COMP1927 Sort Detective Lab Report

by Brian Lam, Matthew Van Schellebeek

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.

1.0 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

1.1 Correctness Analysis

Summary

For the SortA and SortB programs to be correct, they must produce required results for valid inputs. The required results mean that both programs must produce a sorted output, from using valid inputs.

For the correctness test, we used two different input generator programs.

- Program 1 = "Gen"
  - This was the given lab program that could reverse, sorted and random sets of numeric data.
  - The program only produced unique keys.
- Program 2 = "randList"
  - This is a C program that generates ONLY random values.
  - The program could potentially produce duplicates, especially as input size gets larger.

Using both input generator programs was important, as we needed to test against datasets with both unique and duplicate values. By doing this, we could discover more specific information on the stability / instability of the SortA and SortB programs, thus help us determine which specific sorting algorithms were used.

With the **"Gen" program**, we only created sets of data of only **100 input size** as the input generator only creates unique values. The inputs were also produced in three different forms to cover as many cases as possible for sort correctness:

- REVERSE: Descending order
- SORTED: Ascending order
- RANDOM: Random order

However, with the **"RandList program**, we created a set of data of **10,000 input size**, so that we may possibly produce duplicates. This would help test the stability of SortA and SortB. Unlike the "Gen" input generator, we could not produce sorted or reverse datasets, only a random order dataset.

For the final check for correctness, we used **Unix Sort** as the comparison tool with SortA and SortB programs so that we could see if there were any differences in output.

Correctness Analysis Method

For the test using “Gen” we:

1. Generated
   1 x set of REVERSE data, of **size 100**, using “Gen”
   1 x set of SORTED data, of **size 100**, using “Gen”
   1 x set of RANDOM data, all of **size 100**, using “Gen”
   - Put data into a file called *reverseGen, sortedGen* and *randomGen* respectively.
   - Count the number of lines in the data. It should = 100.
2. Put reverseGen data through **SortA, SortB** and **Unix Sort**
   - Put result into *sortedA1*, *sortedB1, sortedU1* respectively.
   - Count the number of lines in data for each file. It should = 100.
3. Put sortedGen data through **SortA, SortB** and **Unix Sort**
   - Put result into *sortedA2*, *sortedB2, sortedU2* respectively.
   - Count the number of lines in data for each file. It should = 100.
4. Put randomGen data through **SortA, SortB** and **Unix Sort**
   - Put result into *sortedA3*, *sortedB3, sortedU3* respectively.
   - Count the number of lines in data for each file. It should = 100.
5. Using Shell’s **diff** utility, we checked the difference between:
   - Both *sortedA1 and sortedB1* against *sortedU1* (from reverseGen input)
   - Both *sortedA2 and sortedB2* against *sortedU2* (from sortedGen input)
   - Both *sortedA3 and sortedB3* against *sortedU3* (from randomGen input)
   - The expected output should be nothing if the programs have sorted correctly.

For the test using “RandList” we:

1. Generated:
   1 x set of RANDOM data, all of **size 10,000** using “randList”
   - Put data into a file called *randomRan*.
   - Count the number of lines in the data. It should = 100.
2. Put randomRan data through **SortA, SortB** and **Unix Sort**
   - Put result into *sortedA*, *sortedB, sortedU* respectively.
   - Count the number of lines in data for each file. It should = 100.
3. Using Shell’s **diff** utility, we checked the difference between:
   - S*ortedA* against *sortedU* (from randomRan input)
   - S*ortedB* against *sortedU* (from randomRan input)
   - The expected output should be nothing if the programs have sorted correctly.

1.2 Performance Analysis

Summary

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

To test execution time, we needed to generate larger data sets than the original sets we created of size 100 for comparison. We created datasets of size:

- **1,000 / 5,000 / 10,000 for SortA.**
- **5M / 10M / 20M for SortB** as it was difficult to gather timing data for the program due the speed it executes for smaller sizes (e.g. the input sizes used for SortA)

Again, this was done with both "Gen" and "RandList" programs as we needed to investigate SortA and SortB's stability, since one dataset would have unique keys and another dataset have duplicates.

Similar to the Correctness Analysis, we also used a mix of REVERSE, SORTED and RANDOM data, as this helps match the performance of SortA / SortB programs with the characteristics of the 11 possible algorithms since they perform differently depending on the input type.

Because of the way timing works on Unix/Linux (by sampling), we decided to repeat each timing run at least 5 times to get an average figure. This helps us produce more accurate performance results for use.

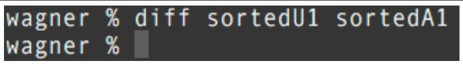
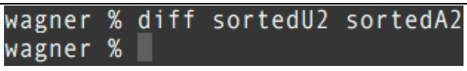
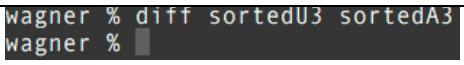
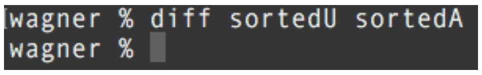
Performance Analysis Method:

1. Ran the **SortA** program 5 times for each REVERSE / SORTED / RANDOM inputs produced from "Gen" to get execution timing data.
    - We collected the following:
        - 5 x sets of execution timing data of REVERSE type.
        - 5 x sets of execution timing data of SORTED type.
        - 5 x sets of execution timing data of RANDOM type.
    - We took an average of the 5 x sets for each type and tabulated the data.
2. Repeat step **#1** for input sizes **5,000 /10,000** for **SortA**.
3. Ran the **SortB** program 5 times for each REVERSE / SORTED / RANDOM inputs produced from "Gen" to get execution timing data.
    - We collected the following:
        - 5 x sets of execution timing data of REVERSE type.
        - 5 x sets of execution timing data of SORTED type.
        - 5 x sets of execution timing data of RANDOM type.
    - We took an average of the 5 x sets for each type and tabulated the data.
4. Repeat step **#3** for input sizes **5M / 10M / 20M** for **SortB**.
5. Repeated the above steps for "RandList" inputs.

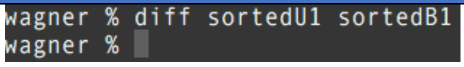
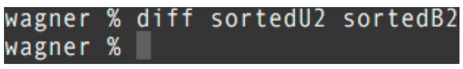
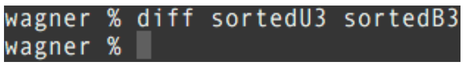
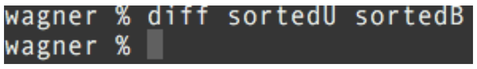
2.0 Experimental Results

2.1 Correctness Experiments

SortA results

Input Generator	Type	Line Count (wc –l)	Output Differences? (Yes/ No)
Gen	Reverse	100	NO  `wagner % diff sortedU1 sortedA1` `wagner %`
Gen	Sorted	100	NO  `wagner % diff sortedU2 sortedA2` `wagner %`
Gen	Random	100	NO  `wagner % diff sortedU3 sortedA3` `wagner %`
RandList	Random	10,000	NO  `wagner % diff sortedU sortedA` `wagner %`

SortB results

Input Generator	Type	Line Count (wc –l)	Output Differences? (Yes/ No)
Gen	Reverse	100	NO  `wagner % diff sortedU1 sortedB1` `wagner %`
Gen	Sorted	100	NO  `wagner % diff sortedU2 sortedB2` `wagner %`
Gen	Random	100	NO  `wagner % diff sortedU3 sortedB3` `wagner %`
RandList	Random	10,000	NO  `wagner % diff sortedU sortedB` `wagner %`

From our observations, both SortA and SortB results produced no output when compared with the Unix Sort results, regardless of the input generator used.

As a result, we can assume that both SortA and SortB are using stable algorithms. We can also safely state that both SortA and SortB are producing the required results, thus are correct.

2.2 Performance Experiments

<u>SortA Observations</u>

For Program SortA, we observed that for:

- REVERSE type input:
    - For input size of 1000, average execution time was **0.547s**[Table 1].
    - For input size of 5000, average execution time was **52s**[Table 1].
    - For input size of 10000, average execution time was **6m45s**[Table 1].
- SORTED type input, execution timing stayed constant as input size increased.
    - For each input size, timing stayed constant at about **<0.01s**[Table 1].
- RANDOM type input:
    - For input size of 1000, average execution time was **0.148s**[Table 1].
    - For input size of 5000, average execution time was **7s**[Table 1].
    - For input size of 10000, average execution time was **45s**[Table 1].

These observations indicate that the algorithm underlying SortA has the following characteristics:

- Against REVERSE type input:
    - Cost complexity of the SortA algorithm used seems to be of **nLogn**, as growth in cost slows down as the size increases.
- Against SORTED type input:
    - Cost complexity of the SortA algorithm used seems to be of **O(1)**, as execution time remained constant regardless of the size.
- Against RANDOM type input:
    - Complexity of the SortA algorithm used seems to be of **nLogn**, as growth in cost slows down as the size increases.

SortB Observations

For Program SortB, we observed that for:

- REVERSE type input, execution timing stayed constant as input size increased.
  - For input size of 5 Million, average execution time was **0.478s**[Table 2].
  - For input size of 10 Million, average execution time was **0.451s**[Table 2].
  - For input size of 20 Million, average execution time was **0.458s**[Table 2].
- SORTED type input, execution timing stayed constant as input size increased.
  - For input size of 5 Million, average execution time was **0.434s**[Table 2].
  - For input size of 10 Million, average execution time was **0.457s**[Table 2].
  - For input size of 20 Million, average execution time was **0.446s**[Table 2].
- RANDOM type input, execution timing stayed constant as input size increased.
  - For input size of 5 Million, average execution time was **0.556s**[Table 2].
  - For input size of 10 Million, average execution time was **0.552s**[Table 2].
  - For input size of 20 Million, average execution time was **0.533s**[Table 2].

These observations indicate that the algorithm underlying SortA has the following characteristics:

- Against REVERSE type input:
  - Cost complexity of the SortA algorithm used seems to be of **O(1)**, as execution time remained constant regardless of the size.
  - However, this is likely because the input size is not large enough
    (as absolute execution time is < 1 second)
- Against SORTED type input:
  - Cost complexity of the SortA algorithm used seems to be of **O(1)**, as execution time remained constant regardless of the size.
  - However, this is likely because the input size is not large enough
    (as absolute execution time is < 1 second)
- Against RANDOM type input:
  - Cost complexity of the SortA algorithm used seems to be of **O(1)**, as execution time remained constant regardless of the size.
  - However, this is likely because the input size is not large enough
    (as absolute execution time is < 1 second)

Conclusions

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

- SortA implements the **Insertion Sort** algorithm
- SortB implements the **Merge Sort** algorithm.

We chose the specific algorithm for SortA because:

- In terms of absolute execution time, the algorithm performed slower for REVERSE and RANDOM input types relative to the SORTED input type.
    - This is because with sorted input, an Insertion Sort algorithm only needs to traverse the list once, since every element after the previous would be larger.
- Also, Insertion Sort's AVERAGE and WORST case complexity are of O(nLogn) complexity (if the list is unsorted i.e. random or reverse order) and the BEST case is of O(n) complexity (i.e. a sorted list)

We chose the specific algorithm for SortB because:

- The algorithm's absolute execution time is very similar regardless of whether the input was REVERSE, SORTED or RANDOM, due to the "divide and conquer" technique / characteristics of the Merge Sort Algorithm.
- Also, Merge Sort's BEST, AVERAGE and WORST case complexity are equivalent, of O(nLogn) cost complexity.

However, some critique for our SortB experiment is that:

- The input sizes used were not large enough since they only recorded execution times of < 1 second, thus insignificant to determine the characteristics of the algorithm.
- If we tried to generate larger input sizes using "Gen" or "RandList", the file sizes become to large / they take too long to generate any size more than 20,000,000 numbers.

Appendix

Table 1: SortA performance results

Dataset	Type	Avg Execution Time (n = 1,000)	Avg Execution Time (n = 5,000)	Avg Execution Time (n = 10,000)
Gen	Reverse	Run 1: 0.492s Run 2: 0.552s Run 3: 0.576s Run 4: 0.568s Run 5: 0.548s **Rounded Ave = 0.547s**	Run 1: 52.287s Run 2: 54.139s Run 3: 50. 812s Run 4: 52.874s Run 5: 48.312s **Rounded Ave = 52s**	Run 1: 6m51.158s Run 2: 6m48.390s Run 3: 6m35.688s Run 4: 6m54.204s Run 5: 6m40.382s **Rounded Ave = 6m46s**
Gen	Sorted	Run 1: 0.000s Run 2: 0.004s Run 3: 0.000s Run 4: 0.000s Run 5: 0.000s **Rounded Ave = <0.01s**	Run 1: 0.000s Run 2: 0.004s Run 3: 0.000s Run 4: 0.004s Run 5: 0.000s **Rounded Ave = <0.01s**	Run 1: 0.004s Run 2: 0.004s Run 3: 0.000s Run 4: 0.004s Run 5: 0.000s **Rounded Ave = <0.01s**
Gen	Random	Run 1: 0.148s Run 2: 0.144s Run 3: 0.148s Run 4: 0.152s Run 5: 0.148s **Rounded Ave = 0.148s**	Run 1: 6.512s Run 2: 6.376s Run 3: 8.589s Run 4: 6.588s Run 5: 7.016s **Rounded Ave = 7s**	Run 1: 46.39s Run 2: 52.032s Run 3: 41.696s Run 4: 40.468s Run 5: 43.312s **Rounded Ave = 45s**
RandList	Random	Run 1: 0.184s Run 2: 0.200s Run 3: 0.188s Run 4: 0.148s Run 5: 0.148s **Rounded Ave = 0.174s**	Run 1: 7.684s Run 2: 7.780s Run 3: 8.673s Run 4: 7.864s Run 5: 8.097s **Rounded Ave = 8.012s**	Run 1: 53.351s Run 2: 1m2.72s Run 3: 58.158s Run 4: 55.388s Run 5: 53.122s **Rounded Ave = 57s**

Table 2: SortB performance results

Dataset	Type	Avg Execution Time (n = 5,000,000)	Avg Execution Time (n = 10,000,000)	Avg Execution Time (n = 20,000,000)
Gen	Reverse	Run 1: 0.560s Run 2: 0.472s Run 3: 0.448s Run 4: 0.468s Run 5: 0.440s **Rounded Ave = 0.478s**	Run 1: 0.452s Run 2: 0.472s Run 3: 0.436s Run 4: 0.444s Run 5: 0.452s **Rounded Ave = 0.451s**	Run 1: 0.460s Run 2: 0.468s Run 3: 0.472s Run 4: 0.436s Run 5: 0.456s **Rounded Ave = 0.458s**
Gen	Sorted	Run 1: 0.440s Run 2: 0.420s Run 3: 0.428s Run 4: 0.436s Run 5: 0.448s **Rounded Ave = 0.434s**	Run 1: 0.444s Run 2: 0.460s Run 3: 0.482s Run 4: 0.448s Run 5: 0.452s **Rounded Ave = 0.457s**	Run 1: 0.476s Run 2: 0.412s Run 3: 0.440s Run 4: 0.436s Run 5: 0.468s **Rounded Ave = 0.446s**
Gen	Random	Run 1: 0.528s Run 2: 0.564s Run 3: 0.572s Run 4: 0.556s	Run 1: 0.552s Run 2: 0.592s Run 3: 0.452s Run 4: 0.596s	Run 1: 0.572s Run 2: 0.560s Run 3: 0.932s Run 4: 0.628s

		Run 5: 0.560s **Rounded Ave = 0.556s**	Run 5: 0.572s **Rounded Ave = 0.552s**	Run 5: 0.604s **Rounded Ave = 0.533s**
RandList	Random	Run 1: 37.92s Run 2: s Run 3: 0.580s Run 4: 0.560s Run 5: 0.560s **Rounded Ave = 0.565s**	Not feasible. File size too large.	Not feasible. File size too large.