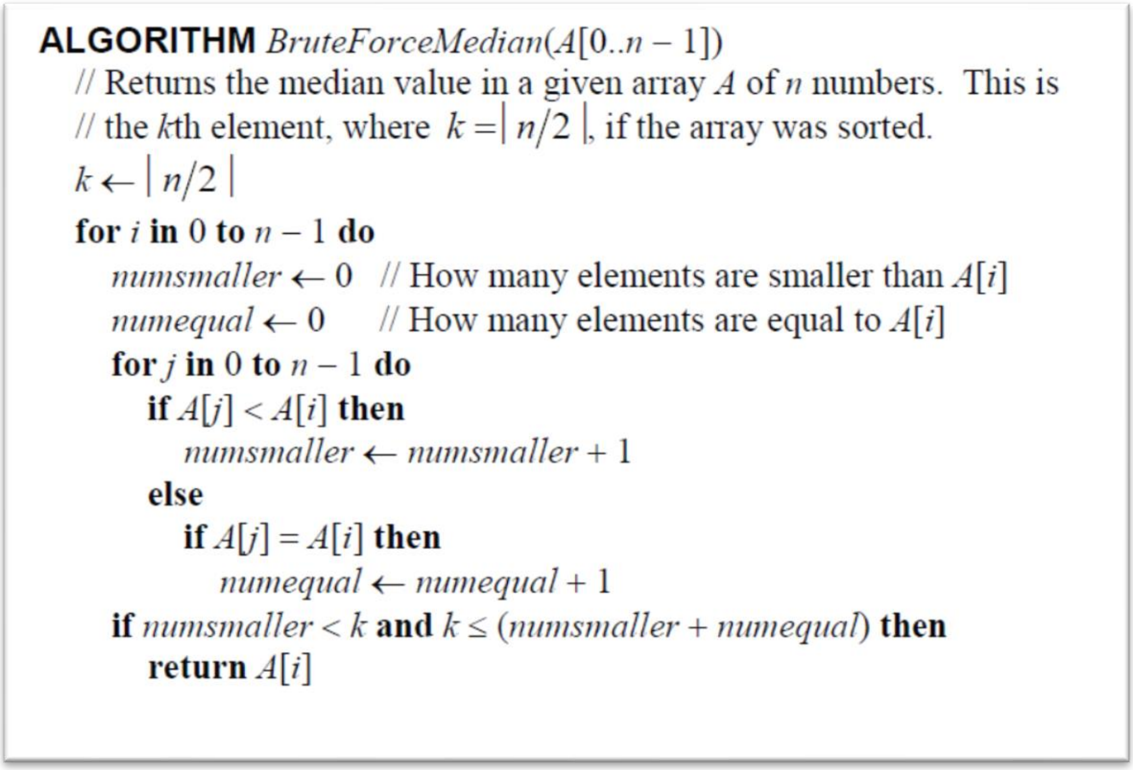
CAB301 Assignment 1 Report

Name: *Vaibhav Vachhani*

Student number: *9796134*

Table of Contents

Description of Algorithm



```
ALGORITHM BruteForceMedian($A[0..n-1]$)
    // Returns the median value in a given array $A$ of $n$ numbers. This is
    // the $k$th element, where $k = \lfloor n/2 \rfloor$, if the array was sorted.
    $k \leftarrow \lfloor n/2 \rfloor$
    for $i$ in 0 to $n-1$ do
        $numsmaller \leftarrow 0$   // How many elements are smaller than $A[i]$
        $numequal \leftarrow 0$     // How many elements are equal to $A[i]$
        for $j$ in 0 to $n-1$ do
            if $A[j] < A[i]$ then
                $numsmaller \leftarrow numsmaller + 1$
            else
                if $A[j] = A[i]$ then
                    $numequal \leftarrow numequal + 1$
        if $numsmaller < k$ and $k \le (numsmaller + numequal)$ then
            return $A[i]$
```

Figure 1: Pseudocode for Algorithm

This algorithm returns the median of an array given the array is sorted. It mainly consists of two for loops. Hence, its efficiency class is Big-theta (n^2). Which means the run time of this algorithm grows quadratically as its input size increases.

The behaviour:

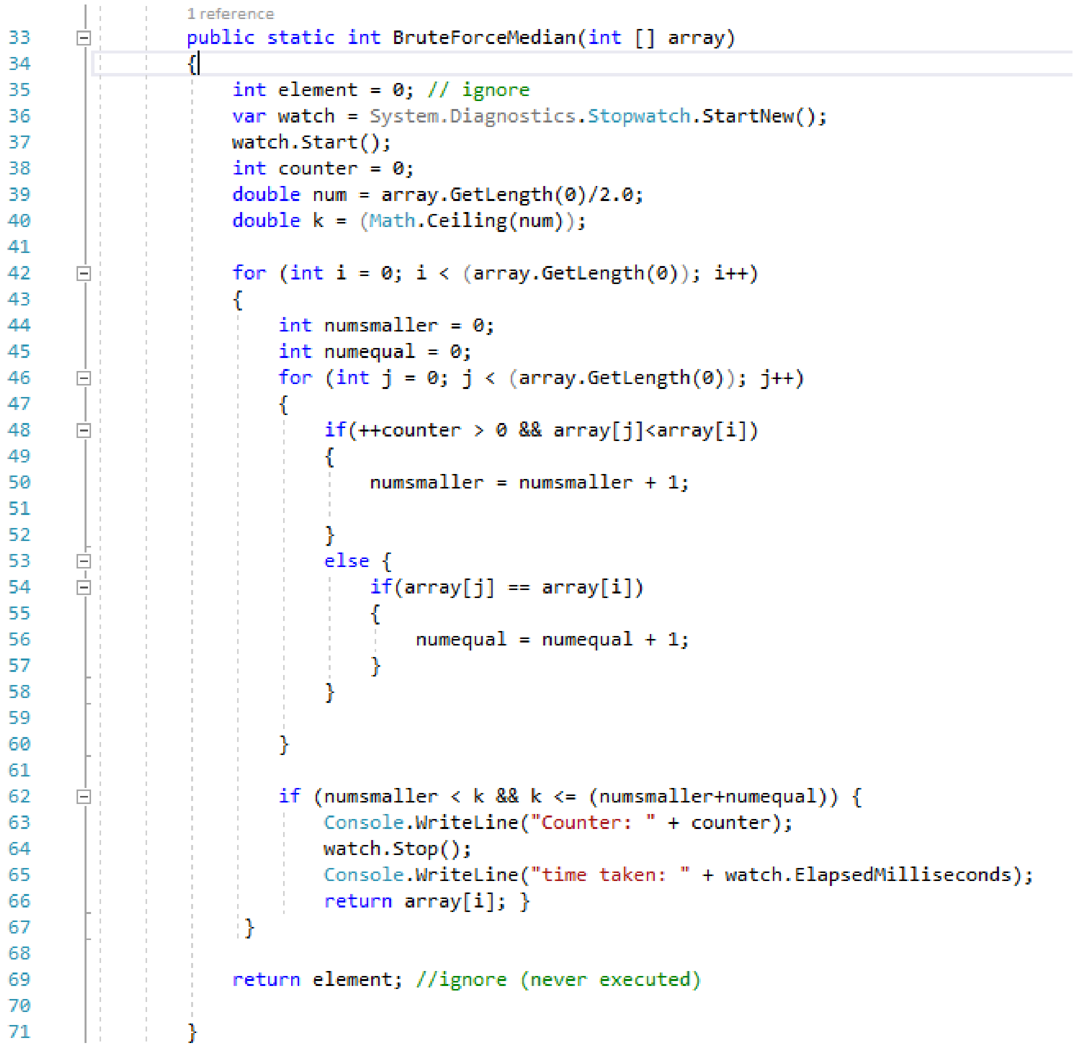
The algorithm will choose the middle most element if the number of elements in array given is even. For example, if array = [1,2,3…20], it will return 10. If array = [1,2,3…50], it will return 25. For odd number of elements (n), it returns the upper bound or the 'ceiling value' of n/2. For example, if array = [1,2,3…15], it will return 8 (15/2=7.5).

Basically, for any given sorted array with even number of elements (n), it will return the element at the index [(n/2)-1]. For odd number of elements, it will return the element at the index [floor(n/2)].

The functionality:

The algorithm compares each element in the array with each element in the array. If element is smaller than the other element – if not, are they equal? The variable *numsmaller* is incremented by 1 in case of the first comparison being true and the variable *numequal* is incremented by 1 in case of the second comparison being true. It finally returns the element where the variables satisfy the condition (*if (numsmaller < k && k <= (numsmaller+ numequal))*) where k is the ceiling value of lengthOfArray / 2.

Implementation of the Algorithm



```
   1 reference
33 public static int BruteForceMedian(int [] array)
34 {
35     int element = 0; // ignore
36     var watch = System.Diagnostics.Stopwatch.StartNew();
37     watch.Start();
38     int counter = 0;
39     double num = array.GetLength(0)/2.0;
40     double k = (Math.Ceiling(num));
41 
42     for (int i = 0; i < (array.GetLength(0)); i++)
43     {
44         int numsmaller = 0;
45         int numequal = 0;
46         for (int j = 0; j < (array.GetLength(0)); j++)
47         {
48             if(++counter > 0 && array[j]<array[i])
49             {
50                 numsmaller = numsmaller + 1;
51 
52             }
53             else {
54                 if(array[j] == array[i])
55                 {
56                     numequal = numequal + 1;
57                 }
58             }
59 
60         }
61 
62         if (numsmaller < k && k <= (numsmaller+numequal)) {
63             Console.WriteLine("Counter: " + counter);
64             watch.Stop();
65             Console.WriteLine("time taken: " + watch.ElapsedMilliseconds);
66             return array[i]; }
67     }
68 
69     return element; //ignore (never executed)
70 
71 }
```

Figure 2: C# method implementing the Algorithm

My choice of language is C# with Visual studio 2017 as IDE. As the return type of this method is ***int***, I was required to return a dummy variable to overcome the error '*not all code paths return a value*'. This variable does not affect the code in any way as the return statement on line 69 is never executed.

The stopwatch starts before variable declaration to ensure each operation is accounted for and hence get more accurate results. I have used built-in method *GetLength* with an argument of 0 as our array will be always 1 dimensional, to retrieve the length of the array. Next, I have used built-in method *Ceiling* from *Math* library to get the ceiling value of (lengthOfArray/2). [Appendix 1]

The main body of the algorithm consists of nested for loops. Both for loops have the same parameters, to go through every element of the array. The purpose of the for loops is to compare each element of the array with the whole array – even with itself. The first for loop goes through each element, and the nested for loop performs the comparisons. Two comparisons are performed with each element, is the element(j) smaller than – if not, is it equal to the variable set from the outer loop. If it is smaller, the variable *numsmaller* is incremented by 1 and if it is equal, the variable *numequal* is incremented by 1. After comparing one element with the whole array, we check if it satisfies the condition on line 62 (*if (numsmaller < k && k <= (numsmaller+ numequal))*) and return the respective element if true. If false, we reset the variables numsmaller and numequal to 0 and increment the variable for outer for loop by 1 to check the next element of the array.

This algorithm is complete – it will always return a value from the array given to it.

Testing and Results

Array = [1,2,3,4,5]   Result = 3

Array = `[1, 4, 7, 9, 10, 35, 63, 76, 111, 150, 167, 203]`   `Result = 35`

Formal proof of correctness

Array = [1,2,3,4,5]

Variable k will be 3 because number of elements = 5, hence ceiling value of (n/2) =3.

Stepping through both for loops manually:

First iteration of the outer loop (i=0)

Array[i]'s value is 1 & will not change throughout this iteration

array[j] equals array[i] so variable numequal will be increased from 0 to 1.

After 1st iteration of the inner loop the variables *numequal* and *numsmaller* will not change cause none of the other elements is equal to or smaller than array[0] which is 1. Hence the equation will not change.

Value of i	Value of i	Equation	Explanation
0	0	0 < 3 <= 1	array[j] equals array[i] so variable *numequal* will be increased from 0 to 1.
0	1	0 < 3 <=1	
0	2	0 < 3 <=1	
0	3	0 < 3 <=1	
0	4	0 < 3 <=1	

Second iteration of the outer loop (i=1)

Array[i]'s value is 2 & will not change throughout this iteration

Value of i	Value of j	Equation	Explanation
1	0	1<3<=1	array[j] < array[i] so variable *numsmaller* will be increased from 0 to 1.
1	1	1<3<=2	array[j] = array[i] so variable *numequal* will be increased from 0 to 1.
1	2	1<3<=2	
1	3	1<3<=2	
1	4	1<3<=2	

After 2nd iteration of the inner loop the variables *numequal* and *numsmaller* will not change cause none of the other elements is equal to or smaller than array[1] which is 2. Hence the equation will not change.

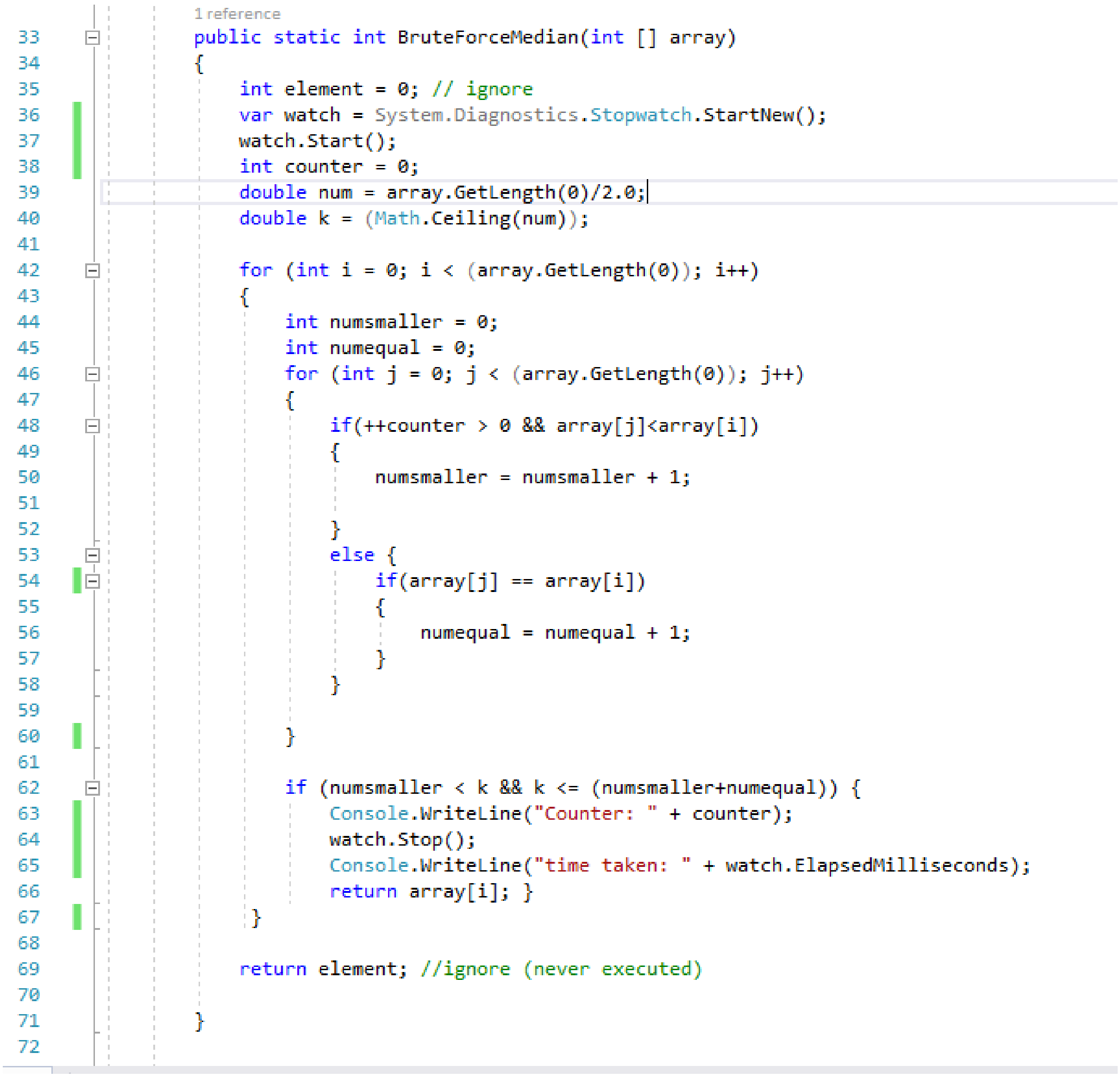
Third iteration of the outer loop (i=2)

Array[i]'s value is 3 & will not change throughout this iteration

Value of i	Value of j	Equation	Explanation
2	0	1<3<=1	array[j] < array[i] so variable *numsmaller* will be increased from 0 to 1.
2	1	2<3<=2	array[j] < array[i] so variable *numsmaller* will be increased from 1 to 2.
2	2	2<3<=3	array[j] = array[i] so variable *numequal* will be increased from 0 to 1.
2	3	2<3<=3	
2	4	2<3<=3	

After 3rd iteration of the inner loop the variables *numequal* and *numsmaller* will not change cause none of the other elements is equal to or smaller than array[2] which is 3. Hence the equation will not change. This equation is mathematically correct hence our element is array[2] = 3.

Design of experiments



```
public static int BruteForceMedian(int [] array)
{
    int element = 0; // ignore
    var watch = System.Diagnostics.Stopwatch.StartNew();
    watch.Start();
    int counter = 0;
    double num = array.GetLength(0)/2.0;
    double k = (Math.Ceiling(num));

    for (int i = 0; i < (array.GetLength(0)); i++)
    {
        int numsmaller = 0;
        int numequal = 0;
        for (int j = 0; j < (array.GetLength(0)); j++)
        {
            if(++counter > 0 && array[j]<array[i])
            {
                numsmaller = numsmaller + 1;

            }
            else {
                if(array[j] == array[i])
                {
                    numequal = numequal + 1;
                }
            }

        }

        if (numsmaller < k && k <= (numsmaller+numequal)) {
            Console.WriteLine("Counter: " + counter);
            watch.Stop();
            Console.WriteLine("time taken: " + watch.ElapsedMilliseconds);
            return array[i]; }
     }

    return element; //ignore (never executed)

}
```

Figure 3: Algorithm implementation in C#.

I have measured the number of times the basic operation is performed, and the time taken for the algorithm to execute in respective variables *counter & watch*.

DEVELOPMENT ENVIRONMENT:

This algorithm was developed and tested on a machine with the following specifications:

Operating System	Windows 10 Home
RAM	8.00 GB
System Type	64 bit, x64 processor
Processor	Intel Core i5-7200U 2.5GHz
Visual Studio Version	15.9.6
C# Tools Version	2.10.0

TEST DATA:

I have selected 11 data points ranging from 1000 to 65000 to plot a graph of *Array Size* vs *Counter* and *ArraySize vs Time*. Each array will be sorted. I avoided choosing small array sizes like 200 or 500 because its data points would group closely near the lower part of the graph (near the origin). We opt for larger data size so we can plot our graph for greater size and predict other values with greater accuracy. There is no trend through the data set chosen.

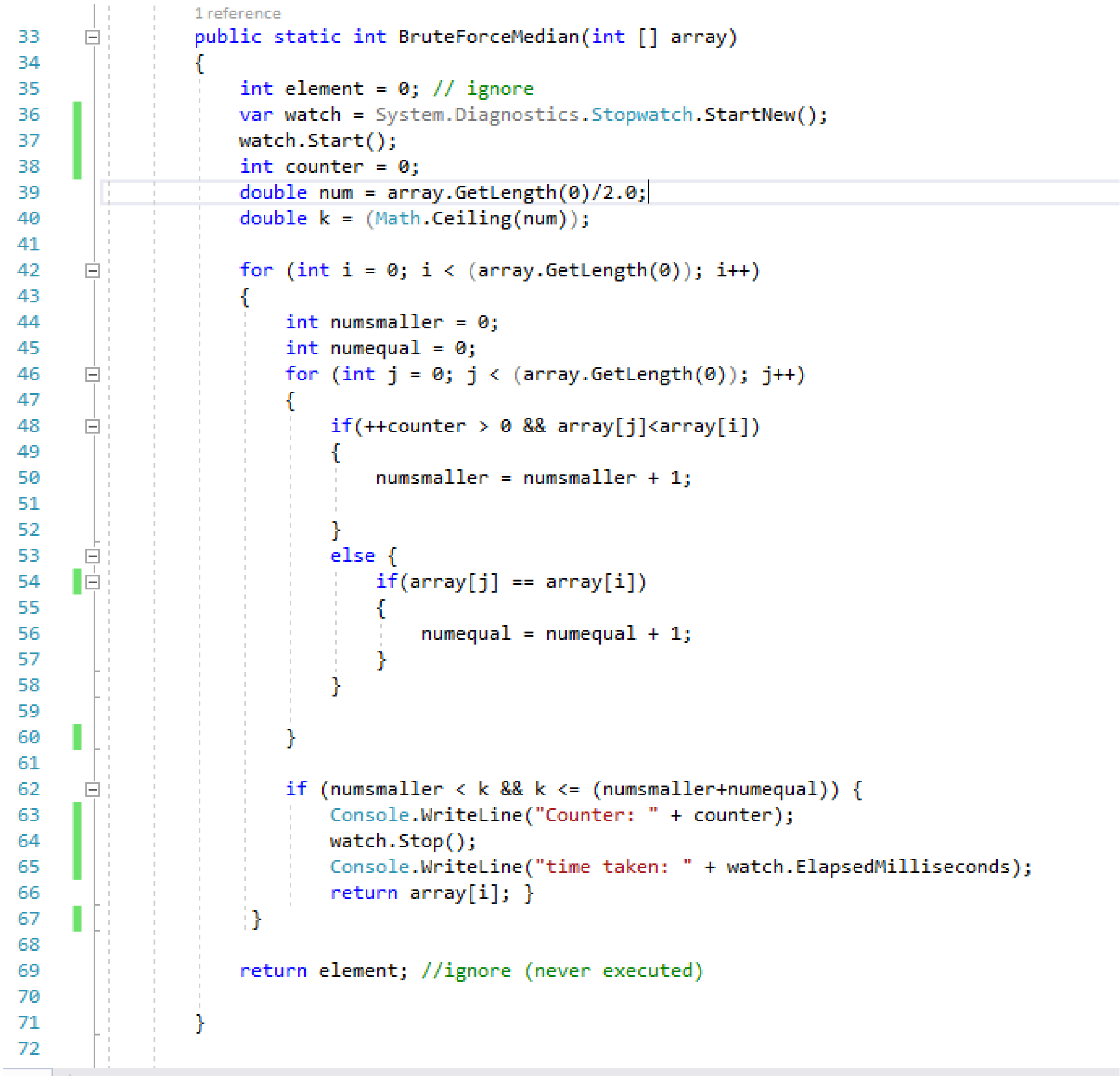
Array Size	Counter
1000	500000
2000	2000000
4000	8000000
6000	18000000
10000	50000000
15000	112500000
25000	312500000
40000	800000000
50000	1250000000
60000	1800000000
65000	2112500000

I have reserved 4 data points to check the consistency of the graph. I have used the trendline feature of MS Excel which generates a best fit line through the data points with an equation. In my case, the equation was y =0.5*(x)^2. Predicted value of the counter is evaluated from the equation where x is *ArraySize*. Actual value was the output from the code - number of times the basic operation was performed when I passed in an array of length *ArraySize.* As you can see, results were very consistent.

Array Size	Predicted Value	Actual Value
24581	302112781	302125071
34567	597438745	597456028
46987	1103889085	1103912578
54321	1475385521	1475412681

Experimental Results

Basic operation of the algorithm is the comparison *if (array[j]<array[i])* because this operation will be performed the most times given any sorted array. Note that even if the statement is not true, it will still make the comparison for every element as it is before the other comparison. As it compares an element with the entire array, it will be executed roughly (n^2) times where *n* is the length of the array.



```
33  public static int BruteForceMedian(int [] array)
34  {
35      int element = 0; // ignore
36      var watch = System.Diagnostics.Stopwatch.StartNew();
37      watch.Start();
38      int counter = 0;
39      double num = array.GetLength(0)/2.0;
40      double k = (Math.Ceiling(num));
41
42      for (int i = 0; i < (array.GetLength(0)); i++)
43      {
44          int numsmaller = 0;
45          int numequal = 0;
46          for (int j = 0; j < (array.GetLength(0)); j++)
47          {
48              if(++counter > 0 && array[j]<array[i])
49              {
50                  numsmaller = numsmaller + 1;
51
52              }
53              else {
54                  if(array[j] == array[i])
55                  {
56                      numequal = numequal + 1;
57                  }
58              }
59
60          }
61
62          if (numsmaller < k && k <= (numsmaller+numequal)) {
63              Console.WriteLine("Counter: " + counter);
64              watch.Stop();
65              Console.WriteLine("time taken: " + watch.ElapsedMilliseconds);
66              return array[i]; }
67       }
68
69      return element; //ignore (never executed)
70
71  }
72
```

I have placed the code to pre-increment the *counter* variable with the comparison such that it increments the variable first and then makes the comparison. This ensures that the comparison is accounted for even if it did not evaluate to true. (line 48)

The timer starts before any variable declaration (line 37). The timer is stopped just before the return statement as any code after the return statement won’t be executed.

Array Size	Counter	Time Taken (ms)
1000	500000	5
2000	2000000	20
4000	8000000	76
6000	18000000	167
10000	50000000	450
15000	112500000	1009
25000	312500000	2810
40000	800000000	7063
50000	1250000000	11002
60000	1800000000	15874
65000	2112500000	19553

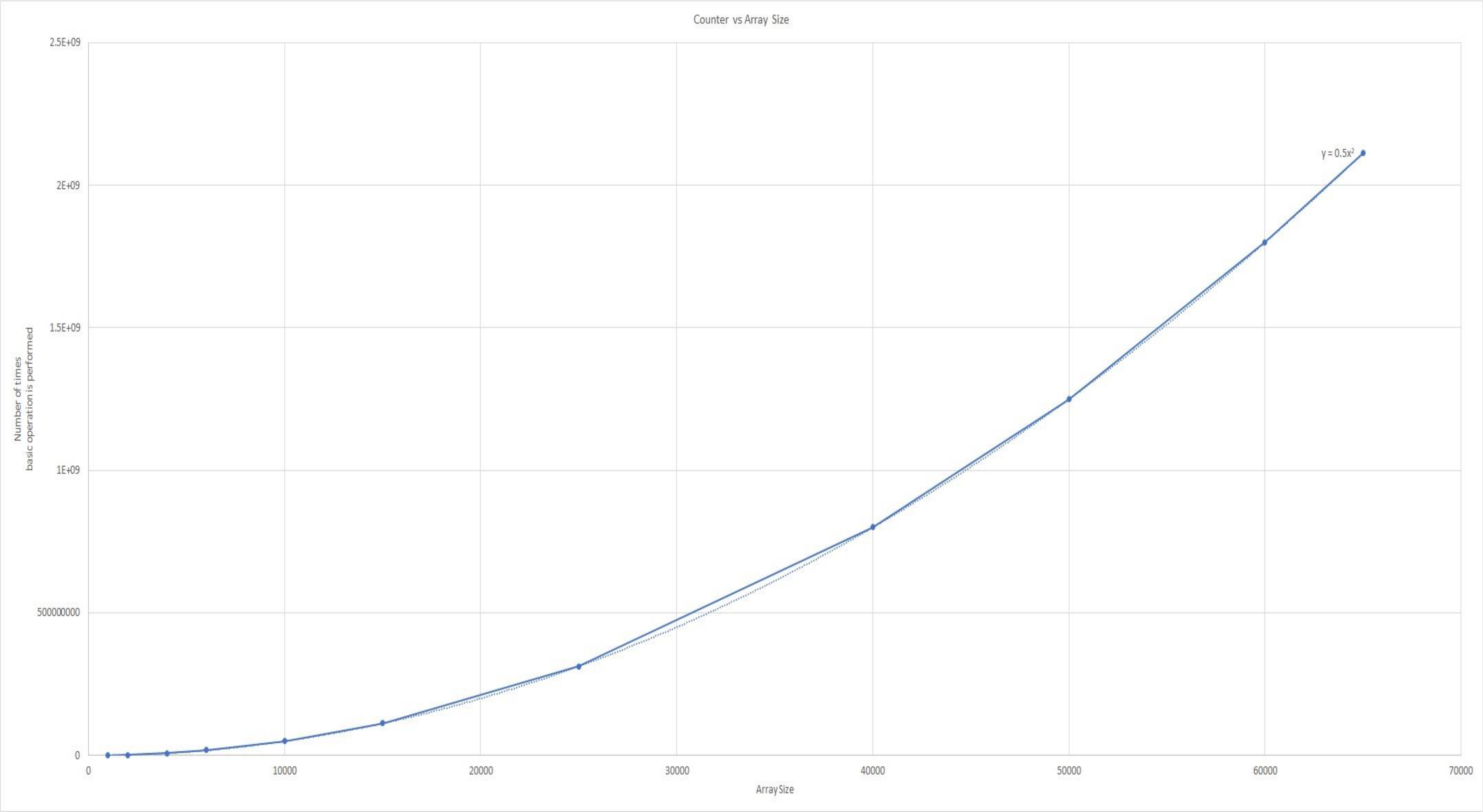
This is the data set used to measure the number of times the basic operation is performed, and time taken in milliseconds.

Analysis of Experimental Results

The graph below is the plotted with the following data points.

Array Size	Counter
1000	500000
2000	2000000
4000	8000000
6000	18000000
10000	50000000
15000	112500000
25000	312500000
40000	800000000
50000	1250000000
60000	1800000000
65000	2112500000

The best fit line through the points has the equation *y=0.5(x^2)*. This is consistent with the suggested big-oh notation of the algorithm which is *n^2.* The equation can be simplified to the big-oh notation as we ignore any constants present, which gives us *y=x^2* (x = array size). We can also confirm this manually without the help of MS Excel. We use the equation *y=c(x^2)*, we take two array sizes and its respective number of times the basic operation was performed to find an average value of c and use it to check other test data points. Taking data point (25000, 312500000) gives us the same value of c – 0.5.


Counter vs Array Size
Number of times
basic operation is performed
2.5E+09
2E+09
1.5E+09
1E+09
500000000
0
0
10000
20000
30000
40000
50000
60000
70000
Array Size
y = 0.5x²

This are the data points gathered from timing experiments.

Array Size	Time Taken (ms)
1000	5
2000	20
4000	76
6000	167
10000	450
15000	1009
25000	2810
40000	7063
50000	11002
60000	15874
65000	19553

The best fit line through the points has the equation *y=(6e-6)*x^1.97*. This can be considered consistent with the suggested big-oh notation of the algorithm which is *n^2* if we round the power to 2. The equation can be simplified to the big-oh notation as we ignore any constants present, which gives us *y=x^2* (x = array size).

The reasons behind the equation not being as accurate could be due to the unit of time measurement and the speed of execution. I experimented with some smaller array sizes and the output for time taken was always 0 milliseconds. Anyhow, if we manually try to find c in the equation y=c*x^2 using the last two data points, the average value of c is 0.00000451868. Using that value to predict the runtime for an array of size 55000 is 13669. But upon executing the algorithm with an array of size 55000, I found out the runtime to be 17063. This suggests that there are other factors affecting the runtime like cache memory.

Overall, both experimental results suggest that the algorithm's big-oh notation belongs to the class (n^2) where n is the size of array.

I have also attached the excel file containing the experiment data and respective graphs. Sheet 1 contains *ArraySize vs Counter* and sheet 2 contains *ArraySize vs Timer.*

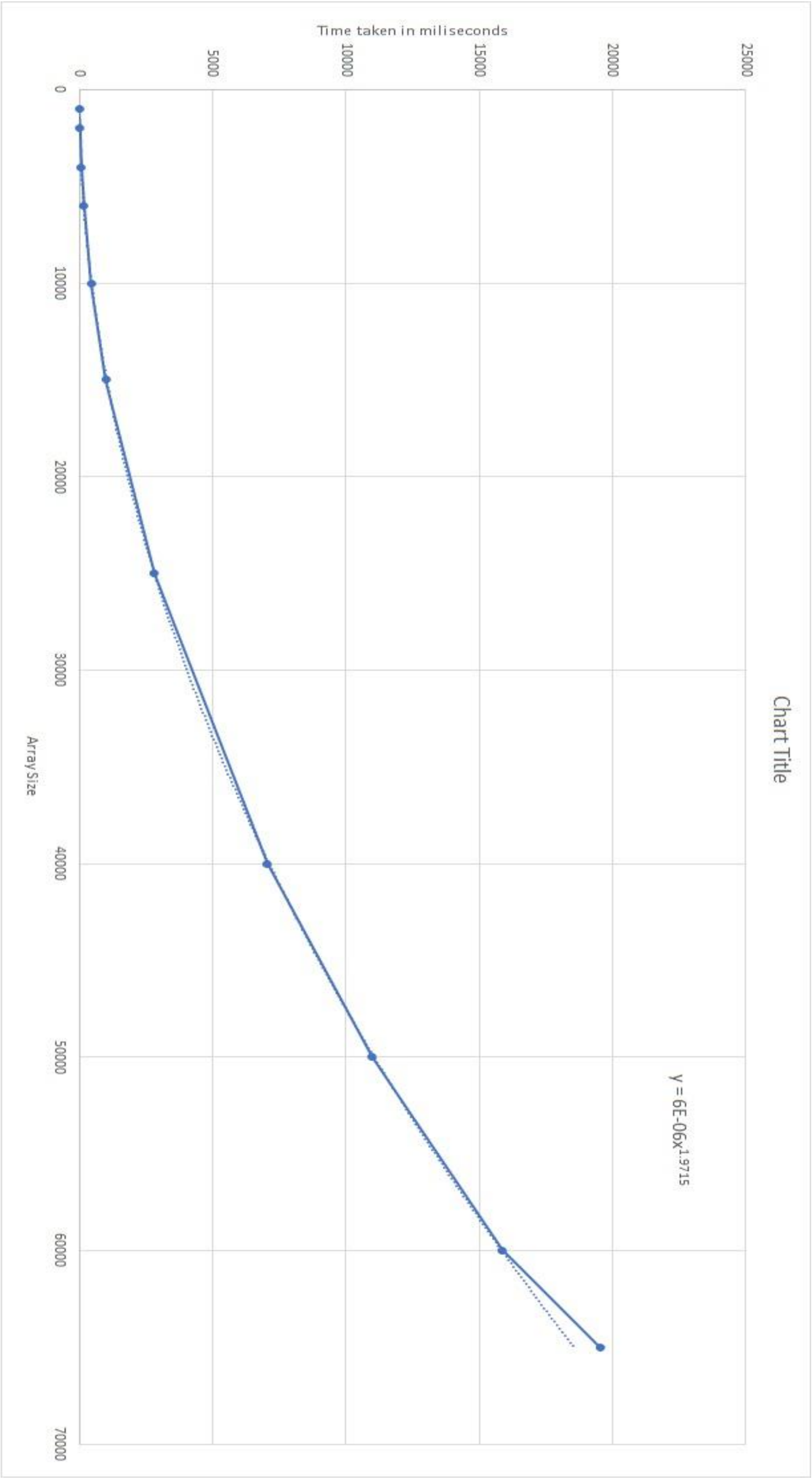


Figure 4: Time Taken vs Array Size Graph

Appendices

Math.Ceiling - https://docs.microsoft.com/en-us/dotnet/api/system.math.ceiling?view=netframework-4.7.2

Returns a ceiling value of the argument. For example, Math.Ceiling(5.4) would return 6.

MS EXCEL trendline – feature (https://support.office.com/en-ie/article/choosing-the-best-trendline-for-your-data-1bb3c9e7-0280-45b5-9ab0-d0c93161daa8)

It enables the user to generate a best fit line/curve through the data points and generate an expression for that line/curve. You can choose an option from (linear, logarithmic, exponential, power, polynomial, etc.) I opted for power as our suggested big-oh notation was *n^2.*

Submission zip file contains a MS Visual Studio Solution file, a MS EXCEL file, 2 photos of the graphs and this report. The contents of the solution are:

- Algo class – contains implementation of the algorithm with the counter. (*BruteForceMedianWithCounter* method)
- Algo1 class – contains implementation of the algorithm with the timer. (*BruteForceMedianWithTimer* method)
- Program class – contains implementation of the algorithm **without** timer or counter and a main class to call methods.