CAB301 Assignment 2

Empirical Comparison of Two Algorithms

Student name: John Santias, Shivam Sachdeva
Student no. n9983244, n9642587
Due Date: 21st May 2018

Summary

This report compares two different algorithms that both have the same function. The Median and the Brute Force Median algorithm both work to find the Median value inside a given array, however, both may have different efficiencies during the process. In this report, we analyze each algorithm by measuring the basic operations and execution time then compare the results to determine which algorithm is more efficient. Both programs are analyzed exactly in the same conditions to ensure that the results are comparable.

1 Description of the Algorithm

1.1 Brute Force Median Algorithm

The brute force median algorithm returns the median value in a given array. In a sorted list the median would always be in the middle, however, by having an unordered list it can be harder to find the value. "Brute force" means the algorithm will try to find many possible ways to find the median value of a given array [2, Section 2]. It is also known as an exhaustive search [2, Section 2]. This algorithm's pseudocode can be found in Appendix A. It starts off by getting the size of the array and divides it by two, this value is assigned to a variable named '*k*'. This is indicating the index of the array where the median will be. When finding the median inside an uneven array size the program will choose the value in the middle. With an unevenly sized array, it will choose the left of the two middle elements ($\text{size of the array} \div 2$). A for-loop is then executed to work its way through the array from left to right. Each array value is used as a pivot. This loop firstly assigns the two variables '*numsmaller*' and '*numequal*' with a value of zero. Afterward, another for-loop is executed to use each element in the array for comparison with the pivot. Inside this inner loop, there are two comparisons. The first comparison determines if the current element value is less than the pivot's value. If it is, the variable '*numsmaller*' increments by one. The second comparison determines if the values are equal. If they are, then the variable '*numequal*' increments by one. Once all the elements are compared with the pivot, the algorithm breaks out of the for loop and runs a final comparison which determines if the chosen pivot is the correct Median. This is done by comparing the value of '*numsmaller*' if it is less than the k and the value of '*numequal*' is greater than or equal to k. If this comparison is true, then the algorithm will return the pivot's value as the median. Otherwise, the algorithm will repeat the whole process until the final comparison is correct.

1.2 Median Algorithm

The Median algorithm is used to find the median inside an array. There are three separate procedures in this algorithm. This algorithm is a special case of Johnsonbaugh and Schaefer's version of the selection problem algorithm. The pseudocode of the algorithm is shown in Appendix B. The main procedure called median takes the array as a parameter. If the array contains only one element, the algorithm returns that element as the median. If the array contains more than one element, the select procedure is called. The select procedure takes four parameters: an array, the first index value of the array, the index of the median element, and the index of the last element of the array. This procedure then calls the partition procedure which takes the first element of the array as a pivot and compares it with all the other values of the array. Every time the pivot value is greater than the other element, the '*pivotloc'* variable increases by one and swaps the element around the pivot. After the for-loop is executed, the pivot is placed in a position where it should be if the array was sorted. This procedure returns the '*pivotloc'* variable. The value returned from partition procedure is stored in the '*pos'* variable of Select procedure and compared with the index of the median element which is equal to $n/2$ where n is the number of elements in the array. If the value matches, the select procedure returns the median of the array. If the value does not match, the select procedure is called again but now with the sub-array. The partition procedure is called to do partitioning on that sub-array. The returned value of partition procedure is compared again, and this process goes on until the value matches the index of the median element. To simplify, the select algorithm is a recursive method, which is called many times until the median is found. Every recursive call involves an array slice whose size is reduced every time.

Both algorithms have different behaviors when finding the median of an even array size. The Brute Force algorithm returns the left of the two middle values while the Median algorithm returns the right value.

2 Theoretical Analysis of the Algorithm

2.1 Choice of Basic Operation

The basic operation of Brute force median algorithm is the if and else-if conditions within the nested for-loops. Every array element is compared to itself and to the other elements. The block of if and else if condition checks whether the chosen element is greater than or equal to the other array elements. If the chosen array element is greater than the other elements, the '*numsmaller'* variable increments whereas if both elements are equal, the '*numequal'* variable increments. This block of if and else if condition is considered as the one basic operation which determines the efficiency of the brute force median algorithm. The basic operation of the Median algorithm is the if condition within the for-loop of the partition procedure which checks whether the pivot is greater than the other elements. Basically, how many times pivot is checked against other elements or in other words how many times for-loop executes for an array determines the efficiency of the Median algorithm. The partition procedure can be called recursively many times until the median is found. The basic operation chosen for both the algorithms performs the same function which is comparing the element with the other array elements.

2.2 Choice of Problem Size

The problem size given to our algorithms ranges from one to a hundred, incrementing the problem size by one. Each element value in the Array is generated randomly between zero and ten thousand. We are generating random arrays to conduct the experiments for finding the average-case efficiency. As we are comparing two algorithms, we will be giving them the same array for each test.

2.3 Average-case efficiency

Each algorithm has a different behavior to find the median of a given array. For each array that is passed on to the algorithms, we want to analyze which algorithm is more efficient. In this report, we are only analyzing the average-case efficiency, not the best- and worst-case efficiency. In analyzing the average-case efficiency, we will need to pass random generated arrays in random order to our algorithms.

2.3.1 Average-case efficiency for the Brute Force Median algorithm

For analyzing the Brute Force algorithm, the for-loop iterates from $i = 0$ to $i = n - 1$ and the inner for-loop iterates from $j = 0$ to $j = n - 1$. Recalling from section 2.1, we chose the if and else comparisons as our basic operation. As these comparisons are executed inside a for-loop, we use the summation formulae described by Levitin [Levitin, page 62-63], and [5, Summation Formulas], to compute the average case of the Brute Force Median's basic operation.

Mathematically,

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1 = \sum_{i=0}^{n-1}(n-1-0+1) \qquad using\ (\sum_{j=l}^{u} 1 = u - l + 1)$$

$$= \sum_{i=0}^{n-1} n \qquad using\ (\sum_{i=0}^{n-1}(u - i = \frac{u(l+1)}{2})$$

$$= \frac{\mathrm{n}(\mathrm{n}+1)}{2}$$

$$= \frac{n^2 + n}{2}\ \epsilon\ \theta(n^2)$$

The efficiency will have a n^2 trend because of the two nested for-loops. We used an excel spreadsheet and the formula to calculate the number of comparisons for each problem size, we expect our tests results to match to the graph in Appendix H.

2.3.2 Average-case efficiency for the Median algorithm

Recalling from section 2.1, we chose the same basic operation for both algorithms. For the median algorithm, we chose the if-condition within the for-loop of the partition procedure. It iterates from $\mathrm{l} + 1$ to h according to the pseudocode in Appendix B. When the partition procedure is called for the first time, the if-condition is checked $\mathrm{n} - 1$ times [1, page 161]. After executing $\mathrm{n} - 1$ times, it partitions the array into sub-arrays and it can be called recursively until the median is found. The worst-case can have iterations [1, page 161]:

$$C_{worst}(n) = (n-1) + (n-2) + \ldots + 1 = (n-1)n/2 \in (n^2)$$

If the partitions are unbalanced, then we will get the worst-case efficiency. The average-case efficiency of the Median is found to be linear [1, page 161]. It will be linear if good partitions occur, which will always depend on the given array. For example, if we provide the array {1, 2, 7, 4, 5, 3, 6} it will produce the worst-case efficiency because the number basic operations using the Median algorithm will be 21 matching the result using worst-case efficiency formula $((7-1)7)/2 = 21$. The worst-case occurs when the median is in the middle. Whereas if we provide the same array but in a different order {4, 1, 2, 3, 5, 6, 7}, this will produce the best-case efficiency which is equal to six. This is matching the best-case efficiency formula [1, page 161]:

$$C_{best}(n) = (n-1) \in (n)$$

The best-case occurs when the first element is the median. The average-case iterations can go from $(n-1)+(n-3)$ or $(n-1)+(n-2)$) all depending on the given array. The efficiency will come out to be c×n where c is constant. For example, if we have the array {2, 1, 4, 3} the efficiency comes out to be 4 in terms of basic operations because in this case the for-loop is executed $(4-1)+(4-3)$times equaling 4. If we have an array {1, 3, 2, 4} the efficiency is 5, because the for-loop will execute $(4-1)+(4-2)$ times which equals to 5. Berman and Paul explain that the average efficiency of the Select procedure assumes its inputs are all permutations of $l, \ldots, n$ and each permutation are equally likely. The average-case efficiency for this procedure shows [3, page 249]:

$$A(n,t) \leq 4n, \quad for\ all\ n\ and\ all\ t \in \{0, \ldots, n-1\}$$

Where A is average-case, n is the array size, and t is an integer where we can always assume it is zero $0 \leq t \leq n-1$ [3, page 249]. Using this formula, we expect that for each array we give to the Median algorithm it will produce an average-case efficiency of less than or equal to $4n$. Thus, producing a linear line as stated by Levitin [1, page 161] as well as Johnsonbaugh and Schaefer [4, page 262]. We expect our results to be similar to the graph in Appendix H.

3 Methodology, Tools, and Techniques

3.1 Programming Environment

1. We decided to implement the algorithms and perform the experiments in the C# programming language. As it has been used before, we found it easy to run experiments and retrieve results.
2. The experiments were performed on a Windows 10 PC. It had an Intel i7 Core processor running at 3.20GHz, 16GB of RAM and 64-bit operating system. We used the software called visual studio community to run our C# experiments and ensured other programs and windows were minimized to prevent interruptions. We used C#'s random number class to create random values inside our array and its stopwatch class to record the program's execution time.
3. A Microsoft Excel spreadsheet was used for recording our results and producing graphs. This software helped organise our experiment results in separate columns and rows. Using the results, we generated a graph using Excel's graph function.

3.2 Implementation of the Algorithms

Algorithms were implemented in C# following the pseudocodes in Appendix A and B. The algorithms ran on a single file to find the median value of the given arrays. As mentioned in section 2.2, we want to be using the same array for both algorithms in each test so that we can compare it correctly.

Whilst implementing the Brute Force algorithm, we found that making the variable '*k*' an integer would produce the wrong result. This is due to the algorithm being given an uneven array, it returns the left of the actual median value. For example, an array {1, 2, 3, 4, 5} would assign the value '2' to the variable '*k*' because $Array\ size\ \div 2$ (or $5 \div 2 = 2$) is incorrect. We expect the value to be '3' in this case. As have found that using an integer to find the median would produce the wrong result, we defined '*k*' as a double to use decimal numbers and the '*Math.Ceiling*' method to round up the value to 3.

$$5 \div 2 = 2.5$$
$$Math.Ceiling(2.5) = 3$$

The method '*swap*' (see Figure 1 of Appendix E) was implemented in our program to swap two elements in an array. This method is used in the Partition procedure of the Median algorithm to sort the elements. The algorithms implemented in C# can be seen in Appendix C and D.

3.3 Generating Test Data and Running the Experiments

To test the correctness of the implementation of our median algorithms, a function called '*GenerateRandomArray*' (see Figure 2 Appendix E) was included. It takes an array size as a parameter and generates an array of that size with random unique values ranging from zero to ten-thousand. The produced array was then passed on to the two algorithms to find the median.

3.4 Functional Testing

To test the correctness of the program, a test method shown in Figure 1 of Appendix F was used. This test method ran three tests for each array size ranging from one to ten. Each test generates a new unique random array and for that array, the median is calculated by using both algorithms. The results obtained from this test is shown in Figure 1 of Appendix G.

As expected, we wanted each test to produce a new array with unique elements and have both algorithms obtaining the correct median:

```
Array size of one:
Test 1: [2718, ] Median: 2718 Brute Force Median: 2718
Test 2: [2874, ] Median: 2871 Brute Force Median: 2874
Test 3: [6475, ] Median: 6475 Brute Force Median: 6475


Array size of two:
Test 1: [342, 4884, ] Median: 4884 Brute Force Median: 342
Test 2: [430, 5859, ] Median: 5859 Brute Force Median: 430
Test 3: [2036, 6105, ] Median: 6105 Brute Force Median: 2036


Array size of three:
Test 1: [933, 1152, 2408, ] Median: 1152 Brute Force Median: 1152
Test 3: [5024, 5581, 7174, ] Median: 5581 Brute Force Median: 5581
Test 2: [374, 4506, 9484, ] Median: 4506 Brute Force Median: 4506
```

Keeping in mind, from Section 1.1, an even array would have two median numbers, the Median algorithm would select the number on the right whereas the Brute Force median algorithm would select the left number.

We wanted to see if the algorithms can handle bigger sized array with unique values. This time we increased the size of the array by thirty-five until the size was bigger than one-thousand (see Figure 2 of Appendix F). The results came out to be:

Array size of 36:
array: [5459, 2003, 4531, 4810, 205, 665, 1756, 4603, 1779, 5325, 5113, 5047, 302, 2097, 5080, 2925, 550, 1109, 5628, 6957, 7329, 5902, 7104, 7229, 6720, 7273, 6033, 8240, 9731, 9009, 8260, 9623, 9129, 8931, 8297, 9861,]

Median: 5628 **brute Force Median**: 5459

Sorted: [205, 302, 550, 665, 1109, 1756, 1779, 2003, 2097, 2925, 4531, 4603, 4810, 5047, 5080, 5113, 5325, 5459, 5628, 5902, 6033, 6720, 6957, 7104, 7229, 7273, 7329, 8240, 8260, 8297, 8931, 9009, 9129, 9623, 9731, 9861,]

Array size of 71:
array: [437, 1464, 294, 1048, 978, 636, 383, 1329, 161, 632, 1013, 1367, 1146, 1618, 2735, 1702, 2142, 3108, 2975, 3392, 1870, 2825, 2944, 3427, 3701, 4091, 4650, 3829, 4031, 3802, 4847, 3566, 4008, 3485, 4316, 4922, 5563, 5737, 6084, 5571, 5455, 6031, 6306, 5108, 6216, 6651, 5977, 6055, 6838, 7307, 6967, 7108, 7424, 9680, 8458, 9130, 9571, 8380, 7901, 8275, 8489, 9911, 9310, 9178, 8687, 8153, 8629, 7817, 9250, 9329, 8616,]

Median: 4922 **brute Force Median:** 4922

Sorted: [161, 294, 383, 437, 632, 636, 978, 1013, 1048, 1146, 1329, 1367, 1464, 1618, 1702, 1870, 2142, 2735, 2825, 2944, 2975, 3108, 3392, 3427, 3485, 3566, 3701, 3802, 3829, 4008, 4031, 4091, 4316, 4650, 4847, 4922, 5108, 5455, 5563, 5571, 5737, 5977, 6031, 6055, 6084, 6216, 6306, 6651, 6838, 6967, 7108, 7307, 7424, 7817, 7901, 8153, 8275, 8380, 8458, 8489, 8616, 8629, 8687, 9130, 9178, 9250, 9310, 9329, 9571, 9680, 9911,]

Array size of 106:
array: [85, 107, 4289, 618, 1977, 1962, 4299, 502, 3193, 3584, 2202, 4309, 2143, 1418, 3707, 3348, 2144, 3459, 392, 1246, 2615, 3123, 2861, 3532, 2579, 2146, 516, 1578, 2074, 950, 3915, 2104, 469, 401, 3062, 3806, 2978, 3297, 2080, 2095, 4198, 4007, 523, 583, 4737, 2025, 4661, 4760, 1018, 240, 4781, 4957, 4853, 5130, 5563, 5594, 5611, 5677, 5765, 6233, 5616, 6177, 6110, 6316, 5798, 6318, 6844, 7268, 6625, 7071, 6586, 7004, 7844, 7999, 7821, 7374, 7389, 6348, 7824, 7660, 7572, 6421, 6858, 6526, 7468, 6861, 6938, 8133, 8611, 9493, 8816, 9718, 9869, 9150, 9899, 8269, 8886, 9691, 9172, 9521, 9895, 9097, 8676, 9287, 9337, 9418,]

Median: 5130 **brute Force Median**: 4957

Sorted: [85, 107, 240, 392, 401, 469, 502, 516, 523, 583, 618, 950, 1018, 1246, 1418, 1578, 1962, 1977, 2025, 2074, 2080, 2095, 2104, 2143, 2144, 2146, 2202, 2579, 2615, 2861, 2978, 3062, 3123, 3193, 3297, 3348, 3459, 3532, 3584, 3707, 3806, 3915, 4007, 4198, 4289, 4299, 4309, 4661, 4737, 4760, 4781, 4853, 4957, 5130, 5563, 5594, 5611, 5616, 5677, 5765, 5798, 6110, 6177, 6233, 6316, 6318, 6348, 6421, 6526, 6586, 6625, 6844, 6858, 6861, 6938, 7004, 7071, 7268, 7374, 7389, 7468, 7572, 7660, 7821, 7824, 7844, 7999, 8133, 8269, 8611,

```
8676, 8816, 8886, 9097, 9150, 9172, 9287, 9337, 9418, 9493, 9521, 9691, 9718, 9869, 9895,
9899, ]
```

We made the program sort the arrays to make it easier for us to find the median. With these tests results, we have confirmed both algorithms did not produce any errors and are functionally correct. The results obtained from this test is shown in Figure 2 of Appendix G.

4 Calculating the results

4.1 Average-efficiency case comparison of both algorithm in terms of number of basic operation

To calculate the basic operations, we inserted two global variable counters called '*counterForMedian'* and '*counterForBrute'* (see Appendix I). This was to store the number of basic operations for each test in which we can then calculate the average out thirty tests in our '*Main'* function (See Appendix I).

4.1.1 Basic operation counter for Brute Force Median Algorithm

To calculate the basic operation of the Brute Force Median algorithm, we inserted a variable called '*counter'* (See Appendix I). Starting from zero, we continuously incremented the variable by one in each iteration of the inner for-loop. The '*counter'* variable was then added to the global variable '*counterForBrute'* (See Appendix I).

4.1.2 Basic operation counter for Median Algorithm

We inserted a variable called '*count'* in the Median algorithm (See Appendix I) to calculate the basic operation. Referring to section 2.1, we chose the if statement within the for-loop of the partition procedure as the basic operation. For each iteration of the for-loop, we incremented the counter by one. After the for-loop, we added the value of the '*count'* variable to the global variable '*counterForMedian'.*

4.1.3 Calculating the counter's average

For an array size we executed thirty tests and in each test, we produced a random array of its size. For each test, we calculated the number of basic operations performed by both algorithms, which we stored in our global variables named '*counterForMedian'* and '*counterForBrute'* (See Appendix I). These values were further added to the variables *'averageOneForMedian'* and *'averageOneForBrute'* (See Appendix I). After executing all thirty tests, we calculated the average by dividing the *'averageOneForMedian'* and *'averageOneForBrute'* variables with the number of tests run which was thirty in this case. The final average basic operation was then stored in the variables *'averageTwoForMedian'* and *'averageTwoForBrute'* (See Appendix I) to be displayed on the console.

4.2 Average-efficiency case comparison of both algorithms in terms of execution time

To calculate the execution time, we inserted two global variables called '*medianTimer'* and '*bruteTimer'* to retrieve the execution time for both algorithms. The '*sw'* (See appendix J) is a stopwatch of a global scope.

4.2.1 Execution timer for Brute Force Median Algorithm

To calculate the execution time of the Brute Force Median algorithm, a variable named '*timer*' started a new stopwatch before the for-loop. This timer stopped when the median was found. The execution time was then stored in the global variable called '*bruteTimer*'.

4.2.2 Execution timer for Median Algorithm

To calculate the execution time of the Median algorithm, we started the stopwatch in the partition procedure before the for-loop (See appendix J) and stopped the timer in the Select procedure when the Median was found. The execution time was then stored in the global variable called '*medianTimer*'. Afterward, the stopwatch was reset back to zero for another test.

4.2.3 Calculating the average execution time

Similar to section 4.1.3, for an array size we executed thirty tests and with each test we produced a random array for its size. In each test, we calculated the execution time for both algorithms which were then stored in our global variables, '*medianTimer*' and '*bruteTimer*'. The value of the '*medianTimer*' was added to the '*averageMedianTimer*' and the '*bruteTimer*' was added to the '*averageBruteTimer*'. Just after when all the tests were executed, the '*averageMedianTimer*' and the '*averageBruteTimer*' was divided by the number of tests run (thirty in this case) (see Appendix J) which gave the average execution time for both algorithms.

5 Experimental Results

5.1 Average-Case Number of Basic Operations

5.1.1 Average-case Basic Operations for the Brute Force Median algorithm

The experimental results did match our predictions in Section 2.3.1. As predicted, the Brute Force Median line in the graph of Appendix K shows a quadratic line increase as we increase the array size. This brute force line almost matched the graph in Appendix H. As the line quadratically increased, it was visible that the basic operations tend to drop and increase. This fluctuation was because we were giving a random array with random values in random order each test.

5.1.2 Average-case Basic Operations for the Median algorithm

As mentioned in Section 2.3.2, we expected the results would produce a linear line and the efficiency would be equal or less than 4n. The results did match our predictions in section 2.3.2, producing a linear line. The results in a graph can be seen in Appendix K. Here you can see the predicted linear line in the graph of Append H almost matched our results. The results did not exactly match with the prediction as we were using random generated arrays in each test.

5.2 Average-Case Execution Time

5.2.1 Average-case Execution Time for the Brute Force Median algorithm

The experimental results also matched our predictions in Section 2.3.1. The brute force median line in the graph of Appendix L shows a quadratic increase as we increased the array size. Similarly, in section

5.1.1, there was a fluctuation because we were generating random arrays with random values in random order for each test. The reasons for this can be the computer's processes working in the background affecting the performance and using portions of the RAM and CPU. Even if we had all other programs minimized, there would still be internal processes running such as the clock, Ethernet etc.

5.2.2 Average-case Execution Time for the Median algorithm

The results did match the predictions stated in section 2.3.2. The median line in the graph of Appendix L does show a linear trend line. Almost exactly matching our prediction. However, there are fluctuations in the graph. Other reasons for this fluctuation can be the computer's processes as stated in 5.2.1.

6 Conclusion

In conclusion, the experimental results matched our predicted results. The graph in Appendix M shows the predicted and experimental results of both algorithms in terms of basic operations. We found that the Median algorithm is more efficient than the Brute Force Median algorithm for finding the median.

References

[1] Levitin, A., 2011. *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley.

[2] Daniel Yim. 2009. *Topics on analysis of algorithms: brute force.* [ONLINE] Available at: ftp://cosm.sfasu.edu/cs/rball/public_html/342/Daniel_WebProject/index.html. [Accessed 2 May 2018].

[3] A., K., 2004. *Algorithms: Sequential, Parallel, and Distributed.* Cengage Learning.

[4] Johnsonbaugh, R., 2003. Algorithms. TBS.

[5] Rodney Anderson. 2018. *CS Summations*. [ONLINE] Available at: http://everythingcomputerscience.com/discrete_mathematics/Summations.html. [Accessed 10 May 2018].

Appendix A – The Brute Force Algorithm

```
ALGORITHM BruteForceMedian(A[0..n – 1])
 // Returns the median value in a given array A of n numbers. This is
 // the kth element, where k=| n/2 |, if the array was sorted.
       K ← | n/2 |
       for i in 0 to n - 1 do
              numsmaller ← 0 //How many elements are smaller than A[i]
              numequal ← 0 //How many elements are equal to A[i]
              for j in 0 to n – 1 do
                     if A[j] > A[i] then
                            numsmaller ← numsmaller + 1
                     else
                            if A[j] = A[i] then
                                   numequal ← numequal + 1

              if numsmaller < k and k <= (numsmaller + numequal) then
                     return A[i]
```

Appendix B – The Median Algorithm

```
ALGORITHM Median(A[0..n – 1])
// Returns the median value in a given array A of n numbers.
if n = 1 then
       return A[0]
else
       return Select(A, 0, , n - 1) //NB: The third argument is rounded down

ALGORITHM Select(A[0..n - 1], l, m, h)
       // Returns the value at index m in array slice A[l..h], if the slice
       // were sorted into nondecreasing order.
       pos ? Partition(A, l, h)
       if pos = m then
              return A[pos]
       if pos > m then
              return Select(A, l, m, pos - 1)
       if pos < m then
              return Select(A, pos + 1, m, h)

ALGORITHM Partition(A[0..n - 1], l, h)
       // Partitions array slice A[l..h] by moving element A[l] to the position
       // it would have if the array slice was sorted, and by moving all
       // values in the slice smaller than A[l] to earlier positions, and all values
       // larger than or equal to A[l] to later positions. Returns the index at which
       // the ‘pivot’ element formerly at location A[l] is placed.
       pivotval ? A[l] // Choose first value in slice as pivot value
       pivotloc ? l // Location to insert pivot value
       for j in l + 1 to h do
              if A[j] < pivotval then
                     pivotloc ? pivotloc + 1
                     swap(A[pivotloc], A[j]) // Swap elements around pivot
       swap(A[l], A[pivotloc]) // Put pivot element in place
       return pivotloc
```

Appendix C – Implementing the Brute Force Median algorithm in C# programming language

```
1.  /// <summary>
2.  /// Finds the Median value of a given array
3.  /// </summary>
4.  /// <param name="A">Gets the array</param>
5.  /// <returns>Median value</returns>
6.  static int BruteForceMedian(int[] A)
7.  {
8.     double k = Math.Ceiling(A.Length / 2.0); //Gets the position of the Median
9.     for (int i = 0; i < A.Length; i++) //Uses each element of the array as pivot
10.    {
11.        int numsmaller = 0; //Sets variables to zero
12.        int numequal = 0; //Sets variables to zero
13.        for (int j = 0; j < A.Length; j++) //Uses each element of array to compare with the piv
    ot
14.        {
15.            if (A[j] < A[i]) //if the pivot is greater than current element
16.            {
17.                numsmaller = numsmaller + 1; //increments by one
18.            } else {
19.                if (A[j] == A[i]) //if pivot is equal to current element
20.                { //incremenet variable by one
21.                    numequal = numequal + 1;
22.                }
23.            }
24.          }
25.        if (numsmaller < k && k <= (numsmaller + numequal)) //Determines if the value of k is g
    reater than the value of numsmaller and is less than or equal to the numsmaller and numequal c
    ombineds
26.        {
27.            return A[i]; //returns pivot value as the median
28.        }
29.    }
30.    return 0;
31. }
```

Appendix D – Implementing the Median algorithm in C# programming language

```
1.  /// <summary>
2.  /// Finds the median of a given array
3.  /// </summary>
4.  /// <param name="A">Takes in a given array to solve</param>
5.  /// <returns>The median value</returns>
6.  static int Median(int[] A) //Takes in array to find the Median
7.  {
8.     if (A.Length == 1) //Checks if array has one value
9.     {
10.        return A[0]; //Returns the only value in the array as median
11.    } else {
12.        return Select(A, 0, A.Length / 2, A.Length - 1); //Runs select procedure passing the ar
    ray, number zero as a pivot, median index, and length of array
13.    }
14. }
15.
16. static int Select(int[] A, int l, int m, int h)
17. {
18.     int pos = Partition(A, l, h); //calls partition procedure and passing array, pivot, median
     index, array size. Obtains the position of the pivot
19.     if (pos == m) //Checks if position is equal to the median index
20.     {
21.         return A[pos]; //Returns array position as final median
22.     }
23.     if (pos > m) //Checks if position is greater than the median index
24.     {
25.         return Select(A, l, m, pos - 1); //Re-runs this select procedure
26.     }
27.     if (pos < m) //Checks if the position is less than the median index
28.     {
29.         return Select(A, pos + 1, m, h); //Re-runs this select procedure
30.     }
31.     return 0;
32. }
33.
34. static int Partition(int[] A, int l, int h)
35. {
36.     int pivotval = A[l]; //uses the pivot's value
37.     int pivotloc = l; //uses the pivot index
38.     for (int j = l + 1; j <= h; j++) //Uses each element in array to compare with the pivot
39.     {
40.         if (A[j] < pivotval) //if the current element is less than the pivot
41.         {
42.             pivotloc = pivotloc + 1; //increment the pivot by one
43.             swap(A, pivotloc, j); //swap pivot index with the current element
44.         }
45.     }
46.     swap(A, l, pivotloc); //swap pivot index with the pivotloc value
47.     return pivotloc; //return pivot index
48. }
```

Appendix E – Functions to swap values inside an array and generating arrays with random values

Figure 1:

```
1.  /// <summary>
2.  /// takes array, and the two indexes to swap
3.  /// </summary>
4.  /// <param name=”A”>Array</param>
5.  /// <param name=”first”>index one</param>
6.  /// <param name=”second”>index two</param>
7.  static void swap(int[] A, int first, int second)
8.  {
9.      int s = A[second]; //assigns value of index two to variable s
10.     A[second] = A[first]; //assigns value of index two with the value of index one
11.     A[first] = s; //assigns value of index one with value of variable s
12. }
```

Figure 2:

```
1.  /// <summary>
2.  /// Creates a new array
3.  /// </summary>
4.  /// <param name=”size”>the size of the array to generate</param>
5.  /// <returns>the generated array</returns>
6.  static int[] GenerateRandomArray(int size)
7.  {
8.      int[] A = new int[size]; //creates new array with given size
9.      for (int i = 0; i < A.Length; i++) { //iterates the following from zero to one
10.         int n; //creates new variable without assigning any value
11.         do {
12.             n = rand.Next(0, 10000); //generates number between 0 and 10000 //checks if the nu
    mber is already in the array, if true then repeat the do-loop to generate a new number
13.         } while (A.Contains(n));
14.         A[i] = n; //adds generated number to the array
15.     }
16.     return A; //return array
17. }
```

Appendix F – The Main functions to test functionality

Figure 1:

```
1.  /// <summary>
2.  /// Main controller of the program. Controls which algorithms, procedures and functions to run
    /// </summary>
3.  /// <param name="args"></param>
4.  static void Main(string[] args) {
5.      int numberOfTimes = 3; //number of tests for each array size
6.      for (int size = 1; size <= 10; size += 1) {
7.          for (int i = 0; i < numberOfTimes; i++) {
8.              int[] test = GenerateRandomArray(size); //calls procedure to generate a new array

9.              int medianResult = Median(test); //calls median algorithm to find median value
10.             int bruteResult = BruteForceMedian(test); //calls brute algorithm to find median v
    alue
11.             Console.Write("array: [");
12.             foreach(int x in test) {
13.                 Console.Write("{0}, ", x); //prints generated array
14.             }
15.             Console.WriteLine("]  Median: {0} brute Force Median: {1} ", medianResult, bruteRe
    sult); //prints median values for both algorithms
16.         }
17.     }
18.     Console.ReadKey(); //waits for user input to end program
19. }
```

Figure 2:

```
1.  /// <summary>
2.  /// Main controller of the program. Controls which algorithms, procedures and functions to run
    /// </summary>
3.  /// <param name="args"></param>
4.  static void Main(string[] args) {
5.      for (int size = 1; size <= 1000; size += 35) //re-
    iterates increasing the size of the array by 35
6.      {
7.          int[] test = GenerateRandomArray(size); //calls procedure to generate a new array
8.          int medianResult = Median(test); //calls median algorithm to find median value
9.          int bruteResult = BruteForceMedian(test); //calls brute algorithm to find median value

10.         Console.Write("array: [");
11.         foreach(int x in test) {
12.             Console.Write("{0}, ", x); //prints generated array
13.         }
14.         Console.WriteLine("]");
15.         Console.WriteLine("Median: {0} brute Force Median: {1} ", medianResult, bruteResult);
    //prints median values for both algorithms
16.         Console.Write("Sorted: [");
17.         Array.Sort(test); //sorts array in ascending order
18.         foreach(int x in test) {
19.             Console.Write("{0}, ", x); //prints sorted array
20.         }
21.         Console.WriteLine("]");
22.         Console.WriteLine();
23.     }
24.     Console.ReadKey(); //waits for user input to end program
25. }
```

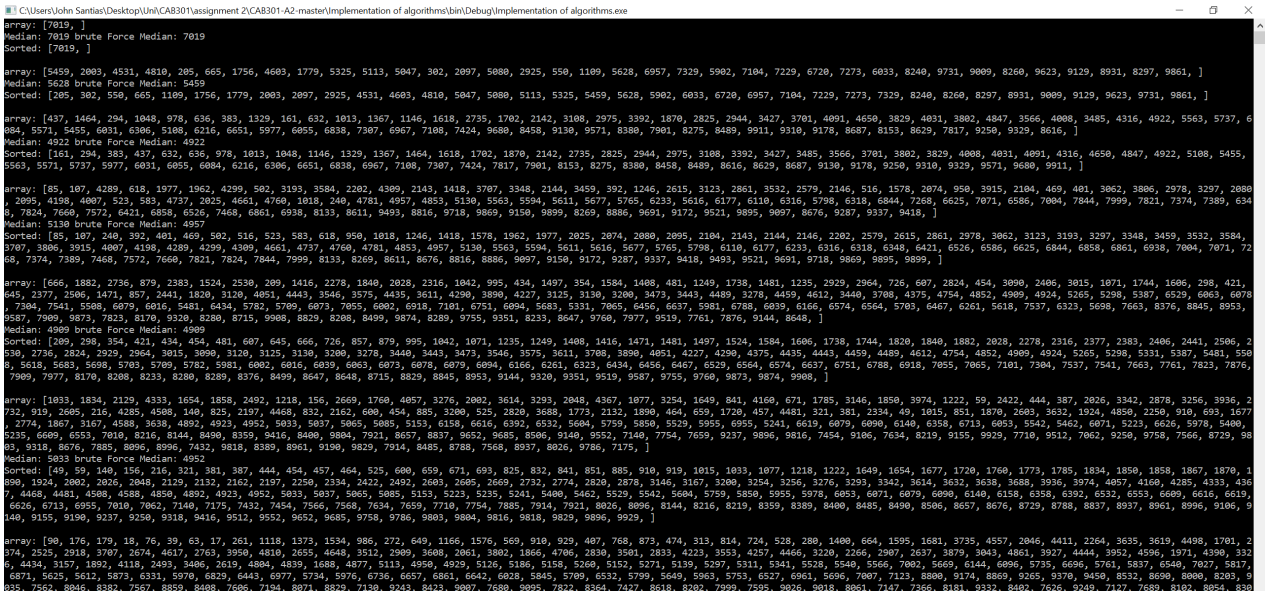
Appendix G – Testing functionality results

Figure 1:

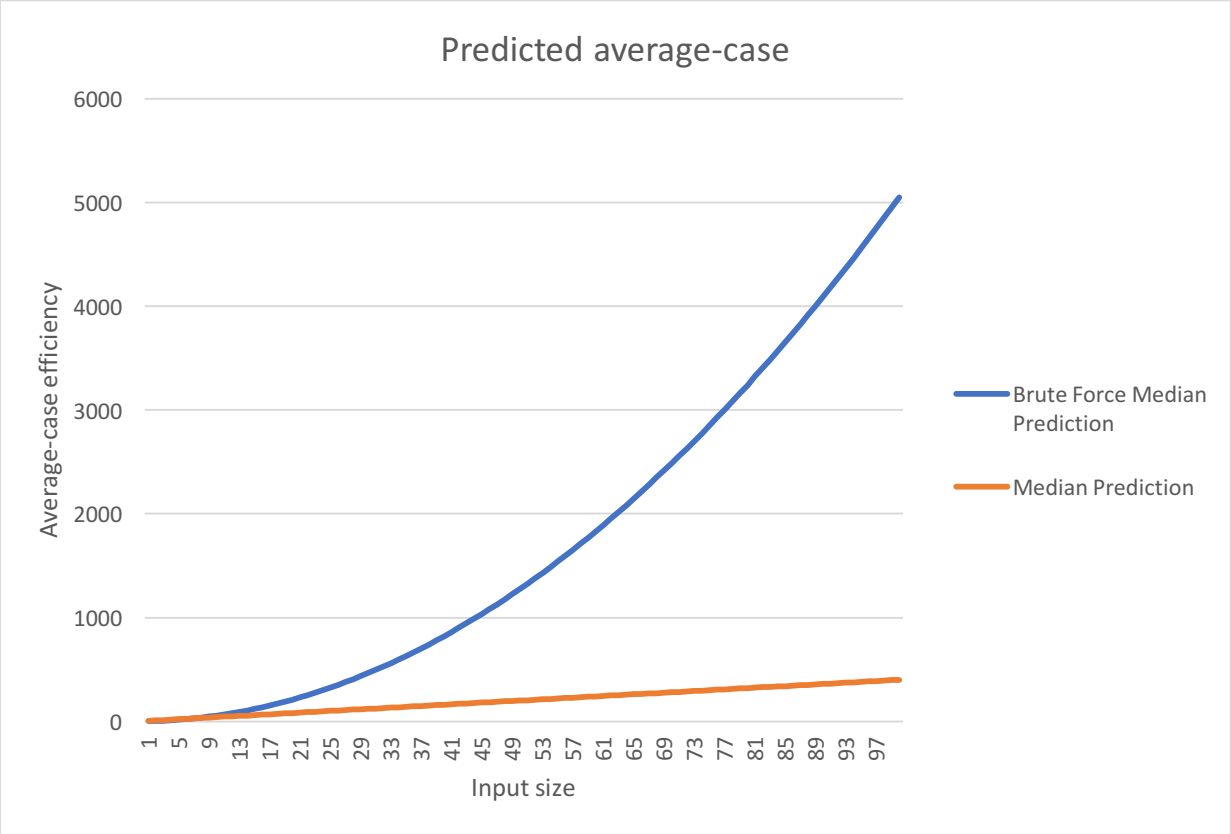
H:\Implementation of algorithms\Implementation of algorithms\bin\Debug\Implementation of algorithms.exe

```
array: {2718, ]  Median: 2718 brute Force Median: 2718
array: [2874, ]  Median: 2874 brute Force Median: 2874
array: [6475, ]  Median: 6475 brute Force Median: 6475
array: [342, 4884, ]  Median: 4884 brute Force Median: 342
array: [430, 5859, ]  Median: 5859 brute Force Median: 430
array: [2036, 6105, ]  Median: 6105 brute Force Median: 2036
array: [933, 1152, 2408, ]  Median: 1152 brute Force Median: 1152
array: [5024, 5581, 7174, ]  Median: 5581 brute Force Median: 5581
array: [374, 4506, 9484, ]  Median: 4506 brute Force Median: 4506
array: [2708, 2754, 5747, 5752, ]  Median: 5747 brute Force Median: 2754
array: [1218, 1898, 3512, 5721, ]  Median: 3512 brute Force Median: 1898
array: [360, 116, 1576, 1677, ]  Median: 1576 brute Force Median: 360
array: [643, 4438, 8501, 8823, 9782, ]  Median: 8501 brute Force Median: 8501
array: [293, 1930, 2064, 3236, 4649, ]  Median: 2064 brute Force Median: 2064
array: [3580, 6735, 9198, 9867, 9277, ]  Median: 9198 brute Force Median: 9198
array: [117, 3521, 3286, 7062, 8516, 8208, ]  Median: 7062 brute Force Median: 3521
array: [9, 1655, 3411, 3707, 6661, 9318, ]  Median: 3707 brute Force Median: 3411
array: [754, 4181, 2985, 5989, 6829, 8854, ]  Median: 5989 brute Force Median: 4181
array: [1151, 1675, 3906, 3937, 4381, 6713, 6757, ]  Median: 3937 brute Force Median: 3937
array: [3476, 3687, 4212, 5725, 5973, 6548, 7553, ]  Median: 5725 brute Force Median: 5725
array: [1350, 1600, 1559, 2141, 8289, 6449, 8377, ]  Median: 2141 brute Force Median: 2141
array: [1176, 463, 633, 1580, 2152, 7777, 5090, 5116, ]  Median: 2152 brute Force Median: 1580
array: [3288, 1663, 3210, 3464, 5016, 5059, 8716, 6866, ]  Median: 5016 brute Force Median: 3464
array: [2973, 963, 3156, 3593, 3995, 6180, 6763, 8905, ]  Median: 3995 brute Force Median: 3593
array: [564, 230, 305, 379, 2621, 4608, 4035, 6862, 8612, ]  Median: 2621 brute Force Median: 2621
array: [2356, 687, 1641, 3747, 4389, 9522, 8935, 8816, 6376, ]  Median: 4389 brute Force Median: 4389
array: [188, 477, 748, 913, 1532, 4061, 2634, 5274, 3293, ]  Median: 1532 brute Force Median: 1532
array: [636, 1614, 4885, 4804, 5379, 6597, 6870, 8228, 8156, 9384, ]  Median: 6597 brute Force Median: 5379
array: [1275, 2394, 2563, 3085, 2645, 5537, 7055, 8899, 8029, 7931, ]  Median: 5537 brute Force Median: 3085
array: [42, 3824, 3247, 2495, 4339, 4586, 5121, 5157, 5329, 9531, ]  Median: 4586 brute Force Median: 4339
```

Figure 2



Appendix H – Brute Force and Median average-case prediction



Appendix I – Counting the basic operations

```
1.  private static int counterForMedian; //global variable to store counter value of median
2.  private static int counterForBrute; //global variable to store counter value of Brute
3.  private static Random rand = new Random();
4.
5.  static int Median(int[] A)
6.  {
7.      if (A.Length == 1) {
8.          return A[0];
9.      } else {
10.         return Select(A, 0, A.Length / 2, A.Length – 1);
11.     }
12. }
13.
14. static int Select(int[] A, int l, int m, int h)
15. {
16.     int pos = Partition(A, l, h);
17.     if (pos == m) {
18.         return A[pos];
19.     }
20.     if (pos > m) {
21.         return Select(A, l, m, pos – 1);
22.     }
23.     if (pos < m) {
24.         return Select(A, pos + 1, m, h);
25.     }
26.     return 0;
27. }
28.
29. static int Partition(int[] A, int l, int h)
30. {
31.     int pivotval = A[l];
32.     int pivotloc = l;
33.     int count = 0; //new variable count starting from zero
34.     for (int j = l + 1; j <= h; j++) {
35.         count++; //increment counter by one
36.         if (A[j] < pivotval) {
37.             pivotloc = pivotloc + 1;
38.             swap(A, pivotloc, j);
39.         }
40.     }
41.     swap(A, l, pivotloc);
42.     counterForMedian += count; //add value of counter to global variable
43.     return pivotloc;
44. }
45.
46. static void swap(int[] A, int first, int second)
47. {
48.     int s = A[second];
49.     A[second] = A[first];
50.     A[first] = s;
51. }
52.
53. static int[] GenerateRandomArray(int size)
54. {
55.     int[] A = new int[size];
56.     for (int i = 0; i < A.Length; i++) {
```

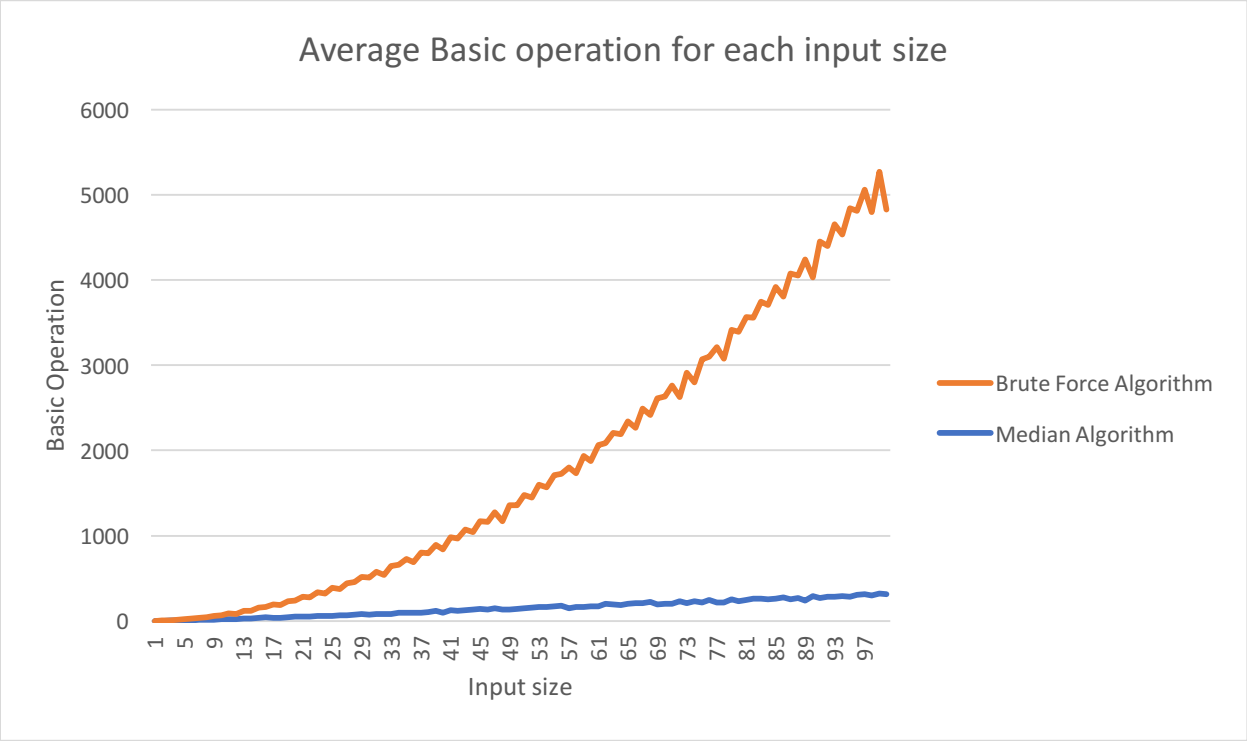
```
57.         int n;
58.         do {
59.             n = rand.Next(0, 10000);
60.         } while (A.Contains(n));
61.         A[i] = n;
62.     }
63.     return A;
64. }
65.
66. static int BruteForceMedian(int[] A)
67. {
68.    double k = Math.Ceiling(A.Length / 2.0);
69.    int counter = 0; //new counter variable starting from zero
70.    for (int i = 0; i < A.Length; i++) {
71.        int numsmaller = 0;
72.        int numequal = 0;
73.        for (int j = 0; j < A.Length; j++) {
74.            counter++; //increment counter by one
75.            if (A[j] < A[i]) {
76.                numsmaller = numsmaller + 1;
77.            } else {
78.                if (A[j] == A[i]) {
79.                    numequal = numequal + 1;
80.                }
81.            }
82.        }
83.        if (numsmaller < k && k <= (numsmaller + numequal)) {
84.            counterForBrute += counter; //Add counter value to global variable
85.            return A[i];
86.        }
87.    }
88.    return 0;
89.
90. static void Main(string[] args)
91. {
92.     int numberOfTimes = 30;
93.     for (int size = 1; size <= 100; size += 1) //number of tests for each array size
94.     {
95.         int averageOneForMedian = 0;  //new variables starting from zero
96.         int averageTwoForMedian = 0;
97.         int averageOneForBrute = 0;
98.         int averagetwoForBrute = 0;
99.  for (int i = 0; i < numberOfTimes; i++) {
100.             counterForMedian = 0;  //resets global variable to zero
101.             counterForBrute = 0;  //resets global variable to zero
102.             int[] test = GenerateRandomArray(size);
103.             Median(test);
104.             BruteForceMedian(test);
105.             averageOneForMedian += counterForMedian;  //add global variable
106.             averageOneForBrute += counterForBrute;  //add global variable
107.         }
108.         averageTwoForMedian = averageOneForMedian / numberOfTimes;  //divide
    averageOneForMedian by number of tests to get average
109.         averagetwoForBrute = averageOneForBrute / numberOfTimes;  //divide averageOneForBrute
    by number of tests to get average
110.
111.         Console.WriteLine("size: {0}  median average: {1} brute average: {2} ", size, averageT
    woForMedian, averagetwoForBrute);
112.        }
113.        Console.ReadKey();
114.        }
```

Appendix J – Execution Time

```
1.  private static double medianTimer; //Global variable
2.  private static double bruteTimer; //global variable
3.  private static Stopwatch sw; //stopwatch
4.  private static Random rand = new Random();
5.
6.  static int Median(int[] A)
7.  {
8.      if (A.Length == 1) {
9.          return A[0];
10.     } else {
11.         return Select(A, 0, A.Length / 2, A.Length – 1);
12.     }
13. }
14. static int Select(int[] A, int l, int m, int h)
15. {
16.     int pos = Partition(A, l, h);
17.     if (pos == m) {
18.         sw.Stop(); //stops stopwatch
19.         medianTimer = sw.Elapsed.TotalMilliseconds; //converts stopwatch into milliseconds and
    assigns it to the global variable medianTimer
20.         sw.Reset(); //Resets stopwatch back to zero
21.         return A[pos];
22.     }
23.     if (pos > m) {
24.         return Select(A, l, m, pos – 1);
25.     }
26.     if (pos < m) {
27.         return Select(A, pos + 1, m, h);
28.     }
29.     return 0;
30. }
31.
32. static int Partition(int[] A, int l, int h)
33. {
34.     int pivotval = A[l];
35.     int pivotloc = l;
36.     sw.Start(); //starts stopwatch
37.     for (int j = l + 1; j <= h; j++) {
38.         if (A[j] < pivotval) {
39.             pivotloc = pivotloc + 1;
40.             swap(A, pivotloc, j);
41.         }
42.     }
43.     swap(A, l, pivotloc);
44.     return pivotloc;
45. }
46.
47. static void swap(int[] A, int first, int second)
48. {
49.     int s = A[second];
50.     A[second] = A[first];
51.     A[first] = s;
52. }
53.
54. static int[] GenerateRandomArray(int size)
55. {
56.     int[] A = new int[size];
57.     for (int i = 0; i < A.Length; i++) {
```

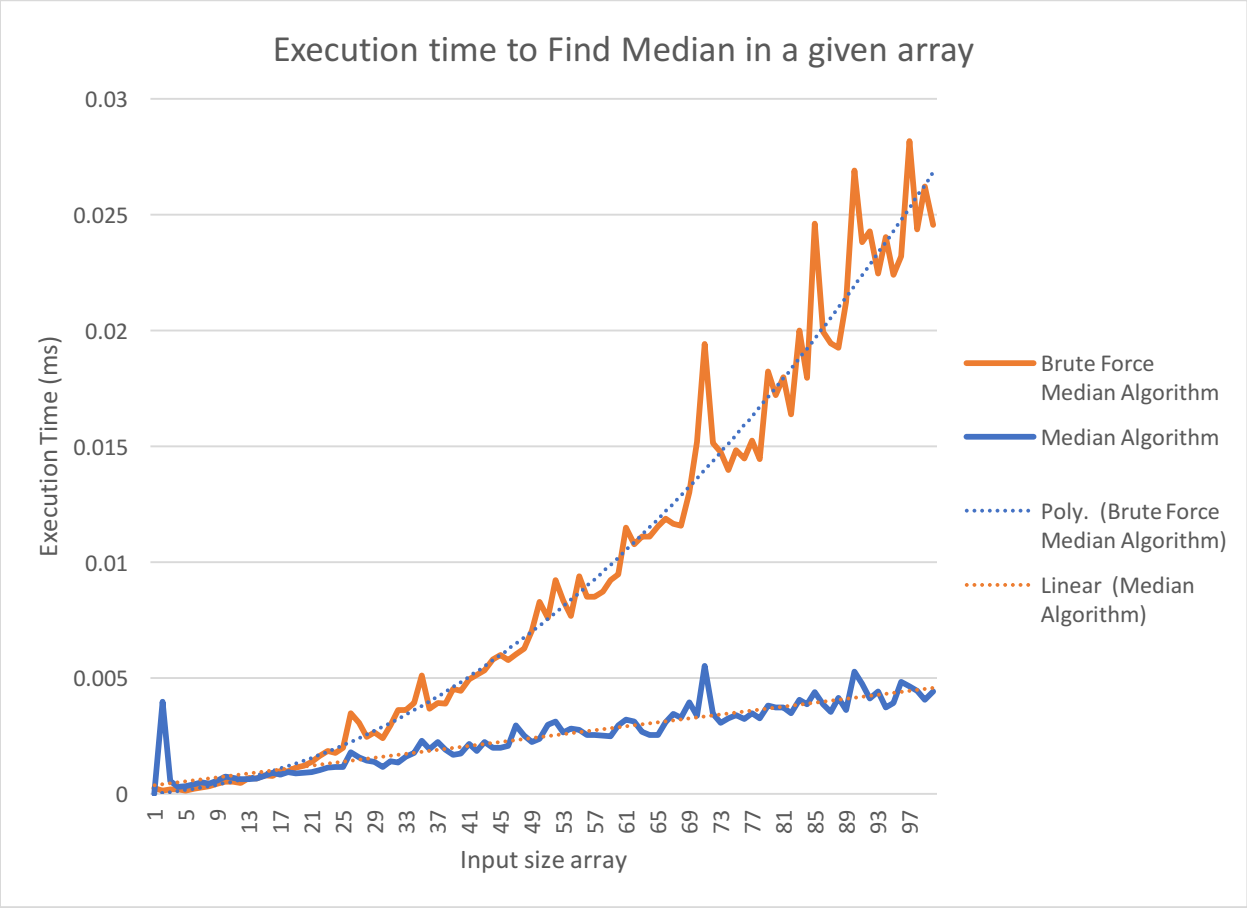
```
58.         int n;
59.         do {
60.             n = rand.Next(0, 10000);
61.         } while (A.Contains(n));
62.         A[i] = n;
63.     }
64.     return A;
65. }
66.
67. static int BruteForceMedian(int[] A)
68. {
69.     double k = Math.Ceiling(A.Length / 2.0);
70.     var timer = System.Diagnostics.Stopwatch.StartNew(); //starts stopwatch
71.     for (int i = 0; i < A.Length; i++) {
72.         int numsmaller = 0;
73.         int numequal = 0;
74.         for (int j = 0; j < A.Length; j++) {
75.             if (A[j] < A[i]) {
76.                 numsmaller = numsmaller + 1;
77.             } else {
78.                 if (A[j] == A[i]) {
79.                     numequal = numequal + 1;
80.                 }
81.             }
82.         }
83.         if (numsmaller < k && k <= (numsmaller + numequal)) {
84.             timer.Stop(); //stops stopwatch
85.             bruteTimer = timer.Elapsed.TotalMilliseconds; //assigns the timer to the global va
    riable
86.             return A[i];
87.         }
88.     }
89.     return 0;
90. }
91.
92. static void Main(string[] args)
93. {
94.     sw = new Stopwatch(); //Creates new stopwatch
95.     int numberOfTimes = 30;
96.     for (int size = 1; size <= 100; size += 1) {
97.         double averageMedianTimer = 0; //new variable with value of zero
98.         double averageBruteForceTimer = 0; //new variable with value of zero
99.         for (int i = 0; i < numberOfTimes; i++) {
100.                 int[] test = GenerateRandomArray(size);
101.                 Median(test);
102.                 BruteForceMedian(test);
103.                 averageMedianTimer += medianTimer; //adds global variable
104.                 averageBruteForceTimer += bruteTimer; //adds global variable
105.             }
106.             averageMedianTimer = averageMedianTimer / numberOfTimes; //divides the value of
    the averageMedianTimer by the number of tests run to get the average
107.             averageBruteForceTimer = averageBruteForceTimer / numberOfTimes; //divides the
    value of the averageBruteForceTimer by the number of tests run to get the average
108.             Console.WriteLine("For size {0}, execution time of Median: {1}, execution time
    of BruteForceMedian: {2}", size, Math.Round(averageMedianTimer, 5),
    Math.Round(averageBruteForceTimer, 5)); //prints results in 5 decimal places
109.         }
110.         Console.ReadKey();
111.     }
```

Appendix K – Experimental results for the Basic Operation



Input size	Median Algorithm	Brute Force Algorithm	Input size	Median Algorithm	Brute Force Algorithm	Input size	Median Algorithm	Brute Force Algorithm
1	0	1	**44**	132	913	**87**	251	3828
2	1	2	**45**	137	1035	**88**	271	3781
3	2	6	**46**	132	1028	**89**	238	4005
4	4	7	**47**	150	1128	**90**	291	3741
5	7	15	**48**	133	1036	**91**	267	4186
6	9	17	**49**	136	1225	**92**	282	4121
7	10	28	**50**	141	1213	**93**	284	4371
8	14	28	**51**	151	1326	**94**	293	4239
9	16	45	**52**	157	1294	**95**	281	4560
10	19	47	**53**	166	1431	**96**	306	4505
11	21	66	**54**	160	1404	**97**	311	4753
12	21	62	**55**	171	1540	**98**	299	4498
13	28	91	**56**	175	1554	**99**	324	4950
14	30	88	**57**	150	1653	**100**	315	4513
kk15	34	120	**58**	166	1566			
16	42	122	**59**	163	1770			
17	38	153	**60**	174	1700			
18	39	144	**61**	171	1891			
19	44	190	**62**	199	1886			
20	50	186	**63**	191	2016			
21	50	231	**64**	189	2001			
22	50	228	**65**	200	2145			
23	59	276	**66**	205	2063			
24	55	264	**67**	212	2278			
25	61	325	**68**	221	2194			
26	67	309	**69**	193	2415			
27	63	378	**70**	197	2440			
28	71	383	**71**	203	2556			
29	78	435	**72**	229	2395			
30	76	429	**73**	208	2701			
31	84	496	**74**	229	2567			
32	82	454	**75**	218	2850			
33	81	561	**76**	245	2852			
34	94	562	**77**	213	3003			
35	97	630	**78**	217	2862			
36	94	596	**79**	256	3160			
37	96	703	**80**	233	3157			
38	101	694	**81**	244	3321			
39	115	780	**82**	264	3293			
40	99	737	**83**	259	3486			
41	122	861	**84**	252	3455			
42	117	851	**85**	261	3655			
43	129	946	**86**	278	3531			

Appendix L – Experimental results for the Execution Time



Input size	Median Algorithm (ms)	Brute Force Algorithm (ms)	Input size	Median Algorithm (ms)	Brute Force Algorithm (ms)	Input size	Median Algorithm	Brute Force Algorithm (ms)
1	0	0.00026	**42**	0.00184	0.00513	**83**	0.00406	0.01999
2	0.00398	0.00013	**43**	0.00224	0.00534	**84**	0.00386	0.01797
3	0.00055	0.00019	**44**	0.00199	0.00579	**85**	0.0044	0.02462
4	0.0003	0.00019	**45**	0.002	0.006	**86**	0.00386	0.01994
5	0.00034	0.00015	**46**	0.00206	0.00578	**87**	0.00354	0.01945
6	0.0004	0.00022	**47**	0.00296	0.00602	**88**	0.00413	0.01925
7	0.0005	0.00027	**48**	0.00252	0.00627	**89**	0.00363	0.02126
8	0.00045	0.00032	**49**	0.00225	0.00704	**90**	0.00528	0.02691
9	0.00057	0.00044	**50**	0.00238	0.00828	**91**	0.00475	0.02382
10	0.00074	0.00051	**51**	0.00298	0.00759	**92**	0.00411	0.0243
11	0.00068	0.00052	**52**	0.00313	0.00923	**93**	0.00443	0.02245
12	0.00063	0.00046	**53**	0.00265	0.00834	**94**	0.00372	0.02405
13	0.00062	0.00069	**54**	0.00283	0.00767	**95**	0.00392	0.0224
14	0.00065	0.00065	**55**	0.00275	0.0094	**96**	0.00482	0.0232
15	0.00078	0.00079	**56**	0.00254	0.00852	**97**	0.00464	0.02819
16	0.00092	0.00078	**57**	0.00253	0.00851	**98**	0.00444	0.02436
17	0.00084	0.00098	**58**	0.0025	0.00872	**99**	0.00406	0.02621
18	0.00093	0.00096	**59**	0.00248	0.00922	**100**	0.00442	0.02456
19	0.00089	0.00114	**60**	0.00295	0.00947	**83**	0.00406	0.01999
20	0.00092	0.00122	**61**	0.00319	0.0115	**84**	0.00386	0.01797
21	0.00094	0.00139	**62**	0.00311	0.01078	**85**	0.0044	0.02462
22	0.00101	0.00163	**63**	0.00269	0.0111	**86**	0.00386	0.01994
23	0.00114	0.00186	**64**	0.00255	0.0111	**87**	0.00354	0.01945
24	0.00117	0.00178	**65**	0.00253	0.01156	**88**	0.00413	0.01925
25	0.00117	0.00199	**66**	0.00309	0.01189	**89**	0.00363	0.02126
26	0.00179	0.00349	**67**	0.00345	0.01167	**90**	0.00528	0.02691
27	0.00157	0.00307	**68**	0.00329	0.01158	**91**	0.00475	0.02382
28	0.00143	0.00246	**69**	0.00395	0.01302	**92**	0.00411	0.0243
29	0.00137	0.00264	**70**	0.00338	0.01519	**93**	0.00443	0.02245
30	0.00117	0.0024	**71**	0.00553	0.01942	**94**	0.00372	0.02405
31	0.00142	0.00296	**72**	0.00344	0.01514	**95**	0.00392	0.0224
32	0.00136	0.00362	**73**	0.00307	0.01476	**96**	0.00482	0.0232
33	0.00161	0.00362	**74**	0.00327	0.01399	**97**	0.00464	0.02819
34	0.00176	0.00391	**75**	0.00339	0.01485	**98**	0.00444	0.02436
35	0.00229	0.00511	**76**	0.00324	0.01449	**99**	0.00406	0.02621
36	0.00194	0.00368	**77**	0.00348	0.01524	**100**	0.00442	0.02456
37	0.00224	0.00391	**78**	0.00326	0.01446			
38	0.0019	0.0039	**79**	0.00381	0.01823			
39	0.00168	0.00452	**80**	0.00373	0.0172			
40	0.00174	0.00446	**81**	0.00372	0.018			
41	0.00216	0.00494	**82**	0.00347	0.01639			

Appendix M – Experimental and predicted results for basic operation