|  |  |
| --- | --- |
| CAB301 Assignment 1  Empirical Analysis of an Algorithm | N9845097  Ka Long Lee (Eric) |

Table of Contents

[1. Summary 2](#_Toc5495446)

[2. Description of Algorithm 2](#_Toc5495447)

[3. Implementation of the Algorithm 3](#_Toc5495448)

[3.1. Program Implementation 3](#_Toc5495449)

[3.2. Functional Testing 4](#_Toc5495450)

[4. Design of Experiments (2 Pages) 5](#_Toc5495451)

[4.1. Methodology, tools and techniques 5](#_Toc5495452)

[4.2. Data sizes, Test Data set 5](#_Toc5495453)

[5. Experiential results (4 Pages) 6](#_Toc5495454)

[5.1. Basic Operation identified 6](#_Toc5495455)

[5.2. Average execution time 6](#_Toc5495456)

[5.3. Experience to measure the program execution times 6](#_Toc5495457)

[6. Analysis of Experiential results (4 Pages) 6](#_Toc5495458)

[6.1. Experimental results 6](#_Toc5495459)

[6.2. Comparing against the theoretical efficiency prediction 6](#_Toc5495460)

[7. Reference (1 Pages) 6](#_Toc5495461)

[8. Appendices (4Pages) 7](#_Toc5495462)

# Summary

This report is to analysis the time complexity and the average case efficiency of the Brute Force Median algorithm. Brute Force Median algorithm is a algorithm that takes an array of integers set and return the median value in the array. The algorithm was implemented in a C# 7.3 program. In this report, it measured the number of basic operation and the execution time of the algorithm in order to check the accuracy of the theoretical analysis of the algorithm. The experiment result is obvious that it is consistent with the theoretical prediction.

# Description of Algorithm

The main purpose of the Brute Force Median algorithm is to find out the median value in the given array which contains a set of integer values. Considering the different condition that the array is ordered by. Returning the value in the middle position of the array is not the perfect way to find out the median value of an array. Because the array might be sorted randomly or exist duplicate number in it. That is the reason why we need to implement Brute Force Median algorithm to retrieve the median value in the array accurately in different condition.

The Brute Force Median algorithm takes any array which contains integers and finds the median value of the array. It selects all the element to compare it one by one in a nested for loop. At the end of each iteration, it checks the current selected item and return it to the user if the value of the selected item is median. However, it will keep iterating until the median found in the array or return -1 to the user when the input array is in a invalid format such as empty array.

The algorithm should return the correct median value in the integer value set despite that the order is sorted randomly, in ascending order or in descending order. It also able to return the correct median value whatever the length of the array is odd or even numbers. Duplicate value in the array will not influence the algorithm works. The disadvantage of this algorithm is that the execution time is growing dramatically according to increasing the number of array length. Therefore, the array length is lower, the algorithm is more efficient.

# Implementation of the Algorithm

# Program Implementation

The algorithm was implemented on the C# programming language. The source code of the algorithm displayed in Appendix A is saved in program.cs file. The algorithm takes only one parameter which is an array contains one or more integer numbers and assigned to the variable named as A. First of all, the algorithm creates one local variable named as k. It is for calculating and storing the median position of the input array. There are two steps for calculating the median position of the array. Firstly, dividing the length of the array by two. Secondly, ceiling the previous result to the highest integer value it can be.

If the length of the array is an odd number, ceiling the result can help us pointing to the median position accurately. In contrast, we assume that dividing the length of the array by two is the median position for an array of even length. Because it doesn’t exist a correct way to define the median position for an even length array. Finally, the result will be assigned to the k variable. If the input array was already sorted before it passed to this algorithm, the median value of the array must be located at the k position.

The algorithm then creates a nested for loop which will repeat the operations inside the block according to the numbers exists in the input array. There are total two for loops in the nested loop. Both loops create an indexer variable for selecting elements in the array for different purpose inside each loop. These indexers start from 0 to the length of the array. The main purpose of the outer loop is to determine the array element selected by the indexer of the outer loop whether median or not. On the other hand, the purpose of the inner loop is for calculating how many elements are smaller and equals to the element selected by the indexer of the outer loop.

In the outer loop, it creates an Indexer variable denoted as i. This indexer only allowed to increase once when the outer loop finished. There are two local variables inside this loop. The first variable is named as numsmall which records how many elements in the input array are smaller than the selected element by indexer i which is denoted as A[i]. The second variable is named as numequal which records how many elements are same as A[i].

In the inner loop, Indexer j is created and updated only inside the inner loop. It checks all the elements in the array to compare with A[i] the element. If A[j] which is the selected element by indexer j is smaller than A[i], numsmall variable will increase one. On the other hand, numequal variable will increase one when A[i] equals A[j]. It is significant to note that the basic operation numsmall variable or numequall variable increasing one might not be run when A[j] is bigger than A[i].

At the end of the outer loop, if numsmall variable is smaller than k meanwhile numsmall plus numequal is larger or equals to k, it means that i is at the median position. Therefore, the program will return A[i] the element which selected by indexer i to the user. On the other hand, the invalid output is passed as the parameter. Such as an empty array. The algorithm will return -1 to the user.

# Functional Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case | Test instance | Expected Output | Actual Output | Test Result |
| Return median from Sorted array | A = [1,2,3,4,5,6,7] | 4 | 4 | Correct |
| Return median from an array which the length is an even number and elements sorted randomly | A = [8, 12. 7, 9] | 8 | 8 | Correct |
| Return median from an array which the length is an odd number and elements sorted randomly | A = [8, 5, 9, 1, 4] | 5 | 5 | Correct |
| Return median from a reverse order array | A = [9,8,7,6,5,4,3] | 6 | 6 | Correct |
| Return median from an array which contains negative numbers | A = [-5, -3, 5, 4, -2] | -2 | -2 | Correct |
| Return median from an array which contains duplicate numbers | A = [1,2,2,2,4,5,6,6,7] | 4 | 4 | Correct |
| Return Median from an array which only contains one number | A = [4] | 4 | 4 | Correct |
| Return -1 from an empty array | A = [] | -1 | -1 | Correct |

Table 1. Functional Test Case Result

The Implementation of the algorithm in C# Programming language is displayed in Appendix A. The functional correctness of the algorithm was tested by using test function which defined in TestCast.cs file. Appendix B-1 shows the unit test code for the brute force median algorithm. There are eight test function in total to test out the algorithm should return the median value in the array in different condition. These conditions include sorted, reverse, random, duplicate, single length, invalid length, odd length and even length array.

The table above recorded each test array, expected and actual output for each test case. Appendix B-2 also displayed that all the unit test case run correctly in the Visual Studio IDE. These unit test confirm that the implementation of the algorithm returning the median value successfully in every different condition. However, if the input array is an empty array, it will return -1 to the user as the error code.

# Design of Experiments (2 Pages)

# Methodology, tools and techniques

1. The Brute Force Median algorithm were implemented on the C# programming language. C# is one of the famous object-oriented programming languages in 2019 which is able to develop different types of applications such as a web, mobile, server, console applications. In this case, the algorithm is developed as a unit test application. Visual Studio is a software which used to run the experiment program. It provides lots of tools for the experiment, such as unit testing, debug console. Microsoft also provides Visual Studio software running in most of the popular operating system. Such as MacOS, Window, Linux Ubuntu.
2. The following experiment were performed on a 15-inch MacBook Pro 2018 model. The operating system on this computer is MacOS Mojave with 16Gb of RAM memory, Intel Core i9 Processor running at 2.9GHz. It also has 256GB storage space in a solid-state disk (SSD) to achieve the best performance as accurate as possible.
3. The basic operations and the execution times results are recorded into two separate csv files. It is a Microsoft Excel file than contains table of the record. Using these results, we were able to produce line graphs in Microsoft Excel to analyze the results.
4. The external library CsvHelper was used to write the basic operations and the execution times results into a CSV file.
5. The CSV file will then convert 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 Data set is generated randomly by the program. The array length of the data starts from 1000 to 20000, in increments of 1000.
2. The numbers in the array are generated randomly between 1 – 20000. The random number generator is used the real time as seed. It can ensure that the program will generate unique random number when the program runs at different time.
3. The test data array in the data set is sorted randomly and unpredictable. Every execution time might result different outcome.
4. In order to predict the efficiency of the algorithm accurately, the experiment tested repeatedly thirty times on the same condition with totally different test data set.

# Experiential results (4 Pages)

# Basic Operation identification of the Algorithm

From the implementation of the algorithm displayed in Appendix A, it is significant that the block of code in the innermost for loop has the greatest influence on the execution time of the algorithm. It calculates the number of elements in the array are smaller than or equal to A[i] one by one in the innermost for loop, if A[i] is not the median value of the array, it selects the next element and compare the others element again and so on. This block of code executes most frequently in the algorithm. According to Maolin stated that “For time analysis, basic operation is the operation that we expect to have the most influence on the algorithm’s total running time” It is obvious that two comparison statement in the innermost loop execute most frequent in the algorithm. This operation takes an exponential amount of time O(n^2) according to the length of the array. It takes most of the time for the execution of the algorithm. We can assume that the average value of the basic operation is less or equal than the exponent of the length of the input array.

The basic operation counter should not be placed inside the if-else statement block in the innermost for loop. The reason is that the block of code inside the if-else statement only execute when the condition of the if-else is true. It means that it is not the most frequent part that the algorithm executes comparing with the block inside the innermost for loop. Therefore, basic operation is not inside the if-else block.

# Experiment on Calculating Average of Basic Operation

In the experiment, we calculate the basic operation by increasing a counter in the algorithm. Appendix-C demonstrated that we modified the implemented algorithm to return the number of the basic operation for execute the brute force median algorithm. As mentioned in section 5.1, the basic operation are two comparisons in the innermost for loop. we implemented an integer variable named as basicOperation at the beginning of the algorithm. Each time the innermost for loop was executed, the basicOperation counter will increase one.

The experiment of the basic operation [Appendix-C] were performed thirty times for calculating the average basic operations of the brute force median algorithm. We generated the results from the random positioned array which the size starting from one thousand to twenty thousand, increasing each array size by one thousand a time. These results will then be saved into a csv file. Finally, we use Microsoft Excel to calculate the average basic operation from these thirty tests on each array size. The average basic operation is produced as a line graph in Appendix D and recorded to Table A below.

# Average execution time of the Algorithm

The execution time experiment uses the System.Diagnostics library to record the execution time of the brute force algorithm. The library records the execution time in millisecond. Therefore, it is able to calculate the time accurately even in an extremely rapid execution time condition which human might not able to feel it. Appendix D reveals the implementation of the execution test algorithm in C#. The method is that it starts the measurement before the brute force median algorithm executes. It stops the time measurement After the brute force median algorithm finished the execution.

# Experiment on Calculating Average Execution time

The experiment is same as the basic operation experiment. It measures the execution time from executing the brute force median algorithm with the random positioned array which the size starting from one thousand to twenty thousand, increasing each array size by one thousand a time. It also performs thirty times the experience with the same condition of the test. It then records all thirty-test data into a csv file. Finally, these data are used to generate the average execution time for each array size from 1000 – 20000. The average execution time data is then produced as a line graph in Appendix E and recorded to Table B below.

# Analysis of Experiential results (4 Pages)

# Average Case Number of Basic Operation

# Average Case Execution Time

# Reference (1 Pages)

# Tables

|  |  |  |
| --- | --- | --- |
| Input size | Predicted Basic Operation | Average Basic Operation over 60 test |
| 1000 | 500500 | 822450 |
| 2000 | 2001000 | 2107900 |
| 3000 | 4501500 | 4529600 |
| 4000 | 8002000 | 7036533.33 |
| 5000 | 12502500 | 10878500 |
| 6000 | 18003000 | 17216800 |
| 7000 | 24503500 | 28082950 |
| 8000 | 32004000 | 34265200 |
| 9000 | 40504500 | 38934300 |
| 10000 | 50005000 | 50248000 |
| 11000 | 60505500 | 65135033.3 |
| 12000 | 72006000 | 73498400 |
| 13000 | 84506500 | 74031316.7 |
| 14000 | 98007000 | 95029200 |
| 15000 | 112507500 | 108002000 |
| 16000 | 128008000 | 122271200 |
| 17000 | 144508500 | 142949600 |
| 18000 | 162009000 | 179484900 |
| 19000 | 180509500 | 199069650 |
| 20000 | 200010000 | 191939667 |

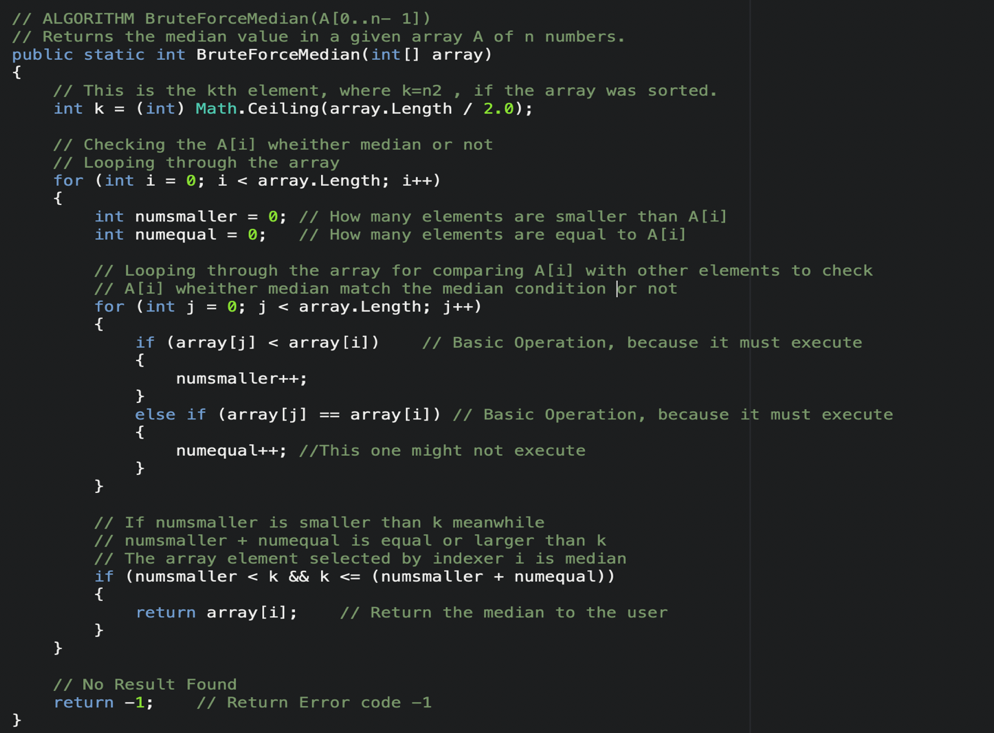
Table 1. Average Basic Operation Comparison table

|  |  |
| --- | --- |
| Input size | Average Execution time over 60 test |
| 1000 | 21.5245902 ms |
| 2000 | 48.0655738 ms |
| 3000 | 82.4098361 ms |
| 4000 | 120.639344 ms |
| 5000 | 163.114754 ms |
| 6000 | 220.459016 ms |
| 7000 | 278.360656 ms |
| 8000 | 366.360656 ms |
| 9000 | 417.311475 ms |
| 10000 | 499.131148 ms |
| 11000 | 590.245902 ms |
| 12000 | 687.639344 ms |
| 13000 | 743.327869 ms |
| 14000 | 943.213115 ms |
| 15000 | 950.606557 ms |
| 16000 | 1163.09836 ms |
| 17000 | 1211.88525 ms |
| 18000 | 1468.52459 ms |
| 19000 | 1539.91803 ms |
| 20000 | 1757.91803 ms |

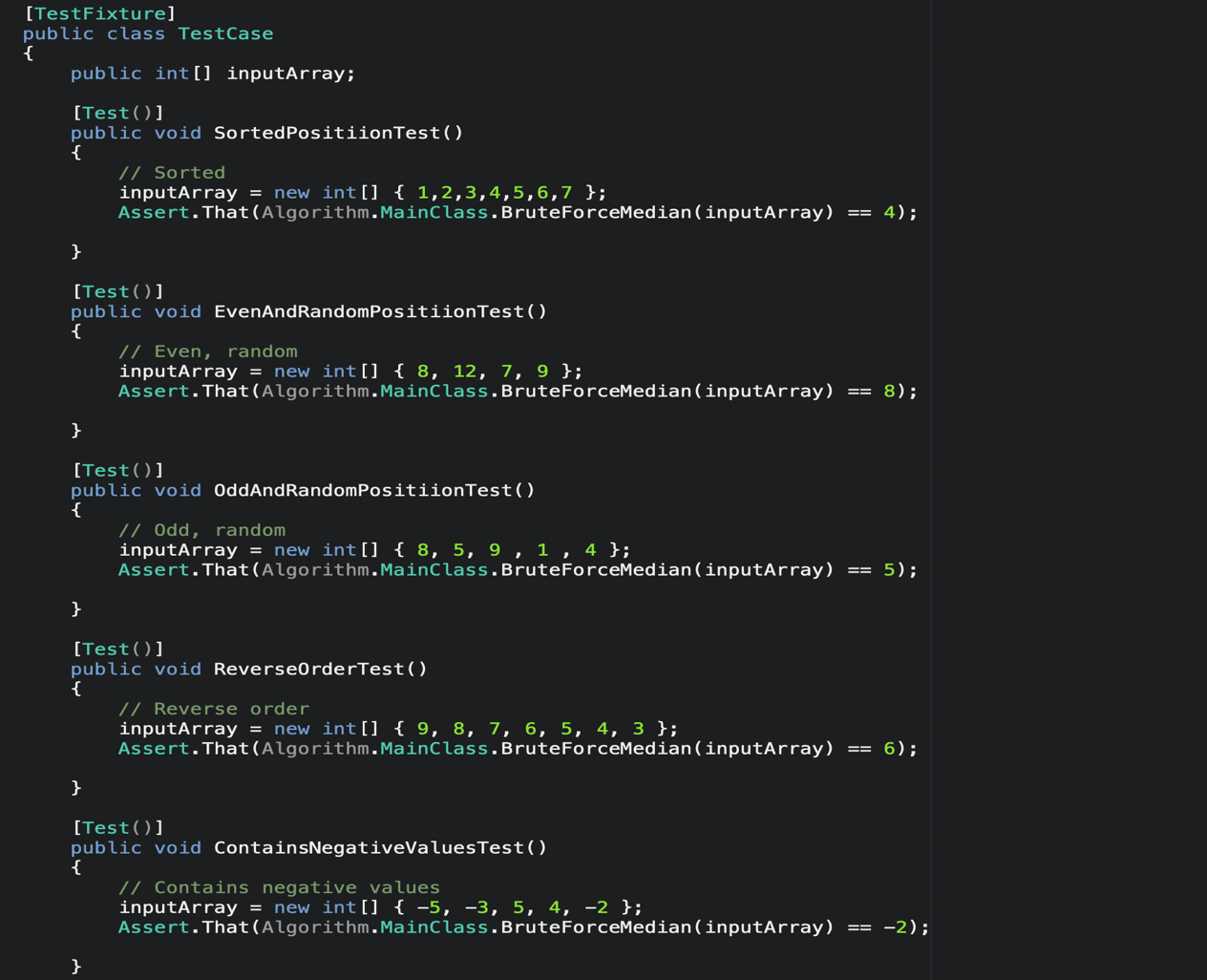
Table 2. Average Execution time Comparison table

# Appendices (4Pages)

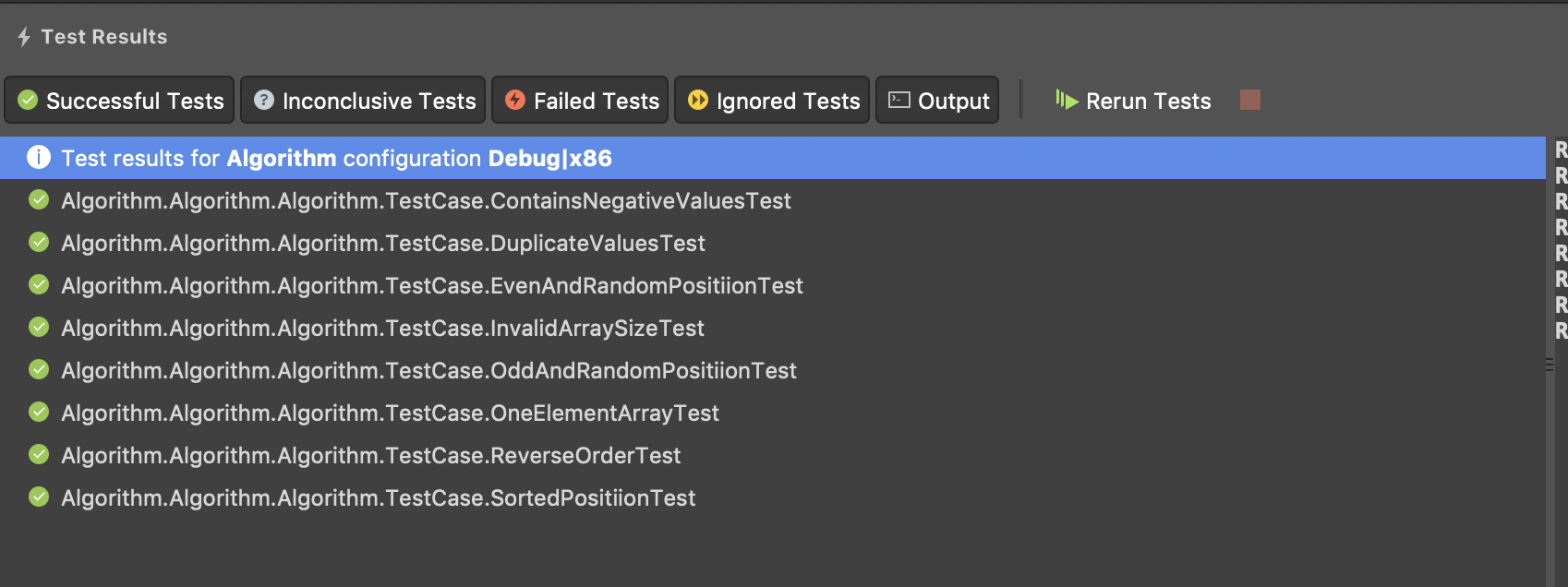
Appendix A – Brute Force Median Algorithm source code in C#



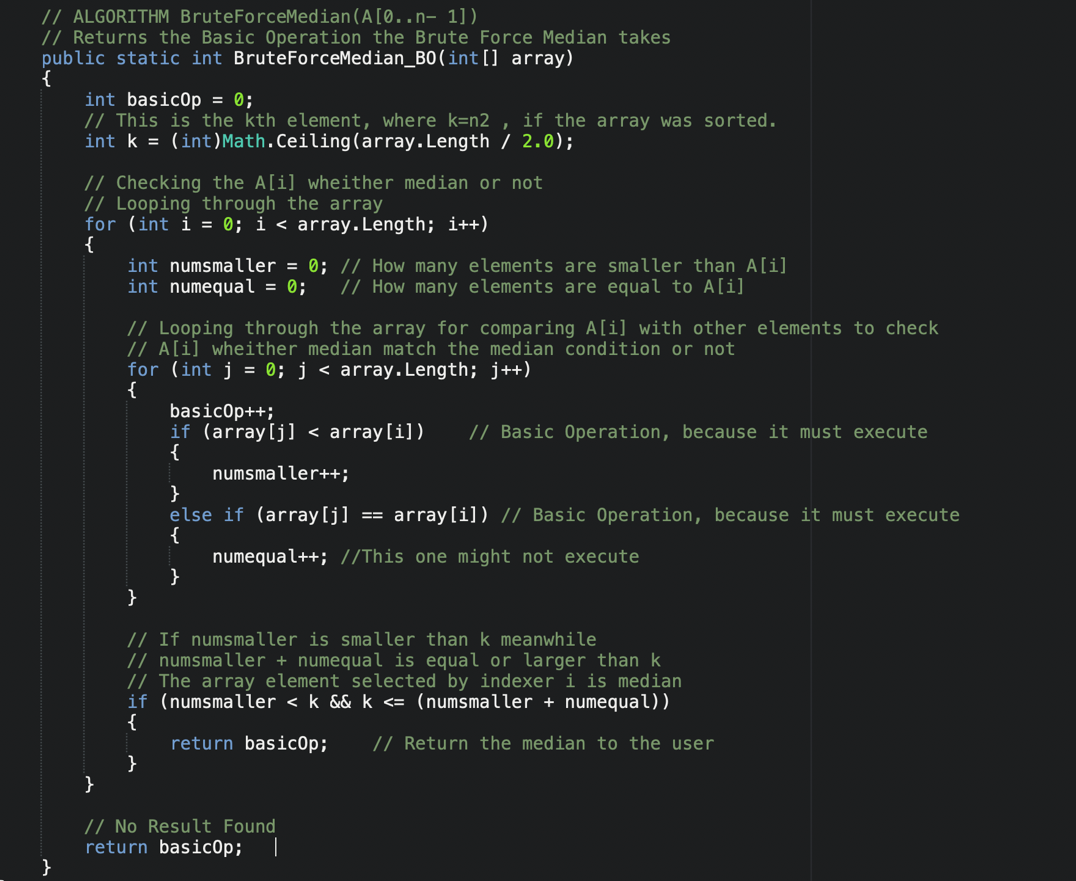
Appendix B-1 – Functional Test code for Brute Force Median Algorithm in C#

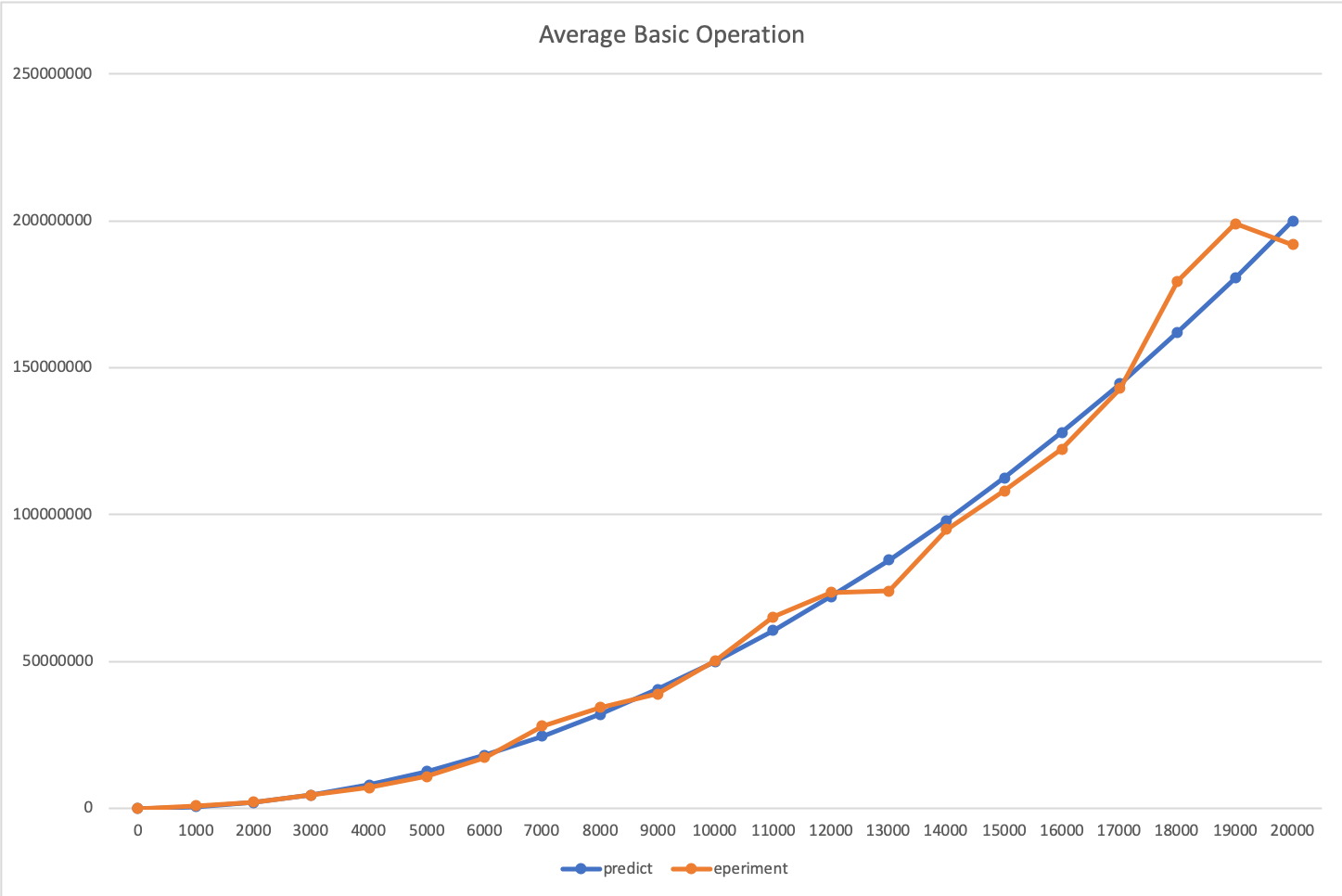
x

Appendix B-2 – Test result for Brute Force Median Algorithm



Appendix C – The Algorithm return basic operation for Brute force median algorithm



Appendix D – Average Basic Operation Graph

Appendix E – Average Execution Time Graph

