CAB301 – Algorithms and Complexity

Semester 1 2019

Andrew Mather n9713671
Liam Dale n9741283

Abstract

This report discusses experiments that compare the efficiency of two minimum distance algorithms. Both have a suggested efficiency class of $\Theta(n^2)$, but the second algorithm is expected to be optimised and perform better. Both algorithms were implemented and tested in the Java programming language on the average-case efficiency. Tests under the same conditions were performed and compared using the metrics of basic operations performed and execution time. The results showed consistent with the theoretical predictions and that the second algorithm is indeed more efficient.

Contents

1.0 Overview of Algorithms

1.1) First Minimum Distance Algorithm Overview

The first version of the minimum distance algorithm (referred to as the 'first algorithm' henceforth) accepts a numeric input array, and returns the minimum distance between any two of its elements. While iterating with two nested for-loops it compares each item in the array to all other items, updating the minimum distance found if required. Distance is calculated by subtracting elements and then made positive with the absolute value function to ensure an accurate comparison. It includes some inefficiencies of repeating the same comparisons, and uses an additional comparison to prevent elements being compared against themselves.

1.1.1) Basic Operation and Problem Size

In evaluating time complexity, the basic operation of a searching algorithm will generally be the key comparison/s that occur in the search. It is assumed that if the first comparison of the if statement evaluates to false, the second will not be performed. In this first algorithm the basic operation is the second comparison of the if statement in the inner for-loop, the one that compares values from the array. See figure 1 below underlining this comparison. The problem size is the number of elements in the input array as there is only one input and which the algorithm loops through. The algorithm assumes an input array with numeric values.

```
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        if i ≠ j and |A[i] − A[j]| < dmin
            dmin ← |A[i] − A[j]|
return dmin
```

Figure 1 –Algorithm 1 basic operation

1.2) Second Minimum Distance Algorithm Overview

The second version of the minimum distance algorithm (referred to as the 'second algorithm' henceforth) has the same functionality as its counterpart, accepting a numeric array and returning the minimum distance between any two of the elements. It does this by comparing each item in the array with all other items following itself (but not itself or preceding) and updates the minimum distance found if required. In this way duplicate comparisons are removed, improving efficiency. Implementation is done simply using two nested for-loops like the first algorithm, but with improved loop variable choices. The distance is calculated in the same manner as the first algorithm , but it is only done once and assigned to a temporary variable to decrease computation time.

1.2.1) Basic Operation and Problem Size

The basic operation in time analysis of a searching algorithm will generally be the key comparisons that occur in the search. In this first algorithm the basic operation is the comparator of the if statement in the inner for-loop. See figure 2 below underlining this comparison. The problem size is the number of elements in the input array as there is only one input parameter which the algorithm loops through. The algorithm assumes an input array with numeric values.

```
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        temp ← |A[i] − A[j]|
        if temp < dmin
            dmin ← temp
```

Figure 2 –Algorithm 2 basic operation

1.3) Theoretical Efficiencies

The analysis performed will look at the average-case efficiency. It is worth noting that the best, worst and average case efficiencies are all synonymous because the iteration is fixed and there are no early exit conditions. As such both algorithms will always theoretically perform the same number of operations for a given input size, remaining unaffected by the input elements or result.

The average-case theoretical efficiency prediction for both algorithms is $\Theta(n^2)$. Showing this empirically is a purpose of the report. This efficiency class is sensible considering there are two nested for-loops, and both loop variables increment by 1. The theoretical calculations for each algorithm are as follows, with summation formulas provided by Anany Levitin in *Introduction to the Design and Analysis of Algorithms*, 2011.

Algorithm One Mathematical Prediction

The outside loop iterates from $i = 0$ to $n - 1$ and nested loop iterates from $j = 0$ to $n - 1$. The basic operation evaluates each time, besides once each outside iteration where $i = j$.

$$\sum_{i=0}^{n-1}\left(\left(\sum_{j=0}^{n-1} 1\right) - 1\right) = \sum_{i=0}^{n-1}(n-1) - 0 + 1 - 1$$ Break down the inner summation (Levitin, 2011)

$$= \sum_{i=0}^{n-1} n - 1$$ Simplify

$$= n^2 - n$$ $n - 1$, n times

$$\approx n^2$$ Removed non-dominant term

This summation simplification predicts an efficiency class of n^2 with a coefficient of 1.

(Garbade, 2018)**Algorithm Two Mathematical Prediction**

The outside loop iterates from $i = 0$ to $n - 2$ and nested loop iterates from $j = i + 1$ to $n - 1$. The basic operation evaluates each time with no exceptions.

$$\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}(n-1) - (i+1) + 1$$ Break down the inner summation (Levitin, 2011)

$$= \sum_{i=0}^{n-2} n - i - 1$$ Simplify

$$= (n-1) + (n-2) + \cdots + 1$$ Expand

$$= \frac{(\mathrm{n}-1)n}{2}$$ Simplify

$$= \frac{n^2}{2} - \frac{n}{2} \approx \frac{n^2}{2}$$ Approximate

This summation simplification predicts an efficiency class of n^2 with a coefficient of 0.5.

2.0) Implementation of the Algorithms

2.1) Programming Implementation

The algorithms were implemented using Java code in a class. The class contains several key methods used in the experiments. A second class contains the JUnit tests for section 2.2.

a) main: Responsible for data generation and running the experiments.
b) minDistanceAlgorithm1: Pure implementation of algorithm 1. Returns the minimum distance calculated, and is used in the execution time tests.
c) minDistanceAlgorithm1Counter: Algorithm 1 modified with a counter of the basic operation, and returns this counter. Not used in timing as it contains the counter.
d) minDistanceAlgorithm2: Pure implementation of algorithm 2. Returns the minimum distance calculated, and is used in the execution time tests.
e) minDistanceAlgorithm2Counter: Algorithm 2 modified with a counter of the basic operation, and returns this counter. Not used in timing as it contains the counter.

See Appendix A for a visual representation of the code matching the original algorithms to the implementations. See Appendix B to see the counter increment locations. The following mentions some of the key features of the implementation.

- The algorithms use two nested for-loops with initialisation including the loop variable and incrementing. For convention, the loop conditions were styled as $i < n$ as opposed to $i \leq n - 1$.

- Implementations accept a double array and return a double result. This was done as for time analysis integers or non-integers will not affect the basic operation count and any effect on execution time will be unimportant to the efficiency class. Other types (such as integers) would be important to consider for space analysis.

- The algorithm uses an absolute value when calculating the distance between two values, which was implemented using Math.abs that is available without any imported modules.

- The algorithm requires the use of infinity as the initial minimum distance, which is implemented using Double.POSITIVE_INFINITY available without any imported modules.

2.2) Functional Testing of Implementation

Both algorithm implementations were verified with functional testing aimed at normal and edge cases, with results displayed in the following tables. Code contains a JUnit testing class that runs the tests in both tables below. Matching the algorithm to the implementation code (Appendix A) and the functional tests have shown a correct implementation.

Table 1 – Minimum distance 1 functional testing

Test case	**Test instance**	**Expected**	**Actual**	**Status**
Positive values	{2.51, 4.02, 8, 16.9, 32.111}	1.51	1.51	Passed
Negative values	{-2.51, -4.02, -8, -16.9, -32.11}	1.51	1.51	Passed
All values the same	{10.1, 10.1, 10.1, 10.1, 10.1}	0	0	Passed
Empty array	{}	+Infinity	+Infinity	Passed
Array of one element	{151}	+Infinity	+Infinity	Passed
Large minimum distance	{12, 987654321, -987654321, -67e9}	987654309	987654309	Passed
Large values	{-54e7, 99e5, 23e9, -43e8}	5.499e8	5.499e8	Passed
Large array	{-398.12, -863.33, -767.5, 598.6, -139.92, 717, 668, -532.01, -926, -305.31, -901, 987.67, -288, 824.54, -631, -294.1, 21}	6.1	6.1	Passed
Non-numeric data	{‘a’, ‘b’, ‘c’}	Compiler error	Compiler error	Passed
				100%

Table 2 – Minimum distance 2 functional testing

Test case	**Test instance**	**Expected**	**Actual**	**Status**
Positive values	{2.51, 4.02, 8, 16.9, 32.111}	1.51	1.51	Passed
Negative values	{-2.51, -4.02, -8, -16.9, -32.11}	1.51	1.51	Passed
All values the same	{10.1, 10.1, 10.1, 10.1, 10.1}	0	0	Passed
Empty array	{}	+Infinity	+Infinity	Passed
Array of one element	{151}	+Infinity	+Infinity	Passed
Large minimum distance	{12, 987654321, -987654321, -67e9}	987654309	987654309	Passed
Large values	{-54e7, 99e5, 23e9, -43e8}	5.499e8	5.499e8	Passed
Large array	{-398.12, -863.33, -767.5, 598.6, -139.92, 717, 668, -532.01, -926, -305.31, -901, 987.67, -288, 824.54, -631, -294.1, 21}	6.1	6.1	Passed
Non-numeric data	{‘a’, ‘b’, ‘c’}	Compiler error	Compiler error	Passed
				100%

3.0) Experimental Design

3.1) Tools

The experiments were performed on a Windows 10 desktop computer, with further specifications provided in Appendix C. During runtime other concurrent processes were closed. They were created with the high-level, general purpose programming language Java, and compiled and executed on the IntelliJ interactive development environment. Java is currently the most popular programming language based on search engine results (Garbade, 2018). Pseudorandom test data was produced in the data generation using Java's random class (java.util.Random). Algorithm execution time was measured in nanoseconds using nanoTime from Java's system class (System.nanoTime), which is the most precise time measurement available (TutorialsPoint, 2019). Basic operation counts were measured by using a simple counter variable inserted into the algorithm. Excel was used to produce charts and table analysis.

3.2) Methodology

The purpose of the experiments is to show that both algorithms have a quadratic efficiency class, and that the second algorithm is more efficient within the class. To prove this, testing must occur under a comprehensive range of scenarios. This led to input testing arrays being generated at a wide range of sizes, with randomised elements. To increase reliability, tests were repeated for problem sizes to get an averaged result, and the same input arrays were used for both algorithms. Two efficiency metrics were measured to understand time complexity: basic operation counts, and algorithm execution time. The different tests are performed with different functions but utilise the same data generation and actual algorithm.

As previously discussed, the average case efficiency is synonymous with the best and the worst-case efficiencies. As such the need to iterate for each data point and generate pseudorandom arrays is significantly curtailed. This methodology is still adopted however to get a standardised result for time execution. It reduces variability from other factors such as non-basic operations and unexpected system complexities and overheads.

3.2.1) Data Generation

The design uses three for-loops to (1) cycle through array sizes, (2) run repetitions for each size, (3) fill an array of random data to test. These three loops are shown in figure 3 below.

```
// First for loop cycles through array lengths.
for (int arrayLen = startSize; arrayLen <= endSize; arrayLen += stepSize) {
    int opsTotal1 = 0;
    int opsTotal2 = 1;
    long timeTotal1 = 0;
    long timeTotal2 = 0;

    // Second for loop cycles through repeating tests.
    for (int test = 0; test < repeats; test++) {
        double[] testArray = new double[arrayLen];

        // Third for loop populates data array for each test.
        for (int item = 0; item < arrayLen; item++) {
            testArray[item] = randDouble.nextDouble() * numRange;
        }
```

Figure 3 – Data generation and test loops

3.2.2) Running Tests

Step two runs the tests which can be seen in figure 4. Code follows the third for-loop (in the second for-loop) that runs both tests on both algorithms for both metrics, with their corresponding method. As many tests are run at each array size totals of the basic operations and execution time are tracked, then averaged in the output. This final line prints the data in a tabular form (for Excel) to the console with one line per array length. Appendix D contains a full view of the testing code.

```
        // Basic operation count tests
        opsTotal1 += minDistanceAlgorithm1Counter(testArray);
        opsTotal2 += minDistanceAlgorithm2Counter(testArray);

        // Execution time measurement test 1
        long start = System.nanoTime();
        minDistanceAlgorithm1(testArray);
        long end = System.nanoTime();
        timeTotal1 += (end-start);

        // Execution time measurement test 2
        start = System.nanoTime();
        minDistanceAlgorithm2(testArray);
        end = System.nanoTime();
        timeTotal2 += (end-start);
    }
    System.out.println(arrayLen + ", " +
            opsTotal1 / repeats + ", " + timeTotal1 / repeats + ", " +
            opsTotal2 / repeats + ", " + timeTotal2 / repeats);
}
```

Figure 4 – Experimental tests and output

3.2.3) Selection of Problem Size Ranges

Step three was to determine appropriate ranges to test for problem size and repetitions. Analysing these algorithms should be about establishing a clear trend to larger problem sizes. Balancing the significant growth in computational use led to the following ranges being selected. Arrays sizes from 10 to 1000 were tested with a step interval of 10. At each size, 500 tests were run to ensure an accurate average for a clear trend. Final values shown in figure 5.

```
Random randInt = new Random();
int startSize = 10;
int endSize = 1000;
int stepSize = 10;
int repeats = 500; //Num of repetitions of each array generated
int numRange = 10000; //Random integer generate 0 to this number
```

Figure 5 – Final setup choices

The final choices used were establishing by testing different combinations, this can be seen in Appendix E. The effectiveness of the choices was determined by graphing the results and applying regression and R^2 analysis. Input values used a range of 0 to 10,000.

4.0) Experimental Results and Analysis

4.1) Basic Operation Count Tests

The first efficiency metric is basic operations performed. The counter increment is placed in the first algorithm to occur whenever the second part of the if statement occurs. The counter increment is placed in the second algorithm to occur whenever the if statement comparison is evaluated. This can be seen in figure 6 and 7 below, or Appendix B for a fuller view.

```
for (int i = 0; i < n; i++) { // i in 0 to n-1 (incl)
    for (int j = 0; j < n; j++) { // j in 0 to n-1 (incl)
        if (i != j && ++counter > 0 && Math.abs(array[i]-array[j]) < dmin) {
            dmin = Math.abs(array[i] - array[j]);
```

Figure 6 – First algorithm counter increment location

```
for (int i = 0; i < n-1; i++) { // i in 0 to n-2 (incl)
    for (int j = i+1; j < n; j++) { // j in i+1 to n-1 (incl)
        double temp = Math.abs(array[i] - array[j]);
        if (counter++ > 0 && temp < dmin) {
            dmin = temp;
```

Figure 7 – Second algorithm counter increment location

The code using the specifications from the previous section was used to run the experiments producing the required results. The output results were copied to Excel to produce charts and analysis tables for this section. Regression lines with R^2 values were produced as well.

The graphs in figures 8 and 9 display the results from the basic operation tests for each algorithm and will be discussed. They are shown plotted together in appendix L and also discussed in this section. This section's graphs will contain 100 data points for 100 tested array sizes, plotted against the count of average-case basic operations along the y-axis.

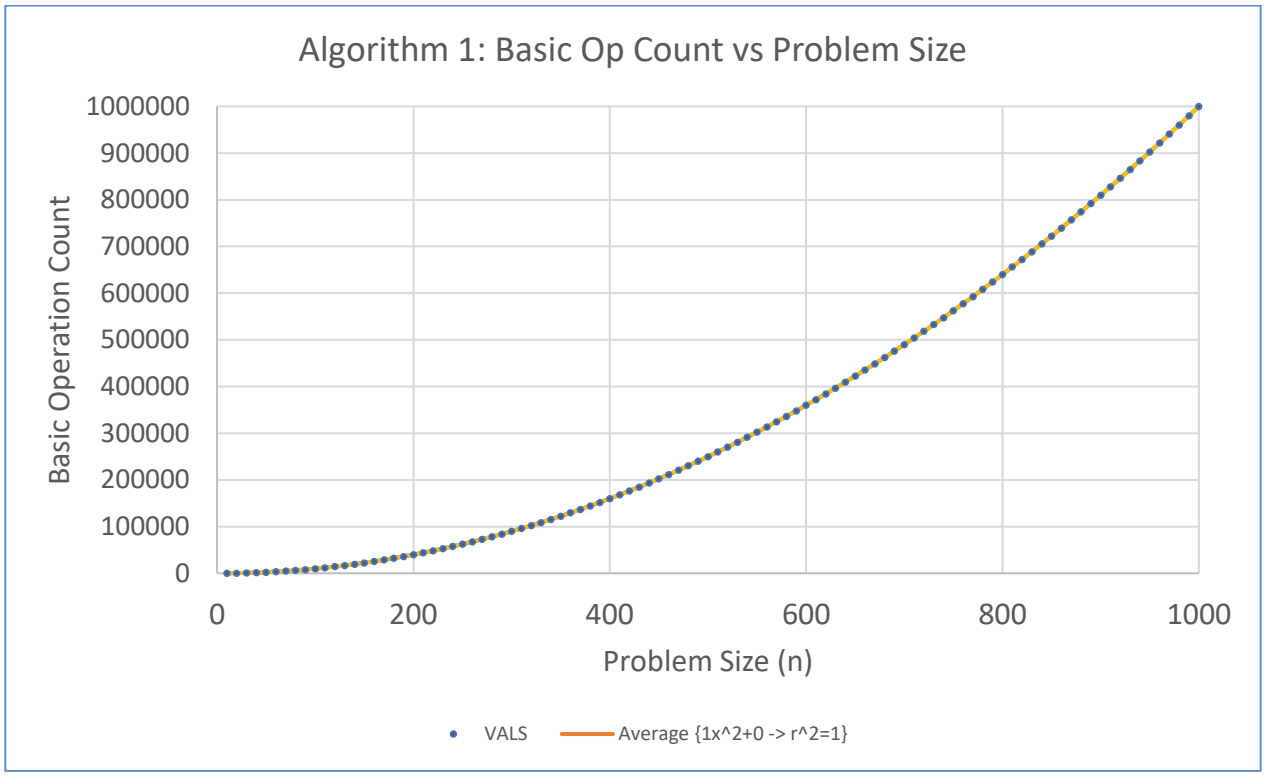

Figure 8 – Experimental results for the first algorithm's basic operation count tests

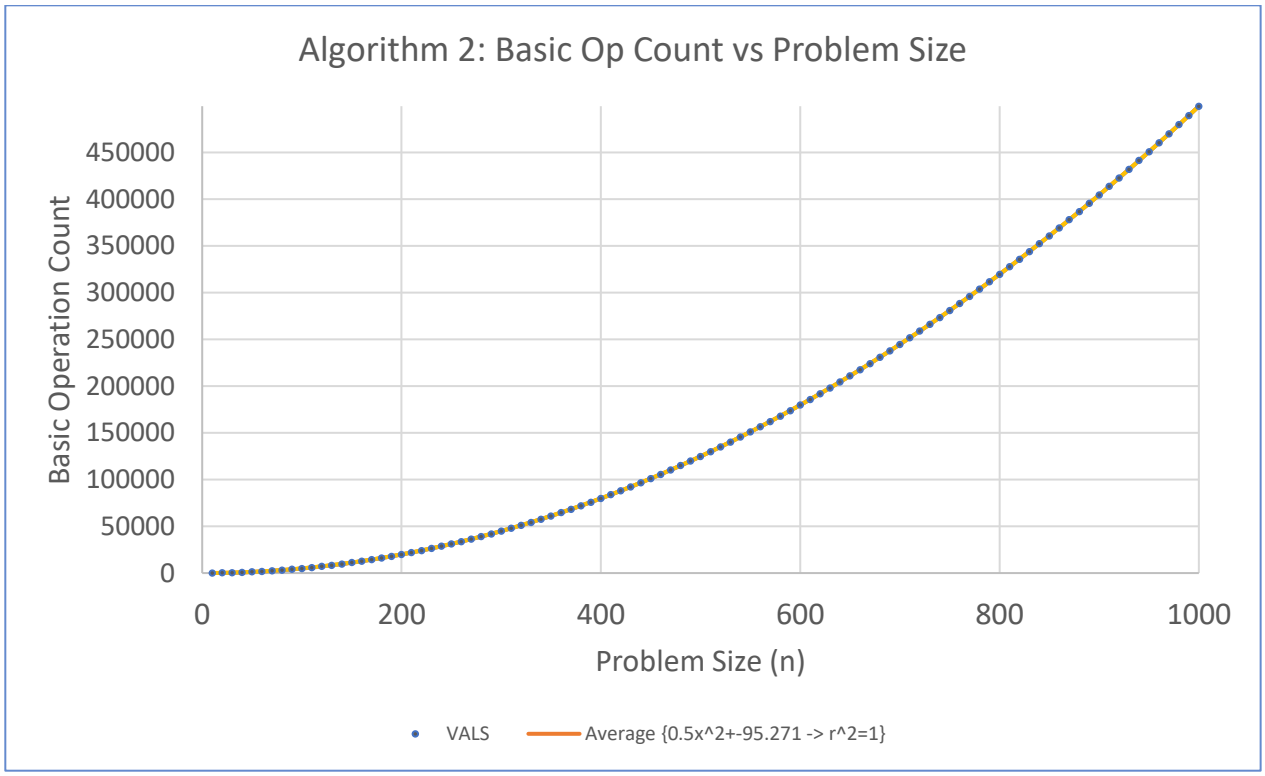

Figure 9 – Experimental results for the second algorithm's basic operation count tests

The theoretical prediction is that both algorithms will have a quadratic efficiency class, and that the second algorithm will be more efficient. This is corroborated by the results shown in the graphs above, both with perfect fit quadratic regression lines and the constant multiplier on the square term being 1.0 for the first and 0.5 for the second algorithm; this metric declares the first algorithm is twice as time complex. In this manner the results are an exact reflection of the theoretical summation calculations.

The difference between the basic operation count, as shown in Appendix L, can be seen when analysing the implementation of the algorithms. As discussed in section one the first algorithm compares all elements twice, whereas the second algorithm compares them only once by offsetting the beginning of the nested for-loop. Comparing elements twice versus only once reflects the constant doubling.

There is a clear trend of polynomial growth, with other types clearly not the case. Both trendlines are highly correlated to the basic operation counts, making them analogous to the tight bound asymptote. This allow them to be compared meaningfully against the predicted class of N^2 and between each algorithm. Neither algorithm's results show any anomalies.

The graphs in Appendix F show the outcome of fitting a linear regressions lines, no higher polynomials need to be considered when the quadratic is a perfect fit. It clearly shows that the linear lines are inadequate representations, evidenced by the poor R^2 values.

The test results are presented in Appendix H and I tables with the addition of some statistical analysis to support graphical analyses. The tables provides analysis on what constant is produced if the data was divided by N, N^2, or N^3, where N is the array size. It shows that the N^2 constant remains correctly steady around its average, with almost no variance – heavily indicating an N^2 efficiency. The linear N class has a high variance, and N^3 continuously decreases towards zero, showing both cannot be the algorithm's efficiency class.

4.2) Execution Time Tests

The second efficiency metric tested is the execution time of the algorithm. This is measured by taking time stamps before and after to nanosecond precision, then finding the difference. This can be seen in figure 10 below. The execution time is averaged across repeated tests for each size. Appendix D may further show this.

```
// Execution time measurement test 1
long start = System.nanoTime();
minDistanceAlgorithm1(testArray);
long end = System.nanoTime();
timeTotal1 += (end-start);
```

```
// Execution time measurement test 2
start = System.nanoTime();
minDistanceAlgorithm2(testArray);
end = System.nanoTime();
timeTotal2 += (end-start);
```

Figure 10 – Execution tests algorithm 1 (left) and test algorithm 2 (right)

The graph in figures 11 and 12 show the results from the execution time tests and will be discussed. They are shown plotted together in appendix M. The graphs in this section also contain 100 data points, each representing one array size, plotted against the execution time (ns) on the y-axis in the average-case. Upper and lower envelopes are also added.

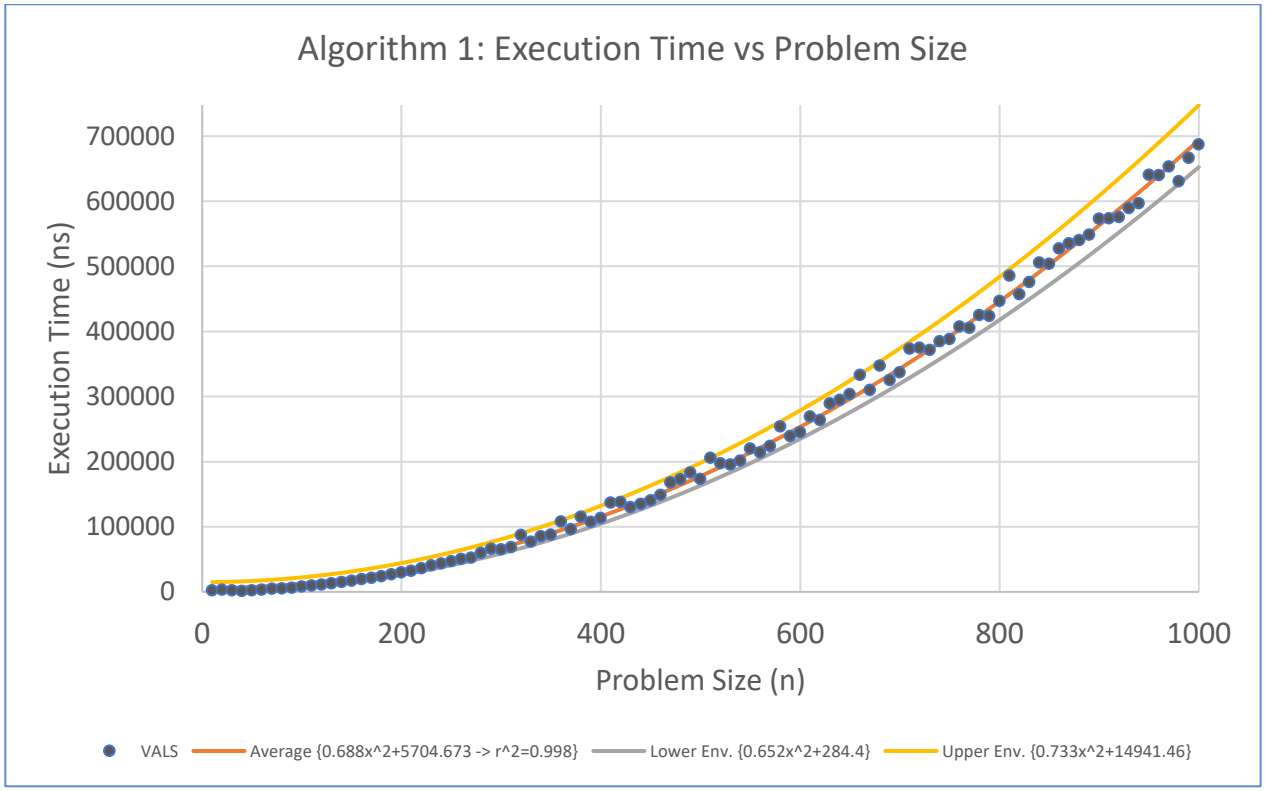

Figure 11 – Experimental results for the first algorithm's execution time tests

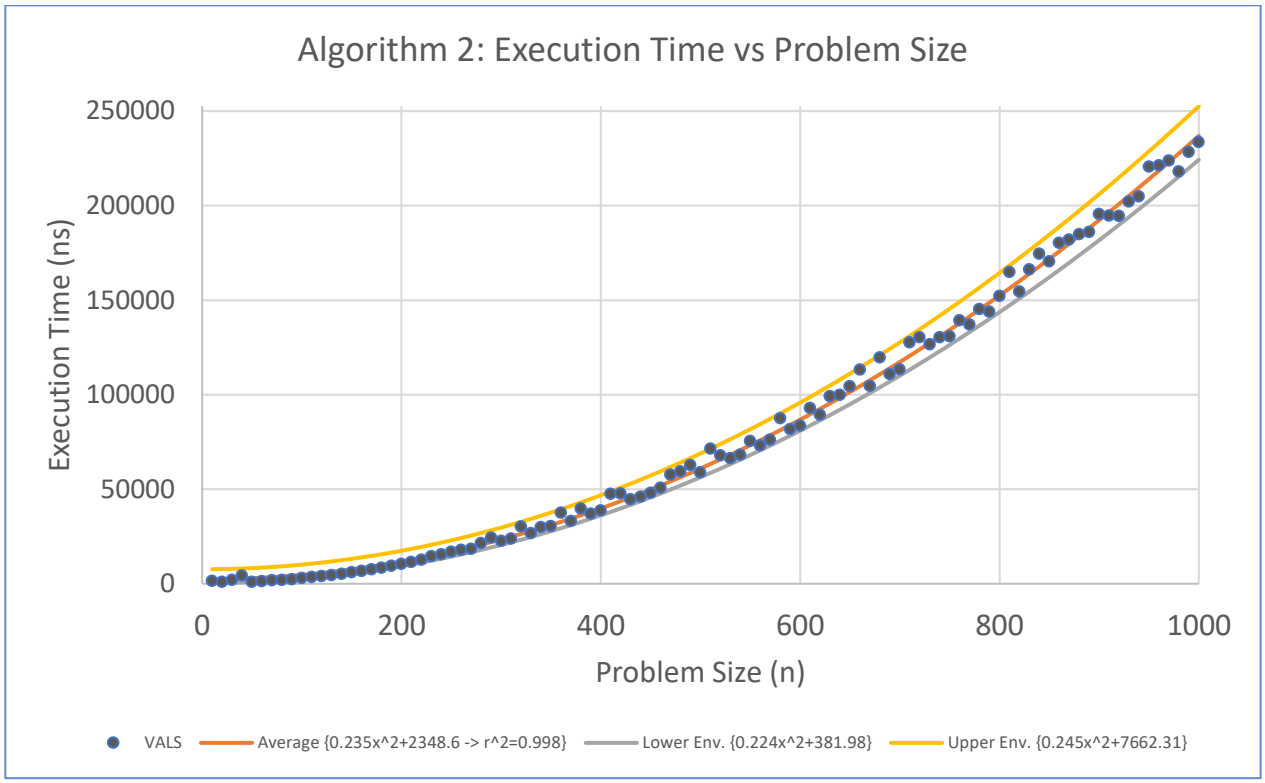

Figure 12 – Experimental results for the second algorithm's execution time tests

The theoretical prediction of an N^2 efficiency class for both algorithms is corroborated by the results in the graphs. This is mathematically shown by the plotting of the lower and upper envelopes which are N^2 multiplied by a constant. For both algorithms all problem sizes examined are tightly encompassed by these bounds.

The coefficient of the two algorithms is most accurately described by the average lines which are highly correlated. They again show that the second algorithm is significantly more effective within the class with a constant multiplier on the square term of 0.235 compared to the first algorithm's multiple of 0.688.

Algorithm one performs twice as many basic operations as algorithm two, whereas as seen in Appendix M, it takes three times as long to execute. This discrepancy is likely attributed to the basic operation not being the sole variable with regards to time complexity; algorithm two makes use of a temporary variable to store distance and there are supplementary operations such as assignments which take differing amounts of time to execute.

There is again a clear trend of polynomial growth. The results from this test are slightly less compelling than the basic operation count tests, however they still support the theoretical prediction as evidenced by the very high R^2 values. It is expected that the increase in data deviations is due computer system overheads throughout testing. Basic operations are only determined from the array size, whereas execution time is impacted by all visible or background system tasks.

The graphs in Appendix G show the outcome of fitting linear or cubic regressions lines. Clearly showing that the linear line is an inadequate representation, evidenced by the poor R^2 value. The cubic function offers no improvement on the quadratic, meaning the algorithm can belong to the quadratic efficiency class. The statistics mentioned further show that a cubic efficiency is not correct.

The test results are presented in Appendix J and K tables with the addition of some statistical analysis to supplement the graphical analyses. The tables provide the same analysis discussed in section 4.1. They show that the N^2 constant remains correctly steady around its average, with a very low variance – providing strong evidence for this class. The linear N class has a high variance, and N^3 continuously decreases and is trending towards zero, showing both cannot be the algorithm's efficiency.

4.3) Results Summary

Both algorithms are shown empirically to have an average-case efficiency class of $\Theta(n^2)$. The first algorithm has a much higher constant multiple within cn^2, which makes the second have a more efficient time complexity. It is double for basic operations, and three times for time execution. This shown in the succeeding graph, where algorithm one is divided by algorithm two for both metrics measured.

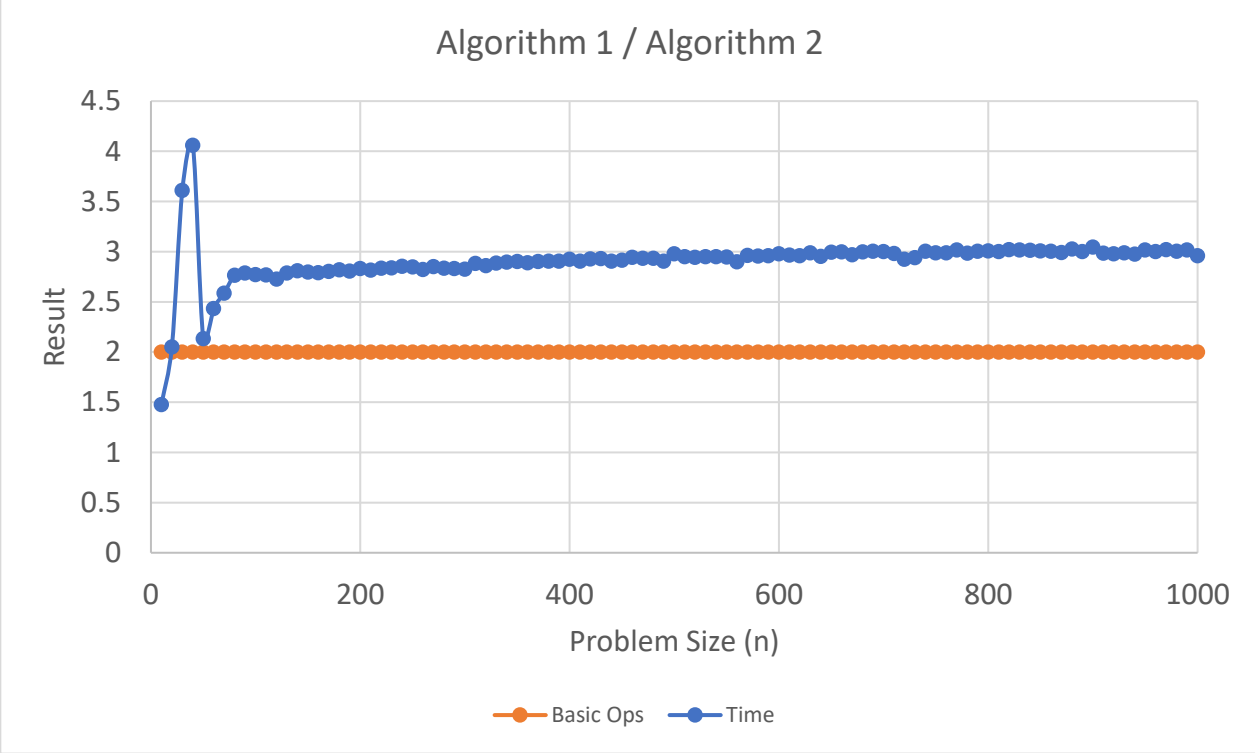

Figure 13 – Algorithm Scalars

As the two algorithms are in the same efficiency class, the scalar difference between the two does not have a large effect at small problem sizes where the execution time is in the microsecond order of magnitude. The difference quickly scales with the size of the input array, where in large scale applications it may be noticeable.

5.0) References

Garbade, M. (2018, August 30). *Top 3 most popular programming languages in 2018* . Retrieved from HackerNoon: https://hackernoon.com/top-3-most-popular-programming-languages-in-2018-and-their-annual-salaries-51b4a7354e06

Levitin, A. (2011). *Introduction to the design & analysis of algorithms — 3rd ed.* Addison-Wesley. Sections 2.1, 2.2, 2.6 and Appendixes A and B.

TutorialsPoint. (2019). *Java.lang.System.nanoTime() Method*. Retrieved from Tutorials Point: https://www.tutorialspoint.com/java/lang/system_nanotime.htm

6.0) Appendices

Appendix A

```java
private static double minDistanceAlgorithm1(double[] array) {
    // Get the minimum distance of any two elements in an array of numbers.
    double dmin = Double.POSITIVE_INFINITY; //isInfinite() method can check
    int n = array.length;

    for (int i = 0; i < n; i++) { // i in 0 to n-1 (incl)
        for (int j = 0; j < n; j++) { // j in 0 to n-1 (incl)
            if (i != j && Math.abs(array[i]-array[j]) < dmin) {
                dmin = Math.abs(array[i] - array[j]);
            }
        }
    }
    return dmin;
}
```

Algorithm *MinDistance*$(A[0..n-1])$
//Input: Array $A[0..n-1]$ of numbers
//Output: Minimum distance between two of its elements
$dmin \leftarrow \infty$
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $n-1$ **do**
    if $i \neq j$ **and** $|A[i]-A[j]| < dmin$
      $dmin \leftarrow |A[i]-A[j]|$
return $dmin$

Figure 14 – First algorithm implementation

```java
public static double minDistanceAlgorithm2(double[] array) {
    // Get the minimum distance of any two elements in an array of numbers.
    double dmin = Double.POSITIVE_INFINITY; //isInfinite() method can check
    int n = array.length;

    for (int i = 0; i < n-1; i++) { // i in 0 to n-2 (incl)
        for (int j = i+1; j < n; j++) { // j in i+1 to n-1 (incl)
            double temp = Math.abs(array[i] - array[j]);

            if (temp < dmin) {
                dmin = temp;
            }
        }
    }
    return dmin;
}
```

Algorithm *MinDistance2*$(A[0..n-1])$
//Input: An array $A[0..n-1]$ of numbers
//Output: The minimum distance d between two of its elements
$dmin \leftarrow \infty$
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    $temp \leftarrow |A[i]-A[j]|$
    if $temp < dmin$
      $dmin \leftarrow temp$
return $dmin$

Figure 15 – Second algorithm implementation

Appendix B

```java
// Returns the basic operation count.
private static int minDistanceAlgorithm1Counter(double[] array) {
    // Get the minimum distance of any two elements in an array of numbers.
    double dmin = Double.POSITIVE_INFINITY; //isInfinite() method can check
    int n = array.length;
    int counter = 0;

    for (int i = 0; i < n; i++) { // i in 0 to n-1 (incl)
        for (int j = 0; j < n; j++) { // j in 0 to n-1 (incl)
            if (i != j && ++counter > 0 && Math.abs(array[i]-array[j]) < dmin) {
                dmin = Math.abs(array[i] - array[j]);
            }
        }
    }
    return counter;
}
```

Figure 16 – Counter version of the first algorithm

```java
// Returns the basic operation count.
private static int minDistanceAlgorithm2Counter(double[] array) {
    // Get the minimum distance of any two elements in an array of numbers.
    double dmin = Double.POSITIVE_INFINITY; //isInfinite() method can check
    int n = array.length;
    int counter = 0;

    for (int i = 0; i < n-1; i++) { // i in 0 to n-2 (incl)
        for (int j = i+1; j < n; j++) { // j in i+1 to n-1 (incl)
            double temp = Math.abs(array[i] - array[j]);
            if (counter++ > 0 && temp < dmin) {
                dmin = temp;
            }
        }
    }
    return counter;
}
```

Figure 17 – Counter version of the second algorithm

Appendix C

OS: Windows 10
CPU: 6 x 3.2GHz
RAM: 16gB
HDD: SSD

Appendix D

```
// First for loop cycles through array lengths.
for (int arrayLen = startSize; arrayLen <= endSize; arrayLen += stepSize) {
    int opsTotal1 = 0;
    int opsTotal2 = 1;
    long timeTotal1 = 0;
    long timeTotal2 = 0;

    // Second for loop cycles through repeating tests.
    for (int test = 0; test < repeats; test++) {
        double[] testArray = new double[arrayLen];

        // Third for loop populates data array for each test.
        for (int item = 0; item < arrayLen; item++) {
            testArray[item] = randDouble.nextDouble() * numRange;
        }
        // Basic operation count tests
        opsTotal1 += minDistanceAlgorithm1Counter(testArray);
        opsTotal2 += minDistanceAlgorithm2Counter(testArray);

        // Execution time measurement test 1
        long start = System.nanoTime();
        minDistanceAlgorithm1(testArray);
        long end = System.nanoTime();
        timeTotal1 += (end-start);

        // Execution time measurement test 2
        start = System.nanoTime();
        minDistanceAlgorithm2(testArray);
        end = System.nanoTime();
        timeTotal2 += (end-start);
    }
    System.out.println(arrayLen + ", " +
            opsTotal1 / repeats + ", " + timeTotal1 / repeats + ", " +
            opsTotal2 / repeats + ", " + timeTotal2 / repeats);
}
```

Figure 18 – Data generation and tests

Appendix E

Table 2 – Problem Size Range Tests

Test	Max size	Step size	Tests	Start size	Outcome
Test 1	800	20	5000	20	Very clear trend, number of repetitions can be reduced significantly, step size too large and leaving noticeable gaps in the chart.
Test 2	800	10	2000	10	Just as clear as test 1, can reduce repetitions further, increase max array size, and perhaps improve detail with a smaller step.
Test 3	1000	5	100	5	Loss of accuracy in the results; more outliers and loss of trend accuracy. Repeats too low.
Test 4	1000	5	200	5	Step size reduction unnoticeable, trend accuracy can be further improved.
Test 5	1000	10	500	10	Looks visually good, regression line accurate.

Appendix F – Additional basic operation count graphs

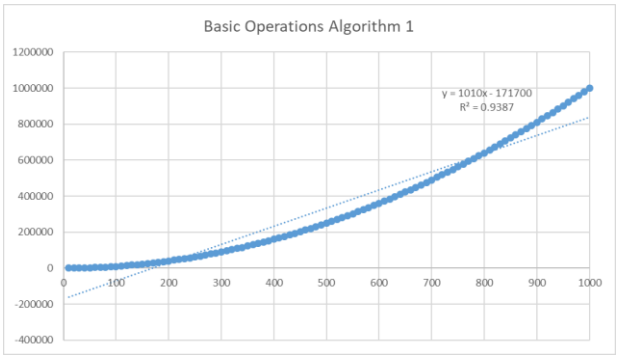

Figure 19 – Algorithm 1 linear regression
$R^2 = 0.9387$

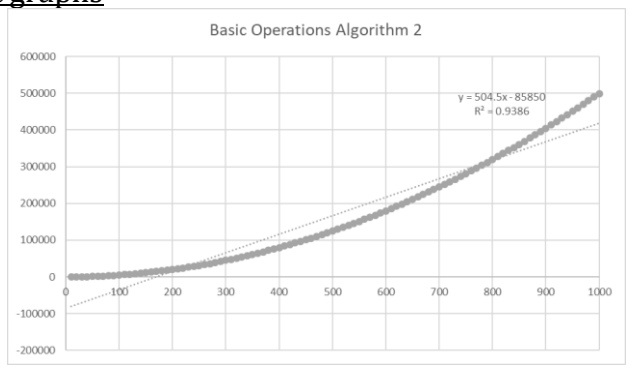

Figure 20 – Algorithm 2 linear regression
$R^2 = 0.9386$

Appendix G – Additional execution time graphs

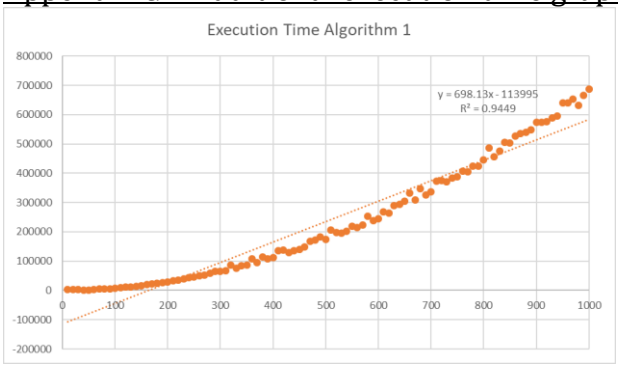

Figure 21 – Algorithm 1 linear regression
$R^2 = 0.9449$

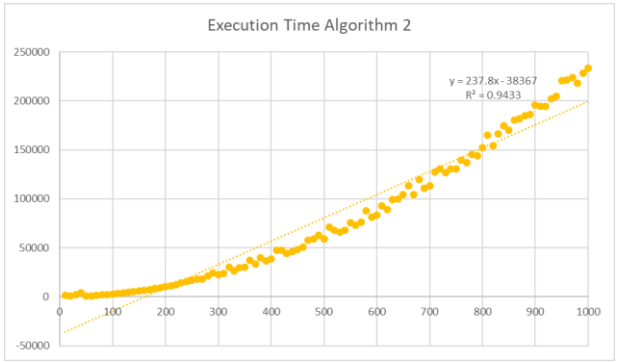

Figure 22 – Algorithm 2 linear regression
$R^2 = 0.9433$

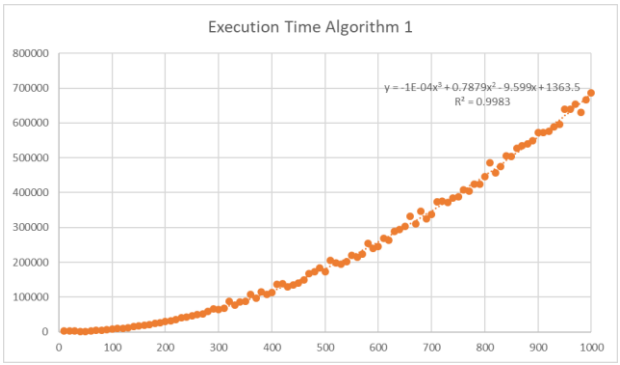

Figure 23 – Algorithm 1 cubic regression
$R^2 = 0.9983$

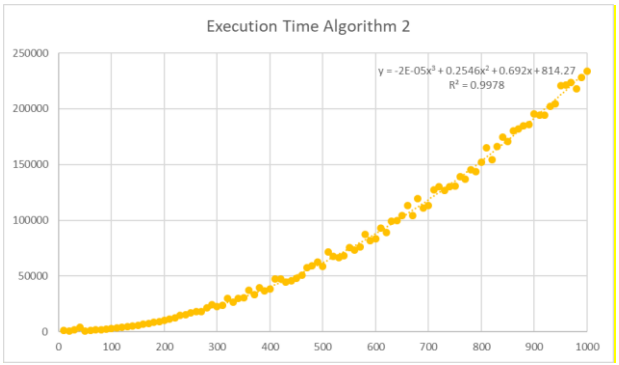

Figure 24 – Algorithm 2 cubic regression
$R^2 = 0.9978$

Appendix H

Size	Count1	n^2 C	Linear C	n^3 C
10	100	1.00	10.00	0.1000
20	400	1.00	20.00	0.0500
30	900	1.00	30.00	0.0333
40	1600	1.00	40.00	0.0250
50	2500	1.00	50.00	0.0200
60	3600	1.00	60.00	0.0167
70	4900	1.00	70.00	0.0143
80	6400	1.00	80.00	0.0125
90	8100	1.00	90.00	0.0111
100	10000	1.00	100.00	0.0100
110	12100	1.00	110.00	0.0091
120	14400	1.00	120.00	0.0083
130	16900	1.00	130.00	0.0077
140	19600	1.00	140.00	0.0071
150	22500	1.00	150.00	0.0067
160	25600	1.00	160.00	0.0063
170	28900	1.00	170.00	0.0059
180	32400	1.00	180.00	0.0056
190	36100	1.00	190.00	0.0053
200	40000	1.00	200.00	0.0050
210	44100	1.00	210.00	0.0048
220	48400	1.00	220.00	0.0045
230	52900	1.00	230.00	0.0043
240	57600	1.00	240.00	0.0042
250	62500	1.00	250.00	0.0040
260	67600	1.00	260.00	0.0038
270	72900	1.00	270.00	0.0037
280	78400	1.00	280.00	0.0036
290	84100	1.00	290.00	0.0034
300	90000	1.00	300.00	0.0033
310	96100	1.00	310.00	0.0032
320	102400	1.00	320.00	0.0031
330	108900	1.00	330.00	0.0030
340	115600	1.00	340.00	0.0029
350	122500	1.00	350.00	0.0029
360	129600	1.00	360.00	0.0028
370	136900	1.00	370.00	0.0027
380	144400	1.00	380.00	0.0026
390	152100	1.00	390.00	0.0026
400	160000	1.00	400.00	0.0025
410	168100	1.00	410.00	0.0024
420	176400	1.00	420.00	0.0024
430	184900	1.00	430.00	0.0023
440	193600	1.00	440.00	0.0023
450	202500	1.00	450.00	0.0022
460	211600	1.00	460.00	0.0022
470	220900	1.00	470.00	0.0021
480	230400	1.00	480.00	0.0021
490	240100	1.00	490.00	0.0020
500	250000	1.00	500.00	0.0020
510	260100	1.00	510.00	0.0020
520	270400	1.00	520.00	0.0019
530	280900	1.00	530.00	0.0019
540	291600	1.00	540.00	0.0019
550	302500	1.00	550.00	0.0018
560	313600	1.00	560.00	0.0018
570	324900	1.00	570.00	0.0018
580	336400	1.00	580.00	0.0017
590	348100	1.00	590.00	0.0017
600	360000	1.00	600.00	0.0017
610	372100	1.00	610.00	0.0016
620	384400	1.00	620.00	0.0016
630	396900	1.00	630.00	0.0016
640	409600	1.00	640.00	0.0016
650	422500	1.00	650.00	0.0015
660	435600	1.00	660.00	0.0015
670	448900	1.00	670.00	0.0015
680	462400	1.00	680.00	0.0015
690	476100	1.00	690.00	0.0014
700	490000	1.00	700.00	0.0014
710	504100	1.00	710.00	0.0014
720	518400	1.00	720.00	0.0014
730	532900	1.00	730.00	0.0014
740	547600	1.00	740.00	0.0014
750	562500	1.00	750.00	0.0013
760	577600	1.00	760.00	0.0013
770	592900	1.00	770.00	0.0013
780	608400	1.00	780.00	0.0013
790	624100	1.00	790.00	0.0013
800	640000	1.00	800.00	0.0013
810	656100	1.00	810.00	0.0012
820	672400	1.00	820.00	0.0012
830	688900	1.00	830.00	0.0012
840	705600	1.00	840.00	0.0012
850	722500	1.00	850.00	0.0012
860	739600	1.00	860.00	0.0012
870	756900	1.00	870.00	0.0011
880	774400	1.00	880.00	0.0011
890	792100	1.00	890.00	0.0011
900	810000	1.00	900.00	0.0011
910	828100	1.00	910.00	0.0011
920	846400	1.00	920.00	0.0011
930	864900	1.00	930.00	0.0011
940	883600	1.00	940.00	0.0011
950	902500	1.00	950.00	0.0011
960	921600	1.00	960.00	0.0010
970	940900	1.00	970.00	0.0010
980	960400	1.00	980.00	0.0010
990	980100	1.00	990.00	0.0010
1000	1000000	1.00	1000.00	0.0010
VAR		0.0000	84167	0.000
AVG		1.0000	530.0	0.0031

Table 3 – Tabulated algorithm 1 basic operation count results

Allow C to stand for 'constant'.
Calculations for:
n^2 C: $Time\ /\ Size^2$
Linear C: $Time\ /\ Size$
n^3 C: $Time\ /\ Size^3$

VAR → Sample variance using Excel function VAR.S().
AVG → Average.
When using these two measures, the first 5 sizes tested were ignored as small n value outliers; more care is placed on larger values of *n*.

The constant calculations and statistical measures show that the n^2 case accurately represents the data whereas the other two do not.

Appendix I

Size	Count2	n^2 C	Linear C	n^3 C
10	45	0.45	4.50	0.0450
20	190	0.48	9.50	0.0238
30	435	0.48	14.50	0.0161
40	780	0.49	19.50	0.0122
50	1225	0.49	24.50	0.0098
60	1770	0.49	29.50	0.0082
70	2415	0.49	34.50	0.0070
80	3160	0.49	39.50	0.0062
90	4005	0.49	44.50	0.0055
100	4950	0.50	49.50	0.0050
110	5995	0.50	54.50	0.0045
120	7140	0.50	59.50	0.0041
130	8385	0.50	64.50	0.0038
140	9730	0.50	69.50	0.0035
150	11175	0.50	74.50	0.0033
160	12720	0.50	79.50	0.0031
170	14365	0.50	84.50	0.0029
180	16110	0.50	89.50	0.0028
190	17955	0.50	94.50	0.0026
200	19900	0.50	99.50	0.0025
210	21945	0.50	104.50	0.0024
220	24090	0.50	109.50	0.0023
230	26335	0.50	114.50	0.0022
240	28680	0.50	119.50	0.0021
250	31125	0.50	124.50	0.0020
260	33670	0.50	129.50	0.0019
270	36315	0.50	134.50	0.0018
280	39060	0.50	139.50	0.0018
290	41905	0.50	144.50	0.0017
300	44850	0.50	149.50	0.0017
310	47895	0.50	154.50	0.0016
320	51040	0.50	159.50	0.0016
330	54285	0.50	164.50	0.0015
340	57630	0.50	169.50	0.0015
350	61075	0.50	174.50	0.0014
360	64620	0.50	179.50	0.0014
370	68265	0.50	184.50	0.0013
380	72010	0.50	189.50	0.0013
390	75855	0.50	194.50	0.0013
400	79800	0.50	199.50	0.0012
410	83845	0.50	204.50	0.0012
420	87990	0.50	209.50	0.0012
430	92235	0.50	214.50	0.0012
440	96580	0.50	219.50	0.0011
450	101025	0.50	224.50	0.0011
460	105570	0.50	229.50	0.0011
470	110215	0.50	234.50	0.0011
480	114960	0.50	239.50	0.0010
490	119805	0.50	244.50	0.0010
500	124750	0.50	249.50	0.0010
510	129795	0.50	254.50	0.0010
520	134940	0.50	259.50	0.0010
530	140185	0.50	264.50	0.0009
540	145530	0.50	269.50	0.0009
550	150975	0.50	274.50	0.0009
560	156520	0.50	279.50	0.0009
570	162165	0.50	284.50	0.0009
580	167910	0.50	289.50	0.0009
590	173755	0.50	294.50	0.0008
600	179700	0.50	299.50	0.0008
610	185745	0.50	304.50	0.0008
620	191890	0.50	309.50	0.0008
630	198135	0.50	314.50	0.0008
640	204480	0.50	319.50	0.0008
650	210925	0.50	324.50	0.0008
660	217470	0.50	329.50	0.0008
670	224115	0.50	334.50	0.0007
680	230860	0.50	339.50	0.0007
690	237705	0.50	344.50	0.0007
700	244650	0.50	349.50	0.0007
710	251695	0.50	354.50	0.0007
720	258840	0.50	359.50	0.0007
730	266085	0.50	364.50	0.0007
740	273430	0.50	369.50	0.0007
750	280875	0.50	374.50	0.0007
760	288420	0.50	379.50	0.0007
770	296065	0.50	384.50	0.0006
780	303810	0.50	389.50	0.0006
790	311655	0.50	394.50	0.0006
800	319600	0.50	399.50	0.0006
810	327645	0.50	404.50	0.0006
820	335790	0.50	409.50	0.0006
830	344035	0.50	414.50	0.0006
840	352380	0.50	419.50	0.0006
850	360825	0.50	424.50	0.0006
860	369370	0.50	429.50	0.0006
870	378015	0.50	434.50	0.0006
880	386760	0.50	439.50	0.0006
890	395605	0.50	444.50	0.0006
900	404550	0.50	449.50	0.0006
910	413595	0.50	454.50	0.0005
920	422740	0.50	459.50	0.0005
930	431985	0.50	464.50	0.0005
940	441330	0.50	469.50	0.0005
950	450775	0.50	474.50	0.0005
960	460320	0.50	479.50	0.0005
970	469965	0.50	484.50	0.0005
980	479710	0.50	489.50	0.0005
990	489555	0.50	494.50	0.0005
1000	499500	0.50	499.50	0.0005
VAR		0.0000	21042	0.0000
AVG		0.4985	264.500	0.0015

Table 4 – Tabulated algorithm 2 basic operation count results

Allow C to stand for 'constant'.
Calculations for:
n^2 C: $Time\ /\ Size^2$
Linear C: $Time\ /\ Size$
n^3 C: $Time\ /\ Size^3$

VAR → Sample variance using Excel function VAR.S().
AVG → Average.
When using these two measures, the first 5 sizes tested were ignored as small n value outliers; more care is placed on larger values of *n*.

The constant calculations and statistical measures show that the n^2 case accurately represents the data whereas the other two do not.

Appendix J

Size	Time1	n^2 C	Linear C	n^3 C
10	2111	21.11	211.10	2.1110
20	3092	7.73	154.60	0.3865
30	2315	2.57	77.17	0.0857
40	1328	0.83	33.20	0.0208
50	1935	0.77	38.70	0.0155
60	2920	0.81	48.67	0.0135
70	4512	0.92	64.46	0.0132
80	5111	0.80	63.89	0.0100
90	6257	0.77	69.52	0.0086
100	8068	0.81	80.68	0.0081
110	9350	0.77	85.00	0.0070
120	10960	0.76	91.33	0.0063
130	12731	0.75	97.93	0.0058
140	14742	0.75	105.30	0.0054
150	16959	0.75	113.06	0.0050
160	19414	0.76	121.34	0.0047
170	21421	0.74	126.01	0.0044
180	23858	0.74	132.54	0.0041
190	26446	0.73	139.19	0.0039
200	29558	0.74	147.79	0.0037
210	32253	0.73	153.59	0.0035
220	35775	0.74	162.61	0.0034
230	40355	0.76	175.46	0.0033
240	43402	0.75	180.84	0.0031
250	46803	0.75	187.21	0.0030
260	50184	0.74	193.02	0.0029
270	52270	0.72	193.59	0.0027
280	59785	0.76	213.52	0.0027
290	66047	0.79	227.75	0.0027
300	65011	0.72	216.70	0.0024
310	68266	0.71	220.21	0.0023
320	87059	0.85	272.06	0.0027
330	76847	0.71	232.87	0.0021
340	85393	0.74	251.16	0.0022
350	87429	0.71	249.80	0.0020
360	108063	0.83	300.18	0.0023
370	96072	0.70	259.65	0.0019
380	115110	0.80	302.92	0.0021
390	107231	0.71	274.95	0.0018
400	113023	0.71	282.56	0.0018
410	136671	0.81	333.34	0.0020
420	137734	0.78	327.94	0.0019
430	129780	0.70	301.81	0.0016
440	134667	0.70	306.06	0.0016
450	140162	0.69	311.47	0.0015
460	148612	0.70	323.07	0.0015
470	167571	0.76	356.53	0.0016
480	172623	0.75	359.63	0.0016
490	183044	0.76	373.56	0.0016
500	173350	0.69	346.70	0.0014
510	205548	0.79	403.04	0.0015
520	197494	0.73	379.80	0.0014
530	195466	0.70	368.80	0.0013
540	201068	0.69	372.35	0.0013
550	219777	0.73	399.59	0.0013
560	214075	0.68	382.28	0.0012
570	224088	0.69	393.14	0.0012
580	253889	0.75	437.74	0.0013
590	239287	0.69	405.57	0.0012
600	245226	0.68	408.71	0.0011
610	269051	0.72	441.07	0.0012
620	263670	0.69	425.27	0.0011
630	289131	0.73	458.94	0.0012
640	294902	0.72	460.78	0.0011
650	303744	0.72	467.30	0.0011
660	332896	0.76	504.39	0.0012
670	309743	0.69	462.30	0.0010
680	347375	0.75	510.85	0.0011
690	325024	0.68	471.05	0.0010
700	336811	0.69	481.16	0.0010
710	373456	0.74	525.99	0.0010
720	374849	0.72	520.62	0.0010
730	371358	0.70	508.71	0.0010
740	384671	0.70	519.83	0.0009
750	388159	0.69	517.55	0.0009
760	407496	0.71	536.18	0.0009
770	405275	0.68	526.33	0.0009
780	424945	0.70	544.80	0.0009
790	423582	0.68	536.18	0.0009
800	446591	0.70	558.24	0.0009
810	485385	0.74	599.24	0.0009
820	456727	0.68	556.98	0.0008
830	475811	0.69	573.27	0.0008
840	505456	0.72	601.73	0.0009
850	503704	0.70	592.59	0.0008
860	527508	0.71	613.38	0.0008
870	535185	0.71	615.16	0.0008
880	539915	0.70	613.54	0.0008
890	548321	0.69	616.09	0.0008
900	573189	0.71	636.88	0.0008
910	573327	0.69	630.03	0.0008
920	575428	0.68	625.47	0.0007
930	589122	0.68	633.46	0.0007
940	596390	0.67	634.46	0.0007
950	640166	0.71	673.86	0.0007
960	640145	0.69	666.82	0.0007
970	653131	0.69	673.33	0.0007
980	630799	0.66	643.67	0.0007
990	666692	0.68	673.43	0.0007
1000	687145	0.69	687.15	0.0007
VAR		0.0020	37148	0.0458
AVG		0.7271	377.5	0.0023

Table 5 – Tabulated algorithm 1 execution time results

The constant calculations and statistical measures show that the n^2 case accurately represents the data whereas the other two do not.

Appendix K

Size	Time2	n^2 C	Linear C	n^3 C
10	1392	13.92	139.20	1.3920
20	973	2.43	48.65	0.1216
30	1861	2.07	62.03	0.0689
40	4315	2.70	107.88	0.0674
50	953	0.38	19.06	0.0076
60	1188	0.33	19.80	0.0055
70	1815	0.37	25.93	0.0053
80	1922	0.30	24.03	0.0038
90	2313	0.29	25.70	0.0032
100	2891	0.29	28.91	0.0029
110	3416	0.28	31.05	0.0026
120	3985	0.28	33.21	0.0023
130	4562	0.27	35.09	0.0021
140	5215	0.27	37.25	0.0019
150	5977	0.27	39.85	0.0018
160	6764	0.26	42.28	0.0017
170	7550	0.26	44.41	0.0015
180	8432	0.26	46.84	0.0014
190	9336	0.26	49.14	0.0014
200	10323	0.26	51.62	0.0013
210	11360	0.26	54.10	0.0012
220	12685	0.26	57.66	0.0012
230	14511	0.27	63.09	0.0012
240	15464	0.27	64.43	0.0011
250	16857	0.27	67.43	0.0011
260	17888	0.26	68.80	0.0010
270	18346	0.25	67.95	0.0009
280	21479	0.27	76.71	0.0010
290	24291	0.29	83.76	0.0010
300	22674	0.25	75.58	0.0008
310	23792	0.25	76.75	0.0008
320	30224	0.30	94.45	0.0009
330	26622	0.24	80.67	0.0007
340	29939	0.26	88.06	0.0008
350	30465	0.25	87.04	0.0007
360	37520	0.29	104.22	0.0008
370	33170	0.24	89.65	0.0007
380	39740	0.28	104.58	0.0007
390	36947	0.24	94.74	0.0006
400	38688	0.24	96.72	0.0006
410	47469	0.28	115.78	0.0007
420	47607	0.27	113.35	0.0006
430	44653	0.24	103.84	0.0006
440	45940	0.24	104.41	0.0005
450	48010	0.24	106.69	0.0005
460	50638	0.24	110.08	0.0005
470	57715	0.26	122.80	0.0006
480	59274	0.26	123.49	0.0005
490	62660	0.26	127.88	0.0005
500	58753	0.24	117.51	0.0005
510	71375	0.27	139.95	0.0005
520	67766	0.25	130.32	0.0005
530	66344	0.24	125.18	0.0004
540	68171	0.23	126.24	0.0004
550	75357	0.25	137.01	0.0005
560	73143	0.23	130.61	0.0004
570	76114	0.23	133.53	0.0004
580	87528	0.26	150.91	0.0004
590	81707	0.23	138.49	0.0004
600	83474	0.23	139.12	0.0004
610	92860	0.25	152.23	0.0004
620	89304	0.23	144.04	0.0004
630	99167	0.25	157.41	0.0004
640	99760	0.24	155.88	0.0004
650	104466	0.25	160.72	0.0004
660	113196	0.26	171.51	0.0004
670	104489	0.23	155.95	0.0003
680	119584	0.26	175.86	0.0004
690	110826	0.23	160.62	0.0003
700	113309	0.23	161.87	0.0003
710	127524	0.25	179.61	0.0004
720	130378	0.25	181.08	0.0003
730	126598	0.24	173.42	0.0003
740	130357	0.24	176.16	0.0003
750	130829	0.23	174.44	0.0003
760	139262	0.24	183.24	0.0003
770	137062	0.23	178.00	0.0003
780	145150	0.24	186.09	0.0003
790	143773	0.23	181.99	0.0003
800	152083	0.24	190.10	0.0003
810	164896	0.25	203.58	0.0003
820	154440	0.23	188.34	0.0003
830	166140	0.24	200.17	0.0003
840	174510	0.25	207.75	0.0003
850	170433	0.24	200.51	0.0003
860	180327	0.24	209.68	0.0003
870	181922	0.24	209.11	0.0003
880	184840	0.24	210.05	0.0003
890	186078	0.23	209.08	0.0003
900	195487	0.24	217.21	0.0003
910	194605	0.24	213.85	0.0003
920	194465	0.23	211.38	0.0002
930	202146	0.23	217.36	0.0003
940	204731	0.23	217.80	0.0002
950	220596	0.24	232.21	0.0003
960	221173	0.24	230.39	0.0002
970	223831	0.24	230.75	0.0002
980	217974	0.23	222.42	0.0002
990	228255	0.23	230.56	0.0002
1000	233664	0.23	233.66	0.0002
VAR		0.0005	4023.9	0.0195
AVG		0.2529	129.7	0.0008

Table 6 – Tabulated algorithm 2 execution time results

The constant calculations and statistical measures show that the n^2 case accurately represents the data whereas the other two do not.

Appendix L

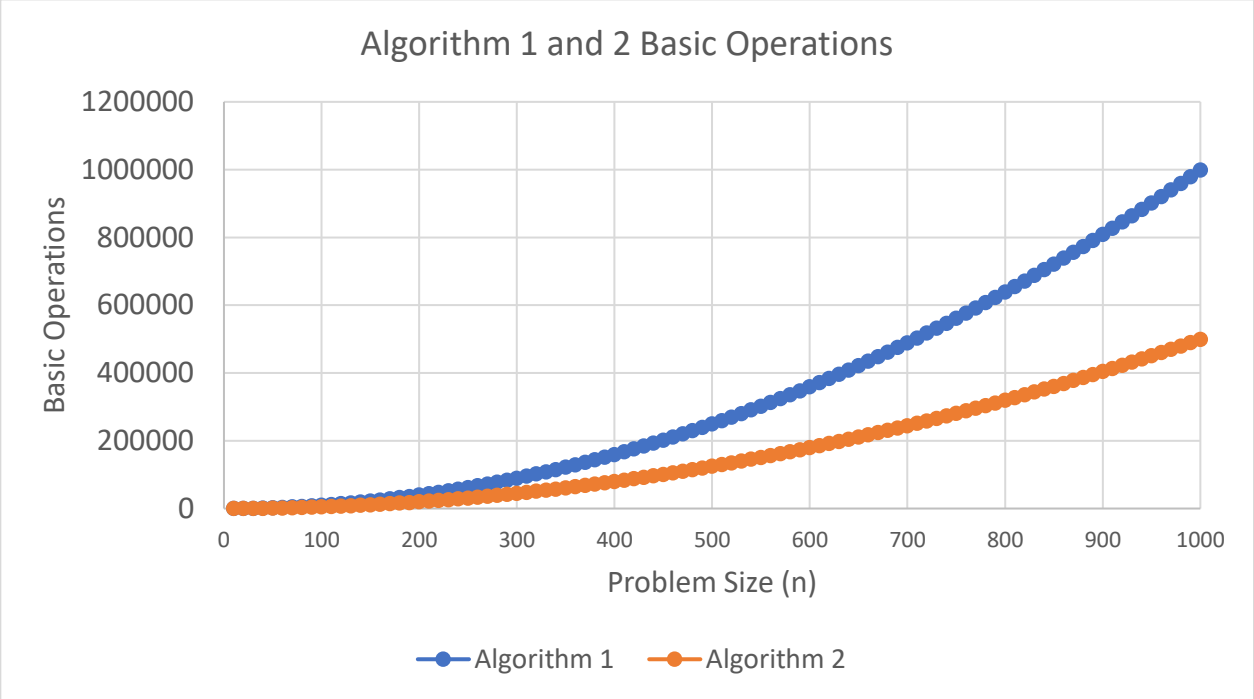

Figure 25 – Basic operations algorithms one and two

Appendix M

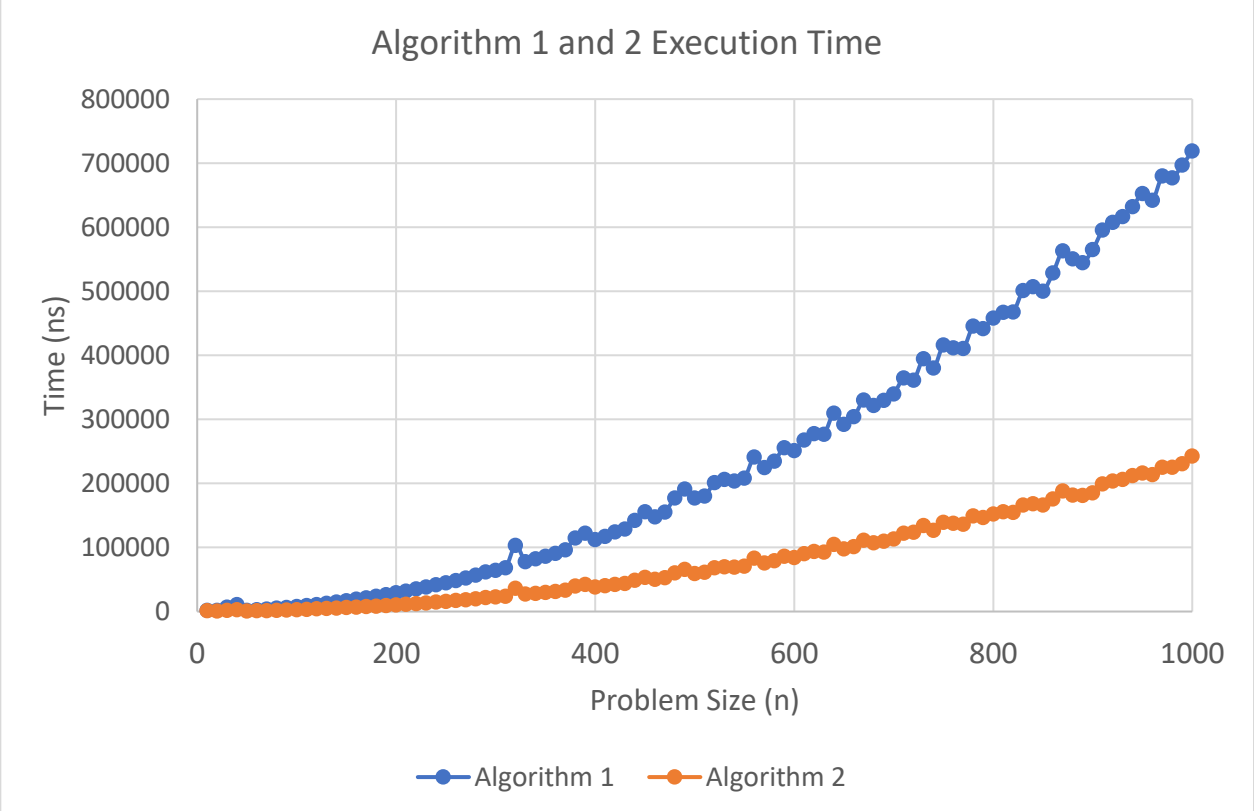

Figure 26 – Time execution algorithms one and two