COMP2521 Sort Detective Lab Report

by Rason, Nathan

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
To ensure that the sorting program's output is sorted, we checked that the output of the sorting algorithm on different inputs of data was sorted. Also, data was sorted by sortA and sortB, and also the linux command “sort” making sure that the output of all sorting algorithms was correct and identical.

Performance analysis
During performance analysis we will use the gen command to generate different amounts of numbers in either ascending, descending or random to attempt to extract any patterns or special features in the execution time of the two sorting algorithms.

Correctness Analysis

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

1. 4,2,56,40,0,2,4,1,1,200,56 -> 0,1,1,2,2,4,4,40,56,56,200 (correct)

2. 500,2,600,0 -> 0,2,500,600 (correct)

3. 0, 0, 1 -> 0,0,1 (correct)

4. 0,0,0,0 -> 0,0,0,0 (Correct)

Procedure in checking sort was correct.

```
# generate some data, put in a file called ""mydata"
$ ./gen 100 R > mydata # Repeated with A, D, R type of data

# count the number of lines in the data (should be 100)
$ wc -l mydata
```

```
# sort the data using sortA, put the result in "sortedA"
$ ./sortA < mydata > sortedA

# sort the data using sortB, put the result in "sortedB"
$ ./sortA < mydata > sortedB

# count the number of lines in "sortedA" (should also be 100)
$ wc -l sortedA

# sort the data using Unix sort
$ sort -n < mydata > sorted

# check that the sortA and sortB programs actaully sorted
$ diff sorted sortedA   Correct Output Showed no diffs
$ diff sorted sortedB   Correct Output Showed no diffs
```

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 will take the average of 100 runs for each of the following test cases:
ascending, descending and random for 1000, 10000, and 100000 numbers.

Using our shell script, we ran the tests 100 times and calculated an average time.

1. Test1 (./gen 1000 A)

sortA: 0.00s
sortB: 0.00s

2. Test2 (./gen 1000 D)

sortA: 0.01s
sortB: 0.00s

3. test3 (./gen 1000 R)

sortA: 0.00s
sortB: 0.00s

4. test4 (./gen 10000 A)

sortA: 0.01s
sortB: 0.01s

5. test5 (./gen 10000 D)

sortA: 0.32, 0.36,0.32,0.30,0.32,0.32,0.30,0.32,0.32,0.33 - >Average: 0.321s
sortB: 0.01,0.00,0.01,0.01,0.00,0.02,0.001,0.001,0.00,0.001 - > Average: 0.01s

6. test6 (./gen 10000 R)

sortA: 0.33,0.33,0.33,0.34,0.33,0.34,0.33,0.33,0.33,0.34 ... - > 0.33s
sortB: 0.01,0.02,0.02,0.01,0.02,0.01,0.01,0.02,0.01,0.02 … - > 0.015s

7. test4 (./gen 100000 A)

sortA: 0.07,0.07,0.05,0.08,0.08,0.06,0.06,0.07,0.08,0.06 … -> 0.068s
sortB: 0.08,0.07,0.10,0.13,0.08,0.05,0.08,0.11,0.10,0.07 … -> 0.087s

8. test5 (./gen 100000 D)

sortA: 24.80,24.67,24.65,25.04,24.92,24.88,24.90,24.92,24.61... -> 22.33s
sortB: 0.09,0.10,0.07,0.08,0.09,0.09,0.08,0.08,0.08,0.09 ... -> 0.085s

9. test6 (./gen 100000 R)

sortA: 34.54,34.92,34.36,34.43,34.48,34.36,34.58,34.84,34.52,34.46 ... -> 34.55s
sortB: 0.19,0.16,0.15,0.18,0.17,0.16,0.14,0.15,0.17,0.17 ... -> 0.164s

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.

Stability
We also investigated the stability of the sorting programs using different sets of data. In a set of numbers, one number would be repeated in random in the the data set with a tag next to it, being its number: FIRST, SECOND, THIRD etc.

This test was done on three different sets of data for 5 runs each set with each run resulting in identical output.
If when sorted, the numbers appeared with the tags in correct order, i.e. FIRST, SECOND etc. then the sorting algorithm maintains the order of the same elements so the algorithm is stable, if not, then the algorithm is unstable.

Stability testing results

Using sortA, The order of the lines with the same number remained the same. Therefore. sortA is **stable**.

Using sortB, The order of the lines with the same number was changed, therefore sortB is **unstable**.

Experimental Results

Correctness Experiments

An example of a test case and the results of that test is ...

4,2,56,40,0,2,4,1,1,200,56 -> 0,1,1,2,2,4,4,40,56,56,200 (correct)

Performance Experiments

For Program A, we observed that the algorithm underlying the program has the following characteristics ...

1. Program A is Stable
2. Possible: Merge, Insertion, Bubble
3. Looking at the Performance Data, sortA performed best in Ascending Ordered Data (Best Case), then Descending Ordered Data (Average) and Random Data (Worst Case).
4. Insertion Sort.

For Program B, we observed that the algorithm underlying the program has the following characteristics ...

1.Program B is NOT Stable
2. Possible: QuickSort, HeapSort, ShellSort
3. Looking at the Performance Data, Ascending Ordered Data (Best Case) and Average (Descending Ordered Data) were completed with similar times but Random Set Data (Worst Case) was completed with a longer time.
4. QuickSort

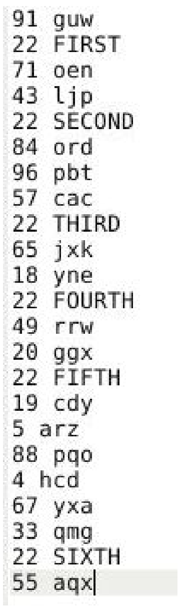
Conclusions
On the basis of our experiments and our analysis above, we believe that:

ProgramA implements the Insertion sorting algorithm
ProgramB implements the QuickSort sorting algorithm

Appendix

Stability testing

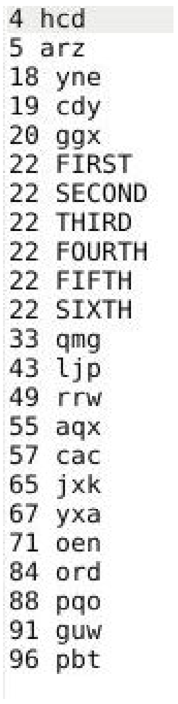
One example of a data set used to test the sorting algorithms is below. The number repeated throughout the set is “22”.



```
91 guw
22 FIRST
71 oen
43 ljp
22 SECOND
84 ord
96 pbt
57 cac
22 THIRD
65 jxk
18 yne
22 FOURTH
49 rrw
20 ggx
22 FIFTH
19 cdy
5 arz
88 pqo
4 hcd
67 yxa
33 qmg
22 SIXTH
55 aqx
```

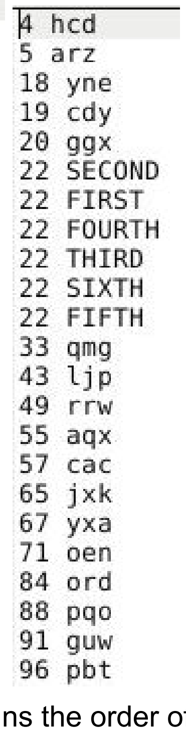
The data is then sorted:

sortA



```
4 hcd
5 arz
18 yne
19 cdy
20 ggx
22 FIRST
22 SECOND
22 THIRD
22 FOURTH
22 FIFTH
22 SIXTH
33 qmg
43 ljp
49 rrw
55 aqx
57 cac
65 jxk
67 yxa
71 oen
84 ord
88 pqo
91 guw
96 pbt
```

sortB



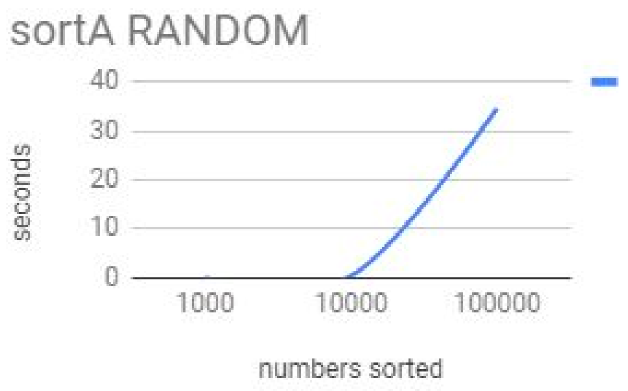
```
4 hcd
5 arz
18 yne
19 cdy
20 ggx
22 SECOND
22 FIRST
22 FOURTH
22 THIRD
22 SIXTH
22 FIFTH
33 qmg
43 ljp
49 rrw
55 aqx
57 cac
65 jxk
67 yxa
71 oen
84 ord
88 pqo
91 guw
96 pbt
```

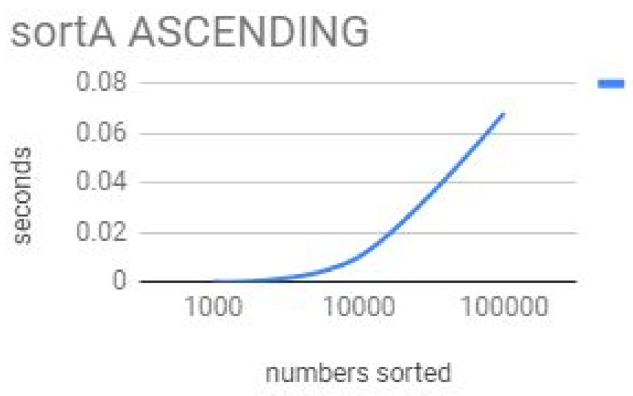
sortA maintains the order of the tags which appear next to “22” whereas sortB does not. Therefore, sortA is stable, sortB is unstable.

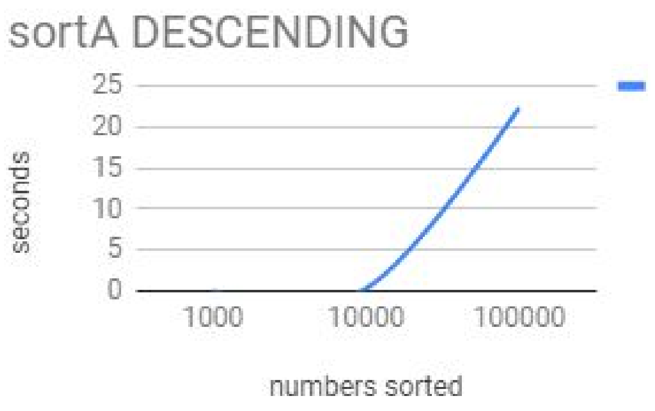
Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)	O(n)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)

Graphs

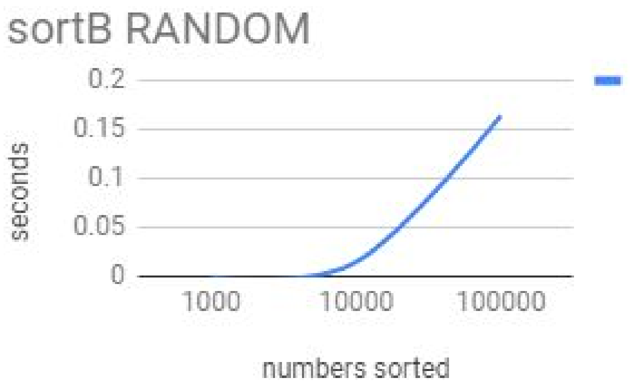
<u>sortA</u>

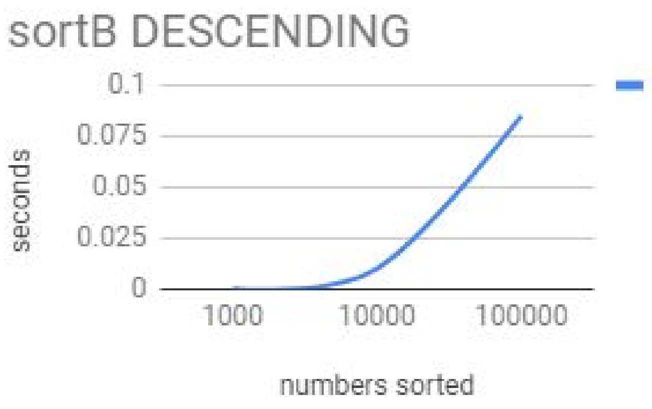


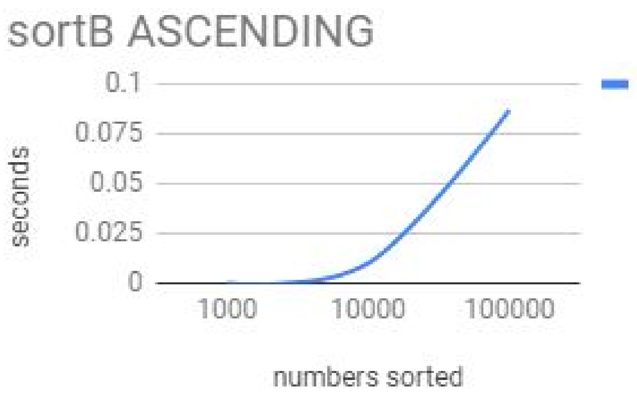



sortA DESCENDING
seconds
25
20
15
10
5
0
1000
10000
100000
numbers sorted

sortB


sortB RANDOM
seconds
0.2
0.15
0.1
0.05
0
1000
10000
100000
numbers sorted


sortB DESCENDING
seconds
0.1
0.075
0.05
0.025
0
1000
10000
100000
numbers sorted


sortB ASCENDING
seconds
0.1
0.075
0.05
0.025
0
1000
10000
100000
numbers sorted