**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.

**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**

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.

For 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, for 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.

1. 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.
2. 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.
3. 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.
4. 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.

1. 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.

**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 did this by following **steps #1** and **#2** in the Correctness Analysis, except for input sizes of 1000, 5000 and 20,000. 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. It was necessary to run the timing test multiple times, so that we could get an average figure.
   * 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. 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.
3. Repeat steps **#1** and **#2** using input generated from “randList”.
4. Repeat steps **#1**, **#2** and **#3** for input sizes **1,000** and **10,000**.

**Experimental Results**

**Correctness Experiments**

SortA results

|  |  |  |  |
| --- | --- | --- | --- |
| **Input Generator** | **Type** | **Line Count (wc –l)** | **Output Differences? (Yes/ No)** |
| **Gen** | Reverse | 100 | NO |
| **Gen** | Sorted | 100 | NO |
| **Gen** | Random | 100 | NO |
| **RandList** | Random | 10,000 | NO |

SortB results

|  |  |  |  |
| --- | --- | --- | --- |
| **Input Generator** | **Type** | **Line Count (wc –l)** | **Output Differences? (Yes/ No)** |
| **Gen** | Reverse | 100 | NO |
| **Gen** | Sorted | 100 | NO |
| **Gen** | Random | 100 | NO |
| **RandList** | Random | 10,000 | NO |

From our observations as outlined in the tables above, both SortA and SortB results produced no output when compared with the Unix Sort results, regardless of the input generator used. We can safely state that both SortA and SortB are producing the required results, thus are correct.

**Performance Experiments**

SortA 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:  Run 2: Run 3: Run 4: Run 5: **Rounded Ave =** | Run 1:  Run 2: Run 3: Run 4: Run 5: **Rounded Ave =** | Run 1:  Run 2: Run 3: Run 4: Run 5: **Rounded Ave =** |

SortB 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: 0.000s Run 2: 0.000s Run 3: 0.000s Run 4: 0.004s Run 5: 0.000s **Rounded Ave = <0.01s** | Run 1: 0.000s Run 2: 0.000s Run 3: 0.004s Run 4: 0.004s Run 5: 0.000s **Rounded Ave = <0.01s** | Run 1: 0.000s Run 2: 0.000s Run 3: 0.004s Run 4: 0.004s Run 5: 0.004s **Rounded Ave = <0.01s** |
| **Gen** | Sorted | Run 1: 0.000s Run 2: 0.000s Run 3: 0.000s Run 4: 0.000s Run 5: 0.000s **Rounded Ave = 0s** | Run 1: 0.004s Run 2: 0.000s Run 3: 0.000s Run 4: 0.000s Run 5: 0.004s **Rounded Ave = <0.01s** | Run 1: 0.000s Run 2: 0.000s Run 3: 0.004s Run 4: 0.004s Run 5: 0.008s **Rounded Ave = <0.01s** |
| **Gen** | Random | Run 1: 0.000s Run 2: 0.000s Run 3: 0.000s Run 4: 0.000s Run 5: 0.000s **Rounded Ave = 0s** | Run 1: 0.000s Run 2: 0.000s Run 3: 0.000s Run 4: 0.000s Run 5: 0.000s **Rounded Ave = 0s** | Run 1: 0.012s Run 2: 0.004s Run 3: 0.012s Run 4: 0.008s Run 5: 0.004s **Rounded Ave = <0.01s** |
| **RandList** | Random | Run 1: 46.39s Run 2: Run 3: Run 4: Run 5: **Rounded Ave =** | Run 1: 46.39s Run 2: Run 3: Run 4: Run 5: **Rounded Ave =** | Run 1: 46.39s Run 2: Run 3: Run 4: Run 5: **Rounded Ave =** |

For Program A, we observed that ...

These observations indicate that the algorithm underlying the program ... *has the following characteristics* ...

For Program B, we observed that ...

These observations indicate that the algorithm underlying the program ... *has the following characteristics* ...

**Conclusions**

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

* ProgramA implements the *uvw* sorting algorithm

This is because

* ProgramB implements the *xyz* sorting algorithm

**Appendix**

*Any large tables of data that you want to present ...*