COMP26120 Lab 2 Report

Louise A. Dennis

August 28, 2022

1 Experiment 1 – Sorting Performance

1.1 Theoretical Best Case

Hypothesis The behaviour for insertion sort on sorted input is $O(n)$.

Theoretically, the best case for insertion sort is $O(n)$ when the input is sorted. The algorithm loops over the input and inserts each element at the start of the sorted list.

Experimental Design To test the hypothesis, random sorted input dictionaries were produced. 5 dictionaries of sizes 10K, 20K, 30K, 40K and 50K were generated. The spell checking program was then run on each dictionary with an input file containing a single word that was not in the dictionary in order that the look-up time should have as little impact on the comparative performance on each dictionary as possible. 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 process of generating dictionaries and computing the time was automated using the shell scripts shown in Appendix B.

Results The results were then plotted using `gnuplot` and `gnuplot`'s **fit** functionality was used 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 A. As can be seen from the graph the the line $f(x) = mx + q$ has a good fit and the values of 0.000002 and 0.22 have been computed for m and q respectively. This confirms the hypothesis about the behaviour of the algorithm on sorted input, and allows us to predict the time taken to sort a dictionary of size x as $0.000002x + 0.22$.

1.2 Theoretical Worst Case

Hypothesis The behaviour for insertion sort on reverse sorted input is $O(n^2)$.

Theoretically, the worst case for insertion sort is $O(n^2)$ when the input is reverse sorted. The algorithm loops over the input and inserts each element at the end of the sorted list.

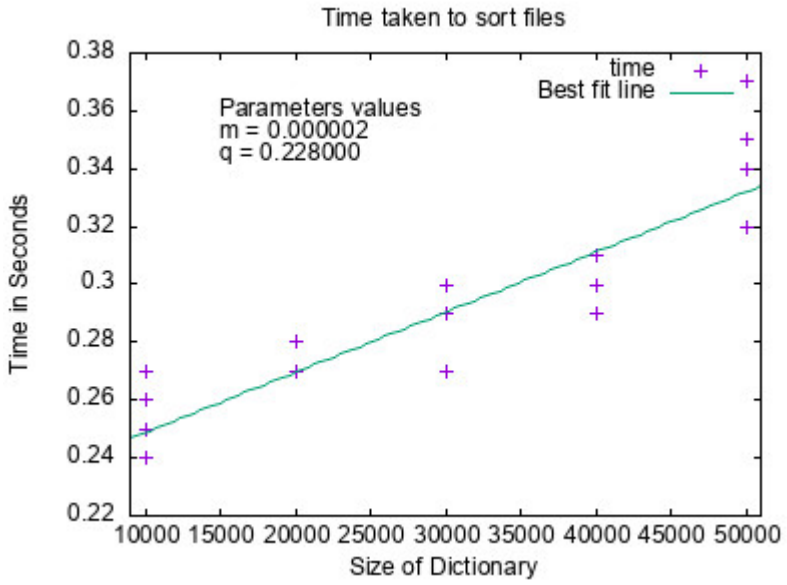

Figure 1: Time taken to look up a word in a sorted dictionary, with best fit line $f(x) = mx + q$ shown and values for the parameters m and q

Experimental Design To test the hypothesis, random reverse sorted input dictionaries were produced. 5 dictionaries of sizes 10K, 20K, 30K, 40K and 50K were generated. The spell checking program was then run on each dictionary with an input file containing a single word that was not in the dictionary in order that the look-up time should have as little impact on the comparative performance on each dictionary as possible. 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 process of generating dictionaries and computing the time was automated using the shell scripts shown in Appendix B.

Results The results were then plotted using `gnuplot` and `gnuplot`'s **fit** functionality was used to fit a line $f(x) = m \times x^2 + q$ calculating values for m and q in the process. Note that q was initially set to 0.2 based on viewing the data points on the graph, and the fitting process didn't vary this value [1]. The results are shown in Figure 2 and the raw data can be seen in Appendix A. As can be seen from the graph the line $f(x) = mx^2 + q$ has a good fit to the data and the values 0.005×10^{-7} and 0.2 have been computed for m and q respectively. This

[1] This seems to be a bug in the gnuplot fitting algorithm where it doesn't vary q for quadratic functions. For a two hour lab setting an initial value was fine, for a scientific paper this wouldn't be acceptable without some justification.

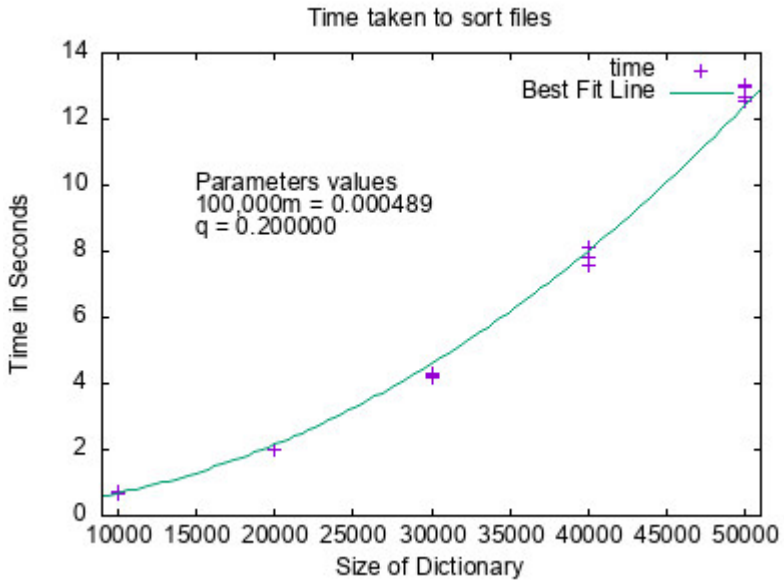

Figure 2: Time taken to look up a word in a reverse sorted dictionary, with best fit line $f(x) = mx^2 + q$ shown and values for the parameters m and q

confirms our hypothesis about the behaviour of the algorithm on reverse sorted input, and allows us to predict the time taken to sort a dictionary of size x as $0.005 \times 10^{-7}x^2 + 0.2$.

1.3 Average Case

Hypothesis The behaviour for insertion sort on random input is somewhere between the theoretical best case $O(n)$ and the theoretical worst case $O(n^2)$.

If we've identified the best and worst cases correctly, then average case performance should lie between these two. However we don't know whether the behaviour will be linear (if slower than best case) or quadratic (if faster than worst case). Plotting the data and comparing to best and worst cases should help us determine this.

Experimental Design To test the hypothesis, random input dictionaries were produced. 5 dictionaries of sizes 10K, 20K, 30K, 40K and 50K were generated. The spell checking program was then run on each dictionary with an input file containing a single word that was not in the dictionary in order that the look-up time should have as little impact on the comparative performance on each dictionary as possible. The time for the program to execute was measured using the UNIX `time` command summing the `user` and `sys` values output by

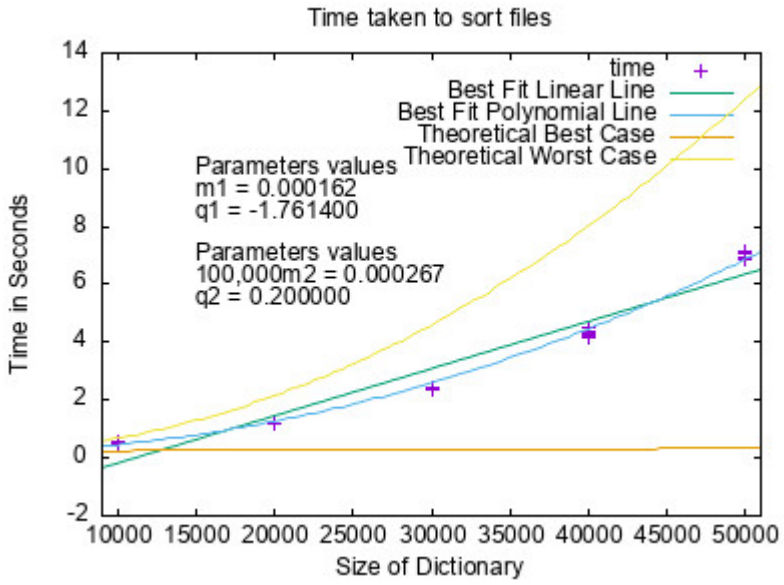

Figure 3: Time taken to look up words in a random dictionary, with best fit lines $f1(x) = m1x + q1$ and $f2(x) = m2x^2 + q2$ shown and values for the parameters $m1$, $m2$, $q1$ and $q2$

the command.

The process of generating dictionaries and computing the time was automated using the shell scripts shown in Appendix B.

Results The results were then plotted using `gnuplot` and `gnuplot`'s **fit** functionality was used to fit two lines, $f1(x) = m1 \times x + q1$ (linear) and $f2(x) = m2 \times x^2 + q2$ (quadratic) calculating values for $m1$, $m2$, $q1$ and $q2$ in the process. As above $q2$ was initially set to 0.2. The results are shown in Figure 3 which also shows the computed lines for best and worst case performance. The raw data can be seen in Appendix A. As can be seen from the graph the line both lines $f1(x)$ and $f2(x)$ can be fitted to the data. However $f2(x) = m2x^2 + q2$ is a better fit than $f1$. For $f2(x)$ the values 0.00026×10^{-8} and 0.2 have been computed for $m2$ and $q2$ respectively. This confirms our hypothesis that the behaviour of the average case lies between that of the best and worst cases and lets us predict that on average the time taken to look up an entry in a dictionary of size x will be $0.00026 \times 10^8 x^2 + 0.2$ seconds.

1.4 Validation and Discussion

While the average case lies between that of the best and worst cases. The fact that the line $f2(x) = m2x^2+q2$ is a better fit for the data than a linear fit means that, in complexity terms, the performance in the average case of insertion sort is polynomial rather than linear. So average case performance of insertion sort more comparable to worst case performance than to best case performance.

2 Experiment 2

Hypothesis Given a dictionary of size, k, and n queries. The point where it becomes quicker to sort the dictionary using insertion sort and then use binary search to check the query, as opposed to using linear search on the unsorted dictionary falls somewhere between $\frac{n}{10}$ queries and $10n$ queries.

Given a dictionary of size k and n queries. The time taken to look up the queries in the dictionary using linear search will be around nk and the time taken to sort the dictionary (average case) and then perform the look up using binary search will be around $k^2 + n \times (log\ k)$. If we assume that, as the numbers become larger. the addition of $n \times (log\ k)$ becomes negligible then we would expect the crossover point to occur around the moment where n becomes larger than k (so nk becomes larger than k^2). For the purpose of our hypothesis we are assuming this is in the range where n and k have the same order of magnitude – so $\frac{k}{10} \leq k \leq 10k$.

Experimental Design To test the hypothesis, random input dictionaries were produced. 5 dictionaries of sizes 10K, 20K, 30K, 40K and 50K were generated. For each dictionary five input files of queries were also produced where, if k is the size of the dictionary the query files were of the size $\frac{k}{10}$, $\frac{k}{2}$, k, $5k$ and $10k$. The spell checking program was then run on each dictionary with each input file in two modes – the first mode performed only linear search and the second mode performed insertion sort followed by binary search. 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 process of generating dictionaries, query input files and computing the time was automated using the shell scripts shown in Appendix D.

Results The results were then plotted using `gnuplot` and `gnuplot`'s **fit** functionality was used to fit the line, $f(x) = m \times x + q$ for each value of k (Note that if k is fixed then both approaches should be linear in the size of n). The results are shown in Figure 4 where the range of x values has been restricted to 0 to $5k$ so the crossover point can be better seen. The raw data can be seen in Appendix C. As can be seen from the graphs the lines for $f1(x)$ and $f2(x)$ cross between 0.5 and 1 for all sizes of dictionary. This confirms our hypothesis that it becomes quicker to sort the dictionary of size n using insertion sort and then use binary search to check the query, as opposed to using linear search on

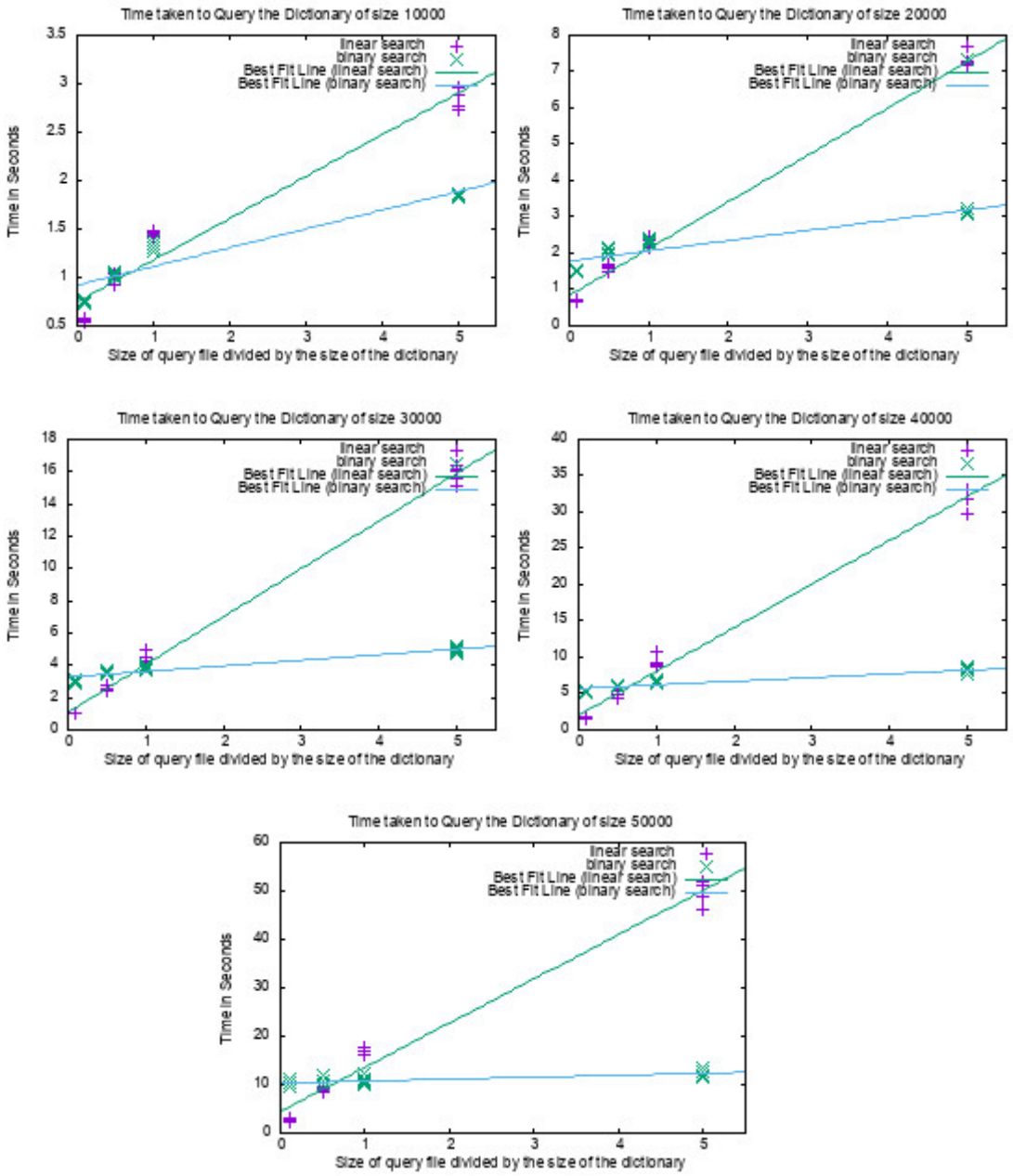

Figure 4: Time taken to look up a word in a random dictionaryies of sizes 10000, 20000, 30000, 40000 and 50000 with best fit lines $f1(x) = m1x + q1$ (linear search) and $f2(x) = m2x^2 + q2$ (insertion sort plus binary search) shown.

the unsorted dictionary somewhere between $\frac{n}{10}$ queries and $10n$ queries. In fact it tells us that this point occurs somewhere around n queries.

A Raw Data for Experiment 1

Sorted Data		Reverse Sorted Data		Random Data	
Size	Time (s)	Size	Time (s)	Size	Time (s)
10000	.24	10000	.75	10000	.50
10000	.24	10000	.75	10000	.56
10000	.25	10000	.74	10000	.53
10000	.26	10000	.68	10000	.50
10000	.27	10000	.71	10000	.50
20000	.28	20000	2.00	20000	1.17
20000	.27	20000	1.99	20000	1.19
20000	.27	20000	2.01	20000	1.19
20000	.27	20000	2.00	20000	1.19
20000	.28	20000	1.99	20000	1.19
30000	.29	30000	4.19	30000	2.40
30000	.30	30000	4.25	30000	2.40
30000	.29	30000	4.26	30000	2.39
30000	.27	30000	4.30	30000	2.42
30000	.27	30000	4.18	30000	2.39
40000	.29	40000	7.56	40000	4.19
40000	.30	40000	8.12	40000	4.37
40000	.31	40000	7.58	40000	4.28
40000	.29	40000	7.60	40000	4.24
40000	.30	40000	7.79	40000	4.48
50000	.32	50000	12.52	50000	7.16
50000	.34	50000	12.65	50000	6.87
50000	.37	50000	12.64	50000	6.96
50000	.34	50000	12.97	50000	7.17
50000	.35	50000	13.05	50000	7.06

B Shell Scripts for Experiment 1

B.1 Generating Dictionaries

```
STATES="sorted reverse none"
SIZES="10000 20000 30000 40000 50000 60000 70000 80000 90000 100000"

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 dict_${SIZE}_${STATE}_$COUNT query_${SIZE}_${STATE}_$COUNT $S

done

done

done
```

B.2 Script for Computing Run Times

```
STATES="sorted reverse none"
SIZES="10000 20000 30000 40000 50000"

rm data/sorted_data_java.dat
rm data/reverse_data_java.dat
rm data/none_data_java.dat
rm data/sorted_data_java.csv
rm data/reverse_data_java.csv
rm data/none_data_java.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
        # java -cp ../../search_and_sort_lab/java  comp26120.speller_darray -d ../dictionari

    ALL_TIME=‘(time -p java -cp ../../search_and_sort_lab/java  comp26120.speller_darray -d


    RUNTIME=0
    for i in $ALL_TIME;
    do RUNTIME=‘echo $RUNTIME + $i|bc‘;
    done
```

```
    echo $SIZE $RUNTIME >> data/${STATE}_data_java.dat
    echo $SIZE, $RUNTIME >> data/${STATE}_data_java.csv
done

done

done
```

C Raw Data for Experiment 2

Dictionary Size:10K, Linear Search	
$\frac{k}{n}$	Time (s)
0.1	.54
0.1	.55
0.1	.55
0.1	.55
0.1	.57
0.5	.93
0.5	.99
0.5	1.03
0.5	.92
0.5	.99
1	1.48
1	1.49
1	1.42
1	1.44
1	1.47
5	2.76
5	2.97
5	2.89
5	2.89
5	2.74
10	4.52
10	4.60
10	4.54
10	4.70
10	4.61

Dictionary Size:10K, Binary Search	
$\frac{k}{n}$	Time (s)
0.1	.77
0.1	.77
0.1	.76
0.1	.74
0.1	.75
0.5	1.02
0.5	1.05
0.5	1.04
0.5	1.04
0.5	1.00
1	1.31
1	1.30
1	1.26
1	1.38
1	1.35
5	1.83
5	1.85
5	1.84
5	1.85
5	1.87
10	2.32
10	2.48
10	2.27
10	2.26
10	2.30

Dictionary Size:20K, Linear Search	
$\frac{k}{n}$	Time (s)
0.1	.70
0.1	.67
0.1	.66
0.1	.71
0.1	.68
0.5	1.61
0.5	1.70
0.5	1.62
0.5	1.64
0.5	1.49
1	2.45
1	2.17
1	2.18
1	2.45
1	2.34
5	7.30
5	7.18
5	7.17
5	7.24
5	7.24
10	13.52
10	13.18
10	12.90
10	13.37
10	14.09

Dictionary Size:20K, Binary Search	
$\frac{k}{n}$	Time (s)
0.1	1.49
0.1	1.53
0.1	1.48
0.1	1.55
0.1	1.49
0.5	1.98
0.5	1.95
0.5	2.11
0.5	2.00
0.5	2.15
1	2.25
1	2.36
1	2.33
1	2.39
1	2.27
5	3.11
5	3.07
5	3.13
5	3.22
5	3.15
10	3.99
10	3.82
10	3.82
10	3.84
10	3.86

Dictionary Size:30K, Linear Search	
$\frac{k}{n}$	Time (s)
0.1	1.00
0.1	1.03
0.1	1.04
0.1	1.04
0.1	1.05
0.5	2.79
0.5	2.55
0.5	2.48
0.5	2.45
0.5	2.40
1	4.26
1	4.07
1	4.98
1	4.95
1	4.46
5	16.39
5	16.10
5	15.10
5	15.53
5	16.00
10	33.85
10	30.76
10	27.90
10	30.54
10	27.46

Dictionary Size:30K, Binary Search	
$\frac{k}{n}$	Time (s)
0.1	2.86
0.1	3.16
0.1	3.03
0.1	2.99
0.1	2.96
0.5	3.59
0.5	3.58
0.5	3.58
0.5	3.73
0.5	3.50
1	3.94
1	3.64
1	3.85
1	4.05
1	3.67
5	4.85
5	5.15
5	4.74
5	5.08
5	4.99
10	6.04
10	6.08
10	6.02
10	6.31
10	6.08

Dictionary Size:40K, Linear Search	
$\frac{k}{n}$	Time (s)
0.1	1.59
0.1	1.66
0.1	1.75
0.1	1.66
0.1	1.64
0.5	5.50
0.5	5.28
0.5	4.98
0.5	4.91
0.5	4.45
1	9.00
1	8.78
1	9.16
1	8.86
1	10.69
5	31.74
5	33.11
5	29.79
5	33.18
5	31.71
10	62.75
10	53.68
10	53.08
10	53.91
10	53.55

Dictionary Size:40K, Binary Search	
$\frac{k}{n}$	Time (s)
0.1	5.28
0.1	5.28
0.1	5.34
0.1	5.09
0.1	5.26
0.5	6.08
0.5	6.07
0.5	5.85
0.5	5.96
0.5	5.93
1	6.76
1	6.79
1	6.52
1	6.86
1	7.00
5	7.81
5	8.09
5	8.67
5	7.59
5	8.38
10	9.30
10	9.34
10	9.48
10	9.18
10	9.20

Dictionary Size:50K, Linear Search		Dictionary Size:50K, Binary Search	
$\frac{k}{n}$	Time (s)	$\frac{k}{n}$	Time (s)
0.1	2.63	0.1	10.41
0.1	2.99	0.1	9.49
0.1	2.46	0.1	9.69
0.1	2.58	0.1	9.74
0.1	3.14	0.1	11.08
0.5	8.83	0.5	11.75
0.5	8.49	0.5	10.92
0.5	9.53	0.5	10.14
0.5	8.61	0.5	9.89
0.5	9.27	0.5	10.72
1	16.08	1	10.30
1	16.21	1	11.29
1	17.03	1	10.77
1	17.79	1	10.07
1	17.66	1	12.40
5	51.26	5	11.75
5	51.85	5	11.71
5	46.17	5	13.54
5	49.66	5	12.56
5	48.76	5	11.91
10	78.46	10	13.58
10	83.17	10	13.72
10	85.35	10	13.54
10	76.64	10	12.32
10	77.61	10	12.98

D Shell Scripts for Experiment 2

D.1 Generating Dictionaries and Queries

```
SIZES="10000 20000 30000 40000 50000"

QUERY_SIZES="0.1 0.5 1 5 10"

for SIZE in $SIZES
do

for QSIZE in $QUERY_SIZES
do

QUERY_SIZE=$( echo "$SIZE * $QSIZE/1" | bc)

echo $QUERY_SIZE
```

```
for COUNT in 1 2 3 4 5
do

     python3 generate.py random dict_${SIZE}_${QUERY_SIZE}_$COUNT query_${SIZE}_${QUERY_SIZE

done

done

done
```

D.2 Computing Run Times

```
SIZES="10000 20000 30000 40000 50000"
QUERY_SIZES="0.1 0.5 1 5 10"


for SIZE in $SIZES
do

rm data/${SIZE}_amortised_data_linear_java.dat
rm data/${SIZE}_amortised_data_binary_java.dat
rm data/${SIZE}_amortised_data_linear_java.csv
rm data/${SIZE}_amortised_data_binary_java.csv

for QSIZE in $QUERY_SIZES
do

QUERY_SIZE=$( echo "$SIZE * $QSIZE/1" | bc)


for COUNT in 1 2 3 4 5
do

    # Debugging command to check program call actually works
    java -cp ../../search_and_sort_lab/java comp26120.speller_darray -d ../dictionaries_and_

    ALL_TIME=‘(time -p java -cp ../../search_and_sort_lab/java comp26120.speller_darray -d

    RUNTIME=0
    for i in $ALL_TIME;
    do RUNTIME=‘echo $RUNTIME + $i|bc‘;
    done
```

```
    echo $QSIZE $RUNTIME >> data/${SIZE}_amortised_data_linear_java.dat
    echo $QSIZE, $RUNTIME >> data/${SIZE}_amortised_data_linear_java.csv

    # Debugging command to check program call actually works
    java -cp ../../search_and_sort_lab/java comp26120.speller_darray -d ../dictionaries_and_

    ALL_TIME2=`(time -p java -cp ../../search_and_sort_lab/java comp26120.speller_darray -d


    RUNTIME2=0
    for i in $ALL_TIME2;
    do RUNTIME2=`echo $RUNTIME2 + $i|bc`;
    done

    echo $QSIZE $RUNTIME2 >> data/${SIZE}_amortised_data_binary_java.dat
    echo $QSIZE, $RUNTIME2 >> data/${SIZE}_amortised_data_binary_java.csv

done

done

done
```