CAB301 Algorithms and Complexity

Assignment 2 – Empirical Comparison of Two Algorithms

Due: Sunday, 19th May 2019

Weight: 30%

Student Name: Kwun Hyo Lee

Student Number: N9748482

Table of Contents

1.0 Introduction....3

2.0 Description of the Algorithms....3

2.1 *MinDistance*(*A*[0..*n* - 1]) Algorithm....3

2.2 *MinDistance2*(*A*[0..*n* - 1]) Algorithm....5

3.0 Theoretical Analysis of the Algorithms....6

3.1 Choice of Basic Operation and Input Size....6

3.2 Average-Case Efficiency....6

3.3 Order of Growth....7

4.0 Implementation of the Algorithms....8

4.1 Algorithm Analysis Methodology, Tools, and Techniques Selection....8

4.2 Algorithm Implementation in C#....9

5.0 Experimental Results....10

5.1 Testing Functionality of Algorithms....10

5.2 Analysis by Basic Operation Count....11

5.2.1 *MinDistance*(*A*[0..*n* - 1]) Algorithm....11

5.2.2 *MinDistance2*(*A*[0..*n* - 1]) Algorithm....14

5.2.3 *MinDistance*(*A*[0..*n* - 1]) and *MinDistance2*(*A*[0..*n* - 1]) Comparison....17

5.3 Analysis by Measuring Execution Time....18

5.3.1 *MinDistance*(*A*[0..*n* - 1]) Algorithm....18

5.3.2 *MinDistance2*(*A*[0..*n* - 1]) Algorithm....19

5.3.3 *MinDistance*(*A*[0..*n* - 1]) and *MinDistance2*(*A*[0..*n* - 1]) Comparison....20

6.0 References....21

7.0 Appendices....22

1.0 Introduction

This report will analyse the *MinDistance*(*A*[0..*n* – 1]) and *MinDistance2*(*A*[0..n – 1]) algorithms (refer to appendix 1 and 2) to determine the time complexity of the given algorithms. The report will discuss the choice of basic operation(s) and the input size, methodology, programming language, and will present an empirical analysis of both algorithms followed by testing. A comparison of efficiency between the two algorithms will also be discussed to provide further insight in which algorithm is more efficient.

2.0 Description of the Algorithms

MinDistance(*A*[0..*n* – 1]) and *MinDistance2*(*A*[0..n – 1]) are algorithms which both calculate the minimum difference between two integers in a given array.

2.1 *MinDistance*(*A*[0..*n* - 1]) Algorithm

The *MinDistance*(*A*[0..*n* – 1]) algorithm manually steps through every potential pair of the given array and determines the difference between the two integers. The smallest difference is determined by reassigning *dmin* when the new difference is smaller than *dmin*. The initial value of *dmin* is set to infinity (or *System.Int32.MaximumValue*) in the algorithm so that *dmin* is always reassigned to the difference between the first pair of integers on the first comparison.

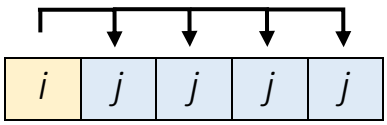
```
Algorithm MinDistance(A[0..n − 1])
// Input: Array A[0..n − 1] of numbers
// Output: Minimum distance between two of its elements
dmin ← ∞
for i ← 0 to n − 1 do
      for j ← 0 to n − 1 do
            if i ≠ j and |A[i] − A[j]| < dmin
                  dmin ← |A[i] − A[j]|
return dmin
```

Figure 1: *MinDistance*(*A*[0..*n* - 1]) Algorithm Pseudocode

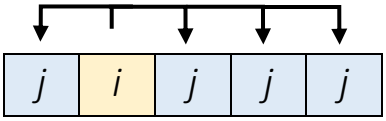
The pseudocode above shows the use of a nested *for* loop to compare an element to the rest of the collection. For each item in the array, the item is always compared with every other item in the collection.

When n = 5,

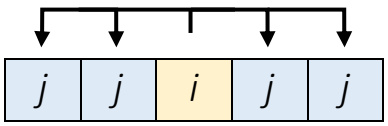
When i = 0,



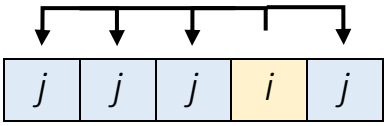
When i = 1,



When i = 2,



When i = 3,



When i = 4,

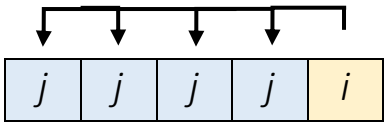


Figure 2: *MinDistance*(*A*[0..*n* - 1]) Algorithm Functionality

As shown in figure 2, the algorithm avoids the situation where the index of the outer loop and the inner loop are equal, as this would result in determining the difference of the same element. Thus, the *if* condition checks and avoids reassigning the *dmin* value when the outer and inner loop indexes are equal.

The *MinDistance*(*A*[0..*n* - 1]) algorithm will finish iterating when the loop counter, *i*, reaches $n - 1$. This is after the last item is compared with all other items of the collection. Assuming the given array has at least a pair of integers, when the algorithm finishes iterating, it will return the newly assigned *dmin* as the minimum difference.

From the functionality of the algorithm, it can be determined that the algorithm is a brute-force algorithm as it calculates every possible outcome for the minimum difference. The algorithm is also evidently unstable, as iterating through every item in the collection does not allow the algorithm to have a best or worst-case efficiency.

2.2 *MinDistance2*(*A*[0..*n* - 1]) Algorithm

The *MinDistance2*(*A*[0..*n* – 1]) algorithm achieves the same outcome as *MinDistance*(*A*[0..*n* – 1]), however, does so in a more efficient approach. While *MinDistance*(*A*[0..*n* – 1]) compares each element with every other element within the collection, *MinDistance2*(*A*[0..*n* – 1]) only compares each element with the elements that follow after it in the collection. This is made possible as the element had already been compared with the previous items of the collection within the algorithm and does not need to be compared again. It is also possible as the algorithm takes the absolute value of the difference, which does not affect finding the difference between two different integers.

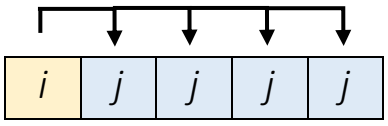
```
Algorithm MinDistance2(A[0..n – 1])
// Input: Array A[0..n – 1] of numbers
// Output: Minimum distance between two of its elements
dmin ← ∞
for i ← 0 to n – 2 do
      for j ← i + 1 to n – 1 do
            temp ← |A[i] – A[j]|
            if temp < dmin
                  dmin ← temp
return dmin
```

Figure 3: *MinDistance2*(*A*[0..*n* - 1]) Algorithm Pseudocode

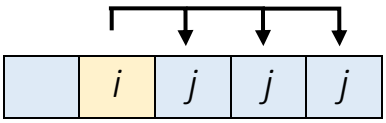
As shown in the pseudocode above, the number of times the nested *for* loop runs is significantly smaller than *MinDistance*(*A*[0..*n* – 1]). The number of times the algorithm iterates reduces by 1 after each iteration as there is one less item to iterate over as shown below.

When *n* = 5,

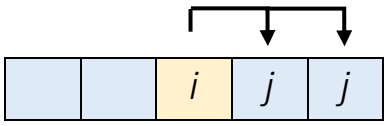
When *i* = 0,



When *i* = 1,



When *i* = 2,



When *i* = 3,

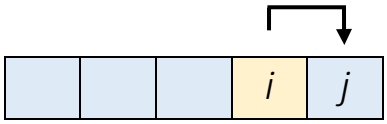


Figure 4: *MinDistance2*(*A*[0..*n* - 1]) Algorithm Functionality

The *MinDistance2*(*A*[0..*n* - 1]) algorithm will finish iterating when the loop counter, *i*, reaches *n* – 2. This is after the second last item is compared with the last item of the collection (refer to figure 4). Assuming the given array has at least a pair of integers, the algorithm will return the reassigned *dmin* as the minimum difference.

3.0 Theoretical Analysis of the Algorithms

3.1 Choice of Basic Operation and Input Size

The basic operation of a given algorithm is defined by the operation which best characterises the efficiency of the algorithm of interest (Tang, 2019). For the *MinDistance*(*A*[0..*n* – 1]) and *MinDistance2*(*A*[0..*n* – 1]) algorithms, the efficiency will be defined by time complexity. Thus, the chosen basic operations will be the conditional statements (in red) within the nested **for** loop as the two comparisons take the largest toll on the execution time.

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

Figure 5: *MinDistance*(*A*[0..*n* – 1]) Basic Operation

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

Figure 6: *MinDistance2*(*A*[0..*n* – 1]) Basic Operation

The chosen input size was the length of the array *n* for both algorithms, as the efficiencies of the given algorithms are dependent on the length of the array.

3.2 Average-Case Efficiency

3.2.1 *MinDistance*(*A*[0..*n* – 1]) Algorithm

The average-case efficiency of the *MinDistance*(*A*[0..*n* – 1]) algorithm can be determined by following the functionality of the algorithm. The algorithm is designed with a nested *for* loop where the basic operation exists within the inner *for* loop as shown in figure 1. As shown in the nested *for* loop, the outer *for* loop is executed *n* – 1 times and the inner *for* loop is also executed *n* – 1 times. However, the basic operation will not run unless it meets the previous condition, $i \neq j$. Now, it is important to consider how many times *i* will be equal to *j* as this is the number of times the basic operation will not be executed. Given an array of length of *n*, it can be determined that the basic operation will not be run *n* times. Thus, the average-case efficiency can be estimated as below,

$$C_{avg}(n) = n(n-1) - n$$

$$= n^2 - n - n$$

$$= n^2 - 2n$$

Figure 7: Average-Case Efficiency of the *MinDistance*(*A*[0..*n* - 1]) Algorithm

The calculations above show a quadratic average-case efficiency class of $C_{avg}(n) \in \theta(n^2)$. The 2*n* can be disregarded in the efficiency class as n^2 has a greater efficiency class than 2*n*. To further prove the accuracy of the calculation above, the average-case efficiency can also be calculated as in appendix 3.

As the *MinDistance*(*A*[0..*n* - 1]) algorithm is unstable, it does not have a best or worst case efficiency and will only have a single efficiency for all arrays of length, *n*.

3.2.2 *MinDistance2*(*A*[0..*n* – 1]) Algorithm

The average-case efficiency of the *MinDistance2*(*A*[0..*n* – 1]) algorithm can be determined by analysing the design of the algorithm. The algorithm consists of a nested *for* loop where the basic operation exists within the inner *for* loop. The outer *for* loop is executed *n* – 2 times, whereas the inner *for* loop executes less frequently as *i* increases. The inner *for* loop would iterate *n* – 1 times when *i* = 0, *n* – 2 times when *i* = 1, *n* – 3 times when *i* = 2, and so on. This would continue until *i* = *n* – *2* where the inner *for* loop would iterate *n* – (*n* – 1) times. Thus, the average-case efficiency can be calculated as shown below,

$$C_{avg}(n) = (n-1) + (n-2) + (n-3) + \ \dots + \big(n-(n-1)\big)$$

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

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

Figure 8: Average-Case Efficiency of the *MinDistance2*(*A*[0..*n* - 1]) Algorithm

While the average-case efficiency class of *MinDistance2*(*A*[0..*n* – 1]) is also quadratic ($C_{avg}(n) \in \theta(n^2)$), the average-case efficiency is approximately half of the efficiency determined for *MinDistance*(*A*[0..*n* – 1]).

As *MinDistance2*(*A*[0..*n* – 1]) is also an unstable algorithm, a worst and best case efficiency is unavailable and will have a single efficiency for any given array of length, *n*.

3.3 Order of Growth

As both algorithms were determined to have an efficiency of $\theta(n^2)$, the expected order of growth of the algorithms is quadratic, with the efficiency being dependant on the length of the given array, *n*. However, despite both algorithms having the same order of growth, *MinDistance2*(*A*[0..*n* – 1]) was determined to be approximately twice as efficient than *MinDistance*(*A*[0..*n* – 1]) (refer to figures 7 and 8).

4.0 Implementation of the Algorithms

4.1 Algorithm Analysis Methodology, Tools, and Technique Selection

1. The analysis and testing of both algorithms were achieved with the C# programming language. C# is a general-purpose, multi-paradigm programming language encompassing strong typing, lexically scoped, imperative, declarative, functional, generic, object-oriented, and component-oriented programming disciplines (Introduction to the C# Language and the .NET Framework, 2015).
2. The testing was performed on an Acer Spin SP513-51 laptop, running Microsoft Windows 10 Home, with an Intel Core i7-7500U CPU @ 2.70GHz. The randomness of the testing was achieved via the Next() function of the Random class, seeded based on the current time (DateTime.Now.Ticks). The Stopwatch class was also used to measure an accurate execution time of the algorithm in milliseconds. While measuring execution times, counters were removed from the algorithms to avoid any unnecessary operations that would risk the validity of the end results. The number of processes open during the execution time testing was also minimised to further ensure accuracy (Lee, 2019).
3. The final results of the algorithm testing were generated from the average of 100 permutations of unique arrays where *n* had ranged from 0 to 20 000. The integers within the arrays had ranged from 0 to Int32.MaxValue (2 147 483 647). For testing, the data was collected in increments of 1000 (Lee, 2019).

4.2 Algorithm Implementation in C#

Figure 9 and 10 shows the implementation of the *MinDistance*(*A*[0..*n* - 1]) and *MinDistance2*(*A*[0..*n* - 1]) algorithms in C#.

```
// The MinDistance() and MinDistance2() algorithms calculate the
minimum difference
// between any two elements in a given array, A.
public int MinDistance(int[] A)
{
    // Input: Array A[0..n - 1] of numbers
    // Output: Minimum distance between two of its elements
    int dmin = int.MaxValue;
    for (int i = 0; i < A.Length; i++)
    {
        for (int j = 0; j < A.Length; j++)
        {
            if ( (i != j) && (Math.Abs(A[i] - A[j]) < dmin) )
            {
                dmin = Math.Abs(A[i] - A[j]);
            }
        }
    }
    return dmin;
}
```

Figure 9: C# Implementation of the *MinDistance*(*A*[0..*n* - 1]) Algorithm

```
public int MinDistance2(int[] A)
{
    // Input: Array A[0..n - 1] of numbers
    // Output: Minimum distance between two of its elements
    int dmin = int.MaxValue;
    for (int i = 0; i < A.Length - 1; i++)
    {
        for (int j = i + 1; j < A.Length; j++)
        {
            int temp = Math.Abs(A[i] - A[j]);
            if (temp < dmin)
            {
                dmin = temp;
            }
        }
    }
    return dmin;
}
```

Figure 10: C# Implementation of the *MinDistance2*(*A*[0..*n* - 1]) Algorithm

5.0 Experimental Results

5.1 Testing Functionality of Algorithms

The table below presents the functionality testing of the *MinDistance*(*A*[0..*n* - 1]) and *MinDistance2*(*A*[0..*n* - 1]) algorithms. Several unique arrays were used to observe the outcome as shown below. An outcome was hypothesised according to the logic of the algorithms and the actual output was recorded, to prove whether the algorithms were implemented as intended.

Test Case	Test Instance	Expected Output	MinDistance Actual Output	MinDistance2 Actual Output	Test Result
Empty Array	A = { }	2 147 483 647	2 147 483 647	2 147 483 647	PASS
Single Element Array	A = { 1 }	2 147 483 647	2 147 483 647	2 147 483 647	PASS
Single Pair Array	A = { 1, 2 }	1	1	1	PASS
Sorted Array	A = { 1, 3, 7, 8, 17, 22, 24, 30, 42, 77 }	1	1	1	PASS
Unsorted Array	A = { 7, 1, 24, 30, 22, 17, 8, 3, 42, 77 }	1	1	1	PASS
Negative Integers Array	A = { -1, -3, -7, -8, -17, -22, -24 }	1	1	1	PASS
Negative and Positive Integers Array	A = { 1, 3, -7, 9, 17, 22, -24 }	2	2	2	PASS

Figure 11: Functionality Testing of the *MinDistance*(*A*[0..*n* - 1]) and *MinDistance2*(*A*[0..*n* - 1]) Algorithms

Figure 11 contains the test case, test instance, the expected output, actual output from both algorithms, and the testing result. The corresponding tests can be seen in appendix 5 and show that the algorithms were indeed implemented as intended.

According to the pseudocode provided in appendix 1 and 2, the empty array was expected to provide an output of 2 147 483 647 as the *for* condition would not be executed in both algorithms. The *for* condition would not be met as the loop counter, $i = 0$, would already be greater than $n - 1$ (*MinDistance*) and $n - 2$ (*MinDistance2*) since n would equal 0. Thus, the *dmin* would not be reassigned and therefore, the algorithm would provide *System.Int32.MaxValue* as the output.

For an array containing only a single element, the algorithms would produce an output of 2 147 483 647 as the *for* condition would again, not be executed. This is due to the loop counter, $i = 0$, from the *for* loop condition already being greater than or equal to $n - 1$ (*MinDistance*) and $n - 2$ (*MinDistance2*).

The rest of the given tests in figure 11 provides the expected results according to the logic of the algorithms. Thus, it is safe to assume that the pseudocode in appendices 1 and 2 had been implemented correctly.

5.2 Analysis by Basic Operation Count

A counter was used to measure the efficiency of the *MinDistance*(*A*[0..*n* - 1]) algorithm. This was achieved by incrementing the counter for every time the basic operation was executed. The counter was placed as shown in appendices 6 and 8. The analysis below presents the hypothesised efficiency and the actual efficiency via using a counter.

5.2.1 *MinDistance*(*A*[0..*n* - 1]) Algorithm

The figure below represents the hypothetical empirical analysis of the *MinDistance*(*A*[0..*n* - 1]) algorithm.

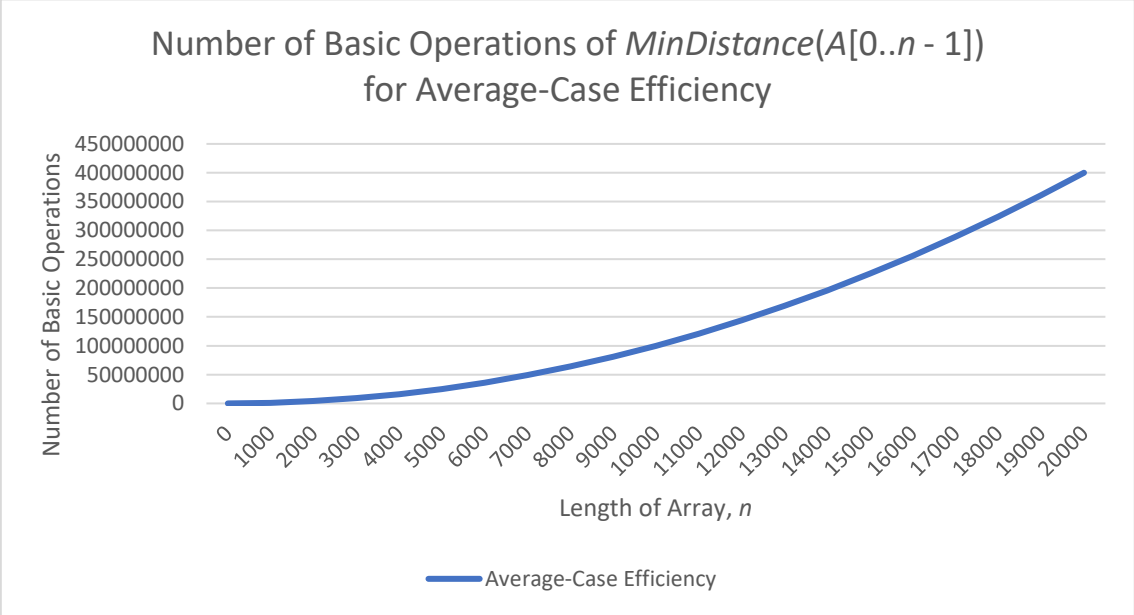


Figure 12: *MinDistance*(*A*[0..*n* - 1]) Algorithm Expected Average-Case Efficiency by Counter

The efficiency class of $C_{avg}(n) = n^2 - 2n$ (refer to figure 7) was used to calculate the graph above. The graph presents the quadratic efficiency of the algorithm as previously hypothesised, where the *x-axis* represents the length of the given array, *n*, and the *y-axis* represents the number of times the basic operation is executed.

It must be considered that the graph provided above is only an approximation given the efficiency calculated in figure 7. The actual efficiency should deviate slightly, however, should mostly stay faithful to the hypothesised efficiency as the algorithm is unstable.

To create a graph which portrays the results obtained in appendix 7, an efficiency class was calculated using the given test results.

The graph below was achieved by calculating the average number of basic operations performed in 100 permutations of an array of *n* = 0 to 20 000.

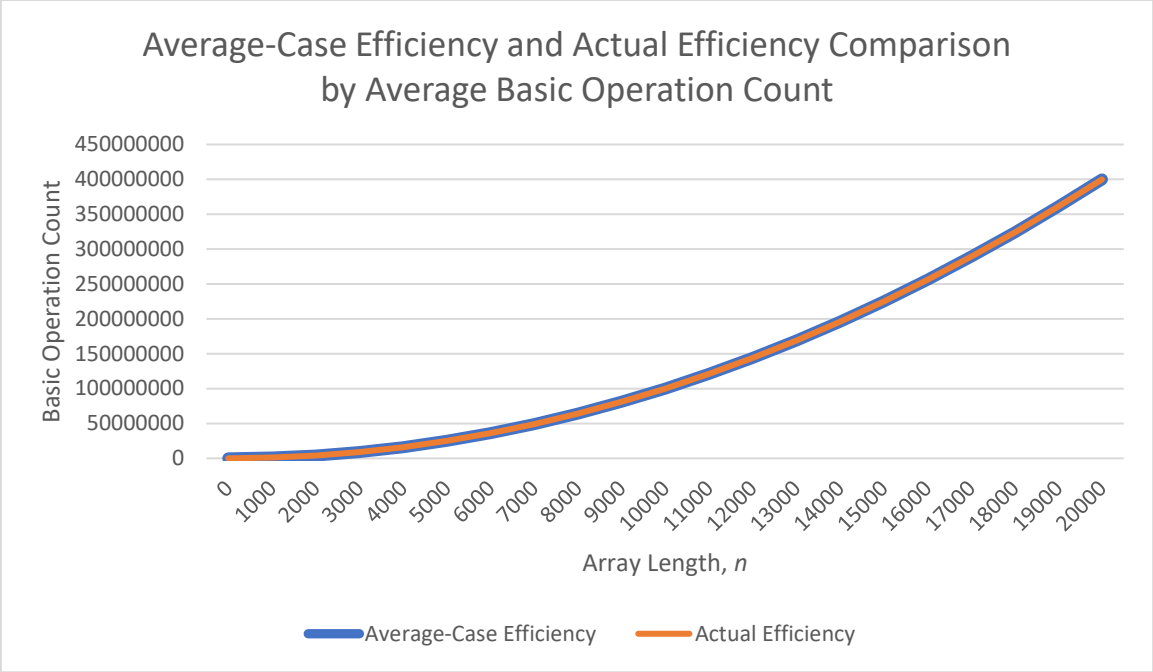


Figure 13: Average-Case and Actual Efficiency Comparison for *MinDistance*(*A*[0..*n* - 1]) Algorithm by Counter

The test results produced in appendix 7 is depicted in figure 13 as the actual efficiency line. The average-case efficiency line from the graph was calculated by hypothesizing a likely efficiency class from the generated data as shown below,

Assuming an efficiency class of n^2,

When $n = 1000, \quad operationCount = 999\,000,$

$$cn^2 = operationCount$$

$$c = \frac{operationCount}{n^2}$$

$$c = \frac{999\,000}{1000^2}$$

$$= 0.999$$

Thus,

$$\in \theta(0.999n^2)$$

Figure 14: *MinDistance*(*A*[0..*n* - 1]) Algorithm Average-Case Efficiency Calculation for *n* = 1000

The mean average efficiency class was calculated by finding the average of every efficiency class from *n* = 1000 to 20 000, as shown below,

From $n = 1000\ to\ 20\ 000$,

$$C_{mean} = \left(\sum_{\substack{i=1000 \\ Step\ by\ 1000}}^{20\ 000} C_i \right) / numOfSteps$$

$$C_{mean} = 0.99982$$

Figure 15: *MinDistance*(*A*[0..*n* - 1]) Algorithm Mean Average-Case Efficiency Class for Basic Operation Count Analysis

Where C_{mean} is the mean C value and C_i is the C value of the current array permutation, i.

As shown in figure 15, an efficiency class of $\theta(0.99982n^2)$ was calculated for the average-case efficiency. The figure below further confirms the accuracy of the calculated efficiency class by calculating the basic operations performed when *n* = 10 000, 15 000, and 20 000.

When $n = 10\ 000$, $operationCount = 99\ 982\ 000$,

$$operationCount = cn^2$$
$$= 0.999 \cdot 10\ 000^2$$
$$= 99\ 900\ 000$$
$$\approx 99\ 982\ 000$$

When $n = 15\ 000$, $operationCount = 224\ 959\ 500$,

$$operationCount = cn^2$$
$$= 0.999 \cdot 20\ 000^2$$
$$= 224\ 775\ 000$$
$$\approx 224\ 959\ 500$$

When $n = 20\ 000$, $operationCount = 399\ 928\ 000$,

$$operationCount = cn^2$$
$$= 0.999 \cdot 20\ 000^2$$
$$= 399\ 600\ 000$$
$$\approx 399\ 928\ 000$$

Figure 16: Estimated Basic Operation Count for Arrays of *n* = 10 000, 15 000, and 20 000

In the calculations above, all the calculated number of basic operation counts vary only slightly from the hypothesised counts. Thus, the accuracy of the newly calculated efficiency class of $\theta(0.99982n^2)$ can be confirmed.

As presented in figure 13, it is clear that the actual efficiency accurately represents the hypothesised average-case efficiency. The lack of deviation in the actual efficiency may be due to the algorithm being unstable, as an unstable algorithm only provides a single potential efficiency.

5.2.2 *MinDistance2*(*A*[0..*n* - 1]) Algorithm

The figure below represents the hypothetical empirical analysis of the *MinDistance2*(*A*[0..*n* - 1]) algorithm.

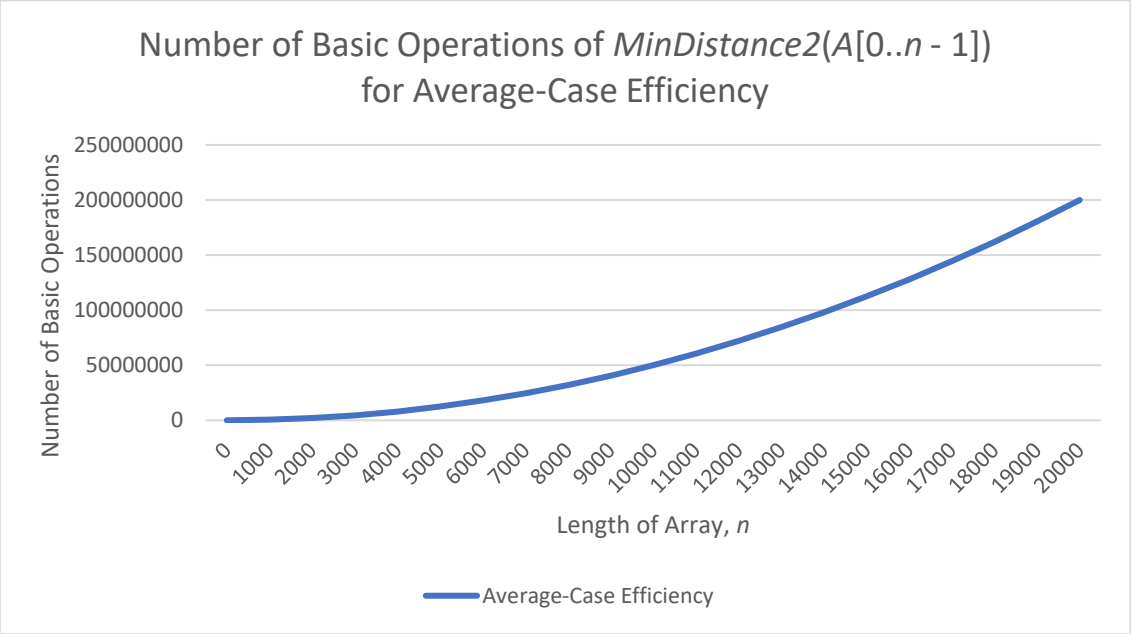


Figure 17: MinDistance2(A[0..n - 1]) Algorithm Expected Average-Case Efficiency by Counter

The efficiency class of $C_{avg}(n) = (n^2 - n)/2$ (refer to figure 8) was used to calculate the graph above. The graph again, presents the quadratic efficiency of n^2 as previously hypothesised, where the *x-axis* of the graph represents the length of the given array, *n*, and the *y-axis* represents the number of times the basic operation is executed.

The graph below presents a comparison between the basic operation count of the average-case efficiency and the actual efficiency for the MinDistance2(A[0..n – 1]) algorithm.

Figure 18 was achieved by calculating the average number of basic operations performed in 100 permutations of an array of *n* = 0 to 20 000.

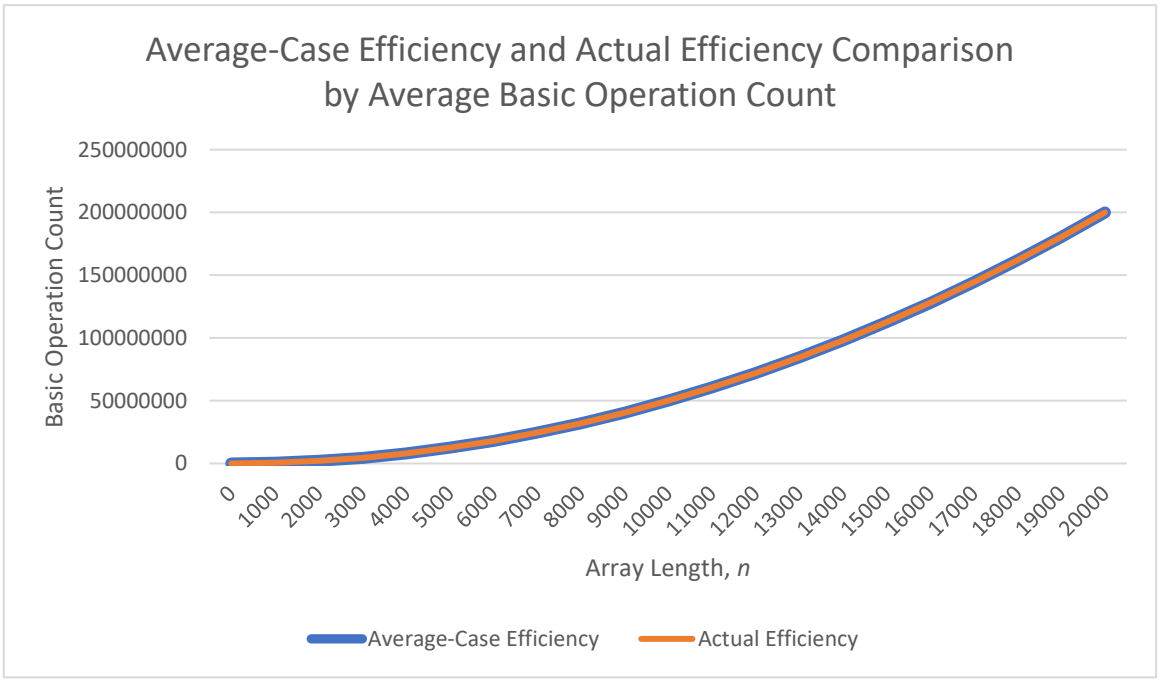


Figure 18: Basic Operation Efficiency Comparison Between the Average-Case and Actual Efficiency of the MinDistance2(A[0..n – 1]) Algorithm

The calculations for the average-case efficiency line in figure 18 are shown as below,

Assuming an efficiency class of n^2,

When $n = 1000, \quad operationCount = 499\,500,$

$$cn^2 = operationCount$$

$$c = \frac{operationCount}{n^2}$$

$$c = \frac{499\,500}{1000^2}$$

$$= 0.4995$$

Thus,

$$\in \theta(0.4995n^2)$$

Figure 19: *MinDistance2*(*A*[0..*n* - 1]) Algorithm Average-Case Efficiency Calculation for *n* = 1000

The mean average efficiency class was calculated by finding the average of every efficiency class from *n* = 1000 to 20 000, as shown below,

From $n = 1000\ to\ 20\ 000$,

$$C_{mean} = \left(\sum_{\substack{i=1000 \\ Step\ by\ 1000}}^{20\ 000} C_i \right) / numOfSteps$$

$$C_{mean} = 0.49991$$

Figure 20: *MinDistance2*(*A*[0..*n* - 1]) Algorithm Mean Average-Case Efficiency Class for Basic Operation Count Analysis

Where C_{mean} is the mean C value and C_i is the C value of the current array permutation, i.

As shown in figure 20, an efficiency class of $\theta(0.49991n^2)$ was calculated for the average-case efficiency and was used for figure 18. The calculations below further confirms the accuracy of the calculated efficiency class by calculating the basic operations performed when *n* = 10 000, 15 000, and 20 000.

When $n = 10\ 000$, $operationCount = 49\ 995\ 000$,

$$operationCount = cn^2$$
$$= 0.49991 \cdot 10\ 000^2$$
$$= 49\ 991\ 000$$
$$\approx 49\ 995\ 000$$

When $n = 15\ 000$, $operationCount = 112\ 492\ 500$,

$$operationCount = cn^2$$
$$= 0.49991 \cdot 20\ 000^2$$
$$= 112\ 479\ 750$$
$$\approx 112\ 492\ 500$$

When $n = 20\ 000$, $operationCount = 199\ 990\ 000$,

$$operationCount = cn^2$$
$$= 0.49991 \cdot 20\ 000^2$$
$$= 199\ 964\ 000$$
$$\approx 199\ 990\ 000$$

Figure 21: Estimated Basic Operation Count for Arrays of *n* = 10 000, 15 000, and 20 000

5.2.3 *MinDistance*(*A*[0..*n* - 1]) and *MinDistance2*(*A*[0..*n* - 1]) Efficiency Comparison

The graph below presents the comparison of the basic operation counts between the *MinDistance*(*A*[0..*n* - 1]) and *MinDistance2*(*A*[0..*n* - 1]) algorithms.

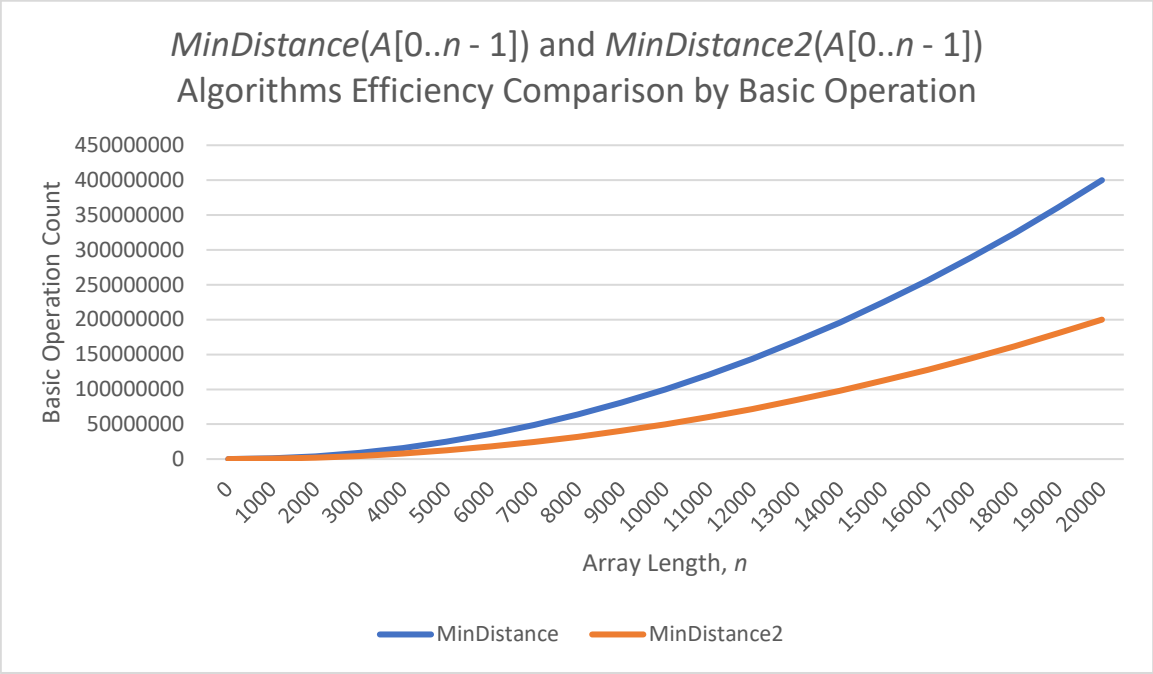


Figure 22: Basic Operation Efficiency Comparison Between *MinDistance*(*A*[0..*n* – 1]) and *MinDistance2*(*A*[0..*n* – 1]) Algorithms

As shown above, both algorithms show a predicted quadratic rate of growth. The *MinDistance2*(*A*[0..*n* - 1]) algorithm is evidently more efficient than the *MinDistance*(*A*[0..*n* - 1]) as it requires less basic operation counts to complete the algorithm. The graph also confirms the hypothesis stated in section 3.2.2, as the *MinDistance2*(*A*[0..*n* - 1]) algorithm requires approximately half the number of basic operations executed in the *MinDistance*(*A*[0..*n* - 1]) algorithm (refer to figure 23).

$$When\ n = 1000,$$

$$= \frac{MinDistance2(A[0..n-1])\ Operation\ Count}{MinDistance(A[0..n-1])\ Operation\ Count}$$

$$= \frac{499\,910}{999\,810}$$

$$= \frac{49\,991}{99\,981} = 0.500005$$

$$\approx \frac{1}{2}$$

$$When\ n = 20\,000,$$

$$= \frac{MinDistance2(A[0..n-1])\ Operation\ Count}{MinDistance(A[0..n-1])\ Operation\ Count}$$

$$= \frac{199\,964\,000}{399\,924\,000}$$

$$= \frac{49\,991}{99\,981} = 0.500005$$

$$\approx \frac{1}{2}$$

Figure 23: Efficiency Comparison Between *MinDistance*(*A*[0..*n* – 1]) and *MinDistance2*(*A*[0..*n* – 1]) at *n* = 1000 and 20 000

Thus, it can be concluded that the *MinDistance2*(*A*[0..*n* - 1]) algorithm is approximately twice as efficient than the *MinDistance*(*A*[0..*n* - 1]) algorithm in terms of time complexity when using a basic operation counter.

5.3 Analysis by Measuring Execution Time

The execution times of the algorithms were also measured to determine the time complexity efficiency of both algorithms. The analysis below presents a comparison between the hypothesised and actual efficiency, as well as a comparison of the efficiency of both algorithms in terms of execution time.

5.3.1 *MinDistance*(*A*[0..*n* - 1]) Algorithm

The graph below presents the comparison of the execution time efficiency between the hypothesised and actual efficiency. The results were achieved via performing the *MinDistance*(*A*[0..*n* - 1]) algorithm in 100 permutations of an array of n = 0 to 20 000.

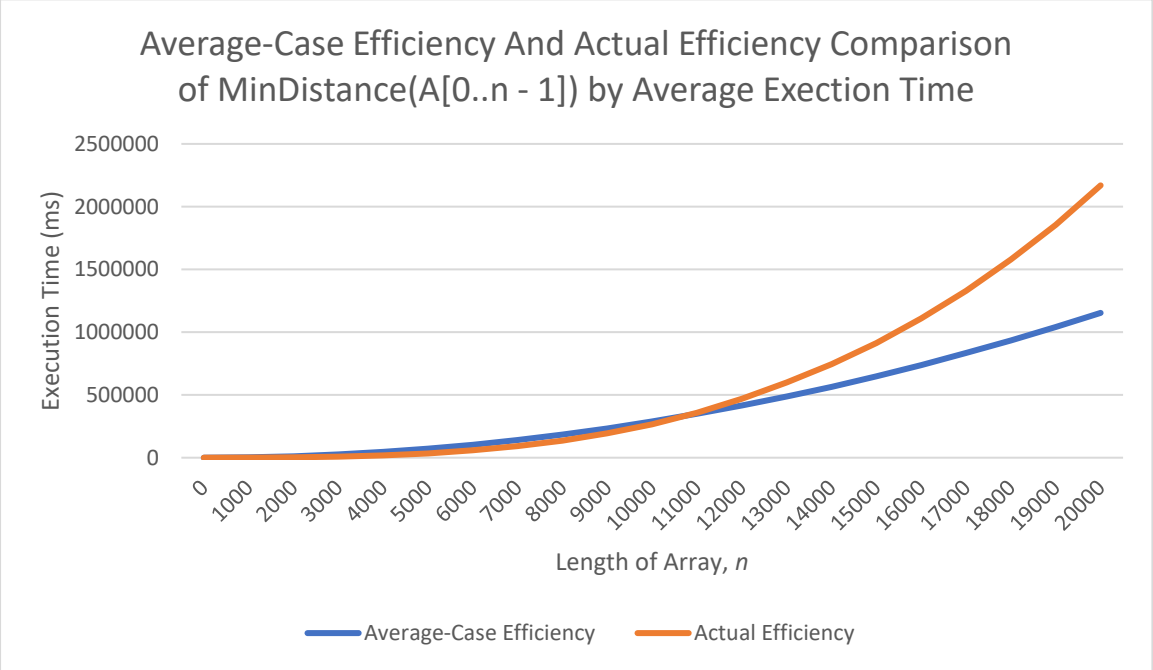


Figure 24: Execution Time Efficiency Comparison Between the Average-Case and Actual Efficiency of the MinDistance(A[0..n – 1]) Algorithm

The calculations below show the calculated mean efficiency class for the average-case efficiency line in figure 24.

From $n = 1000\ to\ 20\ 000$,

$$C_{mean} = \left(\sum_{\substack{i=1000 \\ Step\ by\ 1000}}^{20\ 000} C_i \right) / numOfSteps$$

$$C_{mean} = 0.002884$$

Figure 25: *MinDistance*(*A*[0..*n* - 1]) Algorithm Mean Average-Case Efficiency Class for Execution Time Analysis

In figure 24, the actual efficiency line is depicted from the execution time analysis given in appendix 11.

The graph demonstrates in the actual efficiency, the hypothesised quadratic rate of growth based on the length of the given array, *n*. However, when compared to the average-case efficiency, the actual efficiency deviates significantly when *n* exceeds 11 000. Before *n* = 11 000, the average-case and actual efficiency share close similarities, however, the actual efficiency shows a steeper slope past *n* = 11 000. It can be seen in figure 24 that as *n* increases, the difference between the execution time of the average-case and actual efficiency also increases.

The difference in both efficiencies can be caused by several factors. These include the testing device, CPU utilisation, and language choice. The testing device is according to section 4.1 and is a laptop, which may prioritize battery life over performance. The number of processes running may have also affected the performance of the algorithm as it takes up storage and RAM. The algorithm was also implemented in C#, which may provide slower results than an algorithm implemented in a lower-level language such as C (Lee, 2019).

5.3.2 *MinDistance2*(*A*[0..*n* - 1]) Algorithm

The graph below represents an execution time comparison between the hypothesised and actual efficiency of the *MinDistance2*(*A*[0..*n* - 1]) algorithm. The results were achieved via performing the *MinDistance2*(*A*[0..*n* - 1]) algorithm in 100 permutations of an array from *n* = 0 to 20 000.

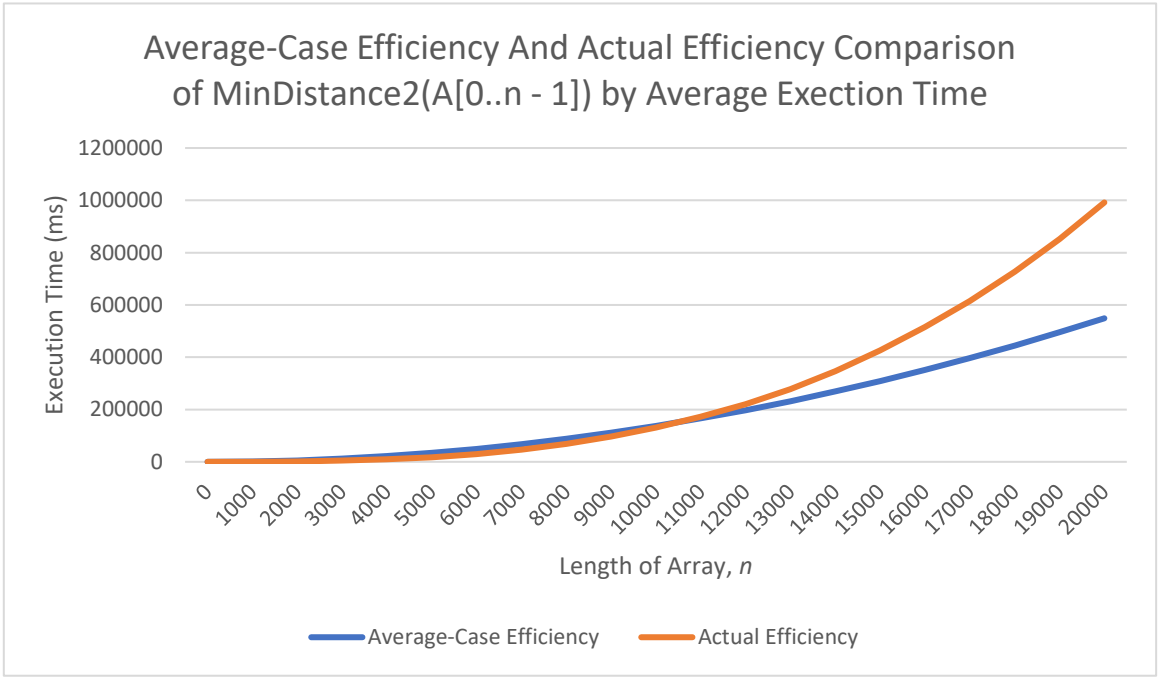


Figure 26: Execution Time Efficiency Comparison Between the Average-Case and Actual Efficiency of the *MinDistance2*(*A*[0..*n* – 1]) Algorithm

The actual efficiency line of the *MinDistance2*(*A*[0..*n* – 1]) shows a similar trend as shown in figure 24. In both graphs, the actual efficiency deviates and becomes steeper than the average-case efficiency at *n* = 11 000. Both graphs also show that as the length of the array, *n*, increases, the difference between the execution time of the average-case and actual efficiency also increases.

Once again, these inaccuracies in the actual efficiency may be due to the factors discussed previously. Such factors include the testing device, CPU utilisation, and language choice (refer to section 5.3.1).

5.3.3 *MinDistance*(*A*[0..*n* - 1]) and *MinDistance2*(*A*[0..*n* - 1]) Efficiency Comparison

The graph below presents a comparison in execution time between the *MinDistance*(*A*[0..*n* - 1]) and *MinDistance2*(*A*[0..*n* - 1]) algorithms.

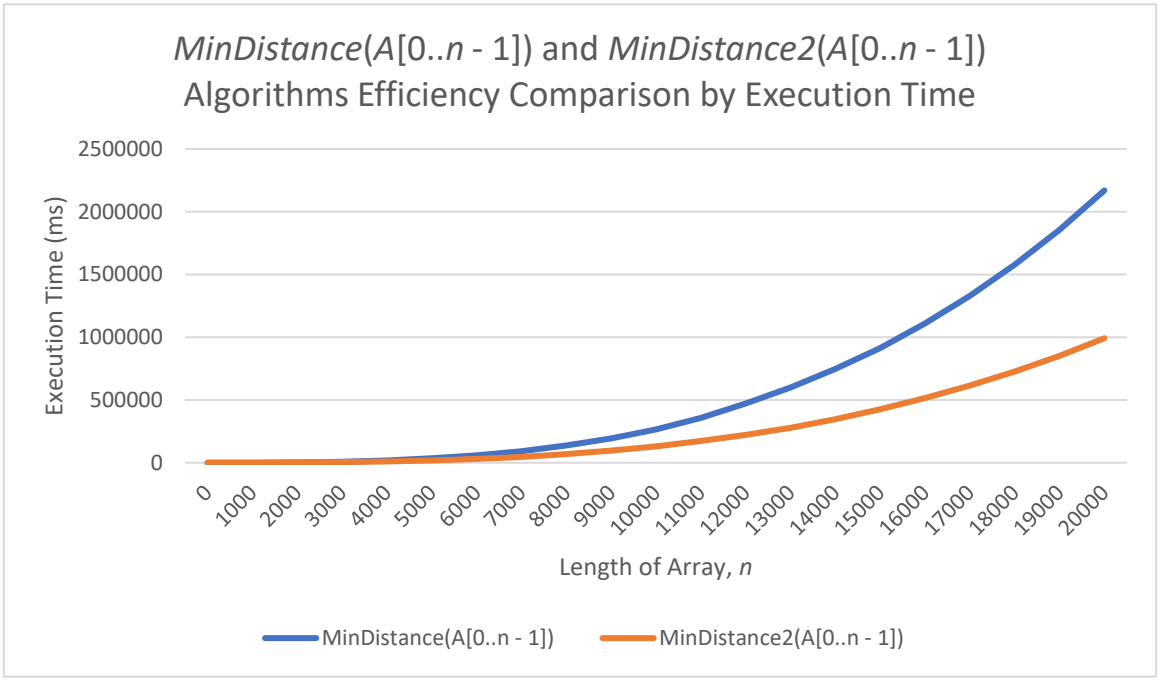


Figure 27: Execution Time Efficiency Comparison Between *MinDistance*(*A*[0..*n* – 1]) and *MinDistance2*(*A*[0..*n* – 1]) Algorithms

Figure 27 shows a clear depiction of the difference in efficiencies between the two algorithms, where the algorithm which takes less time is more time efficient.

Despite the inaccuracies given in figures 24 and 26, figure 27 still shows a predicted quadratic efficiency. It can also be observed that *MinDistance2*(*A*[0..*n* - 1]) is approximately twice as more time efficient than *MinDistance*(*A*[0..*n* - 1]) as hypothesised in section 3.2.2.

As both figures 22 and 27 show the same trends in efficiency in both the *MinDistance*(*A*[0..*n* - 1]) and *MinDistance2*(*A*[0..*n* - 1]) algorithms, it can be safely concluded that the graphs show an accurate representation of the hypothesised quadratic efficiency. Thus, it can be said that the analysis, testing, and calculations for both algorithms confirm the accuracy of the determined basic operation, input size, and analysis methodology.

6.0 References

Introduction to the C# Language and the .NET Framework. (2015, 07 20). Retrieved from Microsoft: https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework

Lee, K. (2019). *CAB301 Assignment - Empirical Analysis of an Algorithm.*

Tang, M. (2019). *CAB301 Lecture 2.*

7.0 Appendices

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 $dmin$

Appendix 1: *MinDistance*(*A*[0..*n* – 1]) Pseudocode

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 $dmin$

Appendix 2: *MinDistance2*(*A*[0..*n* – 1]) Pseudocode

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1$$

$$Using \sum_{i=l}^{u} 1 = u - l + 1, \qquad for\ l \leq u$$

$$= \sum_{i=0}^{n-1}\big((n-1)-0+1\big)$$

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

$$= n \cdot \big((n-1)-0+1\big)$$

$$= n^2$$

$$\therefore \quad M(n) \in \theta(n^2)$$

Appendix 3: Summation Equation *M*(*n*) for the *MinDistance*(*A*[0..*n* – 1]) Algorithm

$$M_2(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$Using \sum_{i=l}^{u} 1 = u - l + 1, \qquad for\ l \leq u$$

$$= \sum_{i=0}^{n-2} \big((n-1) - (i+1) + 1\big)$$

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

$$= \sum_{i=0}^{(n-1)-1} \big((n-1) - i\big)$$

$$Using \sum_{i=0}^{u-1} (u - i) = \frac{u(u+1)}{2},$$

$$= \frac{(n-1)\big((n-1)+1\big)}{2}$$

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

$$\therefore\ M_2(n) \in \theta(n^2)$$

Appendix 3: Summation Equation $M_2(n)$ for the *MinDistance2*(*A*[0..*n* – 1]) Algorithm

```
public void TestFunction()
{
    int[] EmptyArray = { };
    int[] SingleElementArray = { 1 };
    int[] SinglePairArray = { 1, 2 };
    int[] SortedArray = { 1, 3, 7, 8, 17, 22, 24, 30, 42, 77 };
    int[] UnsortedArray = { 7, 1, 24, 30, 22, 17, 8, 3, 42, 77 };
    int[] NegativeIntegersArray = { -1, -3, -7, -8, -17, -22, -24 };
    int[] NegativeAndPositiveIntegersArray = { 1, 3, -7, 9, 17, 22, -24 };

    // Testing Empty Array
    Console.WriteLine("Testing Empty Array...");
    Console.WriteLine("MinDistance: " + test.MinDistance(EmptyArray));
    Console.WriteLine("MinDistance2: " + test.MinDistance2(EmptyArray));
    Console.WriteLine();


        ... // REFER TO THE CODE SOLUTION FOR THE REST OF THE C# CODE

}
```

Appendix 4: C# Implementation of the *MinDistance*(*A*[0..*n* – 1]) and *MinDistance2*(*A*[0..*n* – 1]) Algorithms for Functional Testing (Uses Code from Figure 9)

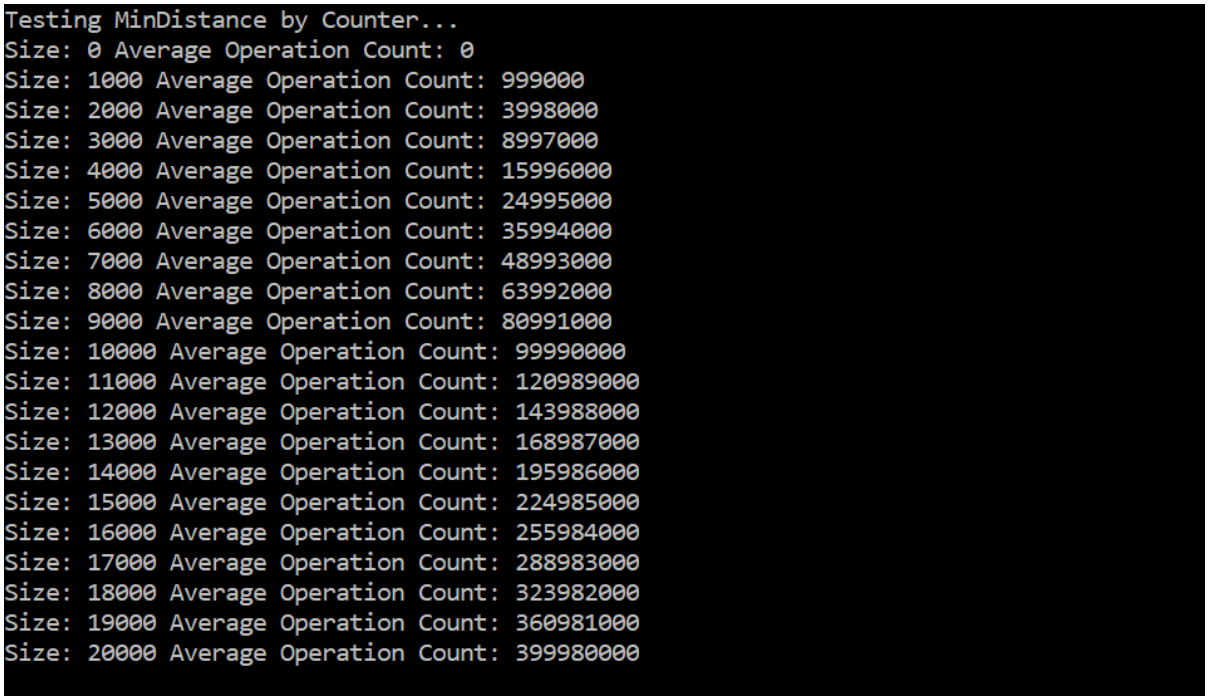
```
Testing Empty Array...
MinDistance: 2147483647
MinDistance2: 2147483647

Testing Single Element Array...
MinDistance: 2147483647
MinDistance2: 2147483647

Testing Single Pair Array...
MinDistance: 1
MinDistance2: 1

Testing Sorted Array...
MinDistance: 1
MinDistance2: 1

Testing Unsorted Array...
MinDistance: 1
MinDistance2: 1

Testing Negative Integers Array...
MinDistance: 1
MinDistance2: 1

Testing Negative and Positive Integers Array...
MinDistance: 2
MinDistance2: 2
```

Appendix 5: Functional Testing for the *MinDistance*(*A*[0..*n* – 1]) and *MinDistance2*(*A*[0..*n* – 1]) Algorithms

```
public int MinDistanceCounter(int[] A)
{
    // Input: Array A[0..n - 1] of numbers.
    // Output: Integer indicating the number of times the basic operation is run
    int c = 0;

    int dmin = int.MaxValue;
    for (int i = 0; i < A.Length; i++)
    {
        for (int j = 0; j < A.Length; j++)
        {
            if ( (i != j) && (++c >= 0) && (Math.Abs(A[i] - A[j]) < dmin) )
            {
                dmin = Math.Abs(A[i] - A[j]);
            }
        }
    }
    return c;
}
```

```
public void TestCounter(Func<int[], int> MinDistance)
{
    for (int size = 0; size < 20000 + 1; size += 1000)
    {
        long c_total = 0;
        double c_average = 0;
        for (int i = 0; i < numOfArrays; i++)
        {
            int[] A = GenerateRandomArray(size);
            int count = MinDistance(A);
            c_total = c_total + count;
        }
        c_average = c_total / numOfArrays;
        Console.WriteLine("Size: " + size + " " + "Average Operation Count: " + c_average);
    }
}
```

Appendix 6: C# Implementation of the *MinDistance*(*A*[0..*n* – 1]) Algorithm for Basic Operation Count Testing



Appendix 7: *MinDistance*(*A*[0..*n* – 1]) Algorithm Basic Operation Count Testing

```
public int MinDistance2Counter(int[] A)
{
    // Input: Array A[0..n - 1] of numbers
    // Output: Minimum distance between two of its elements
    int c = 0;

    int dmin = int.MaxValue;
    for (int i = 0; i < A.Length - 1; i++)
    {
        for (int j = i + 1; j < A.Length; j++)
        {
            int temp = Math.Abs(A[i] - A[j]);
            if ( (++c >= 0) && (temp < dmin) )
            {
                dmin = temp;
            }
        }
    }
    return c;
}
```

Appendix 8: C# Implementation of the *MinDistance2*(*A*[0..*n* – 1]) Algorithm for Basic Operation Count Testing (Refer to Appendix 6 for Testing Code)

```
Testing MinDistance2 by Counter...
Size: 0 Average Operation Count: 0
Size: 1000 Average Operation Count: 499500
Size: 2000 Average Operation Count: 1999000
Size: 3000 Average Operation Count: 4498500
Size: 4000 Average Operation Count: 7998000
Size: 5000 Average Operation Count: 12497500
Size: 6000 Average Operation Count: 17997000
Size: 7000 Average Operation Count: 24496500
Size: 8000 Average Operation Count: 31996000
Size: 9000 Average Operation Count: 40495500
Size: 10000 Average Operation Count: 49995000
Size: 11000 Average Operation Count: 60494500
Size: 12000 Average Operation Count: 71994000
Size: 13000 Average Operation Count: 84493500
Size: 14000 Average Operation Count: 97993000
Size: 15000 Average Operation Count: 112492500
Size: 16000 Average Operation Count: 127992000
Size: 17000 Average Operation Count: 144491500
Size: 18000 Average Operation Count: 161991000
Size: 19000 Average Operation Count: 180490500
Size: 20000 Average Operation Count: 199990000
```

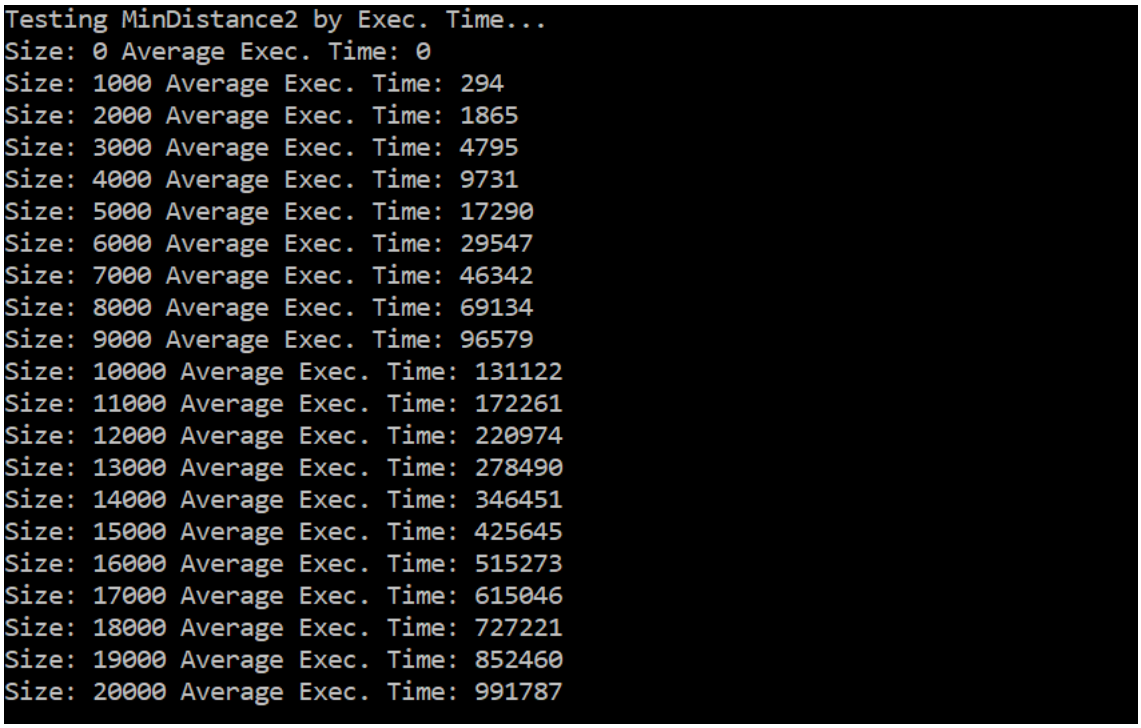
Appendix 9: *MinDistance2*(*A*[0..*n* – 1]) Algorithm Basic Operation Count Testing

```
public void TestExecTime(Func<int[], int> MinDistance)
{
    Stopwatch timer = new Stopwatch();

    for (int size = 0; size < 20000 + 1; size += 1000)
    {
        long t_total = 0;
        double t_average = 0;

        for (int i = 0; i < numOfArrays; i++)
        {
            int[] A = GenerateRandomArray(size);
            timer.Start();
            MinDistance(A);
            timer.Stop();

            long t_elapsed = timer.ElapsedMilliseconds;
            t_total = t_total + t_elapsed;
            }
            t_average = t_total / numOfArrays;
            Console.WriteLine("Size: " + size + " " + "Average Exec. Time: " + t_average);

    }
}
```

Appendix 10: C# Implementation of the *MinDistance*(*A*[0..*n* – 1]) and *MinDistance*(*A*[0..*n* – 1]) Algorithms for Execution Time Testing (Uses Code from Figures 9 and 10)

```
Testing MinDistance by Exec. Time...
Size: 0 Average Exec. Time: 0
Size: 1000 Average Exec. Time: 674
Size: 2000 Average Exec. Time: 3159
Size: 3000 Average Exec. Time: 8784
Size: 4000 Average Exec. Time: 18653
Size: 5000 Average Exec. Time: 34737
Size: 6000 Average Exec. Time: 58635
Size: 7000 Average Exec. Time: 91849
Size: 8000 Average Exec. Time: 136562
Size: 9000 Average Exec. Time: 193646
Size: 10000 Average Exec. Time: 265501
Size: 11000 Average Exec. Time: 358056
Size: 12000 Average Exec. Time: 470601
Size: 13000 Average Exec. Time: 598879
Size: 14000 Average Exec. Time: 746387
Size: 15000 Average Exec. Time: 913855
Size: 16000 Average Exec. Time: 1108934
Size: 17000 Average Exec. Time: 1328657
Size: 18000 Average Exec. Time: 1578134
Size: 19000 Average Exec. Time: 1855368
Size: 20000 Average Exec. Time: 2169516
```

Appendix 11: *MinDistance*(*A*[0..*n* – 1]) Algorithm Execution Time Testing



Appendix 12: *MinDistance2*(*A*[0..*n* – 1]) Algorithm Execution Time Testing