**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**

Introduction

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 random values.
  + This program did not have an option to created reverse or sorted outputs, so we had to re-arrange the lists into reverse or sorted before utilising them with SortA and SortB.
  + The program produced duplicates.

Using both programs to create our set of testing data 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.

To test for correctness, we also created and used sets of data of only 100 input size. We thought that a larger input size was not necessary, as we were not testing for performance. The inputs were also produced in three different forms: REVERSE / SORTED / RANDOM data. This was done to cover as many cases as possible for sort correctness.

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. Ran 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. Ran 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. Ran 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 REVERSE** data, of **size 100**, using “randList”  
**1 x set of SORTED** data, of **size 100**, using “randList”  
**1 x set of RANDOM** data, all of **size 100**, using “randList”

* + Put data into a file called *reverseRan, sortedRan* and *randomRan* respectively.
  + Count the number of lines in the data. It should = 100.

1. Repeated steps **#2**, **#3**. **#4** and **#5** in the above steps for “Gen”, except using randList data.
   * Made sure there were no differences in output for reverse, sorted and random data.

**Performance Analysis**

Introduction

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 1,000 and 10,000. 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**

|  |  |  |  |
| --- | --- | --- | --- |
| **Dataset** | **Type** | **Line Count (wc –l)** | **Output Differences? (Yes/ No)** |
| Gen | Reverse |  |  |
| Gen | Sorted |  |  |
| Gen | Random |  |  |
| RandList | Reverse |  |  |
| RandList | Sorted |  |  |
| RandList | Random |  |  |

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

On all of our test cases, ...

**Performance Experiments**

SortA results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Dataset** | **Type** | **Avg Execution Time (n = 100)** | **Avg Execution Time (n = 1,000)** | **Avg Execution Time (n = 10,000)** |
| Gen | Reverse |  |  |  |
| Gen | Sorted |  |  |  |
| Gen | Random |  |  |  |
| RandList | Reverse |  |  |  |
| RandList | Sorted |  |  |  |
| RandList | Random |  |  |  |

SortB results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Dataset** | **Type** | **Avg Execution Time (n = 100)** | **Avg Execution Time (n = 1,000)** | **Avg Execution Time (n = 10,000)** |
| Gen | Reverse |  |  |  |
| Gen | Sorted |  |  |  |
| Gen | Random |  |  |  |
| RandList | Reverse |  |  |  |
| RandList | Sorted |  |  |  |
| RandList | Random |  |  |  |

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
* ProgramB implements the *xyz* sorting algorithm

**Appendix**

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