COMP2521 Sort Detective Lab Report

by Faiyam Islam (z5258151), Dharani Palanisamy (z5260276)

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

In our experimental design, to determine which sorting algorithm used for sortA and sortB, it is vital to analyse the properties presented to us:

Table 1: Properties of various sorting algorithms

	Bubble Sort	Insertion Sort	Selection Sort	Merge Sort	Naive Quicksort	Median of-three Quicksort	Randomised Quicksort	Bogosort
Stable?	Yes	Yes	No	Yes	No	No	No	No
Adaptive?	Yes	Yes	No	No	Yes	Yes	Yes	Yes
Time complexity (best case)	O(n)	O(n)	$O(n^2)$	nlog(n)	nlog(n)	nlog(n)	nlog(n)	n - 1
Time complexity (worse case)	$O(n^2)$	$O(n^2)$	$O(n^2)$	nlog(n)	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(\infty)$

We measured how each program's execution time varied by considering duplicate keys in input data as well as varying the size and initial sortedness of the input.

Our input for each sorting algorithm consisted of the following data:

1. Stability check with duplicate numbers and letters being alphabetised
2. Input of 10000 numbers being random, sorted and reversed
3. Input of 20000 numbers being random, sorted and reversed
4. Input of 40000 numbers being random, sorted and reversed
5. Input of 80000 numbers being random, sorted and reversed
6. Input of 120000 numbers being random, sorted and reversed
7. Input of 160000 numbers being random, sorted and reversed

We used these test cases to narrow down possible algorithms at each stage starting with testing stability first.

We will test for stability of the algorithms by checking how the algorithm handles a relatively small input data file (numbered in a random order) with multiple duplicate keys indexed with alphabets to help identify them. A sorted stable algorithm would result in two objects with equal keys to appear in the same order as they appear. For example, 1 to 10 is

sorted in ascending order, however the letters are sorted in the way they appear in the input. According to the above table, this can potentially help us rule out whether algorithms belong within one of two categories listed below.

Stable algorithms: Bubble sort, Insertion sort, Merge sort
OR
Unstable algorithms: Selection sort, Quick sort, Bogosort

Once we have narrowed this, as a next step, we aim to test for the time complexity and adaptivity of both algorithms. We will test this by measuring how each program's execution time is varied as the size and initial "sortedness" of the input is varied. Each of these tests was repeated five times to obtain the average of timing results to account for Unix timing variability. Testing using these inputs will determine the best and worst cases for each algorithm and assist in narrowing down the different sorts. In addition, some sorting algorithms have similar time complexities, so it is imperative to check the raw times. The input sizes and sortendess being tested are as listed at the start.

Experimental Results

Program A

For Program A, through running the stability check, we observe that the algorithm is that the order between duplicates is not maintained in the output as shown for an example below for input row 11: "6 ee". This does not appear first in the output.

Input

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

Output

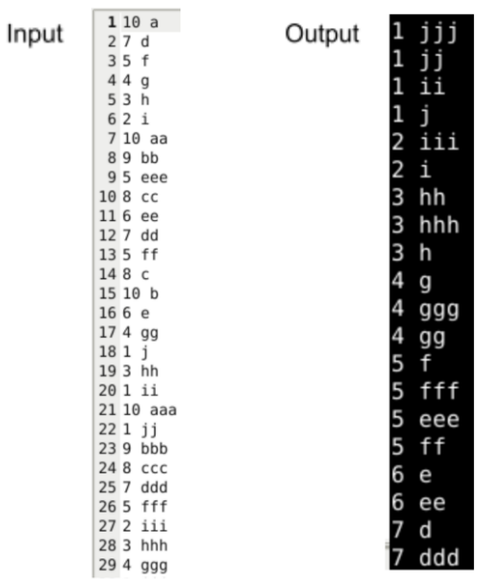
```
1 j
1 ii
1 jj
1 jjj
2 i
2 iii
3 hh
3 h
3 hhh
4 gg
4 ggg
4 g
5 ff
5 f
5 fff
5 eee
6 e
6 ee
7 d
7 ddd
7 dd
8 c
```

These observations indicate that the algorithm underlying the program has the following characteristics. The first test was to check for stability and we concluded that sortA is indeed unstable, leaving us with 5 possible results. Selection sort, naive quicksort,

median-of-three quicksort, randomised quicksort and bogosort. Through analysis of the time complexities we can narrow down the characteristics of the adaptiveness of the sort. In table 2, the time complexity differences between randomised, sorted and reversed showcase a small difference, thus indicating that this program is not adaptive. Chart 1 displays a graph which is more resembling O(n^2), thus ruling out naive quicksort, median-of-three quicksort, randomised quicksort and bogosort. Thus, from close inspection of stability and time complexities, we confirm that the sort is indeed a selection sort.

Program B

For Program B, through running the stability check, we observe that the algorithm is that the order between duplicates is not maintained in the output as shown for an example below for input row 18 : “1 j”.



As evidenced by the line charts in the appendix, we observe that there is a stark contrast in running times between sortA and sortB. Furthermore, sortB there is a clear contrast between timing results of sorted inputs and randomised inputs, with sorted inputs having a lower runtime compared to randomised inputs.

These observations indicate that the algorithm underlying the program is unstable, is adaptive and has average time complexity O(nlogn). Therefore, it is likely to be a quicksort algorithm.

To determine the type of quicksort, we can consider the following arguments.

The algorithm cannot be a naive quicksort since it chooses the left-most element as the pivot element, in which case a sorted input list would take the longest time to process. This is not what we observe (Appendix: Table 3) since randomised inputs show the highest timing results for a high enough input size where difference in timings can be observed.

The algorithm is likely to be median-of-three quicksort as it chooses the first, middle and last elements as the pivot. It can be seen that runtimes (Appendix: Table 3) for a sorted and reverse input are more or less similar, enabling the median of the left and right to be the exact same in both cases. For this reason, we can also deduce that randomised quicksort is slightly more unlikely as it would follow with more inconsistent timing results for any given input size.

Conclusions

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

- sortA implements the *Selection Sort* sorting algorithm
- sortB implements the *Median-of-three quicksort* sorting algorithm

Appendix

Table 2: Time complexity averages for sortA

Note: The time complexities are all to 3 decimal places

Input Size	Random	Reverse	Sorted
10000	0.166	0.150	0.164
20000	0.654	0.618	0.648
40000	2.572	2.474	2.580
80000	10.242	9.840	10.254
120000	22.996	22.116	23.036
160000	40.904	39.274	40.832

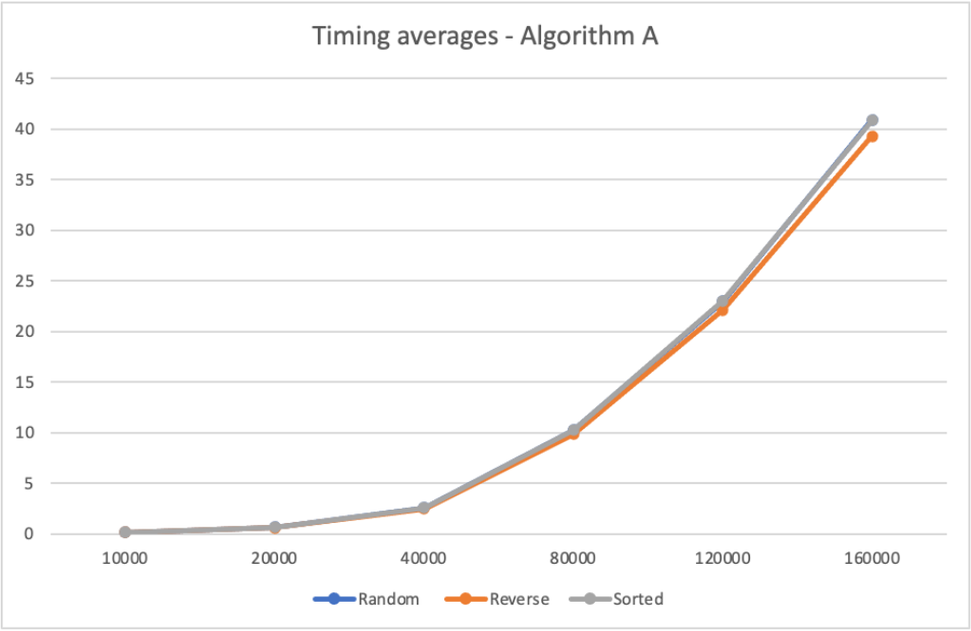
Table 3: Time complexity averages for sortB

Note: The time complexities are all to 3 decimal places

Input Size	Random	Reverse	Sorted
10000	0.000	0.000	0.000
20000	0.000	0.000	0.000
40000	0.010	0.000	0.000
80000	0.020	0.010	0.010
120000	0.030	0.010	0.010

160000	0.040	0.020	0.020
1,000,000 (distinguishing between quicksort)	0.38	0.14	0.13

Chart 1: Time complexities of sortA



(Blue random line is not visible because it is almost the same as reverse and sorted, close to overlap)

Chart 2: Time complexities of sortB