CAB301 Assignment 1: Empirical Analysis of a Brute Force Algorithm for Finding the Median of an Array

Shane Havers N9477756

N9477756

Date Submitted: 11th April 2017

Summary

The purpose of this report is to summarise the outcomes of experiments to measure the executions of the basic operation and execution times of an algorithm that finds the median of a set of numbers. The tests were completed by writing a C++ program in Code Blocks, and outputting the results to Comma Separated Value files. These output files were the imported and graphed in MATLAB, and the results were shown to be consistent with the predicted efficiency of the algorithm.

1. Description of the Algorithm

The algorithm being analysed (shown in Appendix 1 on page 7) searches through an input array to find the median element. It searches through every element of the array, and for each element counts how many elements in the rest of the array are smaller than it and how many elements are equal to it. Based on these counts, if half of the rest of the array is smaller than or equal to the element, that element is deemed to be the median.

It should be noted that this algorithm will only find one median. Properly speaking, if the array's size is an even number the median should be taken as the mean of the two middle elements [1]. This algorithm, however, uses the first median candidate that it finds as the true median, even if the array has two medians. Therefore, this algorithm should not be used to in statistical analysis were finding the true mean is imperative – rather, it should be used to obtain a general idea of the central tendency of the set.

2. Theoretical Analysis of the Algorithm

Identifying the Algorithm's Basic Operation

Clearly the part of the algorithm that has the greatest impact on the algorithm's efficiency is searching through each element, and deciding whether that term is smaller than or equal to the search term. Therefore, the chosen basic operation in this case is the comparison operation, which is performed twice within the innermost for loop - once to see if an element is smaller than the search term, and once to see if the element is equal to.

Also of note is the fact that, in the outermost for loop, there are two more conditional statements that are used to determine whether the current element is the median. The impact of these statements will be much smaller than the conditionals inside the inner for-loop, therefore for the sake of simplicity it will be ignored for this analysis.

Analysis of Average-Case Efficiency

Through observation of the algorithm, the inner for-loop is made up of two distinct parts: the basic operation checking if *A[j]* is less than *A[i]*, and the basic operation checking if the two terms are equal. The first operation is always performed, regardless of what the two elements are; however, the second operation will only be performed if the *A[j]* is, in fact, less than *A[i]*.

If *A[j]* is the smallest value in the array, then the second comparison will be made on every iteration of the inner for-loop. If, however *A[j]* is the largest value, the second comparison will never be made. Both assumptions hold true for non-trivial arrays (i.e. arrays that contain more than one distinct element). Therefore, it is reasonable to assume that values that are between these extremes will execute the second statement some proportion of the time, and that it run half the time for the median value.

Because the entire array is being searched, the first operation will run n times on each iteration of the inner for-loop. The smallest value will cause the second operation to run 0 times, the largest value will cause it to run n times, and the median value will cause it to run $\frac{n}{2}$ times. Therefore, to simplify the analysis, **the assumption that on average the second operation runs $\frac{n}{2}$ times will be made,** and hence **the basic operation runs $n + \frac{n}{2} = \frac{3n}{2}$ times within the inner for loop.**

The analysis of this algorithm was not found in literature – however, it can be thought of as being like Levitin's SequentialSearch algorithm in that it searches each item in the array and if an element meets the search criteria that element is returned [2]. The key differences are:

- Instead of performing only one basic operation to determine if an element meets the criteria, $\frac{3n}{2}$ basic operations are performed; and
- The probability of a successful search, p, will always be 1, as the median of a set must always exist.

Hence, Levitin's formula for the average case efficiency of the SequentialSearch algorithm may be adjusted to: $C_{avg}(n) = \frac{\left(\frac{3n}{2}\right) \times (n-1)}{2}$, which simplifies further to:

$$\boldsymbol{C_{avg}(n) = \frac{3n}{4}(n-1)}$$

Order of Growth

Expanding the equation above gives $C_{avg}(n) = \frac{3n^2}{4} - \frac{3n}{4}$. Clearly, the dominant coefficient in this equation is the $\frac{3n^2}{4}$, which is of the order n^2. Therefore, it can be stated that **this algorithm belongs to the efficiency class, $\boldsymbol{C_{avg}(n) \in \Theta(n^2)}$** . This makes sense, as it implies that as the size of the input becomes greater, the number of comparisons to be made becomes *much* larger.

3. Methodology, Tools and Techniques

The algorithm, tests and experiments were implemented in C++ using the Code Blocks IDE. This programming language is a well-used Object Oriented Programming language dating back as far as 1979, and is still widely used today for both its low-level simplicity and its ability to create classes [3]. The implementation of the Brute Force Median algorithm in C++ is depicted in Appendix 2 on page 8, as the function named BruteForceMedian. The 'double' type was used, so that the both decimal and whole numbers were accommodated for.

To count the algorithm's basic operations and to time it, the algorithm was run for various array sizes, and each size was run for many trials so that an average could be obtained. The array sizes range from 200 to 10,000 in steps of 200, and on each size the algorithm was executed 100 times. The input sizes are deemed to be appropriate, as it will show how the algorithm behaves over a large range of inputs. The number of executions per size is justified because it will smooth out anomalies while keeping the execution time of the program from reaching astronomical times.

The language's 'random' library was used to generate the pseudo-random numbers that filled the arrays in the experiments, with the Mersenne Twister 19937 generator as the seed for the Uniform Integer Distribution between $\pm\ 20{,}000$. This range is used because the maximum input size is 10,000, so a range of this size will reduce the chance of duplicates

The 'fstream' library was used to output data to Comma Separated Value files (.csv), and the 'chrono' library was used to time how long the algorithm ran for. Hinnant's advice on how to record runtime was consulted, as well as BHawk's example on how to output data to a .csv file [4] [5].

These experiments were undertaken on a Microsoft Surface Pro 4, running Microsoft's flagship operating system Windows 10. This was chosen purely out of convenience, as it was the only device/OS available at the time.

The graphing package chosen was MATLAB R2016A. MATLAB has many functions that allow for easy importing of .csv files, data manipulation and data visualisation. The figures were initially exported to the Portable Network Graphics files. They were then added to this report, which was prepared using Microsoft Word 2016. MATLAB was also used to calculate the expected basic operations count; all graphing and data manipulation is completed with the MATLAB script depicted in Appendix 3 on page 9.

4. Experimental Results

Program Setup

The setup of the program can be viewed in Appendix 4 on page 10. This shows the #include statements, constants initialised as #define statements, function prototypes, global variables, main method (the entry point of the program), the method for generating the array of random integers, and the method for generating array sizes.

The program starts by testing the algorithm, by running the AlgorithmTests function. Then after the outcome of the tests are displayed, the basic operations are counted using the AlgorithmBasicOpsCount method. Finally, the algorithm is timed using the AlgorithmTimeTrials method.

Functional Testing

To verify that the implemented algorithm worked as it should, a series of test cases were written. Each test involved passing in a known array with a known median into the algorithm, and verifying that the function returns the correct median. The code for the test is contained in the AlgorithmTests function, as shown in Appendix 5 on page 11, and is executed in the main function. A summary of each test is provided below:

1. A general case
2. When all the elements in the array are the same
3. When there are only two distinct elements
4. When there are an even number of elements
5. When there is only one element in the array
6. When the array contains only two distinct elements and an even number of elements
7. Verifying that the algorithm still works on decimal numbers; and
8. Verifying that the algorithm still works on negative numbers.

The results of the tests are outputted to the standard console. As shown in Figure 1 this function passes all the tests and is therefore fit to be used in the rest of the analysis.

```
Test 1   Expected Median = 42    Calculated Median = 42
Test 2   Expected Median = 10    Calculated Median = 10
Test 3   Expected Median = 1     Calculated Median = 1
Test 4   Expected Median = 44    Calculated Median = 44
Test 5   Expected Median = 5     Calculated Median = 5
Test 6   Expected Median = 1     Calculated Median = 1
Test 7   Expected Median = 8.22  Calculated Median = 8.22
Test 8   Expected Median = -0.44 Calculated Median = -0.44
All Tests Passed
```

Figure 1: Console output upon successful completion of all test cases

Number of Basic Operations

To count the basic operations, the Brute Force Median had to be modified to accommodate the change in purpose. For simplicity, a new function called BruteForceMedian_CountBasicOps (Shown in Appendix 6 on page 12) was created, and this was used instead of the BruteForceMedian function. The differences were:

- A new 'long' variable was created, called basicOperations. 'Long' was used because the numbers stored in the variable can be larger than the maximum allowable 'int' value.
- The basicOperations variable is incremented once if it reaches the first basic operation, and once also if it reaches the second basic operation; and
- Instead of returning the median, the basicOperations variable is returned, and this is reflected by the function's return type.

To execute the algorithm and calculate the counts, the AlgorithmBasicOpsCount (shown in Appendix 7 on page 13) is executed in the main function. Upon completion of the function, the results are output to a .csv file, which was imported into MATLAB for plotting.

Show below in Figure 2 is the expected results of counting the basic operations, and the actual results. This graph confirms the algorithm's expected growth; the actual counts not only follow the expected counts somewhat closely, but it also appears to grow quadratically. The exception to this is that towards the end, the expected results tend to deviate from what is expected. This is most likely because the result of having the median towards the beginning/end of the input is much more exaggerated.

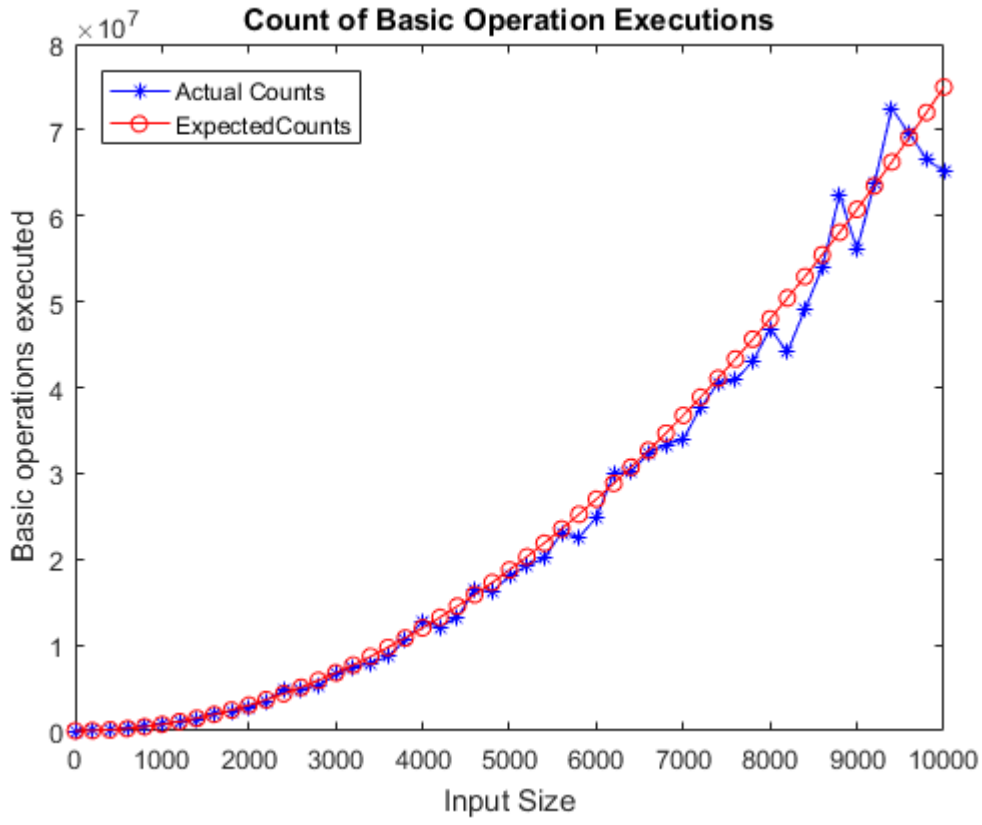

Figure 2: Basic operation executions as a function of input size

Execution Time

Timing the execution of the algorithm was completed using the original, unmodified function, and by taking the difference between before the function started and after the function finished.

The only problem encountered when running the AlgorithmTimeTrials function (shown in Appendix 8 on page 14) was that for relatively small inputs the execution time was recorded as 0 milliseconds. This implies that the resolution of the clock used was not large enough to record the very fast executions. This problem was overcome after the input size reached approximately 600, so these times may be considered quick enough that they are essentially zero without having too large an impact on the rest of the analysis.

The results of the execution time analysis are shown in Figure 3 below. Note that the expected values were not shown on this graph, because the times would vary from computer to computer – however, one can imagine that it would follow the same quadratic trend as with the basic operation counts. The graph below also shows that for larger input sizes, the execution time deviates from the expected trend in a very similar way to how the basic operations count does. This is again likely because early/late medians result in erratic behaviour.

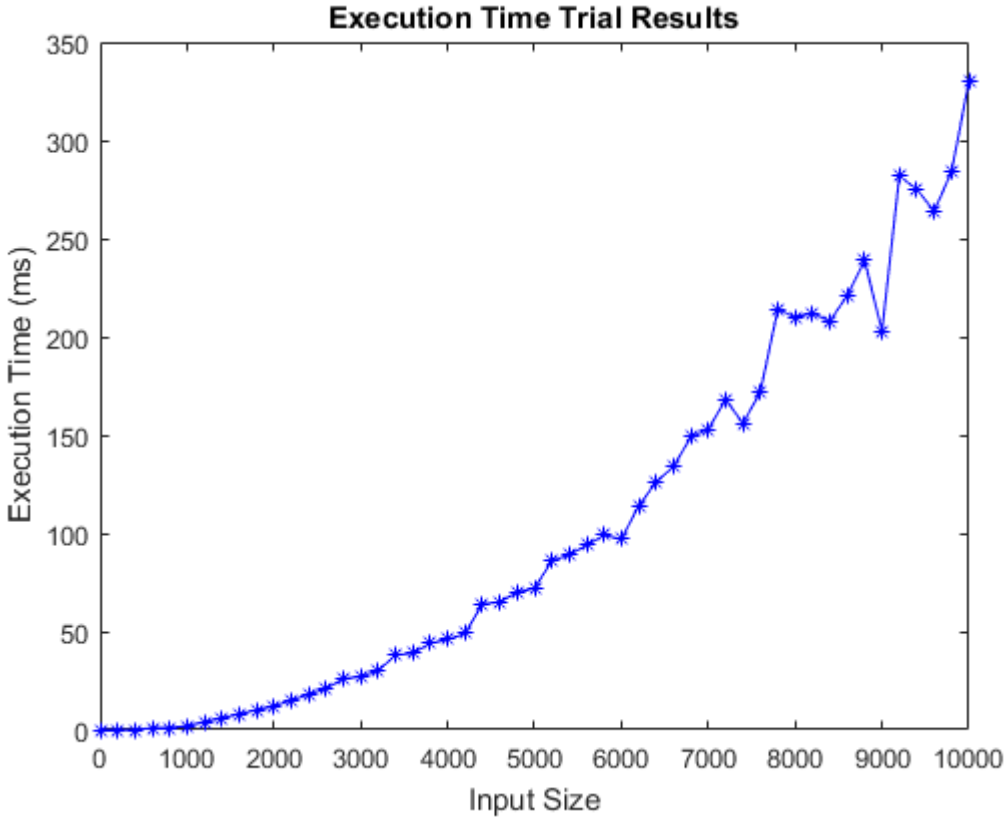

Figure 3: Algorithm execution times as a function of input size

References

[1] "The Median of a Set of Data." [Online]. Available: http://www.mathgoodies.com/lessons/vol8/median.html. [Accessed: 09-Apr-2017].

[2] A. Levitin and S. Mukherjee, *Introduction to the design & analysis of algorithms*. Addison-Wesley Reading, 2003.

[3] "Albatross," "History of C++." [Online]. Available: http://www.cplusplus.com/info/history/. [Accessed: 09-Apr-2017].

[4] H. Hinnant, "c++ - How to get duration, as int milli's and float seconds from <chrono>?," 18-Jan-2013. [Online]. Available: http://stackoverflow.com/questions/14391327/how-to-get-duration-as-int-millis-and-float-seconds-from-chrono. [Accessed: 09-Apr-2017].

[5] "BHawk," "Writing .csv files from C++," 03-Mar-2015. [Online]. Available: http://stackoverflow.com/questions/25201131/writing-csv-files-from-c. [Accessed: 09-Apr-2017].

Appendix 1: BruteForceMedian 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 $k = \lfloor n/2 \rfloor$, 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** $k \leq (numsmaller + numequal)$ **then**

    return $A[i]$

Appendix 2: BruteForceMedian Function Implementation in C++

```
double BruteForceMedian(double A[], int n) {
    // Returns the median value in a given array A of n numbers.
    // This is the 'k'th element, where k = n/2 rounded up if
    // the array was sorted

    // Define Variables
    int numsmaller;
    int numequal;
    int k = (int)ceil(n/2.0);


    // Begin Searching
    for (int i = 0; i <= n-1; i++) {
        // Initialise variables
        numsmaller = 0;
        numequal = 0;

        // Check to see if the 'i'th term is the median
        for (int j = 0; j <= n-1; j++) {
            if (A[j] < A[i]) {
                numsmaller++;
            } else if (A[j] == A[i]) {
                numequal++;
            }
        }
        if ((numsmaller < k) && (k <= (numsmaller + numequal))) {
            return A[i];
        }
    }


}
```

Appendix 3: Graphing and Data Manipulation MATLAB script

```
% Script to plot data from CAB301 Assignment 1, Semester 1, 2017
clear; close all; clc;
%% Basic Operations data
% Import data
basicOps_inputSizes = csvread('basicOps.csv', 1, 0, [1, 0, 51, 0]);
basicOps_counts = csvread('basicOps.csv', 1, 1, [1, 1, 51, 1]);

% Calculate expected values
basicOps_expected = (3 .* basicOps_inputSizes)/4 .* (basicOps_inputSizes -
1);

% Plot
figure;
plot(basicOps_inputSizes, basicOps_counts, '-*b')
hold on
plot(basicOps_inputSizes, basicOps_expected, '-or')
legend('Actual Counts', 'ExpectedCounts', 'Location', 'northwest')
title('Count of Basic Operation Executions')
xlabel('Input Size')
ylabel('Basic operations executed')

%% Time Trial Data
% Import Data
timeTrial_inputSizes = csvread('timeTrials.csv', 1, 0, [1, 0, 51, 0]);
timeTrial_counts = csvread('timeTrials.csv', 1, 1, [1, 1, 51, 1]);

% Plot
figure;
plot(timeTrial_inputSizes, timeTrial_counts, '-*b')
title('Execution Time Trial Results')
xlabel('Input Size')
ylabel('Execution Time (ms)')
```

Appendix 4: Setup of C++ Program

```
#include <iostream>
#include <math.h>
#include <chrono>
#include <fstream>
#include <random>
#include <time.h>
#include <climits>

#define NUMBEROFTRIALS 100


using namespace std;

void AlgorithmTests();
void AlgorithmBasicOpsCount();
void AlgorithmTimeTrials();
double BruteForceMedian(double A[], int n);
long BruteForceMedian_CountBasicOps(double A[], int n);
double* GenerateRandomArray(double inputArray[], int arraySize);
int* GenerateSizeArrays(int arraySizes[], int arraySize);
std::mt19937 gen(time(NULL));
std::uniform_int_distribution<int> distribution(-20000, 20000);

int main()
{
    AlgorithmTests();
    cout << "Begin Counting Basic Operations..." << endl;
    AlgorithmBasicOpsCount();
    cout << "Basic Operations Counted" << endl << "Begin Time Trials..." << endl;
    AlgorithmTimeTrials();
    cout << "Time Trials Complete" << endl;
}

double* GenerateRandomArray (double inputArray[], int arraySize) {

    for (int j = 0; j < arraySize; j++) {
        inputArray[j] = (double)distribution(gen);
    }

    return inputArray;
}

int* GenerateSizeArrays (int arraySizes[], int arraySize) {
    for (int j = 0; j < arraySize; j++) {
        arraySizes[j] = (j+1)*200;
    }

    return arraySizes;

}
```

Appendix 5: AlgorithmTests function for testing the algorithm implementation

```
void AlgorithmTests() {
    // First test - 7 elements, clear median = 42
    double testArray1[] = {17, 63, 59, 23, 78, 42, 13};

    // Second test - 5 elements, all same elements, median = 10
    double testArray2[] = {10, 10, 10, 10, 10};

    // Third test - 7 elements, only 2 distinct elements, median  = 1
    double testArray3[] = {1, 1, 2, 1, 2, 1, 2};

    // Fourth Test - 8 elements, 2 medians, returned median = 44
    double testArray4[] = {46, 43, 42, 48, 41, 45, 44, 47};

    // Fifth Test - 1 element, which should be its own median = 5
    double testArray5[] = {5};

    // Sixth Test - 10 elements, 2 distinct elements, 2 medians, returned median = 1
    double testArray6[] = {1, 1, 1, 1, 1, 2, 2, 2, 2, 2};

    // Seventh Test - 12 elements, all decimal numbers, 2 medians, returned median = 8.22
    double testArray7[] = {5.06, 7.03, 6.82, 3.14, 22.5, 9.0, 45.676, 8.22, 39.2, 10.0, 22.11, 7.031};

    // Eighth Test - 11 elements, some negative, returned median = -0.44
    double testArray8[] = {5.13, -2.5, 3, -0.44, 6.5, -5.66, 77, -0.88, -9, 1.0003, -1.111};


    // Run Tests
    double testResult1 = BruteForceMedian(testArray1, 7);
    double testResult2 = BruteForceMedian(testArray2, 5);
    double testResult3 = BruteForceMedian(testArray3, 7);
    double testResult4 = BruteForceMedian(testArray4, 8);
    double testResult5 = BruteForceMedian(testArray5, 1);
    double testResult6 = BruteForceMedian(testArray6, 10);
    double testResult7 = BruteForceMedian(testArray7, 12);
    double testResult8 = BruteForceMedian(testArray8, 11);

    cout << "Test 1\t Expected Median = 42 \t Calculated Median = " << testResult1 << endl;
    cout << "Test 2\t Expected Median = 10 \t Calculated Median = " << testResult2 << endl;
    cout << "Test 3\t Expected Median = 1 \t Calculated Median = " << testResult3 << endl;
    cout << "Test 4\t Expected Median = 44 \t Calculated Median = " << testResult4 << endl;
    cout << "Test 5\t Expected Median = 5 \t Calculated Median = " << testResult5 << endl;
    cout << "Test 6\t Expected Median = 1 \t Calculated Median = " << testResult6 << endl;
    cout << "Test 7\t Expected Median = 8.22  Calculated Median = " << testResult7 << endl;
    cout << "Test 8\t Expected Median = -0.44 Calculated Median = " << testResult8 << endl;

    if ((testResult1 == 42) && (testResult2 == 10) && (testResult3 == 1) && (testResult4 == 44)
        && (testResult5 == 5) && (testResult6 == 1) && (testResult7 == 8.22)) {
            cout << "All Tests Passed" << endl;
        } else {
            cout << "Testing unsuccessful" << endl;
    }
}
```

Appendix 6: BruteForceMedian_CountBasicOps modified algorithm

```
long BruteForceMedian_CountBasicOps (double A[], int n) {
    // Returns the number of basic operations performed
    // when running the BruteForceMedian Algorithm

    // Define Variables
    int numsmaller;
    int numequal;
    int k = (int)ceil(n/2.0);
    long basicOperations = 0;


    // Begin Searching
    for (int i = 0; i <= n-1; i++) {
        // Initialise variables
        numsmaller = 0;
        numequal = 0;

        // Check to see if the 'i'th term is the median
        for (int j = 0; j <= n-1; j++) {
            // Add to Basic Ops count when reaching this point
            basicOperations++;
            if (A[j] < A[i]) {
                numsmaller++;

            } else {
                // Add to Basic Ops count when reaching this point
                basicOperations++;
                if (A[j] == A[i]) {
                    numequal++;
                }
            }
        }
        if ((numsmaller < k) && (k <= (numsmaller + numequal))) {
            return basicOperations;
        }
    }


}
```

Appendix 7: AlgorithmBasicOpsCount function used to count the number of basic operation executions

```
void AlgorithmBasicOpsCount() {
    // Declare variables
    long long opsCount;
    double *inputArray;

    // Declare Array of input sizes, and the number that there are
    int n = 50;
    int *arraySizes;
    arraySizes = new int[n];
    arraySizes = GenerateSizeArrays(arraySizes, n);

    // Open CSV File for writing
    ofstream basicOps;
    basicOps.open("basicOps.csv");

    // Write Columns
    basicOps << "Input Size, Basic Operations\n0,0\n";

    // Begin testing
    for (int i = 0; i < n; i++) {
        opsCount = 0;
        for (int j = 0; j < NUMBEROFTRIALS; j++) {
            // Fill Input with random values
            inputArray = new double[arraySizes[i]];
            inputArray = GenerateRandomArray(inputArray, arraySizes[i]);

            // Carry out algorithm and count basic operations
            opsCount += BruteForceMedian_CountBasicOps(inputArray, arraySizes[i]);
        }

        // Take the average number of basic operations
        opsCount /= NUMBEROFTRIALS;

        // Output the size of the input and the number of basic operations carried out
        basicOps << arraySizes[i] << "," << opsCount << "\n";

        // Delete the input array so that it can be reused
        delete [] inputArray;
    }

    // Close CSV File
    basicOps.close();
}
```

Appendix 8: AlgorithmTimeTrials function for timing the execution of the algorithm

```
void AlgorithmTimeTrials () {
    // Declare Variables
    std::chrono::duration<float> timeInSeconds;
    double *inputArray;
    std::chrono::steady_clock::time_point startpoint;
    std::chrono::steady_clock::time_point endpoint;

    // Declare Array of input sizes, and the number that there are
    int n = 50;
    int *arraySizes;
    arraySizes = new int[n];
    arraySizes = GenerateSizeArrays(arraySizes, n);


    // Open CSV File for writing
    ofstream timeTrial;
    timeTrial.open("timeTrials.csv");

    // Write Columns and leading 0's
    timeTrial << "Input Size, Time (ms)\n0,0\n";

    // Begin time trials
    for (int i = 0; i < n; i++) {

        timeInSeconds = (std::chrono::duration<float>)0;

        for (int j = 0; j < NUMBEROFTRIALS; j++) {

            // Fill Input with random values
            inputArray = new double[arraySizes[i]];
            inputArray = GenerateRandomArray(inputArray, arraySizes[i]);

            // Begin Timing
            startpoint = std::chrono::steady_clock::now();

            // Run Algorithm
            BruteForceMedian(inputArray, arraySizes[i]);

            // Stop Timing
            endpoint = std::chrono::steady_clock::now();

            // Calculate time, and add to total time
            timeInSeconds += endpoint - startpoint;
        }

        // Take the average time in seconds
        timeInSeconds /= NUMBEROFTRIALS;

        // Convert the time duration to milliseconds
        auto timeInMilli = std::chrono::duration_cast<std::chrono::milliseconds>(timeInSeconds);

        // Write the time taken to CSV
        timeTrial << arraySizes[i] << ","; //<< timeInSeconds << "\n";
        timeTrial << timeInMilli.count() << "\n";

        // Delete the input array so that it can be reused
        delete [] inputArray;
    }

    // Close CSV File
    timeTrial.close();
}
```