COMP26120 Lab 2 Report

Rui Xu
9th, December 2022

THERE ARE 3 EXPERIMENTS IN THIS REPORT.

1 Experiment I: Inserting Performance
Implementation of insert for Hash Set

1.1 Hypothesis

The hypothesis is the expected complexity of the hash-set: Theoretically, the average case for insertion operation is O(n) if the query file is empty when the input is randomly chosen, and the expected complexity of the single insert operation is O(1). The running time of the whole program increases linearly when the dictionary size increases, but the average running time of the insert operation is constant.

1.2 Design

To test the hypothesis, 5 random input dictionaries of sizes 100K, 200K, 300K, 400K, 500K were produced. In order to reduce the probability that some random generated dictionaries have non-typical property of the average case, This experiment generates 5 for each so it can get an average running time for each size.

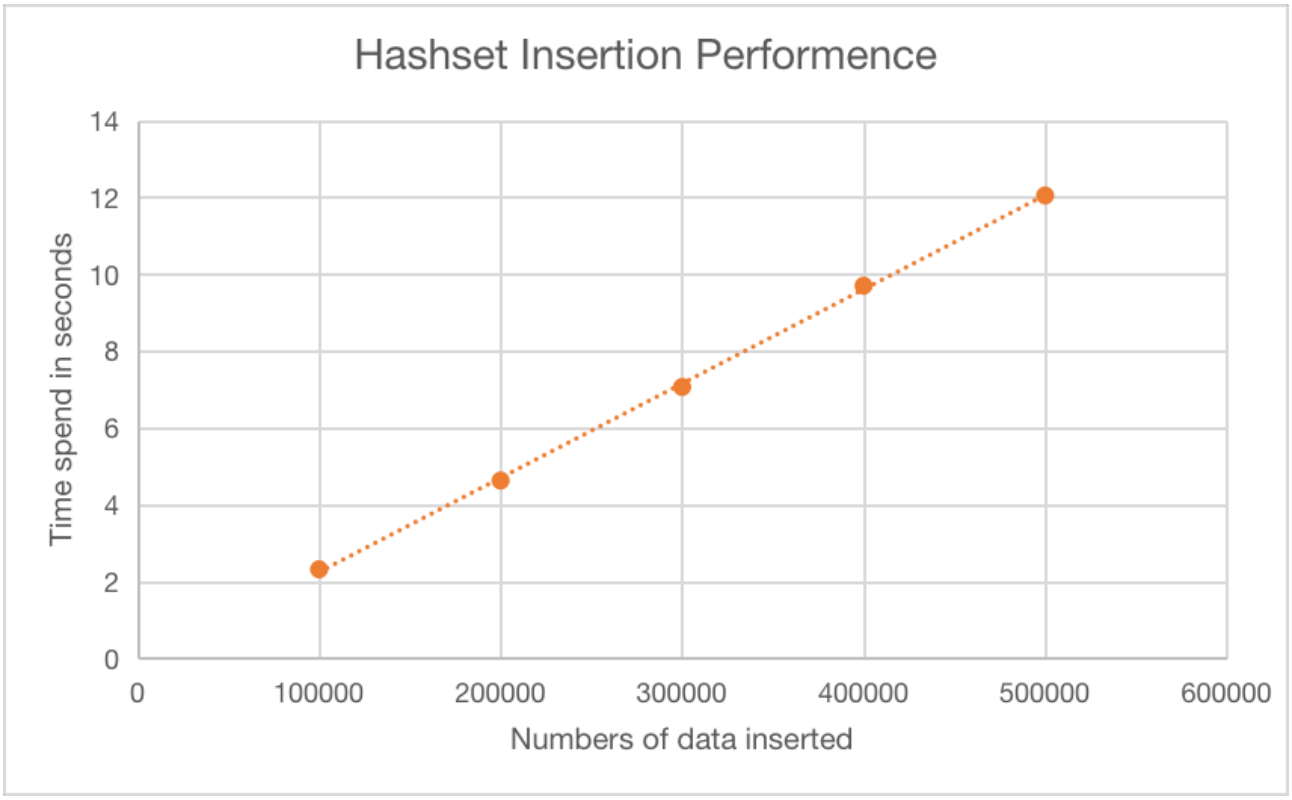
The process of generating dictionaries input was automated using the shell scripts shown in Appendix A. The script for computing the time shows in Appendix B.

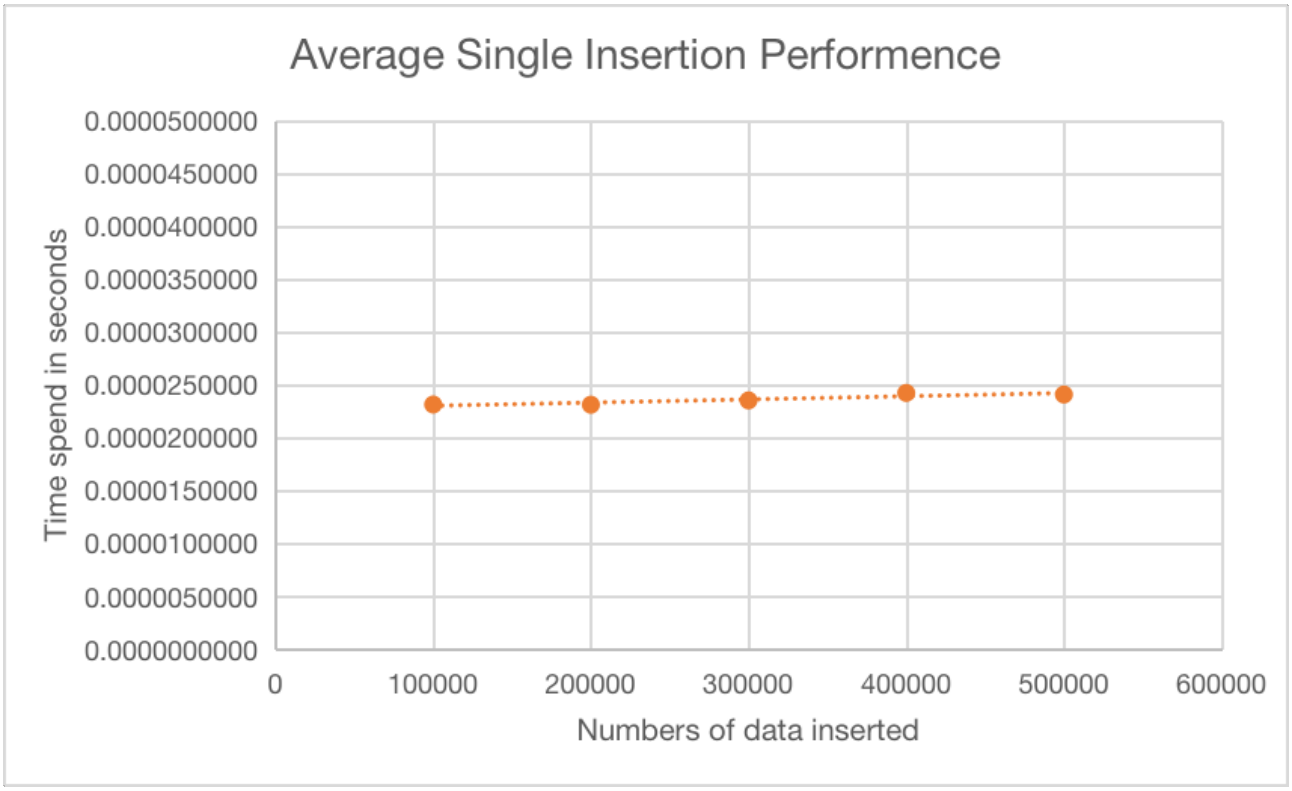
The time for the program to execute was measured using the UNIX time command summing the user and sys values output by the command.

1.3 Results

The results were then plotted using “matplotlib” to fit a line $f(x) = m \times x + q$ calculating values for m and q in the process. The results are shown in Figure 1 and the raw data can be seen in Appendix E. As can be seen from the graph the the line $f(x) = mx+q$ has a good fit and the values of 0.000024546 and 0.050390325 have been computed for m and q respectively. This confirms the hypothesis about the behavior of the hash set on average input, and allows us to predict the time taken to sort a dictionary of size x as 0.000024546 + 0.050390325.

Figure1


Hashset Insertion Performence
Time spend in seconds
14
12
10
8
6
4
2
0
0
100000
200000
300000
400000
500000
600000
Numbers of data inserted


Average Single Insertion Performence
Time spend in seconds
0.0000500000
0.0000450000
0.0000400000
0.0000350000
0.0000300000
0.0000250000
0.0000200000
0.0000150000
0.0000100000
0.0000050000
0.0000000000
0
100000
200000
300000
400000
500000
600000
Numbers of data inserted

1.4 Discussion

The results of the experimental output are basically consistent with the expected hypothesis.

The average case for insertion operation is O(n) if the query file is empty when the input is randomly chosen, and the expected complexity of the single insert operation is O(1).

2 Experiment II: Inserting Performance
Implementation of insert for Binary Search Tree (BSTree)

2.1 Hypothesis

The hypothesis is the expected complexity of the BSTree: The behavior for insertion on binary tree is somewhere between the theoretical average case O(log(n)) and the theoretical worst case O(n).

2.2 Design

To test the hypothesis, 5 random input dictionaries of sizes 100K, 200K, 300K, 400K, 500K were produced. In order to reduce the probability that some random generated dictionaries have non-typical property of the average case, This experiment generates 5 for each so it can get an average running time for each size.

The process of generating dictionaries input was automated using the shell scripts shown in Appendix A. The script for computing the time shows in Appendix C.

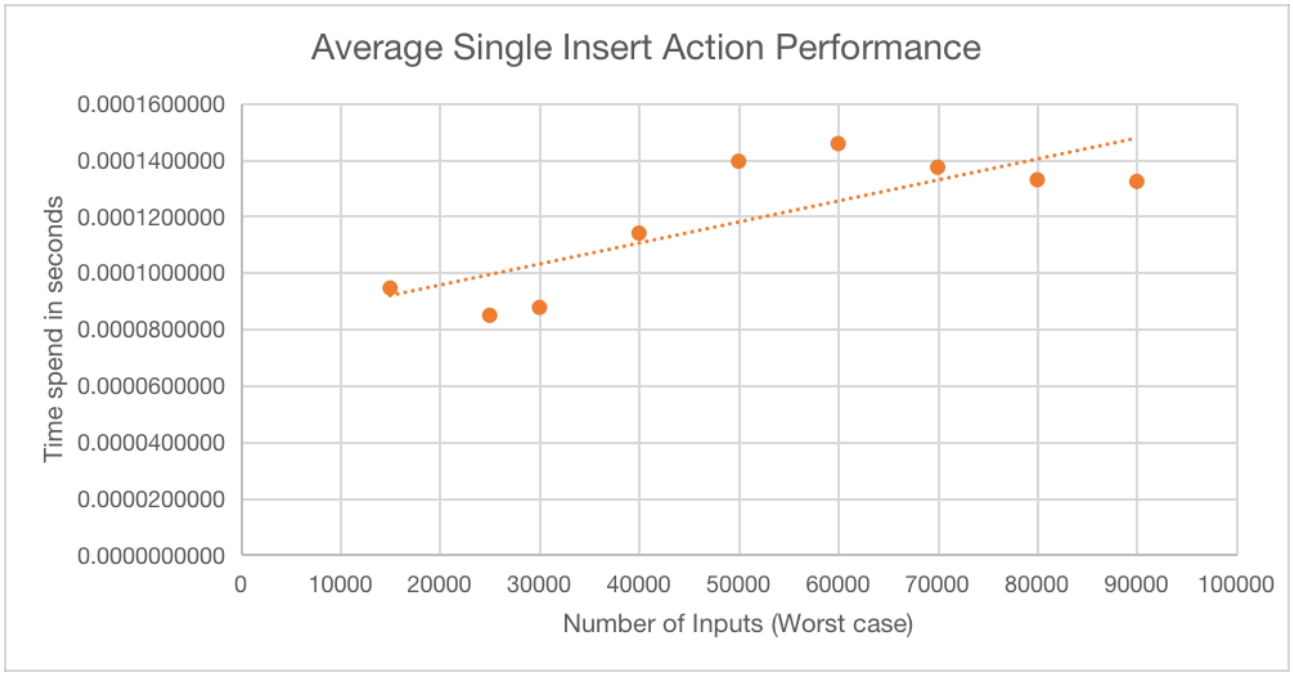
The time for the program to execute was measured using the UNIX time command summing the user and sys values output by the command.

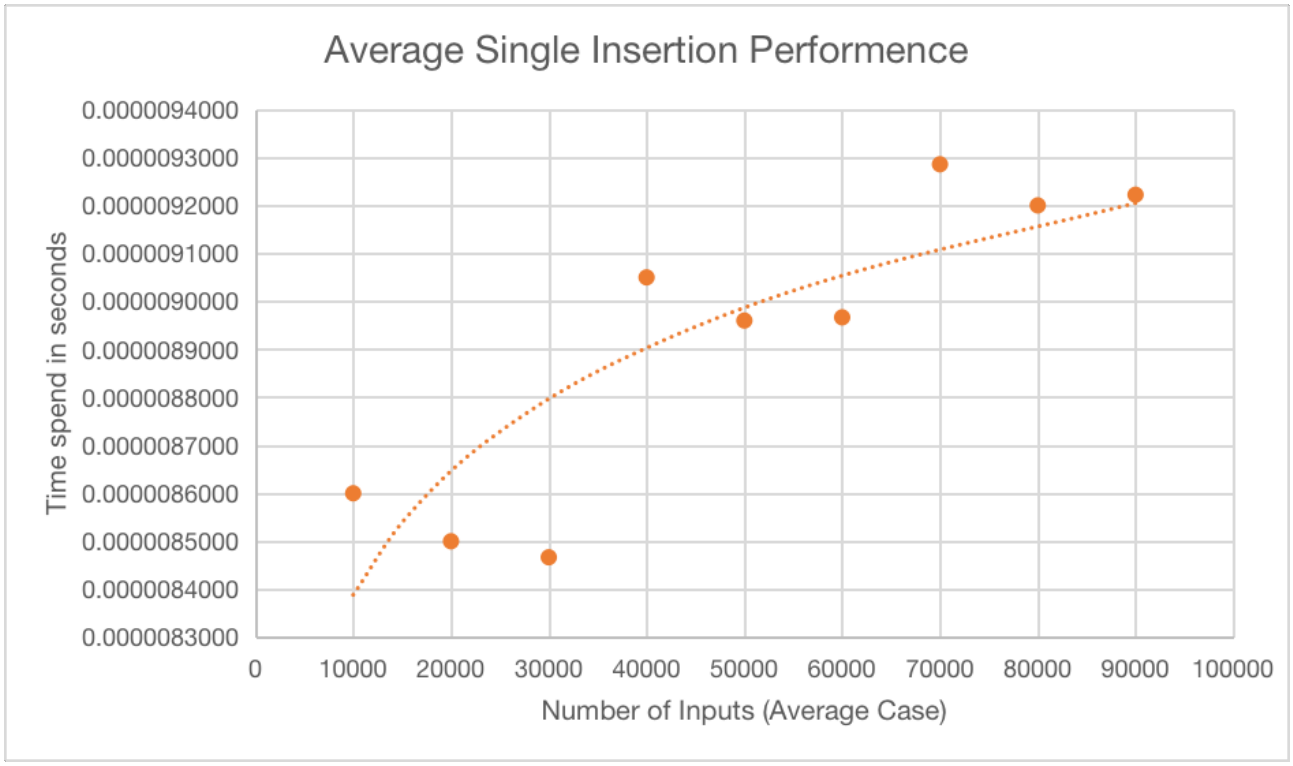
The analysis of two sets of time data was performed by using different data generation methods (random and sorted).

2.3 Results

The results were then plotted using “matplotlib”, which consists of 2 graphs in Figure 2 and the raw data can be seen in Appendix F and Appendix G. When the data is sorted(raw data in Appendix F), the time output is nearly linear: specifically, an upward sloping line, although the beginning part of the time data is somewhat higher. There are also slight fluctuations at the end of the data. When the output data is randomly generated(raw data in Appendix G), the generated image is a curve that is convex upward, and the adjacent data points are very close, divided into 3 segments

Figure2


Average Single Insert Action Performance
Time spend in seconds
0.0001600000
0.0001400000
0.0001200000
0.0001000000
0.0000800000
0.0000600000
0.0000400000
0.0000200000
0.0000000000
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Number of Inputs (Worst case)


Average Single Insertion Performence
Time spend in seconds
0.0000094000
0.0000093000
0.0000092000
0.0000091000
0.0000090000
0.0000089000
0.0000088000
0.0000087000
0.0000086000
0.0000085000
0.0000084000
0.0000083000
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Number of Inputs (Average Case)

2.4 Discussion

The results of the experimental output are basically consistent with the expected assumptions, but not completely in line with the theoretical shape of the graph.

3 Extension Experiment III: Inserting Performance Comparison between 2 modes for Hash Set

3.1 Hypothesis

The hypothesis is about the expected running time of the hash-set: Theoretically, The running time of the linear probing (Mode 0) is faster than the quadratic probing(Mode 1).

Here are some references of these two strategies:

> Quadratic probing lies between the two in terms of cache performance and clustering.
> Linear Probing has the best cache performance but suffers from clustering.*[1]

3.2 Design

This experiment uses the same input files data compared with Experiment I.

To test the hypothesis, 5 random input dictionaries of sizes 100K, 200K, 300K, 400K, 500K were produced. In order to reduce the probability that some random generated dictionaries have non-typical property of the average case, This experiment generates 5 for each so it can get an average running time for each size.

The process of generating dictionaries input was automated using the shell scripts shown in Appendix A. The script for computing the time shows in Appendix D.

The time for the program to execute was measured using the UNIX time command summing the user and sys values output by the command.

3.3 Results

The results are shown in Figure 3 and the raw data can be seen in Appendix H.
By comparing photos and raw data, Although the average running time of the linear probing (Mode 0) is faster than the quadratic probing(Mode 1), there is not much difference in the output time of the two strategies.The performance of the two is very close, and the average difference is generally not more than 0.05 seconds

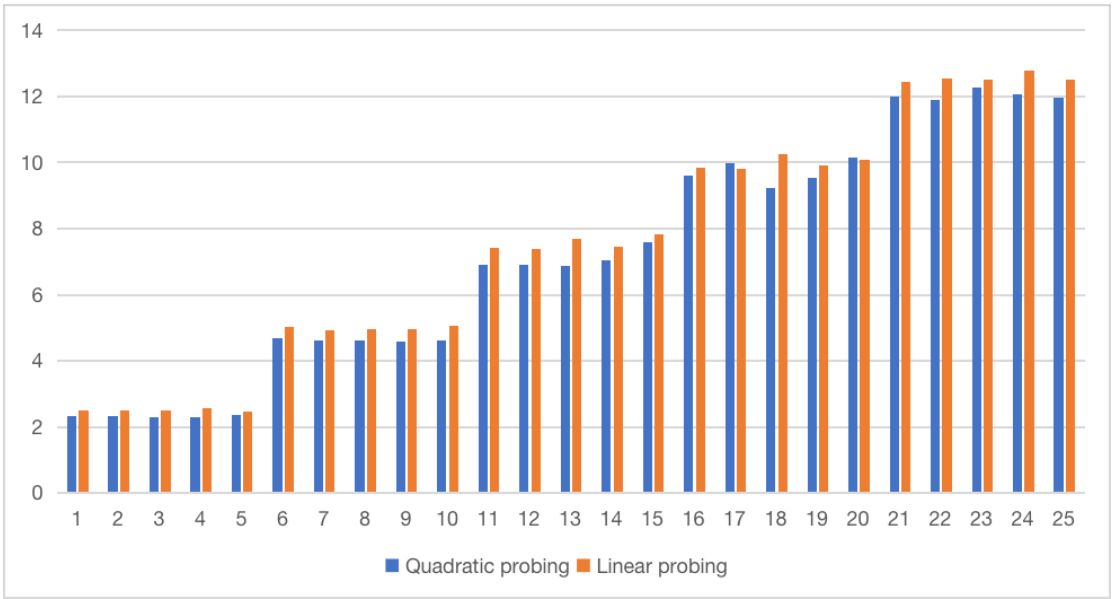
3.4 Discussion

The experimental results almost meet hypothesis expectations, but not sufficiently.
This may be due to overestimating the performance gap between the two when setting the assumptions. Another reason is that due to the excellent performance of the hash function used, it is difficult to distinguish time differences when the amount of data is not large.

---

[1] https://www.andrew.cmu.edu/course/15-310/applications/ln/hashing-review.html

Figure 3



QUADRATIC PROBING VS LINEAR PROBING

4 Conclusion

In general, the results of two experiments are consistent with the hypothesis made before, but the third experiment needs more manipulations to be fully proved. This may require re-analyze the performance theory and increase the sample size to get a better experiment result.

PLEASE NOTE YOU NEED TO CREATE THE CORRESPONDING DIRECTORY FOR SAVING DATA AND OUTPUTS BY YOURSELF

Appendix A:

Please check the script file: test_data_generator.sh

```
STATES="random sorted reverse"
SIZES="100000 200000 300000 400000 500000"

for STATE in $STATES
do

for SIZE in $SIZES
do

echo $SIZE

for COUNT in 1 2 3 4 5
do

    python3 generate.py random ./test_data/test_dict/dict_${SIZE}_${STATE}_$COUNT ./
test_data/test_query/query_${SIZE}_${STATE}_$COUNT $SIZE 0 $STATE 0

done

done

done
```

Appendix B:

Please check the script file: time_output_generator.sh

```
STATES="random sorted reverse"
SIZES="100000 200000 300000 400000 500000"

rm data_output/sorted_data_output_python.dat
rm data_output/reverse_data_output_python.dat
rm data_output/sorted_data_output_python.csv
rm data_output/reverse_data_output_python.csv
rm data_output/random_data_output_python.dat
rm data_output/random_data_output_python.csv

for STATE in $STATES
do

for SIZE in $SIZES
do

for COUNT in 1 2 3 4 5
do

   # Debugging statement to check program calls working as expected
   # python3 ../../search_and_sort_lab/python/speller_darray.py -d ../dictionaries_and_queries/
dict_${SIZE}_${STATE}_$COUNT -m 1 -s ${SIZE} ../dictionaries_and_queries/query_$
{SIZE}_${STATE}_$COUNT

   ALL_TIME=`(time -p python3 ./python/speller_hashset.py -d ./test_data/test_dict/dict_$
{SIZE}_${STATE}_$COUNT -s ${SIZE} -m 0 ./test_data/test_query/query_${SIZE}_$
{STATE}_$COUNT) 2>&1 | grep -E "user|sys" | sed s/[a-z]//g`

   RUNTIME=0
   for i in $ALL_TIME;
   do RUNTIME=`echo $RUNTIME + $i|bc`;
   done
   echo $SIZE $RUNTIME >> data_output/${STATE}_data_output_python.dat
   echo $SIZE, $RUNTIME >> data_output/${STATE}_data_output_python.csv

done

done

done
```

Appendix C:

Please check the script file: time_output_generator2.sh

```
STATES="random sorted reverse"
SIZES="100000 200000 300000 400000 500000"

rm data_output2/sorted_data_output_python.dat
rm data_output2/reverse_data_output_python.dat
rm data_output2/sorted_data_output_python.csv
rm data_output2/reverse_data_output_python.csv
rm data_output2/random_data_output_python.dat
rm data_output2/random_data_output_python.csv

for STATE in $STATES
do

for SIZE in $SIZES
do

for COUNT in 1 2 3 4 5
do

   # Debugging statement to check program calls working as expected
   # python3 ../../search_and_sort_lab/python/speller_darray.py -d ../dictionaries_and_queries/
dict_${SIZE}_${STATE}_$COUNT -m 1 -s ${SIZE} ../dictionaries_and_queries/query_$
{SIZE}_${STATE}_$COUNT

   ALL_TIME="$((time -p python3 ./python/speller_bstree.py -d ./test_data/test_dict/dict_$
{SIZE}_${STATE}_$COUNT -s ${SIZE} -m 0 ./test_data/test_query/query_${SIZE}_$
{STATE}_$COUNT) 2>&1 | grep -E "user|sys" | sed s/[a-z]//g)"

   RUNTIME=0
   for i in $ALL_TIME;
   do RUNTIME=`echo $RUNTIME + $i|bc`;
   done
   echo $SIZE $RUNTIME >> data_output2/${STATE}_data_output_python.dat
   echo $SIZE, $RUNTIME >> data_output2/${STATE}_data_output_python.csv

done

done

done
```

Appendix D:

Please check the script file: time_output_generator2.sh

```
STATES="random sorted reverse"
SIZES="100000 200000 300000 400000 500000"

rm data_output3/sorted_data_output_python.dat
rm data_output3/reverse_data_output_python.dat
rm data_output3/sorted_data_output_python.csv
rm data_output3/reverse_data_output_python.csv
rm data_output3/random_data_output_python.dat
rm data_output3/random_data_output_python.csv

for STATE in $STATES
do

for SIZE in $SIZES
do

for COUNT in 1 2 3 4 5
do

   # Debugging statement to check program calls working as expected
   # python3 ../../search_and_sort_lab/python/speller_darray.py -d ../dictionaries_and_queries/
dict_${SIZE}_${STATE}_$COUNT -m 1 -s ${SIZE} ../dictionaries_and_queries/query_$
{SIZE}_${STATE}_$COUNT

   ALL_TIME=`(time -p python3 ./python/speller_hashset.py -d ./test_data/test_dict/dict_$
{SIZE}_${STATE}_$COUNT -s ${SIZE} -m 1 ./test_data/test_query/query_${SIZE}_$
{STATE}_$COUNT) 2>&1 | grep -E "user|sys" | sed s/[a-z]//g`

   RUNTIME=0
   for i in $ALL_TIME;
   do RUNTIME=`echo $RUNTIME + $i|bc`;
   done
   echo $SIZE $RUNTIME >> data_output3/${STATE}_data_output_python.dat
   echo $SIZE, $RUNTIME >> data_output3/${STATE}_data_output_python.csv

done

done

done
```

Appendix E:

The first column shows data size and and second column represents time (second)

100000	2.34
100000	2.32
100000	2.28
100000	2.28
100000	2.36
200000	4.70
200000	4.62
200000	4.62
200000	4.59
200000	4.61
300000	6.91
300000	6.91
300000	6.86
300000	7.06
300000	7.58
400000	9.59
400000	9.99
400000	9.24
400000	9.53
400000	10.16
500000	12.01
500000	11.90
500000	12.29
500000	12.08
500000	11.98

Appendix F:

10000	1.43
10000	1.4
10000	1.45
10000	1.42
10000	1.39
20000	2.15
20000	2.09
20000	2.15
20000	2.1
20000	2.12
30000	2.56
30000	2.75
30000	2.64
30000	2.57
30000	2.63
40000	4.51
40000	4.58
40000	4.53
40000	4.68
40000	4.5
50000	7.05
50000	6.85
50000	7.26
50000	6.79
50000	6.92

Appendix G:

10000	0.09
10000	0.09
10000	0.07
10000	0.09
10000	0.09
20000	0.17
20000	0.17
20000	0.18
20000	0.16
20000	0.17
30000	0.23
30000	0.23
30000	0.26
30000	0.28
30000	0.27
40000	0.35
40000	0.39
40000	0.34
40000	0.36
40000	0.37
50000	0.45
50000	0.46
50000	0.44
50000	0.45
50000	0.44

Appendix H

100000	2.50
100000	2.48
100000	2.49
100000	2.55
100000	2.45
200000	5.04
200000	4.91
200000	4.95
200000	4.95
200000	5.07
300000	7.42
300000	7.40
300000	7.68
300000	7.45
300000	7.82
400000	9.84
400000	9.81
400000	10.25
400000	9.93
400000	10.10
500000	12.45
500000	12.55
500000	12.51
500000	12.79
500000	12.52