

CAB301

Algorithms and Complexity

ASSIGNMENT1

EMPIRICAL ANALYSIS OF AN

ALGORITHM

Larysa McGookey
n9696563

23/03/19

TABLE OF CONTENTS

1	Description of the Algorithm	2
2	Implementation of the Algorithm	3
3	Design of experiments	7
4	Experimental results	9
5	Analysis of experimental results	13
6	References	19
7	Appendices	20

DESCRIPTION OF THE ALGORITHM

1. The algorithm for this assignment is a BruteForceMedian algorithm and the main goal of the algorithm is to return the median value of a given array of values. The way an algorithm works is:
 - For a given array it calculates a middle element of an array (the kth element). This element is calculated as a ceiling value of total number of elements (n) divided by 2. For an odd number of elements it is an element which is halfway in the array. For an even number of elements it is an element which is located at $n/2$ position of the array. As it will not be a whole integer number, its a ceiling value where a return is a closest integer greater than a $n/2$ position of an element.
 - Algorithm starts with the first element of an array. For each array element (ith) we declare integer variables (numsmaller) and (numequal) of counting array elements. These variables count the number of elements which are smaller than the ith element (numsmaller) and are equal to the ith element (numequal). The initial value of these variables is set to zero. The algorithm goes through each element (i) of the array and compares each element to all other elements (jth) of the array. If a jth element is smaller than the current element (i), the value of numsmaller variable is getting increased by one. But if an jth element is equal than the current element (i), the value of numequal variable is getting increased by one.
 - After an ith element went through a comparison with all the elements of the array, values numsmaller and numequal are compared with the position number of a middle element k. If a number of smaller values (numsmaller) is less than a value of k and a total number of smaller values with the number of equal values is larger or equal k, the ith element is the median value of the array. If not, the algorithm goes to the next ith element – $i+1$ and starts comparing process the same way as for the ith element. These steps continue until condition of the number of smaller values (numsmaller) become less than a value of k and a total number of smaller values (numsmaller) with the number of equal values (numequal) become larger or equal a middle value k. When this condition is true, the current ith element of the array is the median value of an array and an algorithm returns a value of this ith element as a result. [1]

2. IMPLEMENTATION OF THE ALGORITHM

The purpose of BruteForceMedian algorithm is to return a median value of an element in a given array A of n numbers. C# programming language was used for implementation of a given algorithm. Please see below comments in green as an explanation of each algorithm functionality. The output of an algorithm please see in Appendix B.

```
namespace Assignment1_algorithm
{
    class Program
    {
        static void Main(string[] args)
        {
            // initial data for an algorithm- a given array and a number of
            elements in array calculated as a length of an array
            int[] Array = { 4, 7, 2, 8, 9, 11, 1, 3, 23, 0, 5, 92, 34, 15, 45, 67,
21, 34, 78, 563 };
            int n = Array.length;

            // invoking a function of BruceForceMedian algorithm calculation by
            supplying an array length and array elements
            Compare(n, Array);

            //Writing an initial array for a user
            Console.WriteLine("Array :");
            printArray(n, Array);
            Console.WriteLine();

            Console.ReadLine();
        }

        // BruceForceMedian algorithm script represented as a Compare function
        static int Compare(int n, int[] a)
        {
            //Calculation of a median position- kth element of an array, taking
            into account that it must be a ceiling value
            int k = Convert.ToInt32(Math.Ceiling((double)n / 2));

            //Starting from the first element A[i], go through all the array
            elements and for each ith element
            for (int i = 0; i <= (n - 1); i++)
            {
                // variable declaration and its type to count how many elements are
                smaller than a current element A[i] of array. Set a value to zero.
                int numsmaller = 0;
                // variable declaration and its type to count a how many elements are
                equal to a current element A[i] of array. Set a value to zero.
                int numequal = 0;
                //compare current element of an array A[i] to all array elements A[j]
                for (int j = 0; j <= (n - 1); j++)
                {
                    //if each element of array A[j] is smaller than a current element
                    A[i], add 1 to a variable of count smaller values numsmaller
                    if (a[j] < a[i])
                    {
                        numsmaller++;
                    }
                }
            }
        }
    }
}
```

```

    }
    // but if each element of array A[j] is equal to a current
element A[i], add 1 to a variable of count equal values numequal
    else if (a[j] == a[i])
    {
        numequal++;
    }

}
//If amount of small elements (numsmaller) less than a position of
kth element and a sum value of smaller numsmaller and equal values numequal is
larger than a position of kth element, the current array element A[i] is a median.
Write a final median value for the user and return this value to the output. If
not, go to the next ith element (i+1) and define count variable and start
comparing the ith element with the array elements j all over again.

    if ((numsmaller < k) && (k <= (numsmaller + numequal)))
    {
        Console.WriteLine("Final Median value = " + a[i]);

        return a[i];
    }
}
return -1;
}
//print the initial array values for the user
static void printArray(int n, int[] a)
{
    Console.Write("[");

    for (int i = 0; i < n; i++)
    {
        Console.Write(a[i] + " ");
    }
    Console.WriteLine("]");
}
}
}

```

A formal poof of algorithm correctness was performed through the test cases. An attached Excel file “test” (please see Appendix A also) shows that the calculated values of median for an array are equal to the output values from the above script and a program works correctly. Please see Table 1 and Appendix C for the summarised results. Calculated values in Excel were calculated by:

- putting an array in a sorting order,
- finding a total number of elements,
- dividing this number of elements by two,
- taking a ceiling value as an integer- this is a position of a median,
- finding a value of this element from an array.

Test Case	Test Instance	Expected Output	Actual Output	Test result
Positive values of sorted array with an even number of elements	A= { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 }, n = 14	6	6	Pass
Positive values of sorted array with an odd number of elements	A= { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}, n = 15	7	7	Pass
An empty array	A={}, n = 0	No output	No output	Pass
Positive values of unsorted array with an even number of elements	A = { 4, 7, 2, 8, 9, 11, 1, 3, 23, 0, 5, 92, 34, 11, 45, 67, 21, 34, 78, 563}, n = 20	11	11	Pass
Positive values of unsorted array with an odd number of elements	A={ 4, 7, 2, 8, 9, 11, 999999999, 1, 23, 0, 5, 92, 34, 11, 45, 11111, 1000000, 3452, 98765}, n= 19	11	11	Pass
Positive values of reverse sorting array with an even number of elements	A={ 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0}, n= 14	6	6	Pass
Positive values of reverse sorting array with an odd number of elements	A= { 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 }, n= 15	7	7	Pass
Positive duplicate values of array	A= { 4, 3, 4, 1, 1, 3, 10, 5, 5, 7, 45, 6, 45, 22, 34}, n = 15	5	5	Pass
Zero values array	A= { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, n = 10	0	0	Pass
Negative values of array with an even number of elements	A = { -4, -999999999, -2, -8, -9, -11, -1, -3, -23, -2, -9, -8, -7, -34, -111111, -987654, -1000000, -34}, n = 18	-9	-9	Pass
Negative values of array with an odd number of elements;	A = { -4, -7, -999999999, -100000, -2, -8, -9, -11, -1, -3, -23, -2, -9, -23455, -11111, -8, -5, -34, -15}, n = 19	-9	-9	Pass
Mixed values of array with an even number of elements	A = {-7, -2, 999999999, 8, 9, 11, -1, 3, 0, -1111111, -23, 2, -9, 8, 11, -8, -999999999, 1000000}, n = 18	0	0	Pass
Mixed values of array with an odd number of elements	A = {4, -7, -2, 8, 9, 11, 999999999, -1, 3, 0, -23, 2, -9, 8, -999999999, 11, -8}, n = 17	2	2	Pass

Array with all the values of 1 in it	A = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, n = 13	1	1	Pass
Array with one value of array	A = {1}, n = 1	1	1	Pass
Array with the value of first element as median with an even number of elements	A = {6 0 1 2 3 4 5 7 8 9 10 11 12 13}, n = 14	6	6	Pass
Array with the value of last element as median with an even number of elements	A = {0 1 2 3 4 5 7 8 9 10 11 12 13 6 }, n = 14	6	6	Pass
Array with the value of first element as median with an odd number of elements	A = {7 0 1 2 3 4 5 6 8 9 10 11 12 13 14 }, n = 15	7	7	Pass
Array with the value of last element as median with an odd number of elements	A = {0 1 2 3 4 5 6 8 9 10 11 12 13 14 7 }, n = 15	7	7	Pass

Table 1. Summarised test results of BruteForceMedian algorithm calculated using C# and Excel programs.

DESIGN OF EXPERIMENTS

The algorithm and the experiments were implemented in C# programming language. I was using Visual Studio 2017 to code in C# programming language. Visual Studio 2017 is part of Microsoft software package and available for download for free with the Microsoft product user registration. The experiments were performed on a HP Intel Core i7-6700 CPU lab computer of QUT, running Windows 10 operating system. C# random number generator was used to produce test data and C# time module was used to measure execution times. For the experiments involving execution time measurements, the number of other software applications running concurrently with the tests was minimised. Graphs of the experimental results were produced using Microsoft Excel software. The results of computation were pasted from a Visual Studio console to a Microsoft Excel file's columns. By using a graph utility in Excel, the tables and graphs for an algorithm data analysis were generated. This report was written using a Microsoft Word software. For my experiments Test data were selected to check an incongruence between the assertions- what the program supposed to do and what the program actually does. I used:

- positive values of sorted array with an even number of elements;
- positive values of sorted array with an odd number of elements;
- an empty array;
- positive values of unsorted array with an even number of elements;
- positive values of unsorted array with an odd number of elements;
- positive values of reverse sorting array with an even number of elements;
- positive values of reverse sorting array with an odd number of elements;
- positive duplicate values of array;
- zero values array;
- negative values of array with an even number of elements;
- negative values of array with an odd number of elements;
- mixed values of array with an even number of elements;
- mixed values of array with an odd number of elements;
- array with all the values of 1 in it;
- array with one value of array;
- Array with the value of first element as median with an even number of elements;
- Array with the value of last element as median with an even number of elements;
- Array with the value of first element as median with an odd number of elements;
- Array with the value of last element as median with an odd number of elements;

Invalid values of the array are non-numeric values and values which are not an integer type. I used only integer values in my testing as any other values will not be allowed by my code. I checked boundary conditions inside of the valid data. I used 999999999 and -999999999 integer values as a boundary between short and long integer. I used an empty array, 0 and 1 values of the array to observe any anomalies in finding a median of an array.

For counting a number of basic operations performed by the program and for measuring an execution time I produced a set of random values. This way I could produce

average-case results for a particular size of input. I created these test data automatically through the script. I was using Random class to generate a random array. I applied a seed as a current Date and Time. For each element of a particular size array I run a random object to find a random value for each array element. I did not have an upper limit in generating a random value.

```
int[] GenerateRandomArray(int size)
{
    int[] A = new int[size];

    int seed = (int)DateTime.Now.Ticks;
    Random rnd = new Random(seed);

    for (int i = 0; i < A.Length; i++)
    {
        A[i] = rnd.Next(Int32.MaxValue);
    }
    return A;
}
```

EXPERIMENTAL RESULTS

A basic operation: comparison $A[i]$ with each array element $A[j]$. This basic operation appears in two places, " $A[j] < A[i]$ " and " $A[j] = A[i]$ " and it is a key comparison in searching for a median value. These basic operations will be executed the most number of times.

These operations will have the most influence on the algorithm's total running time as each element must be compared with one element of the array between n and n^2 times. n value of comparisons will be when an array will have a median value as its first element and it will be the best case. n^2 number of comparisons will be when a median value of the array will be the last value of the array and it's the worst case. The theoretical average efficiency of this algorithm is $\Theta(n^2)$.

The efficiency of this algorithm is that algorithm will always make at least n key comparisons on every input of size n , where n number may vary between n and 1 elements of array. An order of growth will be $n*n$ elements as each element n can go maximum n times if a median value is last in the array and minimum n times if the first element in a array is a median. It is a quadratic notation.

The algorithm can perform different numbers of basic operations at each iteration so its best-case, worst-case and average-case efficiency may be different.

The best case is when the first element in the array is the median value for the array, because it means that an external for loop runs only once. The algorithm's best-case efficiency is:

$C_{\text{best}}(n) =$

$$\sum_{i=0}^1 \sum_{j=0}^{n-1} 1 = \sum_{i=0}^1 ((n-1) - 0 + 1) = \sum_{i=0}^1 n = n \in \Theta(n)$$

The worst case is when the last array element $A[n-1]$ contains the median value in the array, because it means an external for loop performs $n-1$ times. The algorithm's worst-case efficiency is:

$C_{\text{worst}}(n) =$

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 = \sum_{i=0}^{n-1} ((n-1) - 0 + 1) = \sum_{i=0}^{n-1} n = ((n)(n+1))/2 = n^2 \in \Theta(n^2)$$

The average case efficiency is bounded from above by the worst-case function

$((n)(n+1))/2 \in \Theta(n^2)$, and from below by the best-case function $(n) \in \Theta(n)$. [3]

If to take into account that a probability of $[a_i]$ random element to have a duplicate value in an array is very small, equal to zero, and that an array has a very large number of elements, we can say that an algorithm's average-case efficiency is:

$$\sum_{j=1}^n \frac{1}{2} * \left(\sum_{j=0}^{n-1} (1 + 1/2) \right) = \sum_{i=1}^n 1/2 \left(\frac{3}{2} * n + \frac{1}{2} \right) = \frac{1}{4} n(3n + 1) = n^2 \in \Theta(n^2)$$

The number of basic operations can be calculated through inserting a counter variable into a code for basic operations.

For the count of basic operations, I investigated how many times each basic operation was executed during the execution of the algorithm and added the results together.

I counted basic operations by defining an integer variable **countbase** in the function Compare and assigning its value to 0. Then, I inserted the statements of (++countbase > 0) into conditions **if** (a[j] < a[i]) and **else if** (a[j] == a[i]). I put a WriteLine statement to see a number of time the basic operation was executed together with the WriteLine statement to see a final median value. This is an example of the script.

```
// function to compare each value with the values of the array
static int Compare(int n, int[] a)
{
    //Calculate a median position of element for an array
    int k = Convert.ToInt32(Math.Ceiling((double)n / 2));
    int countbase = 0;

    for (int i = 0; i <= (n - 1); i++)
    {
        int numsmaller = 0;
        int numequal = 0;
        for (int j = 0; j <= (n - 1); j++)
        {
            //count the number of elements smaller to a current values
            if ((++countbase > 0) && (a[j] < a[i]))
            {
                numsmaller++;
            }
            //count the number of elements equal to a current values
            else if ((++countbase > 0) && (a[j] == a[i]))
            {
                numequal++;
            }
        }
        //If the number of small values less than a position of median and
        the values of smaller and equal values are larger than a position of median, the
        current array value is a median.
        if ((numsmaller < k) && (k <= (numsmaller + numequal)))
        {
            Console.WriteLine("Final Median value = " + a[i]);
            Console.WriteLine("Number of times the basic operation was
executed = " + countbase);

            return a[i];
        }
    }
    return -1;
}
```

I implemented count operation on a manual test script to be sure that I am getting a correct results. I created four arrays:

- Array with the value of first element as median with an even number of elements; $n = 14$
- Array with the value of last element as median with an even number of elements; $n = 14$
- Array with the value of first element as median with an odd number of elements; $n = 15$
- Array with the value of last element as median with an odd number of elements; $n = 15$

I checked that a manually calculated number of basic operation matches the results of test cases. Please see Appendix C of an actual output.

Then, I implemented a count operation on a set of random values using a Random class and produced average-case results for an array with the size starting from 1000 to 20000 with the increment every 1000 element. Please see Appendix D for an example of an output. I used the same Compare functionality with the same location of count variable for basic operations, and only difference was that that the arrays were generated randomly.

To accurate measure an execution time of BruteForceMedian algorithm, I used a range of random values, recorded the system 'clock' time at the beginning and at the end of the algorithm, generated 20 different sizes of arrays which were running for 20 times each. I started to run array size of 1000 and increased the value every 1000 elements till 20000 elements of array. Each array was running for 20 times to find the average run time in milliseconds. For calculating an average case I assumed that randomly selected elements are in incorrect order in the array.

For the code, I created a new stopwatch class. For each size of an array I found an average running time of array by using a variable averageMilliSecs. This variable calculates an average milliseconds time for each array by summarising totalMilliSecs time values of each out of 20 runs and divides it on a number of runs (20). By using a for loop, I calculated a total time (totalMilliSecs) of running every array in milliseconds. To do this, I created a variable milliSecs, assigned its value to 0, and run the function of GenerateRandomArray(size) to generate a particular size array with the random values. Then, I start a stopwatch to start recording the time of running the algorithm, run the algorithm itself and then put a stop on a Stopwatch, to stop recording the time. I am getting an elapsed time (milliSecs) through using ElapsedMilliseconds property. I am calculating a total running time (totalMilliSecs) by adding values of milliSecs variable from each run together. Then I divide it on a number of runs (20) to find an average time value in milliseconds (averageMilliSecs). Please see the code below.

```
int totalRunningTimes = 20;
Stopwatch sw = new Stopwatch();
for (int size = 1000; size <= 20000; size += 1000)
{
    long totalMilliSecs = 0;
    double averageMilliSecs = 0;
    for (int i = 1; i <= totalRunningTimes; i++)
    {
        long milliSecs = 0;
        int[] A = GenerateRandomArray(size);
        sw.Start();
```

```

        Compare(A);
        sw.Stop();
        milliSecs = sw.ElapsedMilliseconds;
        totalMilliSecs = totalMilliSecs + milliSecs;
    }
    averageMilliSecs = totalMilliSecs * 1.0 / totalRunningTimes;
    Console.WriteLine("Size: " + size + "; Average Running Time
(MilliSec)= " + averageMilliSecs);
}

```

After receiving data of an average running time for each size of an array, I could proceed to analysing its results. Please see Appendix E for an example of an output.

ANALYSIS OF EXPERIMENTAL RESULTS

By implementing a count variable for a random values algorithms with the different size array, I found a behaviour of the data.

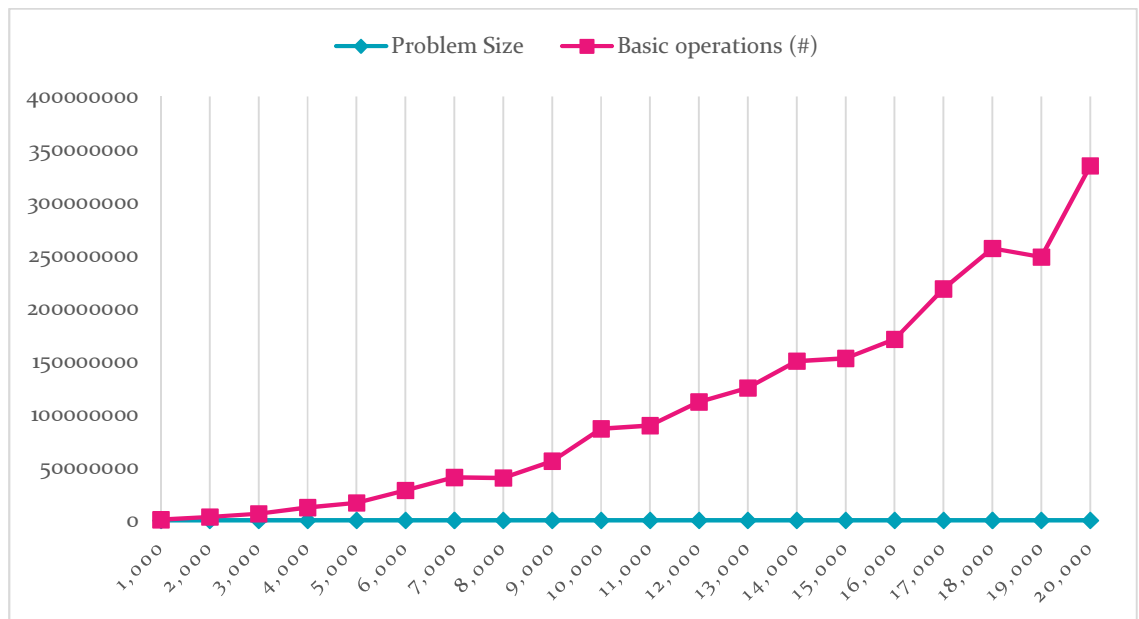
By analysing experiments to count the program's basic operations, I can conclude that the results produce a clear trend which can be compared meaningfully with the algorithm's predicted growth.

By analysing the data obtained from the script output, I formed the hypothesis about algorithm's average case efficiency.

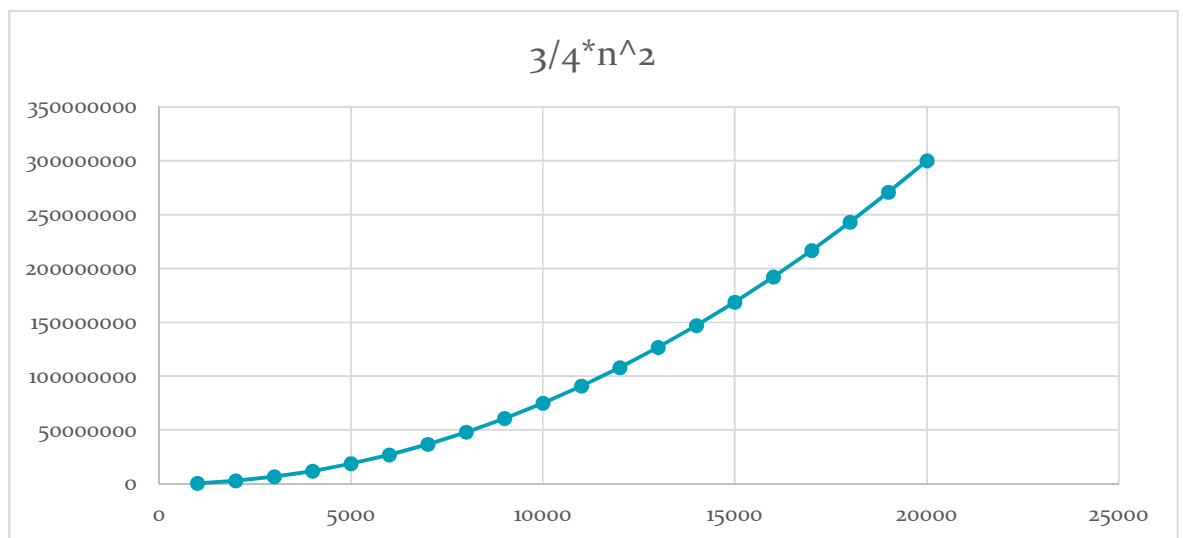
By checking how fast the count values grow with increasing every 1000 elements, I can find that the data exhibits a behaviour indicative of $\frac{3}{4}n^2$. This is an approximate theoretical number of key comparisons BruteForceMedian algorithm. The average efficiency of the algorithm belongs to $\Theta(n^2)$.

Problem Size	Basic operations (#)	$\frac{3}{4}n^2$
1000	538823.1	750000
2000	3197592.9	3000000
3000	6160961.2	6750000
4000	12118591.45	12000000
5000	16375212.4	18750000
6000	28312940.1	27000000
7000	40683871.7	36750000
8000	40157971.55	48000000
9000	56017627.05	60750000
10000	86371572.15	75000000
11000	89563972.35	90750000
12000	111719668.5	108000000
13000	125141841.4	126750000
14000	150330341.4	147000000
15000	152916372.1	168750000
16000	170818791	192000000
17000	218562697.4	216750000
18000	256787267.3	243000000
19000	248481819.5	270750000
20000	334844157.8	300000000

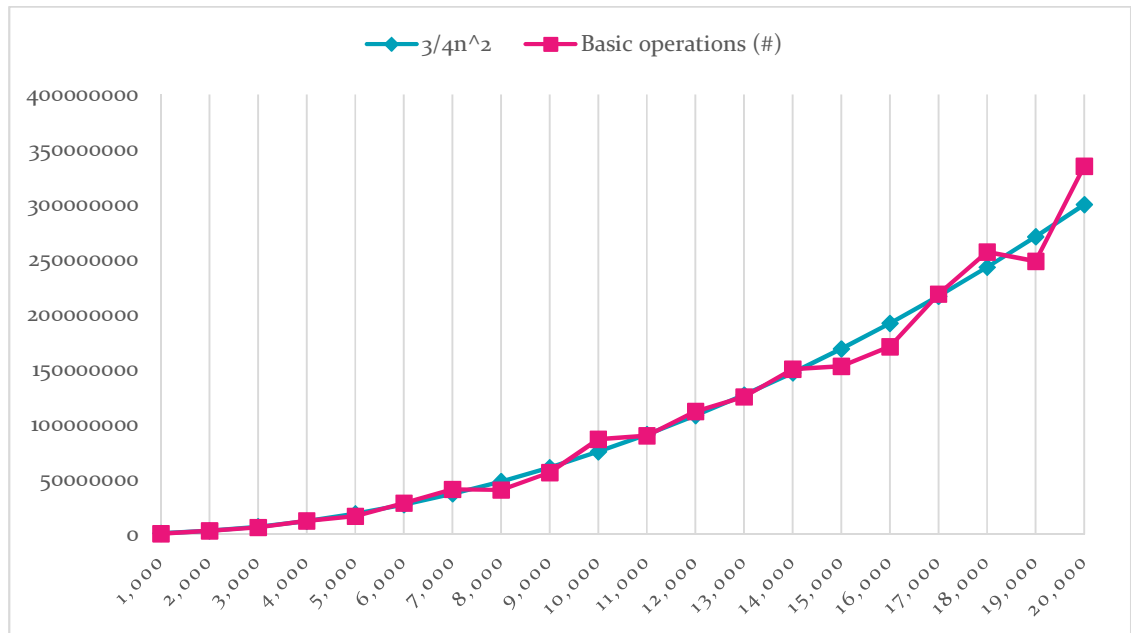
Table 2. A trend between a size of an array, actual number of basic operations and theoretical number of basic operations.



Graph 1. Actual output results of count basic operations of BruteForceMedial algorithm. X axis- size of array, Y-axis- number of basic operations for each array size



Graph 2. Theoretical output of count basic operations by the function $\frac{3}{4} * n^2$. X axis- size of array, Y-axis- number of basic operations for each array size



Graph 3. Combined graphs of actual output number and theoretical output number of basic operation for BruceForceAlgorithm. X axis- size of array, Y-axis- number of basic operations for each array size.

By comparing the results of actual output from the script and theoretical prediction, I can conclude that:

- the data from both sets are very close to each other and actual output follows the $\frac{3}{4}n^2$ parabola,
- this algorithm works efficiently on small inputs,
- the count of basic operations increases with the increasing number of array elements,
- the increase in count of basic operations is in exponential rate of growth as the array length increases,
- the count of basic operations depends n^2 on the size of the input,
- there is a little discrepancy for an actual count of basic operations arrays with the larger size array. The deviation from $\frac{3}{4}n^2$ parabola goes on both sides of the curve.

When input gets large enough, the lower-order terms get insignificant compare to the leading term. When algorithm inputs are small, algorithms is efficient. The final result has any lower-order terms removed and represented as an efficiency of the algorithm $f \in \Theta(n^2)$.

By analysing experiments to measure the program's execution times, I can conclude that the results produce a clear trend which can be compared meaningfully with the algorithm's predicted growth.

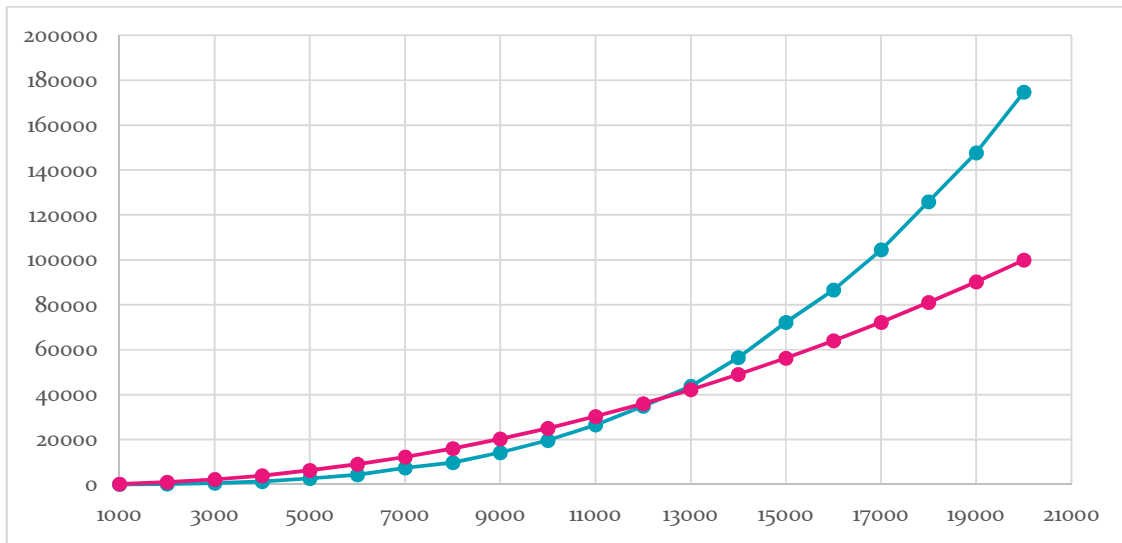
After receiving data of the average running time for each size of an array, I plotted the data on a graph and found that the data exhibits a behaviour indicative of an $n^2/1000 \cdot 1/4$.

This is an approximate theoretical number of key comparisons BruteForceMedian algorithm. The average efficiency of the algorithm belongs to $\Theta(n^2)$.

Problem size	Execution time (msec)	$n^2/1000 \cdot 1/4$
1000	49.8	250
2000	203.45	1000
3000	575.35	2250
4000	1265.15	4000
5000	2751.9	6250
6000	4342.9	9000
7000	7346.75	12250
8000	9825.3	16000
9000	14209.75	20250
10000	19582.45	25000
11000	26494.2	30250
12000	34922.15	36000
13000	43745.9	42250
14000	56455	49000
15000	72148.55	56250
16000	86584.2	64000
17000	104535.75	72250
18000	125873.1	81000
19000	147662.3	90250
20000	174712.7	100000

Table 3. A trend between a size of an array, execution time of an algorithm and a theoretical execution time of the algorithm.

Blue line indicates an actual execution time of each different size array and red line indicates a theoretical behaviour of execution time BruteForceMedian algorithm.



Graph 4. Combined graphs of actual output of an execution time for a BruceForceMedian Algorithm and theoretical output number of an execution time for BruceForceMedian Algorithm. X axis- size of array, Y-axis- execution time in milliseconds for each array size.

It is important to know the exact time it takes to perform a certain operation (in real time systems). It is enough to know how the running time of an algorithm changes as the input gets larger. For this BruteForceMedian algorithm, in our case, we increased an array every 1,000 elements for 20 times. The analysis and a behaviour of the algorithm is not tied to programming but can be used to describe the efficiency of any behaviour that consists of successive operations. The time efficiency analysis is simplified by assuming that all operations that are independent of the size of the input take the same amount of time to execute. The amount of times an algorithm was running is irrelevant as long as the amount is constant. In our case its 20 runs. [1]

By comparing experimental results against the theoretical efficiency prediction I can conclude that:

- this algorithms works efficiently on small inputs,
- the running time increases with the increasing number of array elements,
- the increase in execution time is in exponential rate of growth as the array length increases,
- the execution time depends n^2 on the size of the input,
- the actual execution time follows $n^2/1000*1/4$ parabola until the size of array reaches 1300 elements,
- there is a discrepancy in execution time for array with the sizes starting from 1300 elements. This discrepancy is exponentially getting increased away from $n^2/1000*1/4$ parabola. This behaviour can be explained due to operating system of computer, the number of simultaneous running processes on the computer invisible to a user. When I was running an execution time measurements, the

number of other software applications running concurrently with the tests was minimised but not eliminated. These software applications take a part of RAM and slow down computations and computer responses. This behaviour is more visible with the large size arrays as it takes longer to process data for these arrays. By adding more RAM, a CPU usage and throughput would be higher.

REFERENCES

1. Efficiency and algorithm design, lecture 3, accessed 05 April 2019, <http://www.cs.tut.fi/~tie20106/material/lecture3.pdf>
2. CSC 344 – Algorithms and Complexity, accessed 05 April 2019, <https://home.adelphi.edu/~siegfried/cs344/344l3.pdf>
3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

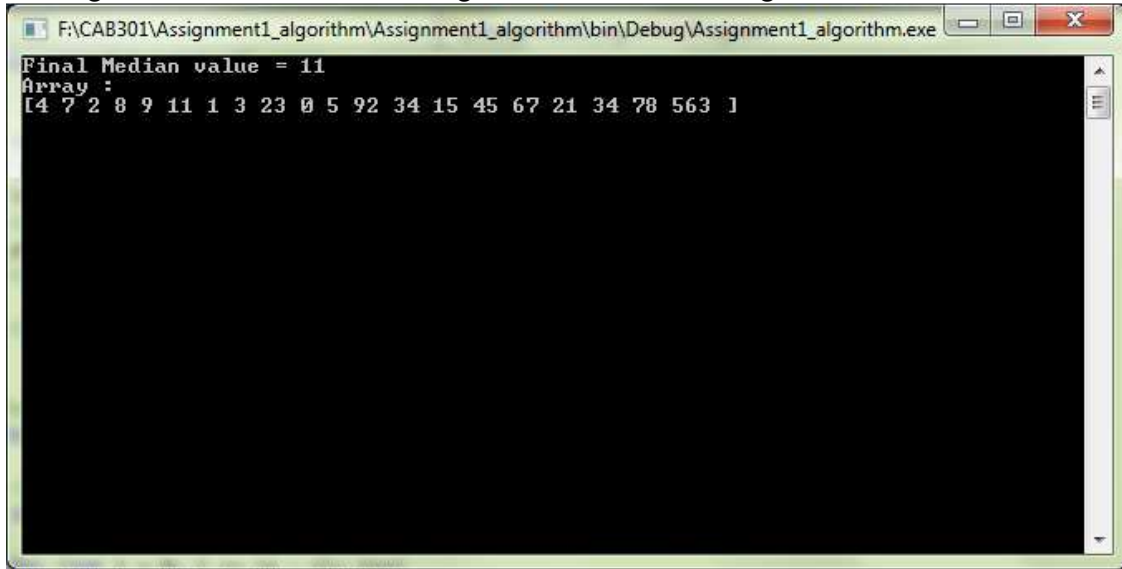
Appendix A

information

Total list of arrays										Order	Each individual array is in a sorted order																		
1										0	0		0	0	0	0	0	1	0	-999999999	-999999999	-999999999	-999999999	1	1	0	0	0	0
2										1	1		1	1	1	1	1	0	-1000000	-111111	-111111	-111111	-23	1	1	1	1	1	
3										2	2		2	2	2	2	3	0	-987654	-100000	-23	-9	1	2	2	2	2		
4										3	3		3	4	3	3	3	0	-111111	-23455	-9	-8	1	3	3	3	3		
5										4	4		4	5	4	4	4	0	-34	-34	-8	-7	1	4	4	4	4		
6										5	5		5	7	5	5	4	0	-34	-23	-7	-2	1	5	5	5	5		
7										6	6		7	8	6	6	5	0	-23	-15	-2	-1	1	6	6	6	6		
8										7	7		8	9	7	7	5	0	-11	-11	-1	0	1	7	7	7	7		
9										8	8		9	11	8	8	6	0	-9	-9	0	2	1	8	8	8	8		
10										9	9		11	11	9	9	7	0	-8	-9	2	3	1	9	9	9	9		
11										10	10		11	23	10	10	10	10	-8	-8	3	4	1	10	10	10	10		
12										11	11		21	34	11	11	22	-7	-8	8	8	1	11	11	11	11			
13										12	12		23	45	12	12	34	-5	-7	8	8	1	12	12	12	12			
14										13	13		34	92	13	13	45	-4	-5	9	9		13	13	13	13			
15										14	14		34	3452	14	14	45	-3	-4	11	11		14	14	14	14			
16													45	11111				-2	-3	11	11								
17													67	98765				-2	-2	1000000	999999999								
18													78	1000000				-1	-2	999999999									
19													92	999999999					-1										
20													563																

Appendix B

Finding an actual median value using BruceForceMedian Algorithm



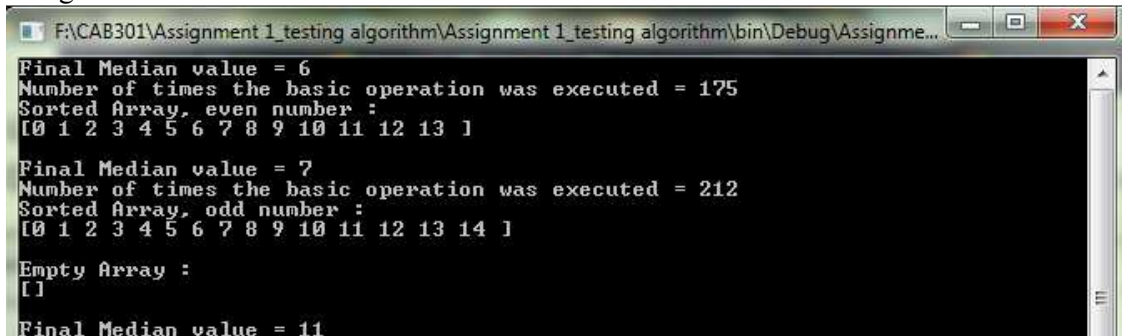
The screenshot shows a Windows command prompt window with the following text:

```
F:\CAB301\Assignment1_algorithm\Assignment1_algorithm\bin\Debug\Assignment1_algorithm.exe  
Final Median value = 11  
Array :  
[4 7 2 8 9 11 1 3 23 0 5 92 34 15 45 67 21 34 78 563 1]
```

The window title bar indicates the file path: F:\CAB301\Assignment1_algorithm\Assignment1_algorithm\bin\Debug\Assignment1_algorithm.exe. The output shows the final median value is 11 and the array contains 20 elements.

Appendix C

Finding an actual median value and a number of times the basic operation was executed using test data



```
F:\CAB301\Assignment 1_testing algorithm\Assignment 1_testing algorithm\bin\Debug\Assignme...
Final Median value = 6
Number of times the basic operation was executed = 175
Sorted Array, even number :
[0 1 2 3 4 5 6 7 8 9 10 11 12 13 ]

Final Median value = 7
Number of times the basic operation was executed = 212
Sorted Array, odd number :
[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 ]

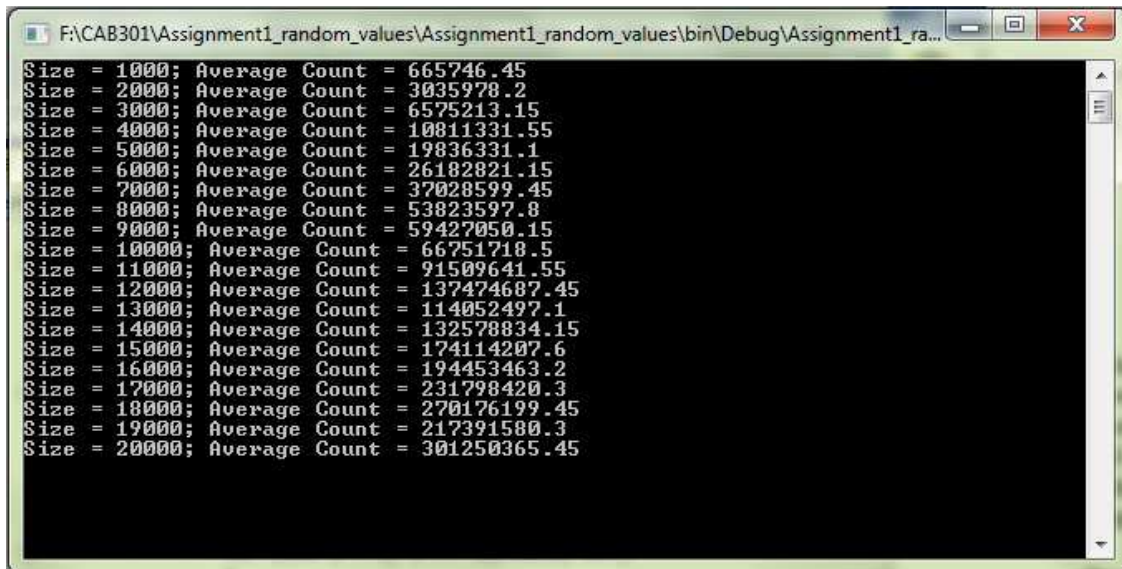
Empty Array :
[]

Final Median value = 11
```

Continue on next page

Appendix D

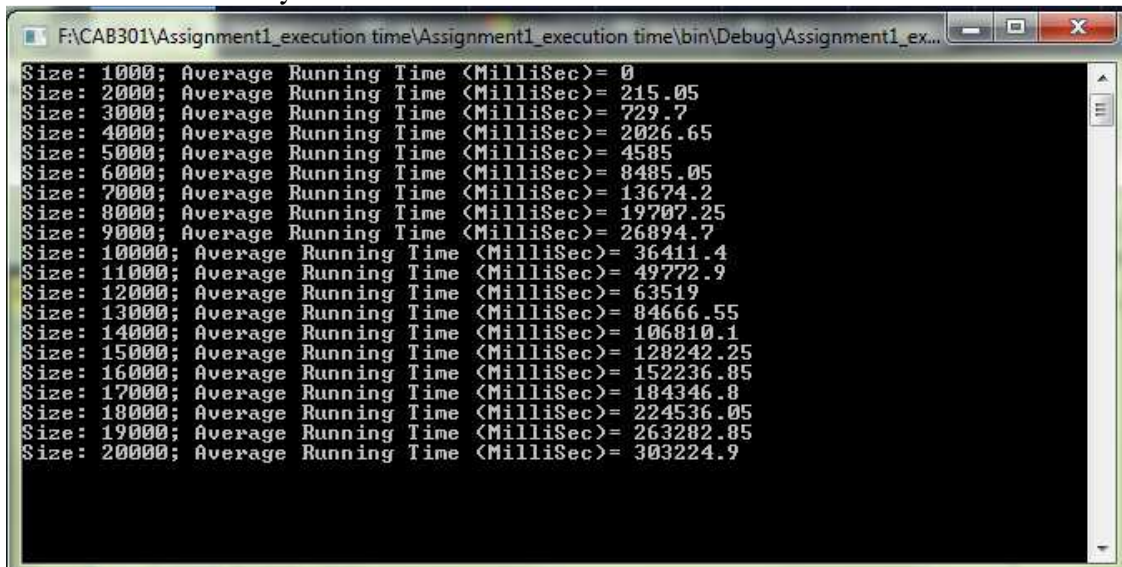
Finding a number of basic operations for different size arrays using random values of array elements



```
F:\CAB301\Assignment1_random_values\Assignment1_random_values\bin\Debug\Assignment1_ra...
Size = 1000; Average Count = 665746.45
Size = 2000; Average Count = 3035978.2
Size = 3000; Average Count = 6575213.15
Size = 4000; Average Count = 10811331.55
Size = 5000; Average Count = 19836331.1
Size = 6000; Average Count = 26182821.15
Size = 7000; Average Count = 37028599.45
Size = 8000; Average Count = 53823597.8
Size = 9000; Average Count = 59427050.15
Size = 10000; Average Count = 66751718.5
Size = 11000; Average Count = 91509641.55
Size = 12000; Average Count = 137474687.45
Size = 13000; Average Count = 114052497.1
Size = 14000; Average Count = 132578834.15
Size = 15000; Average Count = 174114207.6
Size = 16000; Average Count = 194453463.2
Size = 17000; Average Count = 231798420.3
Size = 18000; Average Count = 270176199.45
Size = 19000; Average Count = 217391580.3
Size = 20000; Average Count = 301250365.45
```

Appendix E

Finding an execution time of BruteForceMedian algorithm for different size arrays using random values of array elements



```
F:\CAB301\Assignment1_execution time\Assignment1_execution time\bin\Debug\Assignment1_ex...
Size: 1000; Average Running Time (MilliSec)= 0
Size: 2000; Average Running Time (MilliSec)= 215.05
Size: 3000; Average Running Time (MilliSec)= 729.7
Size: 4000; Average Running Time (MilliSec)= 2026.65
Size: 5000; Average Running Time (MilliSec)= 4585
Size: 6000; Average Running Time (MilliSec)= 8485.05
Size: 7000; Average Running Time (MilliSec)= 13674.2
Size: 8000; Average Running Time (MilliSec)= 19707.25
Size: 9000; Average Running Time (MilliSec)= 26894.7
Size: 10000; Average Running Time (MilliSec)= 36411.4
Size: 11000; Average Running Time (MilliSec)= 49772.9
Size: 12000; Average Running Time (MilliSec)= 63519
Size: 13000; Average Running Time (MilliSec)= 84666.55
Size: 14000; Average Running Time (MilliSec)= 106810.1
Size: 15000; Average Running Time (MilliSec)= 128242.25
Size: 16000; Average Running Time (MilliSec)= 152236.85
Size: 17000; Average Running Time (MilliSec)= 184346.8
Size: 18000; Average Running Time (MilliSec)= 224536.05
Size: 19000; Average Running Time (MilliSec)= 263282.85
Size: 20000; Average Running Time (MilliSec)= 303224.9
```