COMP1927 Sort Detective Report

By Shan Wang(z5119666), Mo Li(z5089148)

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

(Sample)

Data1 (ascending order with no repeat keys)	Data2 (ascending order with repeat keys)	Data3 (descending order with no repeat keys)	Data4 (descending order with repeat keys)	Data5 (random order with no repeat keys)	Data6 (random order with repeat keys)
1 ghh	1 rte	90 shd	78 dbd	53 hsd	66 adf
2 hjh	1 jhj	87 fgg	66 has	76 dfs	53 asd
3 hgh	1 ffd	83 fgj	66 ghh	92 jtr	85 gre
5 uio	2 eet	76 prr	66 rfv	74 ert	73 hjj
6 otr	2 tty	71 edf	48 tgb	65 ioo	76 rty
9 hjk	8 qul	47 hkj	45 yhn	12 nnr	66 ety
14 sdf	13 dsf	34 fsf	44 ujm	47 wet	23 jjj
17 dfg	13 yiu	30 uki	35 edc	7 htd	53 wep
20 wwe	13 ghj	23 dgg	34 sdg	18 dfd	34 sdg
22 qwe	15 uuu	19 gjh	34 ert	78 qwe	53 trh
23 sdf	15 rer	14 qwb	34 qwr	57 rtt	75 wdv
28 erw	15 ryy	11 bnm	15 kdl	24 uyt	67 rgh
33 adt	15 ytu	6 gfg	15 jhd	77 edw	53 ygv
55 dfg	15 opp	1 aop	3 dsf	3 egm	3 tfc
90 rww	45 asf	0 rwe	3 sdd	11 ohc	44 edd

We chose these inputs because it covered different conditions, for example, it already sorted in ascending order, and already sorted in ascending order with repeat keys, and in descending order, and in descending order with repeat keys, and in random order. Firstly, we can test if it works. Secondly, the output shows if these programs work properly, such as if these programs sort all numbers correctly and if it drops some numbers and if it gives more numbers.

Expectation

The expectation is the number of these keys is correct, and after sorting, keys should in ascending order.

Performance Analysis

Execution time & Adaptability

Time

In our performance analysis, we measured how each program's execution time varied as the size and initial sortedness of the input varied.

Adaptability

We also investigated the adaptability of the sorting programs by comparing the time used in sorting ascending, descending and random order numbers when the number of keys is equal.

We used given generator 'gen' to generate different amount of numbers in ascending, descending and random order and record execution time.

(Table to record execution time)

Order / #lines	sortA random/s	sortA ascending/s	sortA descending/s	sortB random/s	sortB ascending/s	sortB descending/s
10						
100						
1,000						
5,000						
10,000						
20,000						
50,000						
80,000						
100,000						
999,999						

We used these test cases because we want to find variation between numbers of keys and time. Therefore, we can work out the complexity of each program. And the reason why the maximum number of lines is 999,999 is that there is a limit on the size of the input each program can process (1,000,000 lines).

Because of the way timing works on Unix/Linux, it was necessary to repeat the same test multiple times (which is not shown on the table above).

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.

Expectation

We expect we can draw a line graph to observe and figure out the complexity of each program.

Stability

We also investigated the stability of the sorting programs by the following kinds of input data.

(Sample)

Data1 (Random)	Data 2 (Ascending)	Data 3 (Descending)
6 aaa	1 aaa	9 aaa
4 aaa	1 bbb	9 bbb
17 aaa	1 ccc	9 ccc
8 aaa	2 aaa	8 aaa
6 bbb	2 bbb	8 bbb
32 aaa	2 ccc	8 ccc
56 aaa	2 ddd	7 aaa
7 aaa	2 eee	7 bbb
4 bbb	2 fff	7 ccc
3 aaa	3 aaa	7 ddd
2 aaa	4 aaa	7 eee
42 aaa	5 aaa	6 aaa
5 aaa	5 bbb	5 aaa
6 ccc	5 ccc	4 aaa
8 bbb	6 aaa	4 bbb

We used these test cases because we can check if the sorted data has the same order as the original data when they have the same key. If it is stable then when key is same, ‘aaa’ should appear before ‘bbb’, ‘bbb’ before ‘ccc’ extra.

Expectation

If the program is unstable, there must be at least one example that data after sorting is not like the principle mentioned above. If the program is stable, then we need to do more test to prove it. For instance, all keys are same.

Experimental Results

Correctness Experiments

The output of sortA shows that it sorts correctly. However, sortB is incorrect, it doesn't work if the input data contains more than two repeated keys, for example

Works	Doesn't work
1 sdf 1 hdf 2 jsd 3 kjs	1 sdf 1 hdf 2 jsd 1 kjs

Therefore, we test stability of sortB by change some inputs to no more than 2 repeat numbers. Furthermore, when test execution time, we used 'gen' to generate no repeat numbers.

Performance Experiments

For sortA, we observed that it is unstable. Moreover, the complexity of random order sorting is O(n^2), ascending order is O(n^2), descending order is O(n^2). Therefore, sortA is non-adaptive.

These observations indicate that the algorithm underlying the program is unstable and non-adaptive.

For sortB, we observed that it is unstable. Furthermore, the complexity of random order sorting is O(n), ascending order is O(n^2), descending order is O(n^2). Thus, sortB is adaptive.

These observations indicate that the algorithm underlying the program is unstable and adaptive.

Conclusions

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

- SortA implements the *Oblivious bubble sort* sorting algorithm
- SortB implements the *Quick sort median of three* sorting algorithm

Appendix

Properties of these sort

	complexity		Stable	Adaptive
	Best	Worst		
Oblivious bubble sort	O(n^2)	O(n^2)	N	N
Bubble sort with early exit	O(n)	O(n^2)	Y	Y
Vanilla insertion sort	O(n)	O(n^2)	Y	Y
Insertion sort with binary search	O(n)	O(nlogn)	Y	Y
Vanilla selection sort	O(n^2)	O(n^2)	N	N
Quadratic selection sort	O(n*sqrt(n))	O(n*sqrt(n))	N	N
Merge sort	O(nlogn)	O(nlogn)	Y	N
Vanilla quick sort	O(nlogn)	O(n^2)	N	Can be Y
Quick sort median of three	O(n)	O(n^2)	N	Y
Randomized quick sort	N/A	N/A	N/A	N/A
Shell sort powers of two	O(nlogn)	O(n(logn)^2)	N	Y
Shell sort Sedgewick	N/A	N/A	N/A	N/A
Bogo sort	O(n)	infinity	N	N

Execution time

Order / #lines	sortA random/s	sortA ascending/s	sortA descending/s	sortB random/s	sortB ascending/s	sortB descending/s
10	0	0	0	0	0	0
100	0	0	0	0	0	0
1,000	0	0	0	0	0	0
5,000	0.1	0.04	0.1	0	0.03	0.04
10,000	0.44	0.18	0.45	0.01	0.14	0.14
20,000	1.75	0.8	1.8	0.02	0.55	0.55
50,000	10.8	4.5	11	0.04	3.4	3.4
80,000	28	11.7	28.5	0.06	8.5	8.5
100,000	45.6	18.1	43.7	0.1	13.5	13.5
999,999	N/A	N/A	N/A	1.1	N/A	N/A

Line charts of sortA and sortB