**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**

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.

The inputs that we generated for the correctness test was of size 100, as we only needed to test for correctness (as opposed to performance) and an unnecessarily large input size would increase our execution time. The inputs will be produced in three different forms: REVERSE / SORTED / RANDOM data. This is done to test that the two programs sort correctly for the three different types of data.

To test for correctness, we:

1. Generated 1 x set of REVERSE data, of size 100, using the “Gen” program.
   * Put data into a file called *reverseGen*
   * Count the number of lines in the data. It should = 100.
2. Generated 1 x set of SORTED data, of size 100, using the “Gen” program.
   * Put data into a file called *sortedGen*
   * Count the number of lines in the data. It should = 100.
3. Generated 1 x set of RANDOM data, of size 100, using the “Gen” program.
   * Put data into a file called *randomGen*
   * Count the number of lines in the data. It should = 100.
4. Ran reverseGen data through **SortA**
   * Put result into *sortedA*
   * Count the number of lines in data. It should = 100.
5. Ran reverseGen data through **SortB**
   * Put result into *sortedB*
   * Count the number of lines of data. It should = 100.
6. Ran reverseGen data through **Shell Sort**
   * Put result into *sortedSH*
   * Count the number of lines of data. It should = 100;
7. Checked that SortA, SortB and Shell Sort produce the same output using diff.
   * The expected output should be nothing, if there is no difference between output from the three sorting programs.
8. Repeat steps **4, 5, 6 and 7** for *sortedGen* and *randomGen* data.

**Performance Analysis**

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

This analysis will help prove that certain algorithm pre-conditions (from the 11 possible algorithms) match the post-conditions from the test results, thus let us determine which algorithms are used in SortA and SortB programs.

1. To set up the analysis experiment by tabulating the various possible sort programs used and list their known characteristics against three types of input: *Reverse, Sorted* and *Random data*.
2. Generate 3 x sets of input data by using the “Gen” program given of size 100.
   * 1 set of REVERSE
   * 1 set of SORTED
   * 1 set of RANDOM
3. Repeat procedure #2, with size 1,000 and then size 10,000.
   * We should have in total 6 sets of input data.
4. Produce 3 x sets of our own data with duplicates (as “Gen” only provides a unique set of keys.)
   * 1 set of REVERSE
   * 1 set of SORTED
   * 1 set of RANDOM
5. Run the **SortA** program 5 times with each REVERSE / SORTED / RANDOM inputs produced from “Gen” to get execution timing data.
   * We should now have:
     + 5 x execution timing data for REVERSE. Take an average of the figures.
     + 5 x execution timing data for SORTED. Take an average of the figures.
     + 5 x execution timing data for RANDOM. Take an average of the figures.
   * You should now have 3 averages, each for reverse / sorted / random tests.
   * Record this average into the table.
6. Run the **SortA** program 5 times with input from our own set to get execution timing data.
   * Take an average of the timing data.
   * Record this average into the table.
7. Run the **SortB** program 5 times with input from “Gen” to get timing data.
   * Take an average of the timing data.
   * Record this average into the table.
8. Run the **SortB** program 5 times with input from our own set to get timing data.
   * Take an average of the timing data.
   * Record this average into the table.

We used the following kinds of input ...

We used these test cases because ...

Because of the way timing works on Unix/Linux, it was necessary to repeat the same test multiple times ...

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.

We also investigated the stability of the sorting programs by ...

We also investigated ... *any other relevant properties* ...

**Experimental Results**

**Correctness Experiments**

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

On all of our test cases, ...

**Performance Experiments**

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 ...*