | 1 ii
4 | 1 jj
5 | 1 jjj
6 | 2 iii
7 | 2 i
8 | 3 hhh
9 | 3 h
10 | 3 hh
11 | 4 ggg
12 | 4 gg
13 | 4 g
14 | 5 f
15 | 5 ff
16 | 5 eee
17 | 5 fff
18 | 6 e
19 | 6 ee
20 | 7 d
21 | 7 dd
22 | 7 ddd
23 | 8 ccc
24 | 8 cc
25 | 8 c
26 | 9 bb
27 | 9 bbb
28 | 10 aaa
29 | 10 aa
30 | 10 b
31 | 10 a
```

Experimental Design

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

We thought by analysing the stability of the runtimes by checking if it can handle duplicate keys, we could get a step closer to finding what kind of sort it is. We also believe that analysing the runtime complexities of the algorithms that would be a key factor in our findings.

We used these test cases because analysing stability we could potentially rule out selection bubble sort insertion sort and bogosort if it is stable or rule them out as the only options if they are unstable. By analysing their orders of complexity (Big O) and the stability we can rule out the remaining cases as they all have different worst case and best case run times.

Because of the way timing works on Unix/Linux, it was necessary to repeat the same test multiple times we analysed each test case 5 times and averaged the result in order to account for Unix variability.

We also investigated the stability of the sorting programs by

Adding structs with duplicate number values for sorting with different string values

We also investigated ... *any other relevant properties*

We also investigated the curve of best fit as whilst their orders (Big O) (n^2 for both insertion and bubble sort) are nearly identical for some the complexity ($n^2/4$ for insertion sort and $n^2/2$ for bubble sort) differs. By plotting a curve of best fit against their complexities we can see which curve it closely correlates to and rule out other possibilities.

Experimental Results

For Program A, we observed that ...

These observations indicate that the algorithm underlying the program ... *has the following characteristics* The first test was to check algorithm stability, we tested the stability of Program A and observed that it was stable, this left us with three possible results. Bubble sort Insertion sort and Bogosort. We analysed the order of complexity and found it was nearly identical to $O(n)$ for the best case and $O(n^2)$ for the worst case. This narrowed out bogosort from our list of possible options. Through further analysis it seemed that the order of complexity of insertion sort and bubble sort are identical which left us to look at the curve of best fit with complexity. Insertion sort has $n^2/4$ complexity whilst bubble sort has $n^2/2$ complexity. Table 2 and Figure 1 shows our analysis of the curves of best fit and it showed a closer resemblance to $n^2/4$ complexity leaving us with one sorting algorithm which met all the criteria.

For Program B, we observed that ...

These observations indicate that the algorithm underlying the program ... *has the following characteristics* The first test was to check algorithm stability, we tested the stability of Program B and observed that it was unstable. This left us with four distinct options. Selection sort, merge sort, naive quicksort and median-of-three quicksort. Through further analysis of the complexity (Table 1) we found it was almost identical to $O(n \log n)$. This immediately narrowed out selection sort, naive quicksort, median-of-three quicksort and randomised quicksort. This left us with one option with no analysis of the curve of best fit required.

Conclusions

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

- **sortA implements the *Insertion Sort* sorting algorithm**
- **sortB implements the *Merge Sort* sorting algorithm**

Appendix

Table 1:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Sort A								Sort B							
2	Random								Random							
3	Size (n)	10000	20000	40000	80000	160000			Size (n)	10000	20000	40000	80000	160000		
4	1	0.42	1.62	6.49	27.32	108.5			1	0	0	0.01	0.02	0.09		
5	2	0.42	1.61	6.47	27.42	106.16			2	0	0	0	0.02	0.04		
6	3	0.38	1.55	6.25	27.39	105.19			3	0	0	0.02	0.04	0.04		
7	4	0.38	1.65	6.34	26.54	104.55			4	0	0	0.02	0.02	0.07		
8	5	0.39	1.61	6.6	27.16	104.94			5	0	0	0.02	0.02	0.06		
9	Average	0.398	1.608	6.43	27.166	105.868			Average	0	0	0.014	0.024	0.06		
10		3.98E-09	4.02E-09	4.019E-09	4.24E-09	4.14E-09										
11																
12	Sorted								Sorted							
13	Size (n)	10000	20000	40000	80000	160000			Size (n)	10000	20000	40000	80000	160000		
14	1	0	0	0	0.01	0.02			1	0	0	0.01	0.02	0.05		
15	2	0	0	0	0.01	0.01			2	0	0	0.01	0.01	0.06		
16	3	0	0	0	0	0.01			3	0	0	0.01	0.01	0.04		
17	4	0	0	0.01	0	0.02			4	0	0	0.01	0.01	0.06		
18	5	0	0	0	0.01	0.04			5	0	0	0.01	0.02	0.04		
19	Average	0	0	0.002	0.006	0.02			Average	0	0	0.01	0.014	0.05		
20				5E-08	7.5E-08	1.25E-07										
21																
22	Reversed								Reversed							
23	Size (n)	10000	20000	40000	80000	160000			Size (n)	10000	20000	40000	80000	160000		
24	1	0.42	1.68	6.64	27.01	106.54			1	0	0	0.01	0.02	0.04		
25	2	0.4	1.75	6.62	27.11	104.32			2	0	0	0.01	0.02	0.04		
26	3	0.48	1.69	6.79	27.05	106.28			3	0	0	0.01	0.02	0.06		
27	4	0.47	1.65	6.76	26.88	104.9			4	0	0	0.01	0.02	0.03		
28	5	0.48	1.74	6.59	26.27	106.21			5	0	0	0.01	0.02	0.05		
29	Average	0.45	1.702	6.68	26.864	105.65			Average	0	0	0.01	0.02	0.044		
30			3.782222	3.9247944	4.021557	3.932772										
31		4.5E-09	4.26E-09	4.175E-09	4.2E-09	4.13E-09										
32																
33																
34	a	1.7424E-08														
35	n^2/2	0.8712	3.4848	13.9392	55.7568	223.0272										
36	n^2/4	0.4356	1.7424	6.9696	27.8784	111.5136										
37																

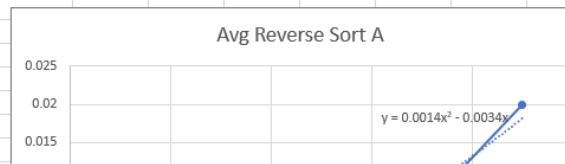


Table 2:

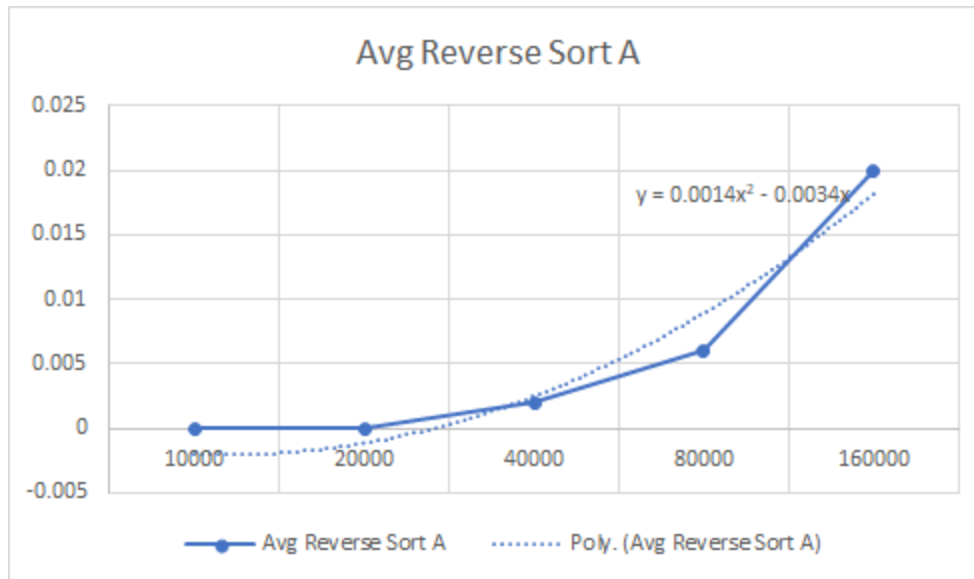


Figure 1:

$n^2/2$	0.8712	3.4848	13.9392	55.7568	223.0272
$n^2/4$	0.4356	1.7424	6.9696	27.8784	111.5136