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

[1.Description of Algorithm (1 Page) 2](#_Toc5389909)

[2. Implementation of the Algorithm (2 Pages) 2](#_Toc5389910)

[2.1. Functional correctness 2](#_Toc5389911)

[3. Experiment Design (2 Pages) 3](#_Toc5389912)

[3.1. Methodology, tools and techniques 3](#_Toc5389913)

[3.2. Data sizes, Test Data set 3](#_Toc5389914)

[4. Experiential results (4 Pages) 3](#_Toc5389915)

[4.1. Basic Operation identified 3](#_Toc5389916)

[4.2. Average execution time 3](#_Toc5389917)

[4.3. Experience to measure the program execution times 3](#_Toc5389918)

[5. Analysis of Experiential results (4 Pages) 3](#_Toc5389919)

[5.1. Experimental results 3](#_Toc5389920)

[5.2. Comparing against the theoretical efficiency prediction 3](#_Toc5389921)

[6. Reference (1 Pages) 3](#_Toc5389922)

[7. Appendices (4Pages) 3](#_Toc5389923)

# Summary

# 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. This algorithm only takes one parameter which only accept a array of integers. Considering the different condition that the array is ordered by, we cannot simply take the value in the middle position of the array as the median value of the data set. Because the array might be sorted randomly or exist duplicate number in it at random position. That is the reason why we need to implement Brute Force Median algorithm to retrieve the median value in the array accurately.

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 returns the current selected item value if it finds the value is the median.

Discusss more the effiency

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 even number. Because it has no correct way to define the median position for even number of the array length. 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 how many integer numbers exists in the input array. There are total two for loops in this case. Both loops create an indexer variable for selecting elements in the array for different purpose inside the loop. These indexers start from 0 to the number of the array length. 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 which selected by the indexer in the outer loop.

In the outer loop, it creates an Indexer variable denoted as i. This indexer only allowed to update inside the outer loop. There are two local variables inside this loop. The first variable is named as numsmall which is responsible to record 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 loops through 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 |

# 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. The experimental results should be accurate to compare with the theoretical predictions.

1. The results were recorded on two separate excel spreadsheets to record fifty tests of the basic operations and the execution times. Using the results, we were able to produce graphs using the built-in line graph function to view and analyse the results.

# Data sizes, Test Data set

# Experiential results (4 Pages)

# Basic Operation identified

# Average execution time

# Experience to measure the program execution times

# Analysis of Experiential results (4 Pages)

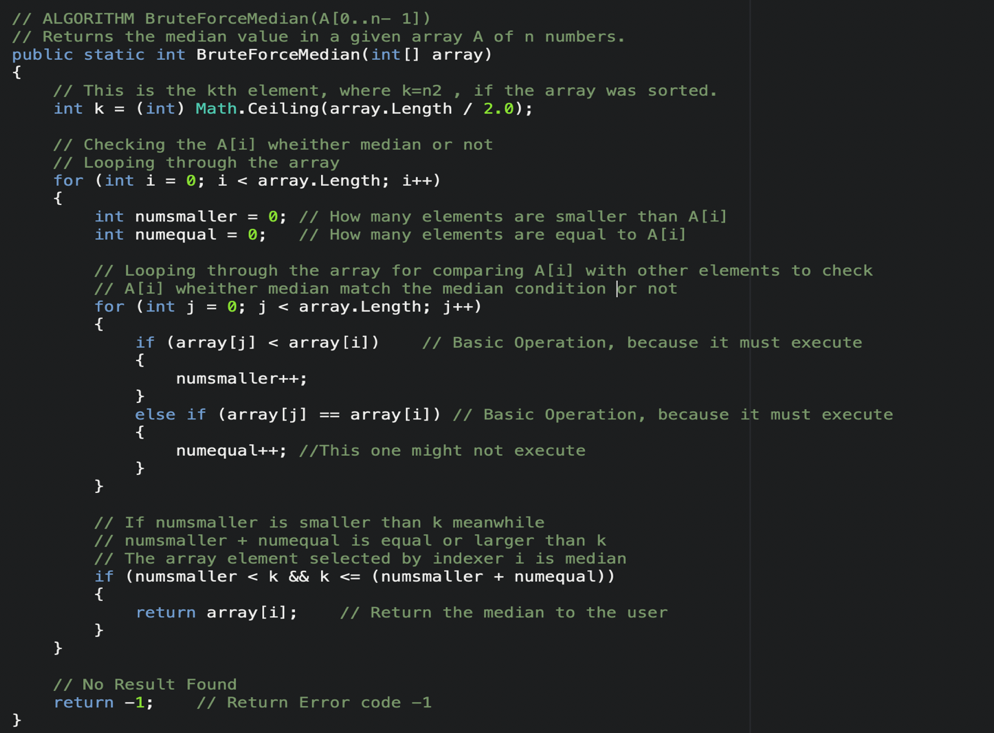
# Experimental results

# Comparing against the theoretical efficiency prediction

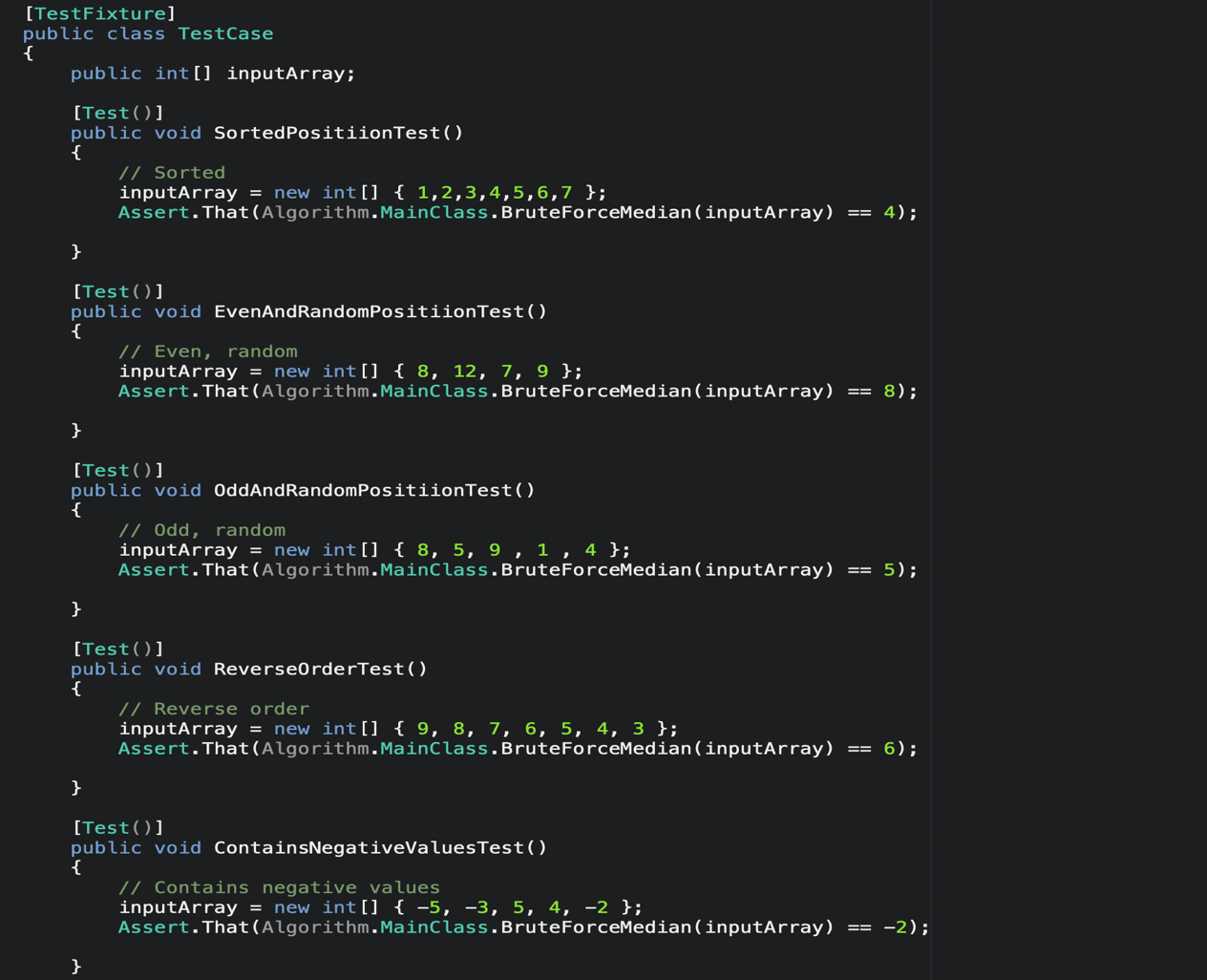
# Reference (1 Pages)

# 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#



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

