|  |  |
| --- | --- |
| CAB 301 ASSIGNMENT 2  Empirical Analysis of an Algorithm | N9845097  Ka Long Lee (Eric)  N9694315  Stanislav Berezin |

[1. Summary 2](#_Toc9374226)

[2. Description of Algorithm 2](#_Toc9374227)

[3. Theoretical analysis of the algorithms 3](#_Toc9374228)

[3.1. Basic Operation identification of the Algorithm 3](#_Toc9374229)

[3.2. Problem Size 4](#_Toc9374230)

[3.3. Average Case Efficiency 4](#_Toc9374231)

[4. Implementation of the Algorithm 5](#_Toc9374232)

[4.1. Program Implementation 5](#_Toc9374233)

[4.2. Functional Testing 5](#_Toc9374234)

[5. Design of Experiments 6](#_Toc9374235)

[5.1. Methodology, tools and techniques 6](#_Toc9374236)

[5.2. Data sizes, Test Data set 7](#_Toc9374237)

[6. Experiential results 8](#_Toc9374238)

[6.1. Calculating Basic Operation for both Algorithms 8](#_Toc9374239)

[6.2. Experiment on Calculating Average of Basic Operation 8](#_Toc9374240)

[6.3. Average execution time of the Algorithm 9](#_Toc9374241)

[6.4. Experiment on Calculating Average Execution time 9](#_Toc9374242)

[7. Analysis of Experiential results 10](#_Toc9374243)

[7.1. Average of Basic Operation Analysis 10](#_Toc9374244)

[7.2. Average of Execution Time Analysis 10](#_Toc9374245)

[8. Reference 12](#_Toc9374246)

[9. Appendices 13](#_Toc9374247)

# Summary

The purpose of this report is to compare the average case efficiency and the time complexity of the “Min Distance Algorithm” in two different versions. The implementations of specified algorithms were implemented in C# utilising Visual Studio Code which assisted this research greatly in providing accurate results and producing CSV documents with ease. Additionally, unit tests were developed to ensure the high quality of this report and provide evidence of its successful completion. The implementation of algorithms was developed in accordance with CRA specification, meaning tracking a record of basic operation and execution time for both algorithms.

The report begins with a description of the algorithm and goes into greater details explaining its intricacy and functionality to build a cohesive understanding of this report. Once the core idea of algorithms is conveyed to the reader, the report goes into theoretical analysis of the algorithm which discusses the technical analysis of implementation like an array, loops and other elements used during practical implementation. The theoretical analysis predicted both algorithms performance in a mathematical way.

Later the report delves into methodology, tools and techniques, which essentially helps to understand how the entire research was conducted in terms of hardware, programming language, third-party packages and provides justification for its use where necessary.

Lastly, the report goes into experimental results where actual results of algorithms provided, including CSV graphs, calculation of basic operations and execution time with relevant images. Then this report analyses those results which help to understand the difference between the two and conclude the final verdict in terms of efficiency.

# Description of Algorithm

The main purpose of the Min Distance algorithm is to find out the difference between each element in the given array which contains one or more integer values and return the smallest difference value to the user. In this experiment, we will study two different version of min distance algorithm. The first version of the min distance algorithm, *MinDistance*, was presented by Levitin and the Pseudo code displayed in **Appendix A-1**. On the contrary, The second version of the min distance algorithm, *MinDistance2*, was invented to work as *MinDistance* but performs much more efficient than it. The Pseudo code of MinDistance2 can be found in **Appendix A-2.**

Both Algorithms take an array which contains one or more integer values as the input and should return the minimum distance value regardless of input array in different condition. Such as random positioned, contains negative or contains duplicate values.

The *MinDistance* algorithm starts by creating the variable dmin and assign the maximum value of the integer to it. After that, it creates a two-layer loop and creates two index variables, i and j in order to iterates over comparing all the elements in the input array A. Both indexers start from 0 to the length of the array. Then, the algorithm begins checking the elements selected by indexer i and j whether to match the specified conditions or not. There are two specified conditions in this case. The first condition is that indexer i must not holding the same element as indexer j (check i ≠ j). The second condition is that the distance value between selected elements A[i] and A[j], must less than the current minimum distance value held by dmin (check Math.Abs (A[i] - A[j] < dmin ) ). If both conditions are matched in the current iteration, the distance value between selected elements A[i] and A[j] will assign to dmin. In the end, the algorithm returns the minimum distance value which held by dmin and quit when the iteration of the nested for loop is finished.

The *MinDistance2* algorithm starts the same way as *MinDistance* algorithm. It also creates the variable *dmin* and assigns the maximum value of the integer to it at the beginning.

After that, it creates a two-layer nested for loop and create two index variable, i and j in order to iterates over comparing all the elements in the input array A. However, it is different with *MinDistance* algorithm from here. The indexer i starts from 0 to the length of the array. In contrast, the indexer j always starts from i + 1 to the length of the array in each iteration of the outermost loop. It allows us to skips comparing the same element held by both indexer (Do not need to check i ≠ j). Then, the distance value between selected elements A[i] and A[j] will assign to a variable named as *temp* every time in each iteration of the innermost loop. After that, it assigns the value of *temp* to *dmin* if the current minimum distance value held by dmin is larger than temp. In the end, the algorithm returns the minimum distance value which held by *dmin* and quit when the iteration of the nested for loop is finished.

The average case efficiency for *MinDistance* algorithm is expected to be n(n - 1) time. because the nested loop always performs quadratic times of the array length. It would not skip the execution of the nested for loop even the minimum distance value has been found. MinDistance algorithm is normally the same with MinDistance algorithm. However, the innermost for loop is always initialized by the index of outermost for loop plus one. It makes the algorithm will never hold the same element to compare. It performs twice time faster than *MinDistance* algorithm. Therefore, it inherits the equation from MinDistance and divides the result by two.

# Theoretical analysis of the algorithms

# Basic Operation identification of the Algorithm

In order to compare both algorithms accurately, it must choose an operation that is common to two algorithms to serve as the basic operation. The common operation is calculating the absolute value of the difference value between A[i] and A[j] which is located inside the innermost for loop in both algorithms.

In *MinDistance* algorithm*, the* common operation has the potential that can be performed twice per iteration in the innermost for loop. The first chance the operation perform is as one part of the if condition. The second opportunity the operation performs is to assign the operation value to *dmin* variable if the previous if conditions are true.

In *MinDistance2* algorithm, the common operation only performs one time per iteration of the innermost for loop in order to save the absolute value of the difference value between A[i] and A[j] to the *temp* variable. In comparison to MinDistance algorithm, the common operation in *MinDistance2* is not optional and it performs every time when the innermost for-loop iterates.

The selected operation mentioned above is not only the most performed operation but also the most expensive operation in both algorithms. Maolin (2019) stated that “For time analysis, the basic operation is the operation that we expect to have the most influence on the algorithm’s total running time” It is obvious that the selected operation executes most frequent in the innermost loop of both algorithms. In comparison with other operations, they are either assignment or comparison. the selected operation not only calculates the difference between A[i] and A[j] but also calculate its absolute value. It is significant that this operation has a huge influence on both algorithm execution times.

# Problem Size

The problem size for two algorithms is related to the number of the elements in the input array. Another approach is that both algorithms expect another integer input n as the number of the array length. However, we will not use this approach in this experiment. Because C# is a high-level programming language that able to get the length of the array by accessing the property of the array class (Array.Length).

# Average Case Efficiency

**Appendix B-1** is the mathematical function to predict the average case efficiency for *MinDistance* algorithmin numbers of n array elements. From **Appendix B-1,** in the inner part of the function (n - 1), is derived from the algorithm performing n - 1 with each iteration times in the innermost for loop. Reducing one is the reason that the algorithm will not performs the basic operation when both indexers holding the same element (A[i] = A[j] and i = j). i will always equal to j in each of the iterations of the innermost loop. Therefore, it is n-1 in the inner part of the equation. on the other hand, the outermost for loop will always iterate according to the length of the array. Therefore, the times performs in innermost loop times the times performs in the outermost loop. The full equation is n \* (n - 1).

**Appendix B-2** is the function for predicting average case efficiency for *MinDistance2* algorithm with numbers of n array elements. The upper part of the equation, n \* (n -1), is the same logic as *minDistance.* However, there is a tiny different part from *MinDistance* which improve the algorithm dramatically. The different part is that the indexer j is always starts from i + 1. It makes that the element selected by indexer i will never be same as the element selected by indexer j. This method reduces the operation inside the nested for loop instead of the first approach which is using if statement to check the condition whether both indexers is holding the same element. By the end of the nested for-loop iteration, the operations will only perform half of the number that MinDistance Algorithm does. Therefore, it inherits the equation from MinDistance and divide the result by two.

# Implementation of the Algorithm

# Program Implementation

Two different versions of the Algorithms were implemented on C# programming language provided by Microsoft. The source code of *MinDistance* algorithm is displayed in **Appendix C-1.** On the other hand, the source code of *MinDistance2* algorithm is shown in **Appendix C-2.** The source code for two algorithms is saved in Algorithm.cs file in Assignment Workspace folder.

# Functional Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case | Test instance | Expected Output | Actual Output | Test Result |
| Both Algorithms should return minimum distance value from a general array | A = [1, 5, 9, 52, 46, 10, 13, 22 ] | 1 | 1 | Correct |
| Both Algorithms should return minimum distance value from an array which the numbers sorted from small to big. | A = [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61 ] | 1 | 1 | Correct |
| Both Algorithms should return minimum distance value from an array which the numbers sorted from big to small. | A = [61,59,53,47,43,41,37,31,29,23,19,17,13,11,7,5,3,2 ] | 1 | 1 | Correct |
| Both Algorithms should return minimum distance value from an array which contains duplicate numbers. | A = [1, 65535, 65535, 3, 6, 2, 0, 2, 7 ] | 0 | 0 | Correct |
| Both Algorithms should return minimum distance value from an array which the minimum value is in the last two elements | A = [50, 7, 30, 21, 56, 78, 15, 40, 44 ] | 4 | 4 | Correct |
| Both Algorithms should return minimum distance value from an array which contains duplicate number at begin and end position | A = [65535, 65535, 3, 6, 2, 0, 2, 7 , 65535 ] | 0 | 0 | Correct |
| Both Algorithms should return minimum distance value from an array which the numbers are sorted randomly | A = [ 26 ,24 ,21 ,30 ,1 ,27 ,11, 20 ,16 ,8 ,14 ,25 ,9 ,14 ,5 ,17 ] | 0 | 0 | Correct |
| Both Algorithms should return minimum distance value from an array which contains negative values | A = [-9, -23, -32, 0 ] | 9 | 9 | Correct |

Table 1. Functional Test Case Result

Functional testing was also performed during this research to ensure a high-quality level of this research. Each function test ensures both algorithms should return the expected value of each test. The function test will fail if either one of the algorithms does not return the expected output. A separate file in Microsoft Visual Studio was created and named TestCase.cs which contains most of the test performed during the experimental session. To put it shortly the types of testing an array which was done are:

* General array
* Small to big number
* Big to small number
* Duplicate numbers array
* Duplicate in the beginning and end array
* When the smallest distance is comparing the last two elements of the array
* Random array
* Negative numbers array

Having a diverse range of arrays allowed this research to test functionality and correctness of the implementation of Minimum Distance Algorithm. Table 1 reveals each the expected output, actual output and the input example for each unit test case. The test results can be viewed in **Appendix D-2** and its coding implementation in **Appendix D-1**.

# Design of Experiments

# Methodology, tools and techniques

1. Two different versions of MinDistance algorithm were implemented on the C# programming language. In this case, the algorithm is developed as a console based application. Visual Studio is the software which used to run the program. It provides lots of tools for the experiment, such as terminal, unit testing, debug console. The program also can run in most of the popular operating system. Such as MacOS, Window, Linux Ubuntu.
2. The following experiment was performed on a 15-inch MacBook Pro 2018 Winter model. The OS on this computer is MacOS Mojave with Intel Core i9 Processor running at 2.9GHz and 16Gb of RAM memory. The hard drive on this computer contains 256GB storage space in a solid-state disk (SSD) to achieve the best performance as accurately as possible.
3. There is a tiny program in the program.cs file. It recorded the result of the basic operations and the execution times takes in both *minDistance* algorithm into two separate csv files. It is a file that contains the table of the experiment records. Using these results, we were able to produce line graphs in Microsoft Excel to analyze the results.
4. CsvHelper, third-party C# library, was used to write the basic operations and the execution times results in a CSV file. In this case, we make every test as one row using the write Anonymous Type Objects method (CSVhelper, 2019) to record each experiment test in each column.
5. The CSV file then converts to xlsm file for Microsoft Excel to create a line graph to represent the growth of execution time and the growth of the basic operation takes related to the length of the array growth.

# Data sizes, Test Data set

1. The Test Dataset is generated randomly by the program.
2. The size of the array starts from 0 to 10000, in increments of 1000.
3. The numbers in the array are generated randomly depends on the array size the algorithm is testing.
4. The range of the numbers started from -1073741824 to +1073741824 Which are the half of the maximum and minimum value of Integer.
5. The random number generator is used the real-time as seed and sorted the number randomly. It ensures that the numbers in the array have a unique chance to be assigned when the program runs at a different time.
6. The test data array in the data set is shuffled randomly and unpredictable.
7. In order to predict the efficiency of the algorithm accurately, the experiment tested repeatedly 60 times on the same condition with totally different data set as the condition mentioned above. It means that every type of arrays which listed in the unit test case has an equal chance to appear in every test.

# Experiential results

# Calculating Basic Operation for both Algorithms

In order to achieve and procure records of basic operations, a variable was created and placed inside of the internal loop and if statements to calculate the proper amount of basic operations performed by algorithms.

The functionality to calculate basic operations is located in Algorithm.cs file, there we have a method called MinDistanceBO and MinDistance2BO which actually implement the calculation of basic operations. Both methods consist of two loops, and inside of an internal loop we have the most important element as this is the precise location where basic operations are calculated. An array is produced randomly and method for its generation is located inside of a Helper.cs file, and that is to ensure to quality and persistence in analysis provided in this report.

# Experiment on Calculating Average of Basic Operation

This research experimented on calculation of basic operation for both algorithms

with different array sizes. This table below illustrated the number of basic operations performed for both algorithms and array size that was used during experimentation process.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array Size | *MinDistance* Predicted Basic Operation | *MinDistance*  Average Basic operations | *MinDistance2* Predicted Basic Operation | *MinDistance2* Average Basic operations |
| 0 | 0 | 0 | 0 | 0 |
| 1000 | 999999 | 1015663 | 499500 | 507825 |
| 2000 | 3999999 | 4064649 | 1999000 | 2032316 |
| 3000 | 8999999 | 9146965 | 4498500 | 4573475 |
| 4000 | 15999999 | 16262617 | 7998000 | 8131300 |
| 5000 | 24999999 | 25411600 | 12497500 | 12705791 |
| 6000 | 35999999 | 36593917 | 17997000 | 18296950 |
| 7000 | 48999999 | 49809566 | 24496500 | 24904775 |
| 8000 | 63999999 | 65058551 | 31996000 | 32529266 |
| 9000 | 80999999 | 82340867 | 40495500 | 41170425 |
| 10000 | 99999999 | 101656518 | 49995000 | 50828250 |

**Table 5.A Average Basic Operation Comparison table**

To gather accurate statistical data both algorithms were run 60 times, with array size increasing by 1000 up to 10000. The location of this implementation is located in CSV.cs, there is a loop that increments the array size as mentioned previously by 1000 and inside of it another loop that runs 60 times with a purpose to acquire statistical data and provide this research with evidence of its conclusions.

The results of such experiments were saved in CSV files to produce appropriate graphs and help readers of this report to visualise the results with ease. You can view the graph in **Appendix G** where it clearly displays the increase of basic operations as array size grows. The first algorithm required more basic operations to calculate to distance in comparison to second algorithm. The prediction provided in table above is calculated by utilising theoretical prediction and mathematics to ensure high quality of this research.

# Average execution time of the Algorithm

A similar solution and approach was applied when calculating average execution time. Average execution time allowed this research to compare two algorithms and compare them in terms of time efficiency. Theoretically speaking, the longer it takes to process an array with Minimum Distance algorithms the less effective it would be as time is crucial when it comes to effectiveness of algorithms. Microsoft Visual Studio provides tools to calculate the proper execution time of a method and in this case we have utilised it to our advantage to produce accurate results and support this report with evidence of average execution time.

# Experiment on Calculating Average Execution time

In the scope of this project and experiment a diverse range of arrays in terms of length and randomness were tested for both algorithms and compared with one another. The array size begins from 0 and grows to 10 000 by incrementing itself by 1000 on each new iteration. Due to the differences between two algorithms and their implementation it is expected that one should outperform another one and table below provides us with this evidence.

|  |  |  |
| --- | --- | --- |
| Size | *MinDistance*  Average Execution Time | *MinDistance2*  Average Execution Time |
| 0 | 0 ms | 0 ms |
| 1000 | 11ms | 5 ms |
| 2000 | 47 ms | 20 ms |
| 3000 | 107 ms | 46 ms |
| 4000 | 191 ms | 83 ms |
| 5000 | 300 ms | 131 ms |
| 6000 | 430 ms | 188 ms |
| 7000 | 583 ms | 254 ms |
| 8000 | 763 ms | 333 ms |
| 9000 | 970 ms | 422 ms |
| 10000 | 1223 ms | 527 ms |

**Table 5.B Average Execution time Comparison table**

Based on this graph and evidence produced during experimental session it should be reasonable to conclude that the second algorithm proved itself to be far more effective comparing to the first one. The CSV graph that was produced during this experiment is provided in **Appendix H**. The effectiveness of the second Minimum Distance algorithm almost doubles first algorithm in terms of time efficiency and CSV graph also proves quite a similar result. The coding implementation of average execution time can be viewed on **Appendix F-1** and **Appendix F-2**.

# Analysis of Experiential results

# Average of Basic Operation Analysis

Based on the charts in **Appendix G** it would be appropriate to conclude that the the second algorithm proved itself to be far more effective comparing to the first one. The prediction which were produced based from formulas **Appendix B-1** and **Appendix B-2** provided us with accurate estimation in terms of basic operations that are expected to be performed in both cases. For example the mathematical estimation for the first algorithm would be 999 999 for an array with length 1000 and the actual result was 1 015 6663, and the second algorithm had 499 500 against actual 507 825, both of these cases were tested on local machine multiple times and actual results nearly matched the prediction which ensured the correctness of this research.

From charts in Appendix, it is also observable that the second algorithm outperform the first one almost by half. The reason for such difference is that the element selected by indexer i will never be same as the element selected by indexer j in *MinDistance2* . This method reduce the operation inside the nested for loop instead of the approach in *MinDistance* which is using if statement to check the condition whether both indexer is holding the same element.

|  |  |  |  |
| --- | --- | --- | --- |
| 999999 | 1015663 | 499500 | 507825 |

# Average of Execution Time Analysis

The execution time of the second algorithm was also expected to outperform the first one. Considering the fact that the second required far lesser basic operations, it was within logical reason to assume shorter execution time. Based on the graph in **Appendix H**, it can be noted that the second algorithm is indeed outperformed the first based due to its effectiveness when it comes to finding Min. Distance in the array. An interesting phenomena and similarity which should be noted is that execution time of the first algorithm doubles the time of a second one and exactly the same thing happens to basic operations, where the first one doubles comparing to the second one. Hence, the second algorithm is better in terms of average execution time, as well as basic operations which makes twice as more effective than the second one.

# Reference

CsvHelper. (2019). Retrieved from <https://joshclose.github.io/CsvHelper/>

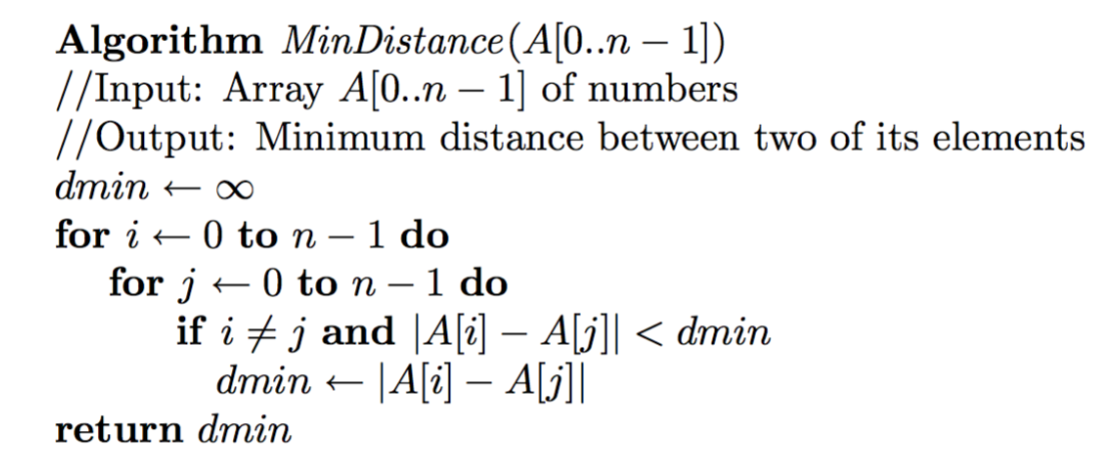
CsvHelper. (2019). Retrieved from <https://joshclose.github.io/CsvHelper/examples/writing/write-anonymous-type-objects/>

Dimitrov, D., Gönül, S., Foster, J., Agnew, B., & PRANTO, R. (2012). Calculate the execution time of a method. Retrieved from <https://stackoverflow.com/questions/14019510/calculate-the-execution-time-of-a-method>

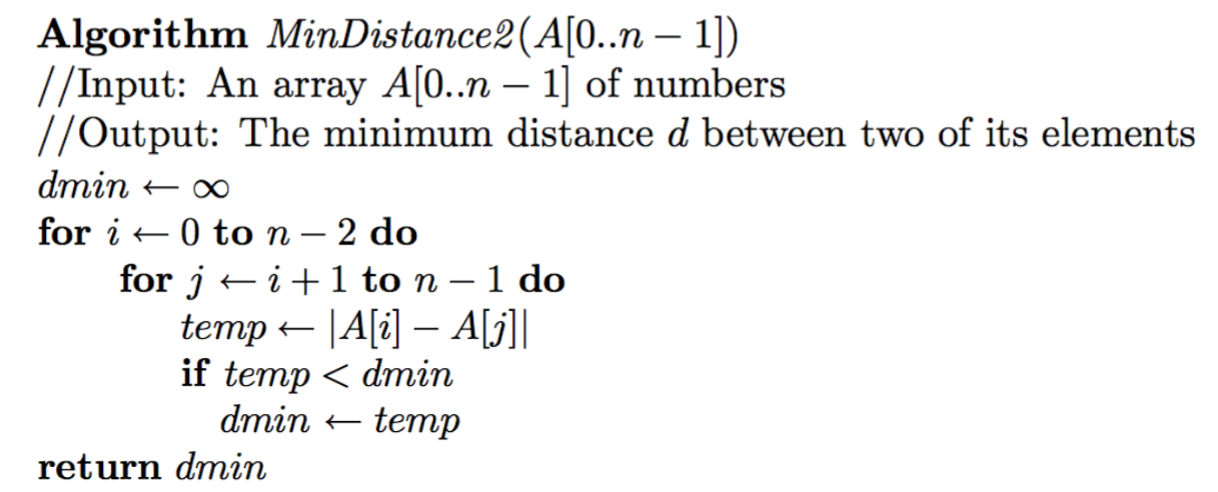
Tang, M. (2019). CAB301-Lecture2. Retrieved from <https://blackboard.qut.edu.au/bbcswebdav/pid-7900328-dt-content-rid-23380690_1/courses/CAB301_19se1/CAB301-Lecture2%282%29.pdf>

# Appendices

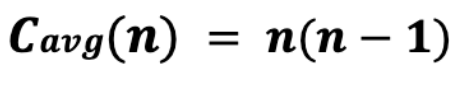
**Appendix A-1 - The Pseudo code of *MinDistance* Algorithm**



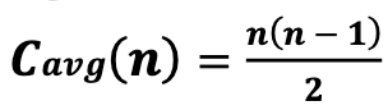
**Appendix A-2 - The Pseudo code of *MinDistance2* Algorithm**



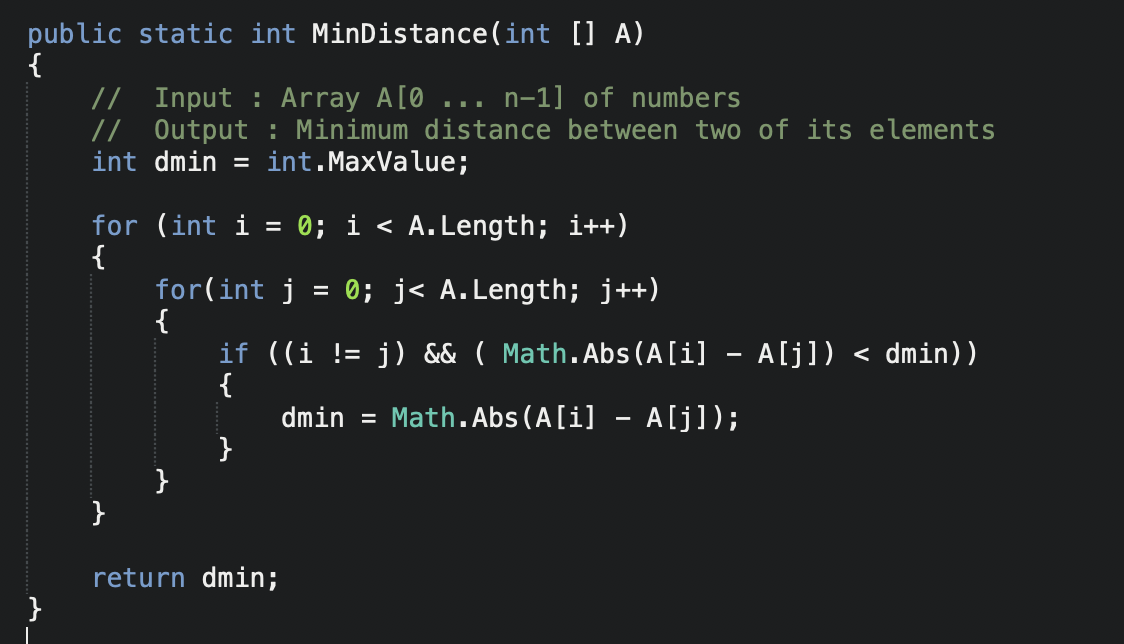
**Appendix B-1 - Mathematical function for Calculating Average-Case Efficiency of *MinDistance* algorithm**



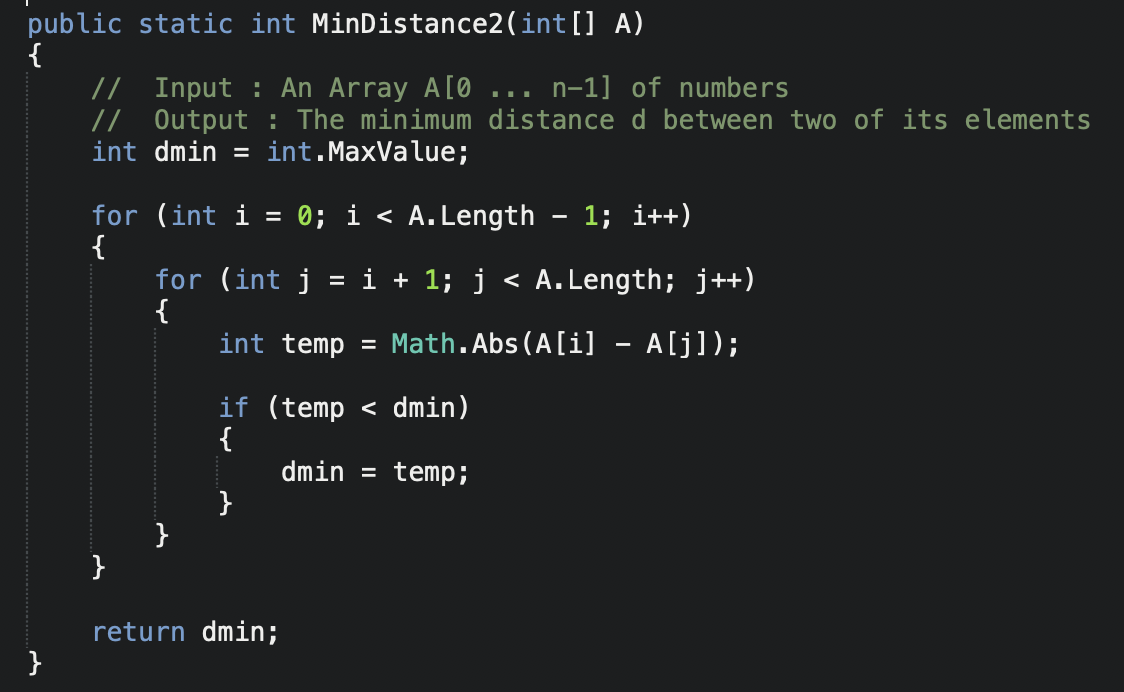
**Appendix B-2 - Mathematical function for Calculating Average-Case Efficiency of *MinDistance2* algorithm**



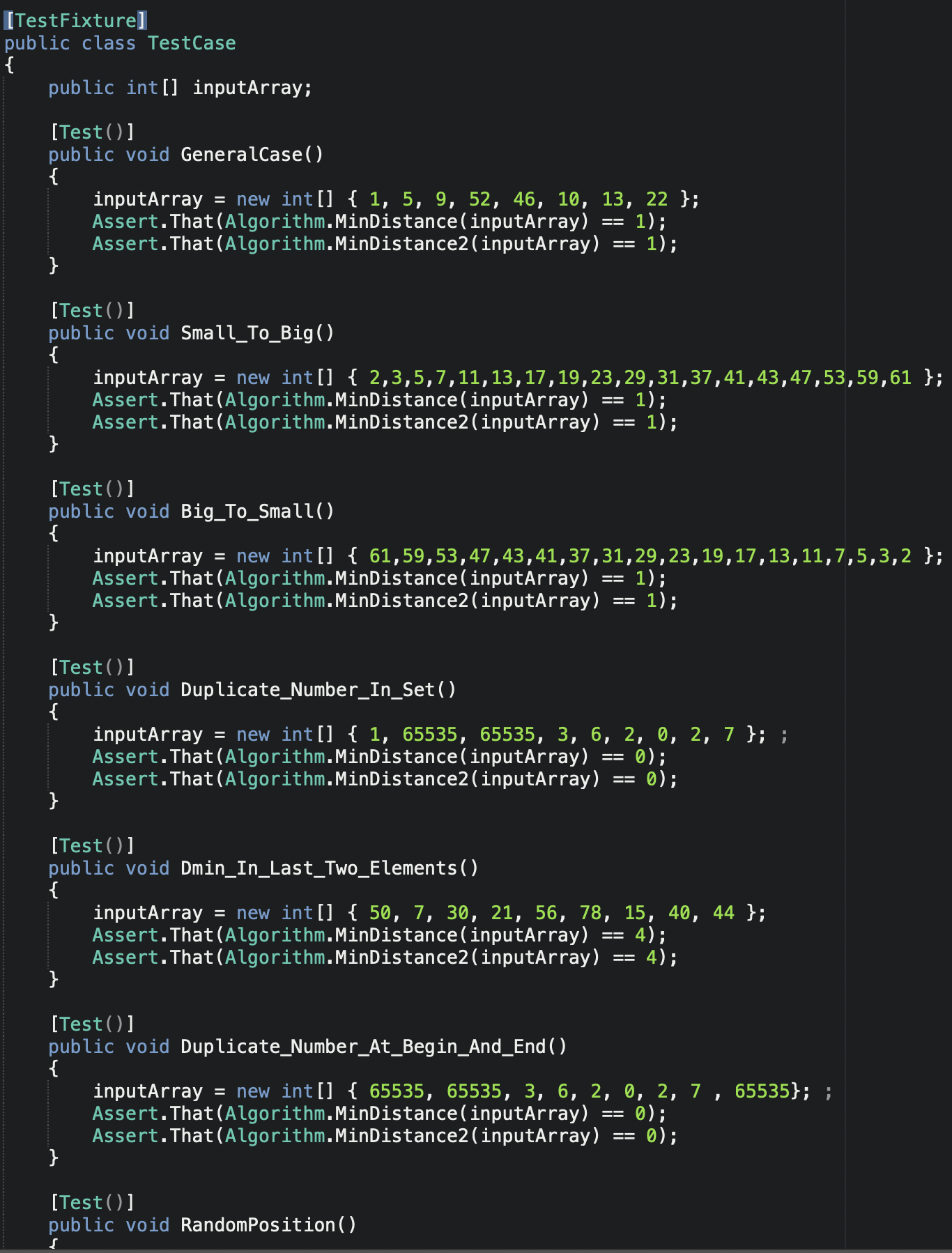
**Appendix C-1 – *MinDistance* Algorithm source code in C#**



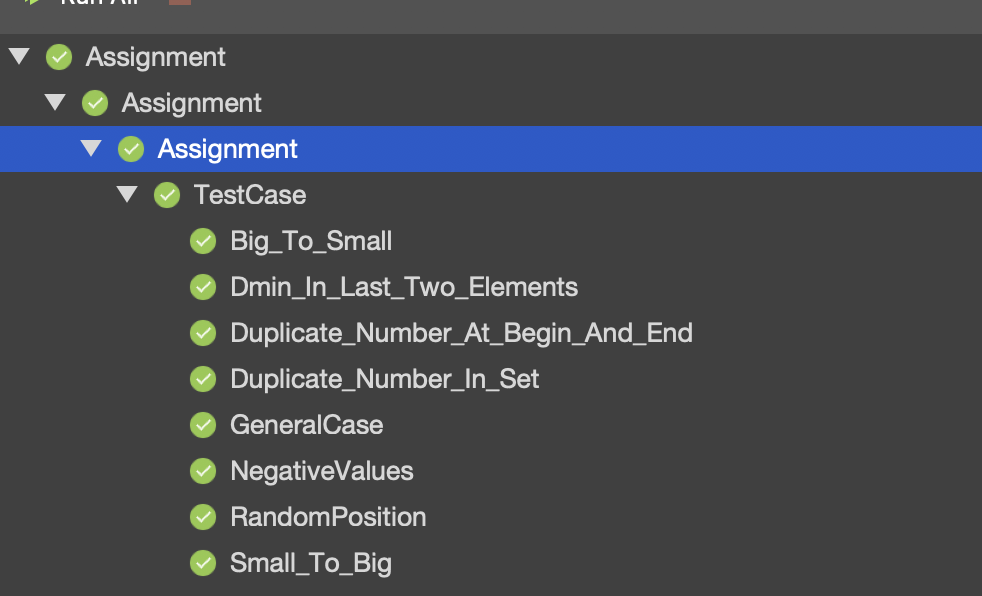
**Appendix C-2 – *MinDistance2* Algorithm source code in C#**



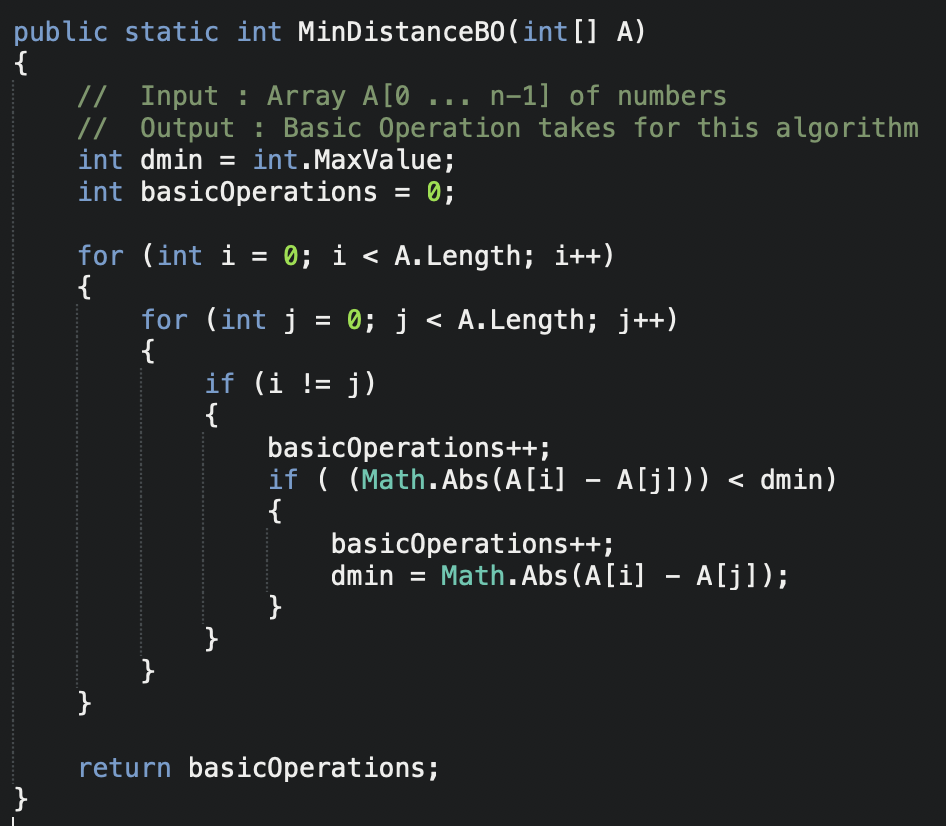
**Appendix D-1 – Functional Test code for Two Min Distance Algorithm in C#**



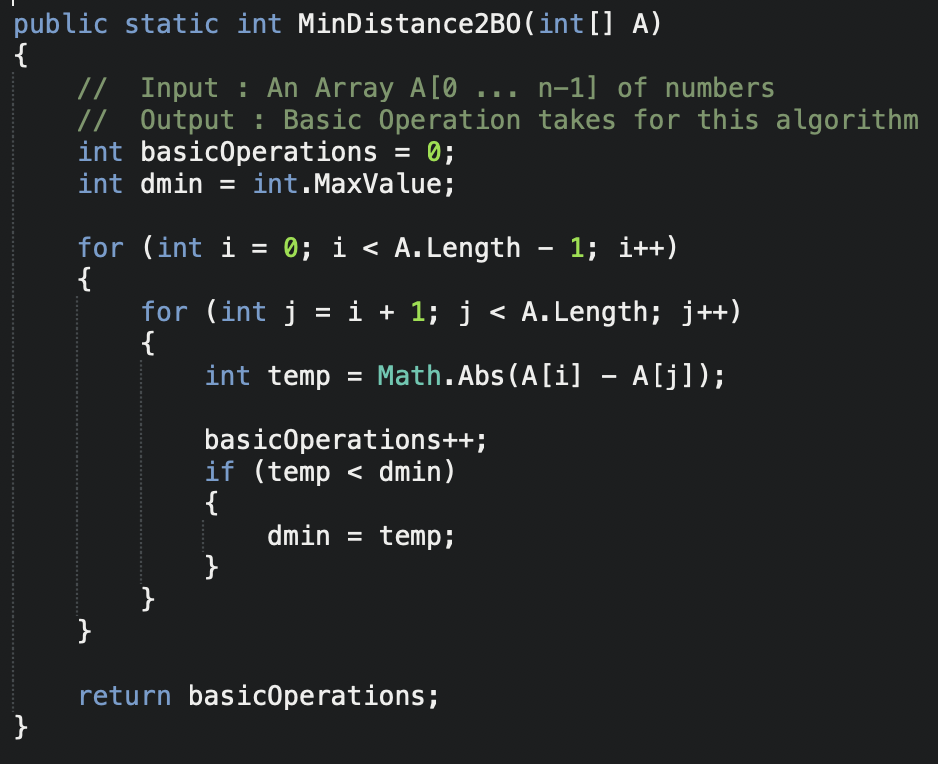
**Appendix D-2 – Test result for Brute Force Median Algorithm**



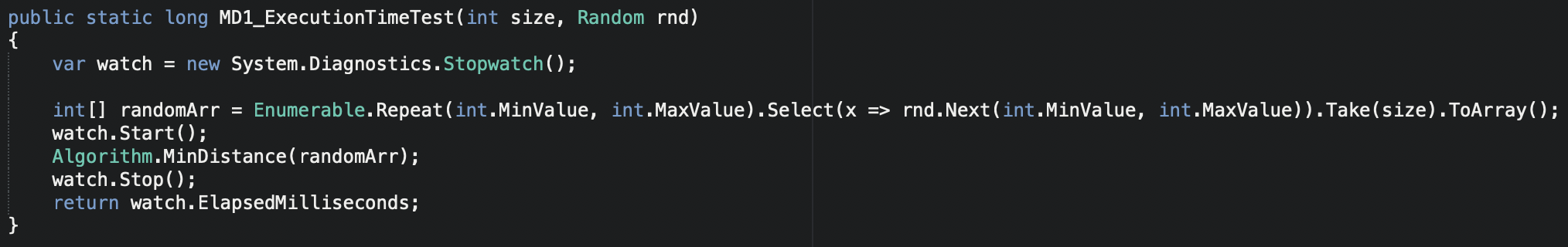
**Appendix E-1 – The Algorithm return basic operation for *MinDistance* algorithm**



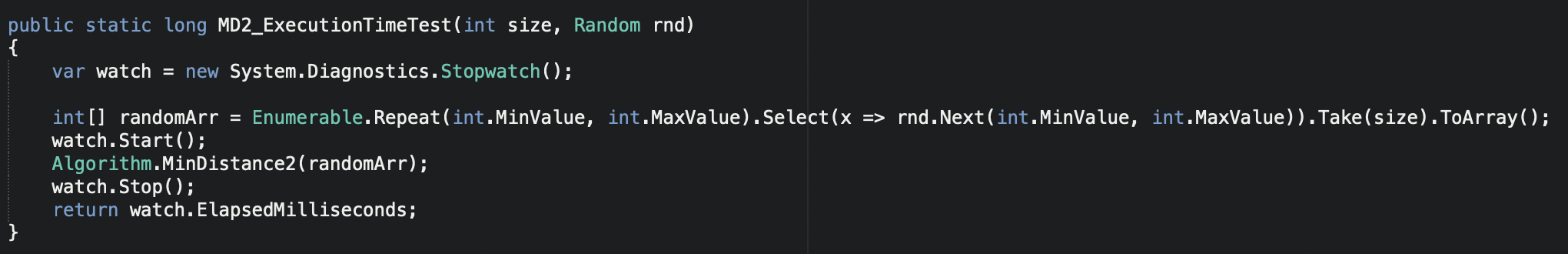
**Appendix E-2 – The Algorithm return basic operation for *MinDistance2* algorithm**



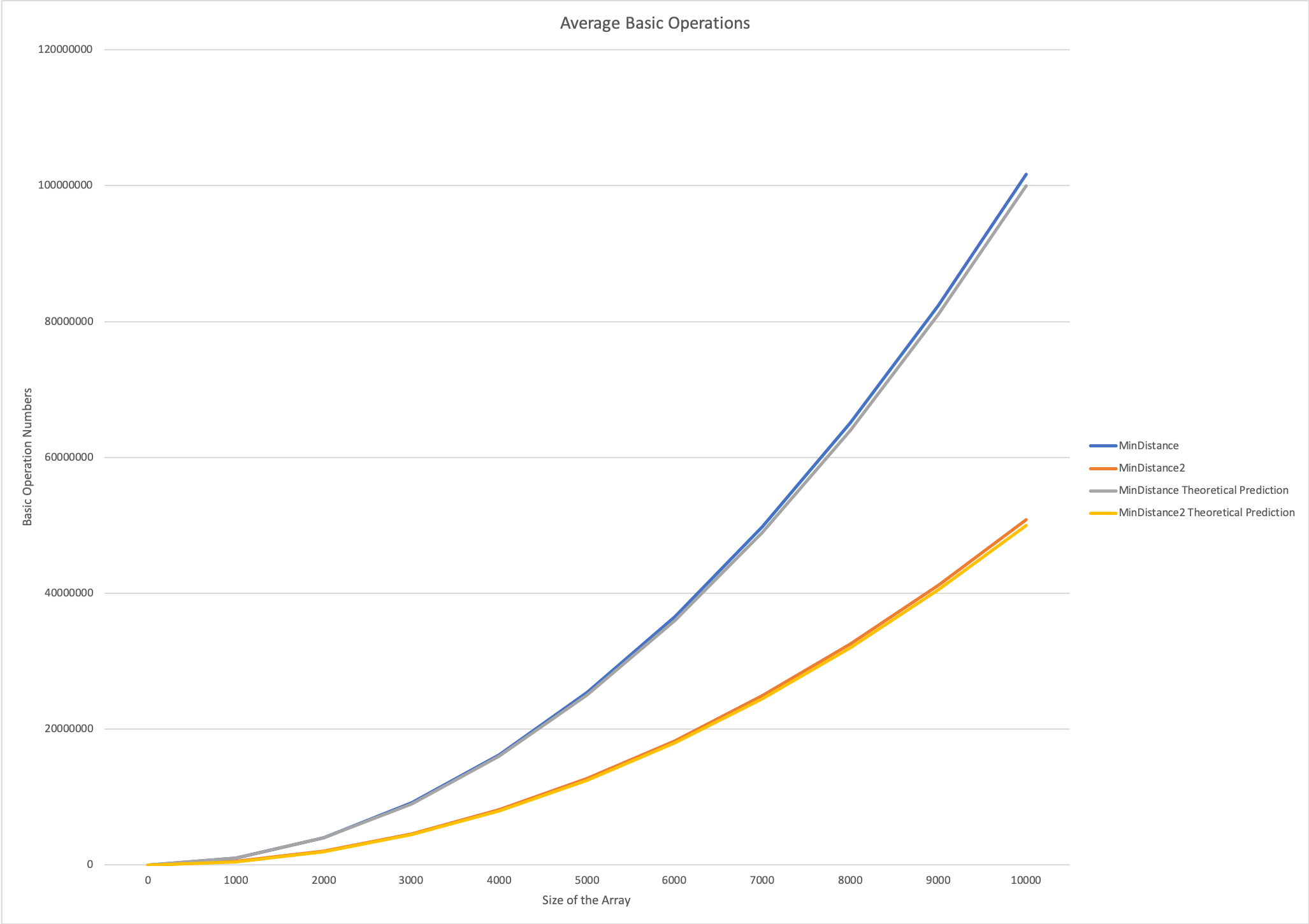
**Appendix F-1 – *MinDistance* algorithm Execution time test code**



**Appendix F-2 –*MinDistance2 algorithm* Execution time test code**



**Appendix G – Average Basic Operation Graph of the algorithm**



**Appendix H – Average Execution Time Graph of the algorithm**

