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

Table of Contents

[1. Summary 2](#_Toc6085177)

[2. Description of Algorithm 2](#_Toc6085178)

[3. Implementation of the Algorithm 2](#_Toc6085179)

[3.1. Program Implementation 2](#_Toc6085180)

[3.2. Functional Testing 3](#_Toc6085181)

[4. Design of Experiments 5](#_Toc6085182)

[4.1. Methodology, tools and techniques 5](#_Toc6085183)

[4.2. Data sizes, Test Data set 5](#_Toc6085184)

[5. Experiential results 6](#_Toc6085185)

[5.1. Basic Operation identification of the Algorithm 6](#_Toc6085186)

[5.2. Experiment on Calculating Average of Basic Operation 6](#_Toc6085187)

[5.3. Average execution time of the Algorithm 7](#_Toc6085188)

[5.4. Experiment on Calculating Average Execution time 8](#_Toc6085189)

[6. Analysis of Experiential results 8](#_Toc6085190)

[6.1. Theoretical Efficiency Prediction of the algorithm 8](#_Toc6085191)

[6.2. Average of Basic Operation Analysis 8](#_Toc6085192)

[6.3. Average of Execution Time Analysis 9](#_Toc6085193)

[7. Reference 10](#_Toc6085194)

[8. Appendices 11](#_Toc6085195)

# 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 an algorithm that takes an array of integers set and return the median value in the array. The algorithm was implemented in a C# Version 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 numbers in it. That is the reason why we need to implement the Brute Force Median algorithm to retrieve the median value in the array accurately no matter which kind of array the input is. The different types of the array include sorted, reverse, random, duplicate, single length, invalid length, odd length and even length array.

The Brute Force Median algorithm takes an array which contains integers and finds the median value of the array. It selects all the elements to compare it one by one in a nested for loop. In the iteration of the innermost loop, it checks the currently selected item by the outermost loop, returns it to the user if the value 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 an invalid format such as an 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 is also able to return the correct median value whatever the length of the array is odd or even numbers. Duplicate values 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. In theoretical perspective, the base case of the algorithm will only execute n of time if the first element of the array is always median. The worst case of the algorithm will execute n times exponentially because of assuming the median always at the last position. We can expect that the average case should takes the execution time between best case and worst case. 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 BFM.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.

Ceiling the result can help us pointing to the median position accurately If the length of the array is an odd number. 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 repeats 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 indexes 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 outermost 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 innermost 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 TestCase.cs file. **Appendix B-1** shows the unit test code for the brute force median algorithm. There are eight test functions 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 test cases should run in the unit test pad in the Visual Studio IDE. It provides an efficiency way to test the code in the project without running the main function in the program.cs file. These unit tests confirm that the implementation of the algorithm returning the median value accurately 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

# Methodology, tools and techniques

1. The Brute Force Median algorithm was 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 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 was 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. There is a tiny program in the program.cs file. It recorded the result of the basic operations and the execution times takes in the brute force algorithm into two separate csv files. It is a file then 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. The third-party library CsvHelper 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 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 0 to 20000, in increments of 1000.
2. The numbers in the array are generated randomly depends on the array size the algorithm is testing. The random number generator is used the real-time as seed and sorted the number randomly. It can ensure that the numbers in the array have a unique chance to be assigned to another position when the program runs at a different time.
3. The test data array in the data set is shuffled randomly and unpredictable. Every execution time might result in different outcome.
4. In order to predict the efficiency of the algorithm accurately, the experiment tested repeatedly sixty times on the same condition with totally different order test data set as the condition mentioned above. It means the chance that reversed, nearly sorted, sorted, random positioned array has an equal chance to appear in every test.

# Experiential results

# 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 numbers 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 other's element again and so on. This block of code executes most frequently in the algorithm. According to 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 two comparison statement in the innermost loop executes 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. Therefore, the basic operations in the brute force median algorithm are the two comparisons statement in the if-else statement which is inside the innermost for loop.

# Experiment on Calculating Average of Basic Operation

|  |  |  |
| --- | --- | --- |
| Input size | Predicted Basic Operation | Average Basic Operation over 60 Test |
| 0 | 0 | 0 |
| 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 5.A Average Basic Operation Comparison table

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 to execute the brute force median algorithm. As mentioned in section 5.1, the basic operations 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 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 of the algorithm executes comparing with the block inside the innermost for loop. Therefore, the basic operation counter should not be placed inside the if-else block. It should be placed at the first line of the innermost for loop. It can make sure that the counter has the same execution time likes the comparison statements in the innermost for loop.

The experiment of the basic operation was performed sixty times for calculating the number of basic operations performs in the brute force median algorithm. We generated the results from the random positioned array which the size starting from zero 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 sixty tests on each array size. The average basic operation is recorded to the Table above. The result reveals that all the result of the average case in the experiment is relatively closed to the result of the theoretical prediction.

# Average execution time of the Algorithm

The experiment of execution time uses the System.Diagnostics library in C# to record the execution time of the brute force algorithm. Dimitrov (2012) stated that Stopwatch object in System.Diagnostics library is designed for calculating the execution time of a method and is one of the best ways to measure time execution in .NET. 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. The library provides an easy and fast method to recording the execution time accurately in the C#.

**Appendix F** 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. Then, it executes the brute force median algorithm with the test dataset. The algorithm does not count the basic operations in this experiment in order to record the execution time as accurate as possible. Finally, it stops the time measurement After the algorithm finished the execution.

# Experiment on Calculating Average Execution time

|  |  |
| --- | --- |
| 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 5.B Average Execution time Comparison table

The experiment is the 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 zero to twenty thousand, increasing each array size by one thousand a time. It also performs sixty times the experience with the same condition of the test. It then records all sixty-test data into a csv file. Finally, these data are used to generate the average execution time for each array size from 0 – 20000. The average execution time data is then recorded in Table B above.

# Analysis of Experiential results

# Theoretical Efficiency Prediction of the algorithm

The average case calculated by theoretical prediction is assumed the fact that each number in the array has an equal chance of being the median value. It means that the median values have 1/ n – 1 chance lies in any position. **APPENDIX E** shows the mathematical function that calculates the average prediction. For instance, an array contains 5000 elements should operate around 12502500 basic operations. If the length of the array is 10000 should operate around 50005000 basic operations.

# Average of Basic Operation Analysis

**APPENDIX F** reveals the average basic operation in the line graph. The yellow line represents the result of the experiment. On the other hand, the blue line highlights the result of the mathematical prediction. It is significant to note that the result from the experiment is extremely similar to the result of the mathematical prediction. Although, the result of the experiment is slightly smaller or bigger than the mathematical prediction results in some points. because the data set is not always generated fairly in some of the points. In a statistic perspective, the experiment results are in a 90% confidence interval of the mathematical prediction results in all the time calculated by the result from **Table 5A** . We can conclude that the experience proved that the mathematical function predicts accurately for the algorithm.

# Average of Execution Time Analysis

**APPENDIX G** shows the average execution time of the algorithm. The execution time grows extremely similarly likes the basic operation line in **APPENDIX F.** The line graph demonstrated that the execution time growing exponentially as the array length increase. To conclude that the execution time matches the theoretical prediction mentioned above.

# Reference

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

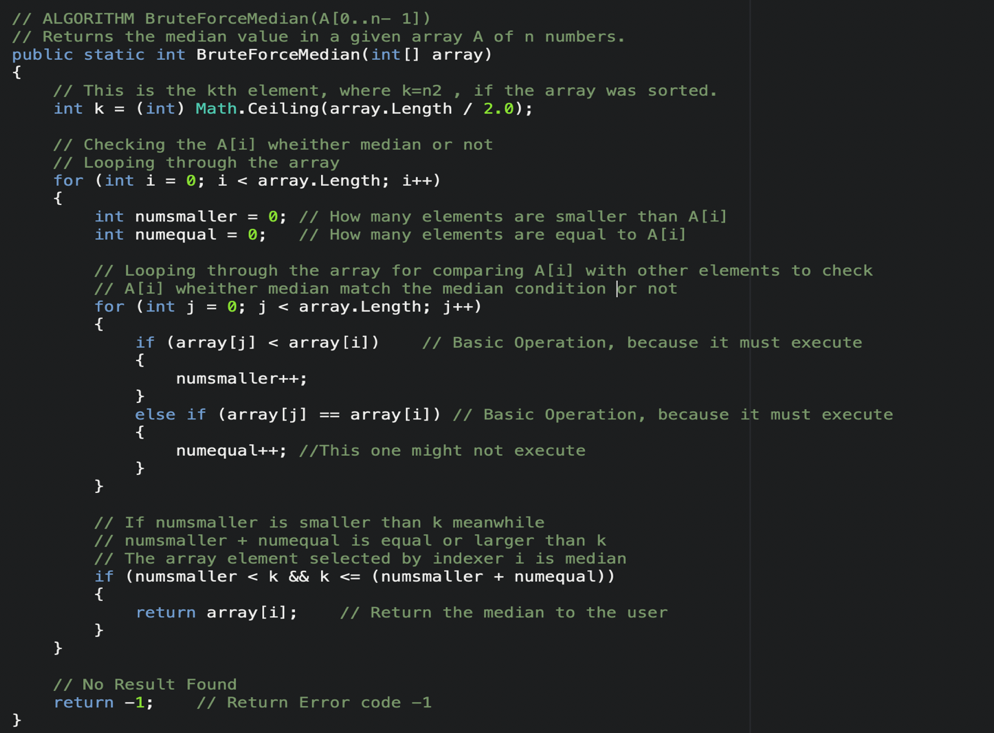
CsvHelper. (2019). Retrieved from <https://joshclose.github.io/CsvHelper/examples/writing/write-anonymous-type-objects/>

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>

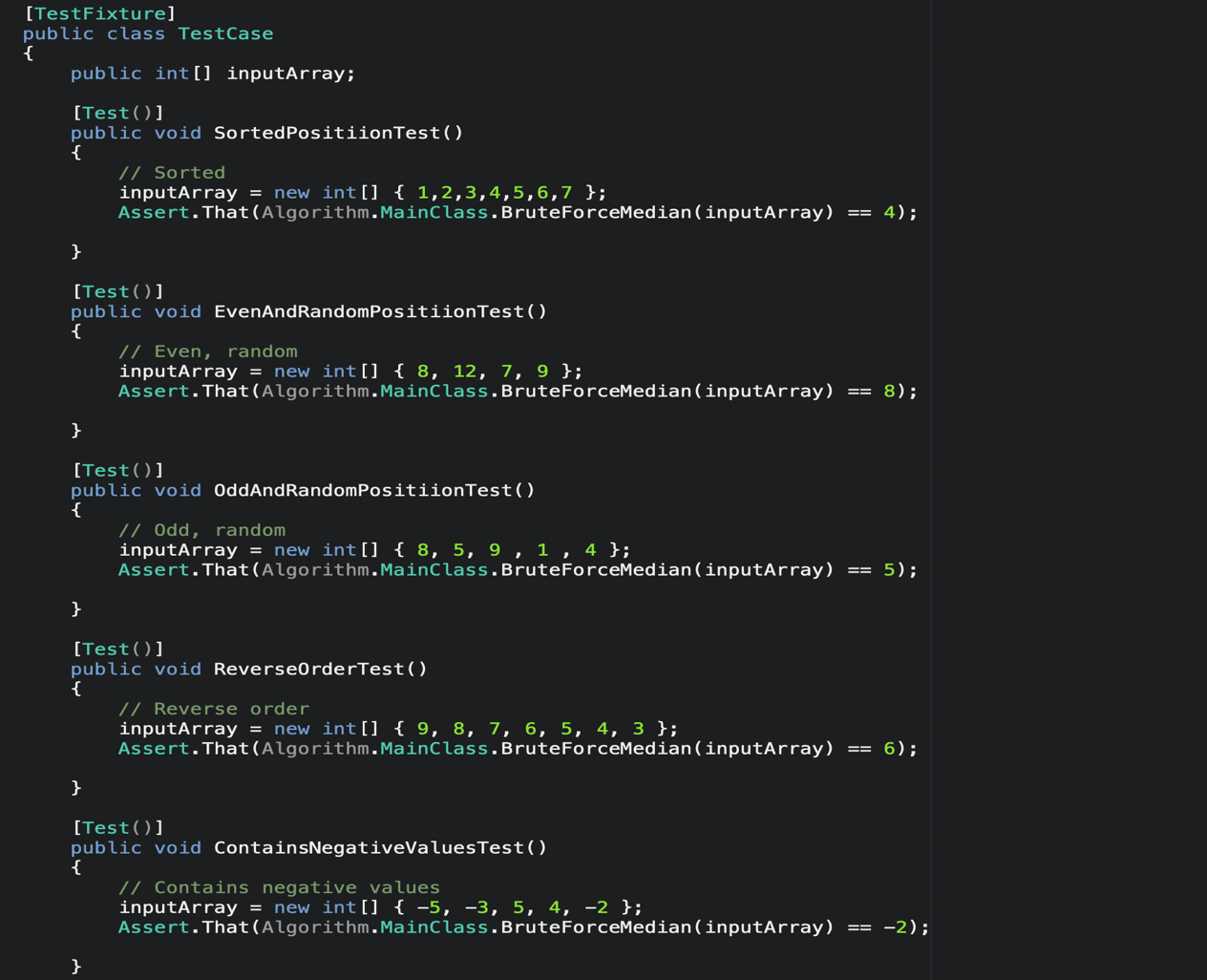
Tang, M. (2019). CAB301-Lecture2. Retrieved from <https://blackboard.qut.edu.au/bbcswebdav/pid-7900328-dt-content-rid-23380690_1/courses/CAB301_19se1/CAB301-Lecture2%282%29.pdf>

# Appendices

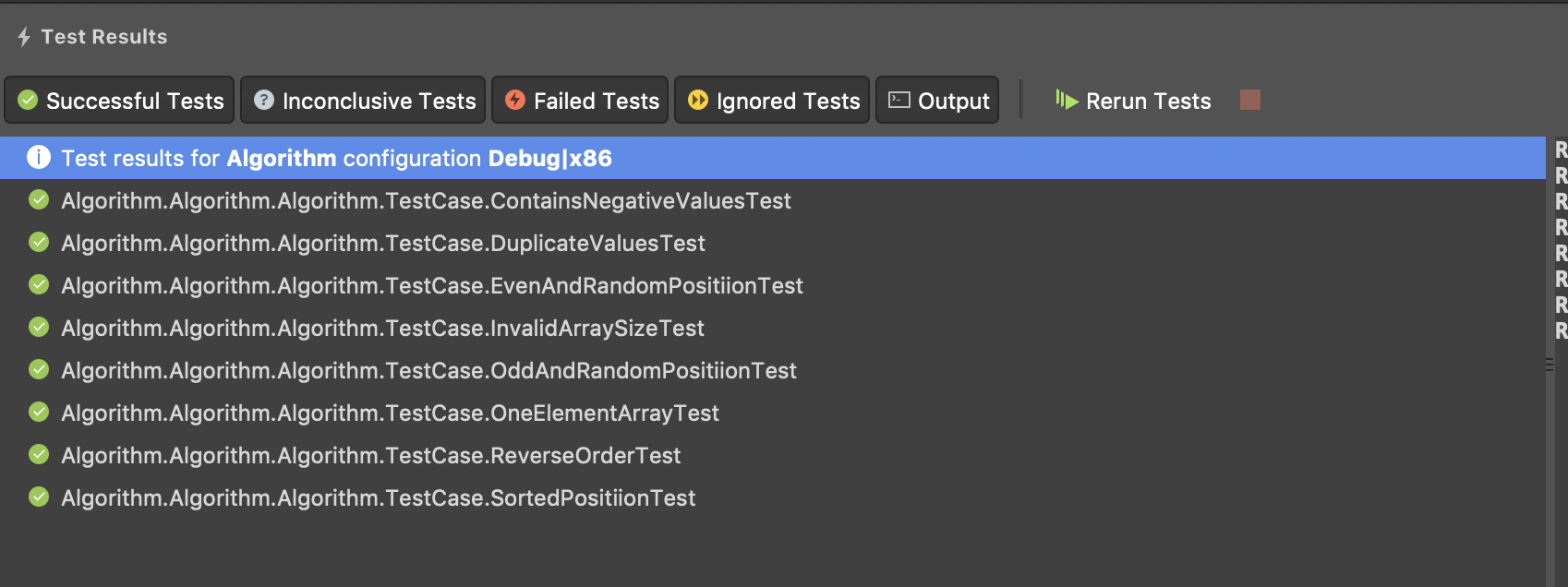
Appendix A – Brute Force Median Algorithm source code in C#



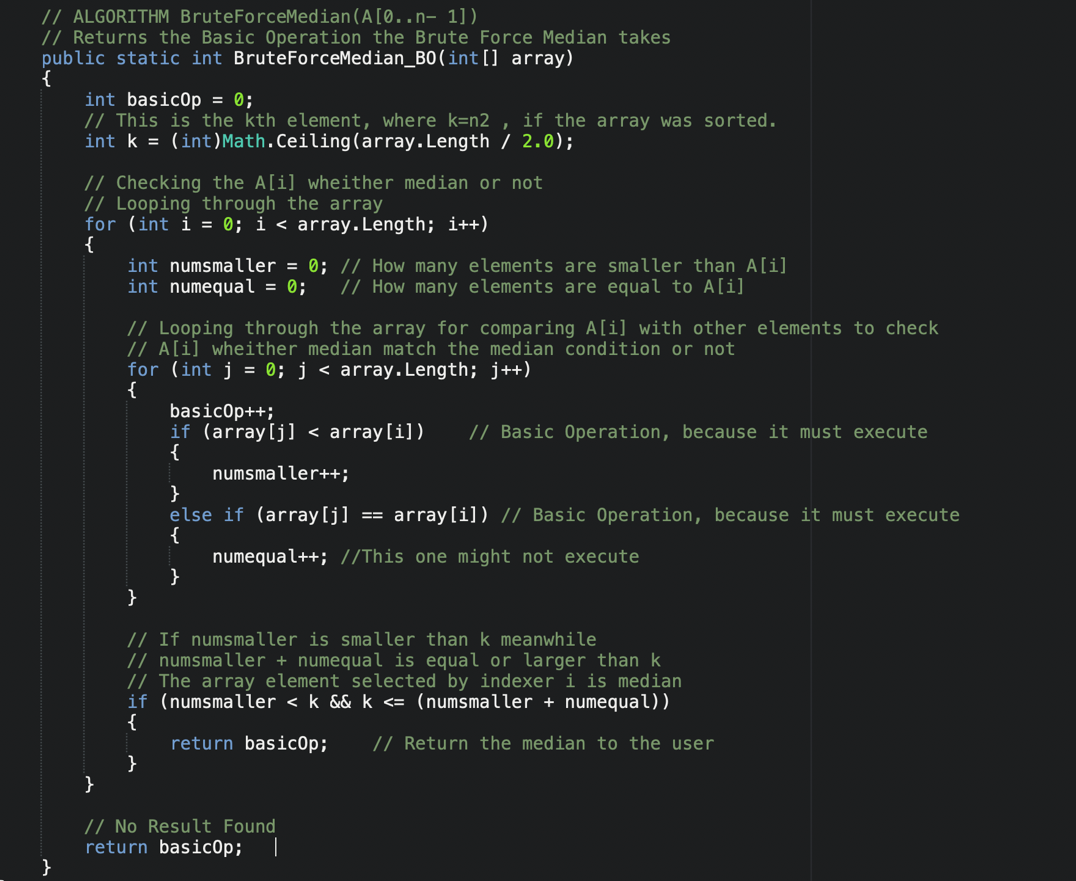
Appendix B-1 – Functional Test code for Brute Force Median Algorithm in C#

x

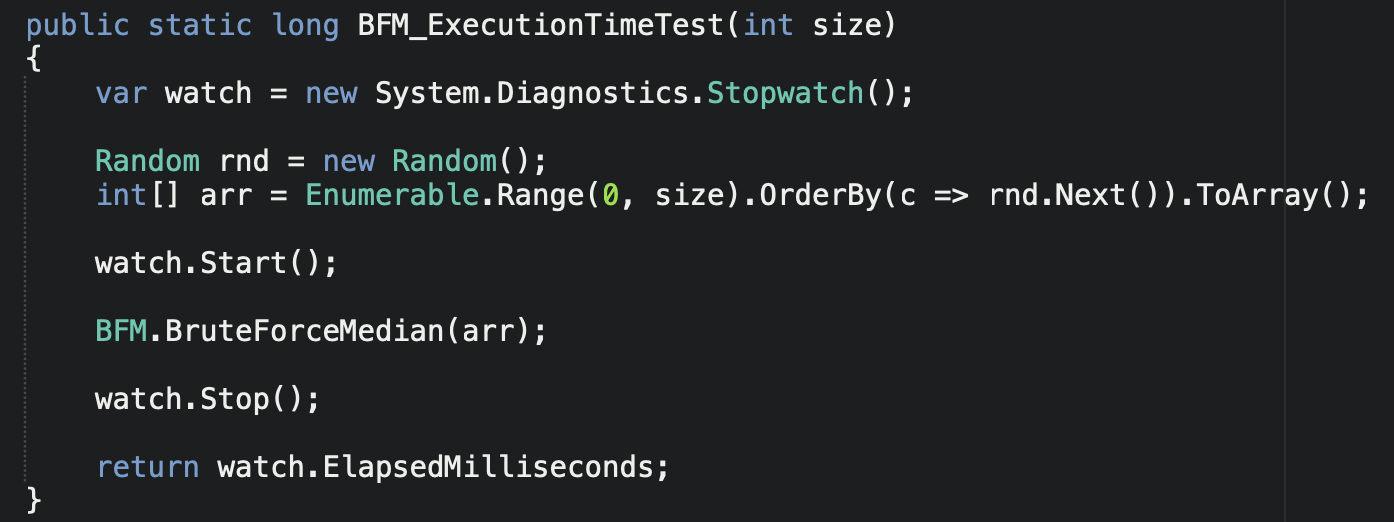
Appendix B-2 – Test result for Brute Force Median Algorithm



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

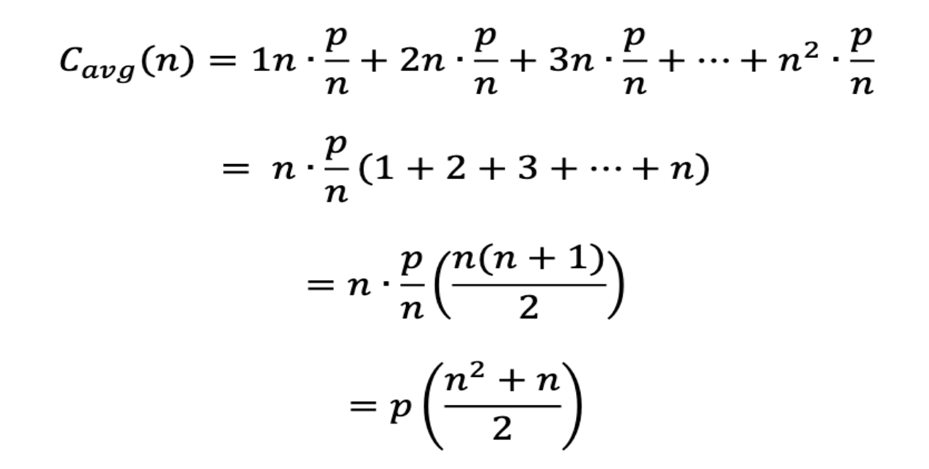


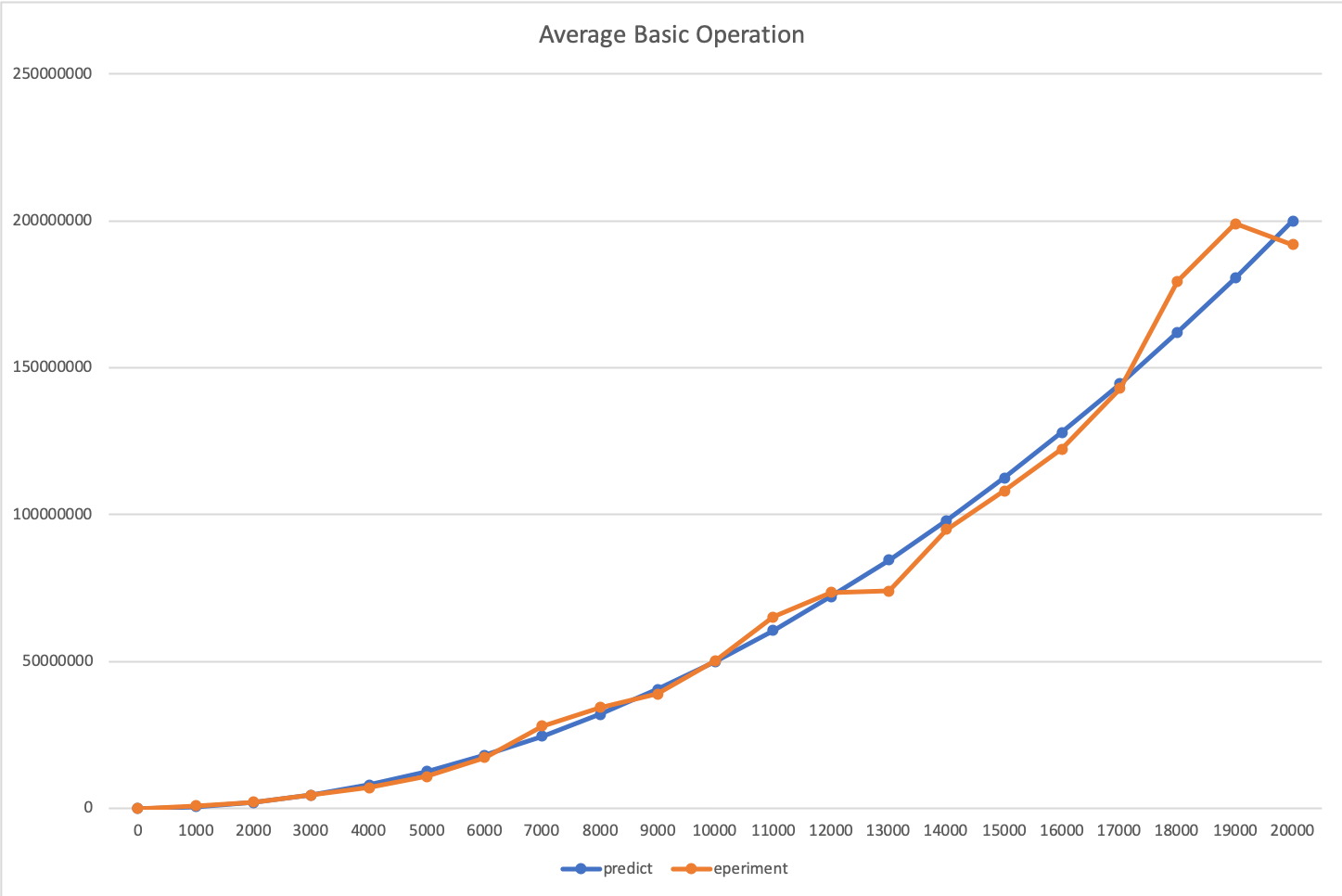
Appendix D – Brute Force Median Algorithm Execution time test code



Appendix E – Mathematical function for Calculating Average-Case Efficiency

Of Brute force median algorithm



Appendix F – Average Basic Operation Graph of the algorithm

Appendix G – Average Execution Time Graph of the algorithm

