APRIL 15, 2017

EMPIRICAL ANALYSIS OF AN ALGORITHM

CAB301 ASSIGNMENT 1

JESSE STANGER
N9162259

CONTENTS

Summary

This report will analyse the average case efficiency and time complexity of a Brute Force Median algorithm. The algorithm is implemented in C++11. The number of basic operations and the execution time of the algorithm were compared across various array lengths, and averages were formed for each array length to ensure consistency. The average case efficiency for basic operations and execution time was found to be consistent with the theoretical predictions.

1. Description of Algorithm

The relevant brute force median algorithm described in Appendix A takes any array of integers, and finds the median value of the entire array. From the first item in the array to N - 1, it compares each sequential value against the rest of the array, and if at the end of each iteration it finds the current value is the median, then it returns that value. Thus, the number of basic operations is always a multiple of n. Delving into more detail of how, the algorithm uses two variables, numSmaller and numEqual, to determine when it has found the median, and will break out of the array. This algorithm gets more efficient depending on the distance from the start of the array the median value lies. Therefore, the lower the range between array value min and max, and the higher the array length, the more efficient the algorithm is.

2. Theoretical Analysis of the Algorithm

This section describes the algorithm's anticipated time complexity from a theoretical perspective.

2.1 Identifying the Algorithm's Basic Operation

The part of the algorithm that has the greatest toll on the execution time of the program are the two for loops. In the innermost loop, the program compares every item in the array against the first element, and then every item in the array against the second element, and so on. This operation takes an exponential amount of time $\theta(n^2)$ based on array length, and contributes the greatest to the execution time. The if and else if statements, and all assignment and increment statements, contribute an insignificant and constant amount to the program's execution time, and therefore the for loops are the basic operation of this algorithm.

2.2 Average-Case Efficiency

The average case efficiency is given by the fact that each element in the array has an equal probability of being the median, whether the median lies in the first position of the array or last position. Appendix B outlines the function that gives the average probability for each element in the array being the median. For example, an array length of 1000 should yield 500500 basic operations, and an array length of 10000 should yield 50005000 basic operations. Appendix C shows the average efficiency is an exponential function when graphed, and lies bounded by the upper and lower (best and worst case) bounds of the function (Tang, 2017).

2.3 Order of growth

The highest power in the equation of Appendix B is of efficiency class $\theta(n^2)$. Thus, similarly to the growth in basic operations, we can expect execution time to have quadratic growth as array length increases.

3. Methodology, Tools and Techniques

This section briefly summarises the computing environment used for the experiments.

1. The algorithm and experiments performed were implemented in C++ and compiled with g++ following the C++11 ISO standard. StackOverflow (StackOverflow, n.d.) was used to learn more about how to output to CSV (BHawk, 2015), how to measure execution time between two points using the <chrono> library (user3762106, 2015), implementing vectors, including generating random numbers and shuffling (Maverik, 2015; Tunjic, 2014).
2. The experiments were carried out on a Windows based desktop computer with an Intel 2500K CPU running at 4.3Ghz. The random numbers were generated using the C++ built-in rand() function of the <iostream> library, seeded based on the current time (srand(time(NULL))). The external <chrono> library was used to measure time more accurately (microseconds) than the built in <time.h> library, whose smallest measurement is in seconds. For experiments involving time measurements, the basic operation counter was removed temporarily to ensure the validity of the measurements, and any unnecessary open processes were ended so they did not interfere with the running of the experiment.
3. The results of each experiment were exported to .CSV format for analysis in Microsoft Excel. In Excel, these data points were converted into a PivotTable, whose results were graphed. These graphs demonstrate the exponential nature of the algorithm for each array length n in terms of basic operations and time efficiency.
4. The technique used to generate the data is described by the following points
    a. Data was collected for each array length between 1000 and 20000, in increments of 1000.
    b. One set of random numbers was generated for each unique array length, whose values range between 1 – 1000000. The random numbers were generated using the Mersenne Twister engine std::mt19937 allowing each number to have a uniform chance of generating.
    c. For each unique array length (1000, 2000, 3000, etc), 200 shuffled versions of that array were pushed through the algorithm to determine an accurate average basic operations and execution time. By shuffling the numbers you change the location of the median in the array, while also saving computer resources and reducing experiment output time, as regenerating takes more time and memory. This implementation uses the Mersenne Twister engine std::mt19937 to shuffle the elements, allowing all possible permutations of the same array have an equally likely chance of generating.

4. Experimental Results

This section describes the outcomes of the experiments and compares the results with the theoretical predictions from Section 2. The programming language implementation of the algorithm is shown in Appendix A.

4.1 Functional Testing

The implementation of the algorithm in Appendix A can be found in Appendix D. The functional correctness of this implementation was tested using the test function defined in Appendix E. This function prints the contents of the vector that was pushed through the brute force median algorithm, as well as the median value. Many different instances of arrays were tested, including values that are Sorted, Random, Nearly Sorted, Reversed, Few Unique, and an array of Odd Length. Each vector of values contained numbers from 1-20, and a length of 10 (except Odd Length, with length 9). Appendix F outlines the test arrays and their results after passing through the algorithm. These tests confirm that the implementation finds the median value successfully for all potential permutations of positive integers.

4.2 Average-Case Number of Basic Operations

To measure the number of basic operations of the brute force median algorithm, a counter was placed inside the second for loop of the algorithm. To get an accurate average of the number of basic operations required for each array length, 200 permutations of each array length were passed through the algorithm. The number of basic operations it took to find the median for these 200 permutations were averaged, and plotted on a line graph to compare against the best case, worst case, and average case number of operations. Appendix G shows the average of 200 permutations line (blue) conforming similarly to the theoretical average case number of operations line (yellow).

4.3 Average-Case Execution Time

This implementation uses the <chrono> library to test the execution time of the brute force median algorithm. This library allows for time measurements to the millisecond, and is therefore able to get an accurate measure of execution time. To get the average of the execution time of this algorithm, the implementation of the algorithm in Appendix D was used, except the basicOperationCounter was commented out, so as not to interfere with the true execution time of the algorithm. 200 counts of execution time were measured for each array length. Appendix H shows the results of these 200 averages for each array length tested, and demonstrates the exponential rate of growth in execution time as the array length n increases.

References

BHawk. (2015). *Writing .csv files from C++.* Retrieved on 7th April 2017 from http://stackoverflow.com/questions/25201131/writing-csv-files-from-c

Maverik. (2012). *Pointer and vectors in C++.* Retrieved on 7th April 2017 from http://stackoverflow.com/questions/9216619/pointer-and-vectors-in-c

StackOverflow. (n.d). *Stack Overflow*. Retrieved on 7th April 2017 from http://stackoverflow.com/.

Stephan T. Lavavej. (2013). rand() Considered Harmful. Retrieved on 15th April 2017 from https://channel9.msdn.com/Events/GoingNative/2013/rand-Considered-Harmful

Tang, M. (2017). *CAB301 Algorithms and Complexity: Lecture 2 [Lecture Notes]*. Retrieved from https://blackboard.qut.edu.au/webapps/blackboard/content/listContent.jsp?course_id=_132422_1&content_id=_6685064_1

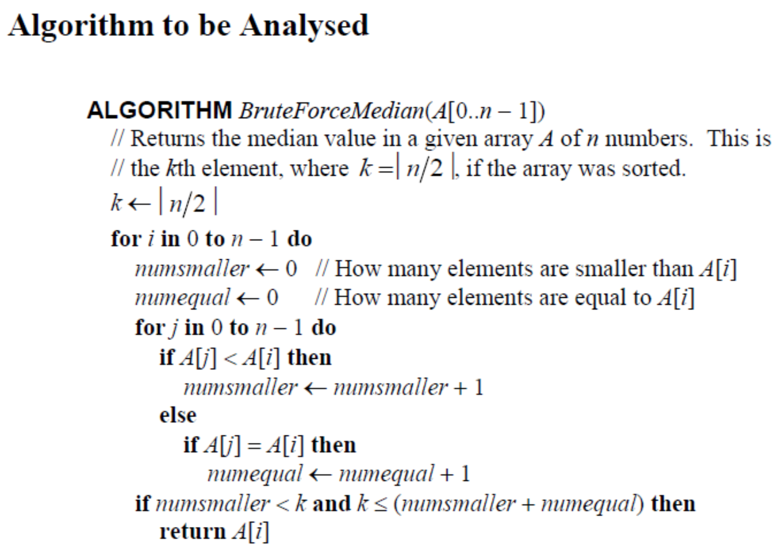
Tunjic, Marko. (2014). *Fill a vector with random numbers c++.* Retrieved on 7th April 2017 from http://stackoverflow.com/questions/21516575/fill-a-vector-with-random-numbers-c

user3762106. (2015). *Easily measure elapsed time.* Retrieved on 7th April 2017 from http://stackoverflow.com/questions/2808398/easily-measure-elapsed-time

APPENDICES

APPENDIX A

The specification for Brute Force Median analysed and implemented in this report (Tang, 2017). Sorted and unsorted arrays are acceptable, empty arrays are assumed to not be entered into the algorithm.



Algorithm to be Analysed

```
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 ← 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

Two assumptions are made about the inputs to this array. The median is equally likely to appear in any position in the array, and the probability that the median is in the array is always 1 (guaranteed).

$$C_{avg}(n) = 1n \cdot \frac{p}{n} + 2n \cdot \frac{p}{n} + 3n \cdot \frac{p}{n} + \cdots + n^2 \cdot \frac{p}{n}$$

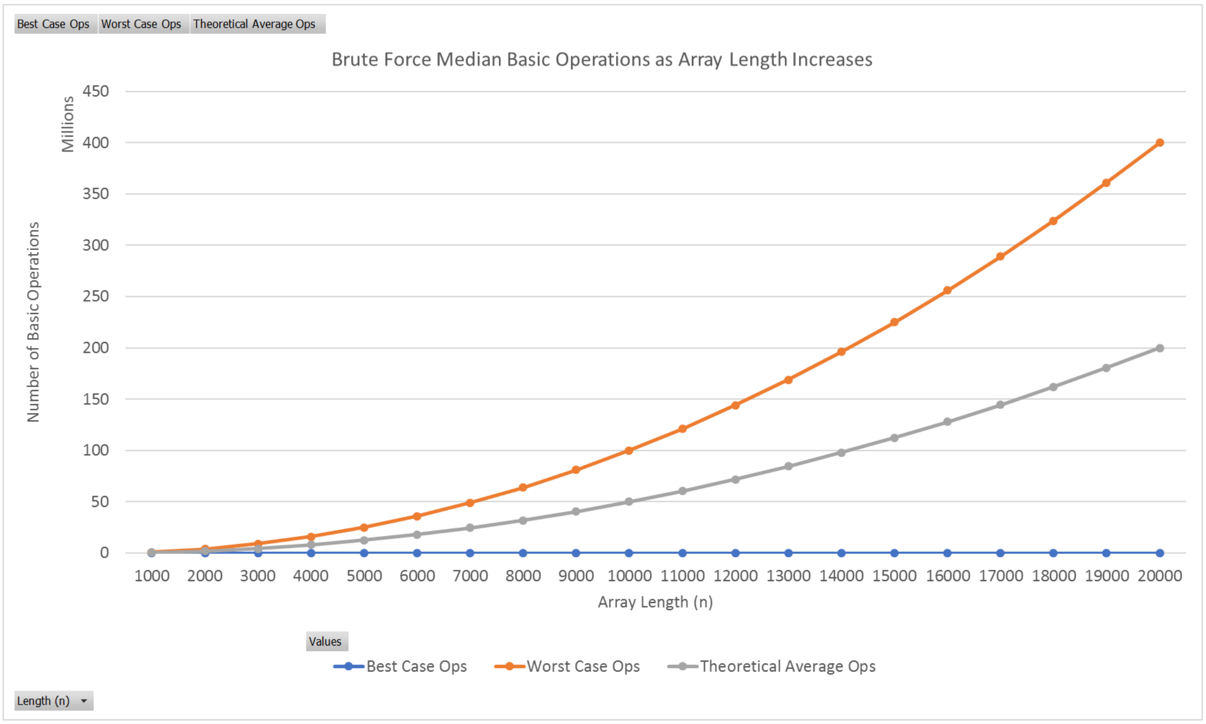
$$= n \cdot \frac{p}{n}(1 + 2 + 3 + \cdots + n)$$

$$= n \cdot \frac{p}{n}\left(\frac{n(n+1)}{2}\right)$$

$$= p\left(\frac{n^2 + n}{2}\right)$$

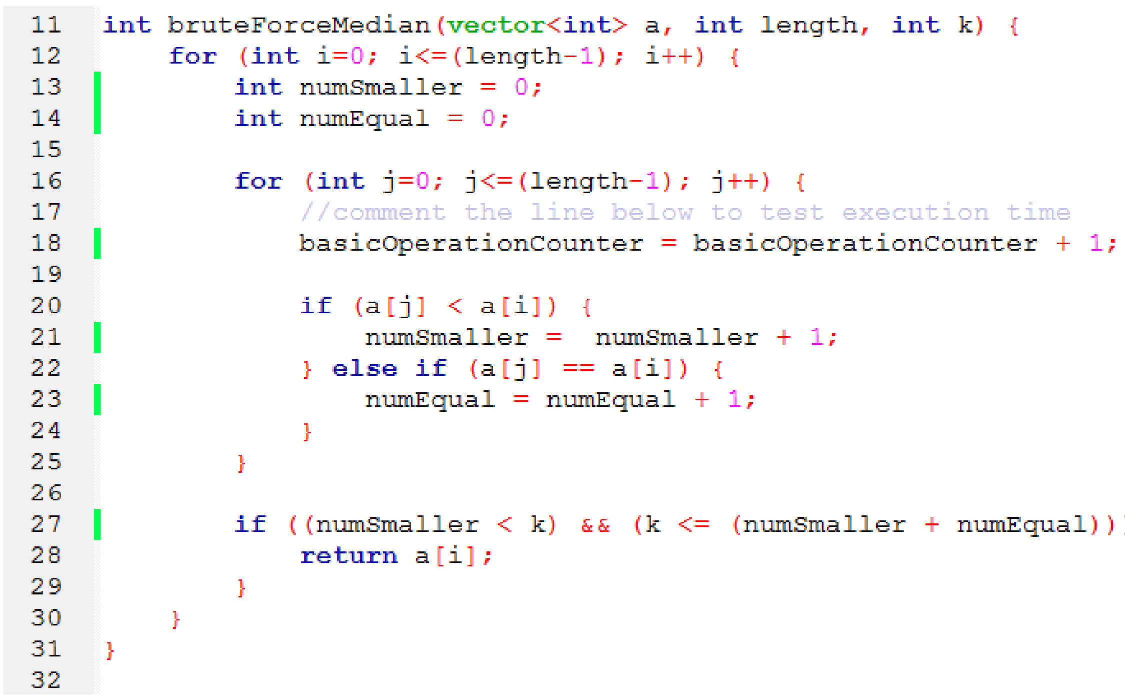
APPENDIX C

The best, worst, and average case number of basic operations for each array length after passing through the brute force median algorithm. These data points are constructed using the average efficiency equation in APPENDIX B.



APPENDIX D

C++ implementation of the brute force median algorithm outlined in APPENDIX A.



```cpp
int bruteForceMedian(vector<int> a, int length, int k) {
    for (int i=0; i<=(length-1); i++) {
        int numSmaller = 0;
        int numEqual = 0;

        for (int j=0; j<=(length-1); j++) {
            //comment the line below to test execution time
            basicOperationCounter = basicOperationCounter + 1;

            if (a[j] < a[i]) {
                numSmaller =  numSmaller + 1;
            } else if (a[j] == a[i]) {
                numEqual = numEqual + 1;
            }
        }

        if ((numSmaller < k) && (k <= (numSmaller + numEqual))) {
            return a[i];
        }
    }
}

```

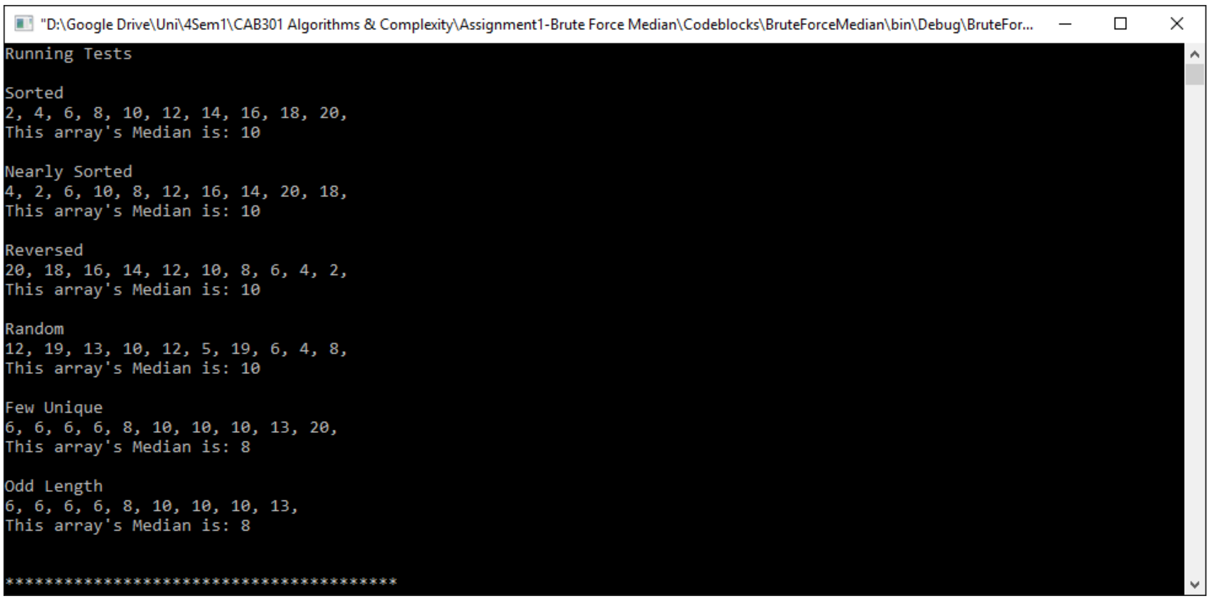
Appendix E

Function used to test the correctness of the implementation in Appendix D.

```cpp
void testImplementationValidity() {
    cout << "Running Tests" << endl << endl;
    int length = 10;
    int medianLoc = ceil(length/2.0);

    vector<int> a = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
    vector<int> b = { 4, 2, 6, 10, 8, 12, 16, 14, 20, 18 };
    vector<int> c = { 20, 18, 16, 14, 12, 10, 8, 6, 4, 2 };
    vector<int> d = { 12, 19, 13, 10, 12, 5, 19, 6, 4, 8 };
    vector<int> e = { 6, 6, 6, 6, 8, 10, 10, 10, 13, 20 };
    vector<int> f = { 6, 6, 6, 6, 8, 10, 10, 10, 13 };

    cout << "Sorted" << endl;
    printArray(a, length, bruteForceMedian(a, length, medianLoc));
    cout << "Nearly Sorted" << endl;
    printArray(b, length, bruteForceMedian(b, length, medianLoc));
    cout << "Reversed" << endl;
    printArray(c, length, bruteForceMedian(c, length, medianLoc));
    cout << "Random" << endl;
    printArray(d, length, bruteForceMedian(d, length, medianLoc));
    cout << "Few Unique" << endl;
    printArray(e, length, bruteForceMedian(e, length, medianLoc));
    cout << "Odd Length" << endl;
    printArray(f, 9, bruteForceMedian(f, 9, medianLoc));
}

```

Appendix F

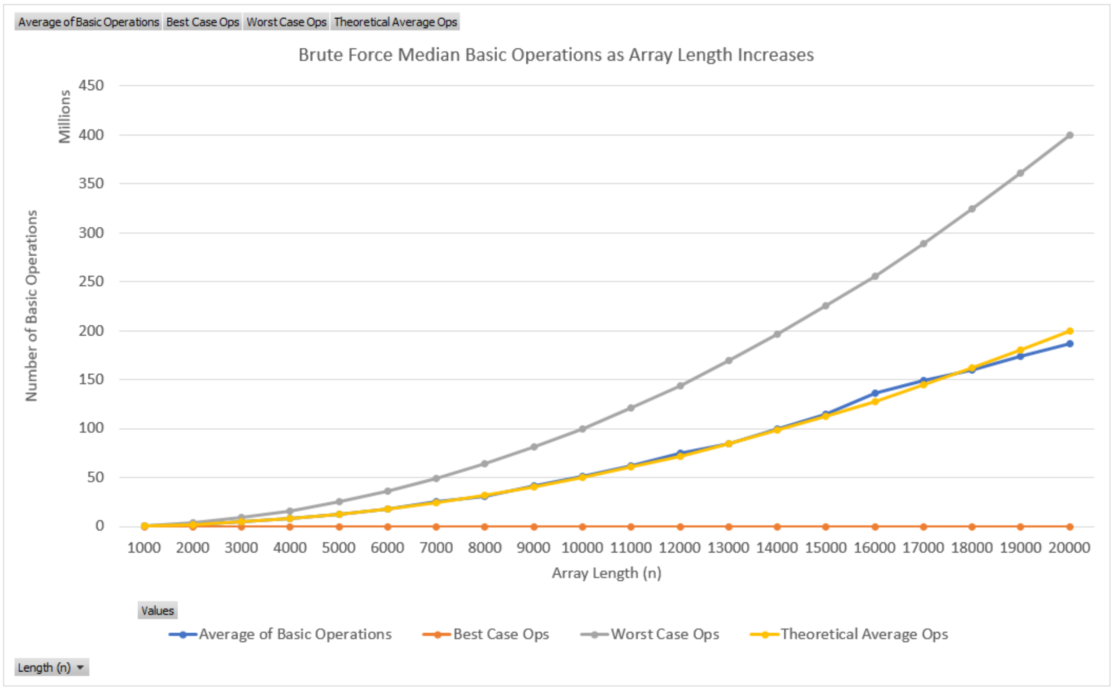
The console output of the function described in Appendix E.


"D:\Google Drive\Uni\4Sem1\CAB301 Algorithms & Complexity\Assignment1-Brute Force Median\Codeblocks\BruteForceMedian\bin\Debug\BruteFor...

```
Running Tests

Sorted
2, 4, 6, 8, 10, 12, 14, 16, 18, 20,
This array's Median is: 10

Nearly Sorted
4, 2, 6, 10, 8, 12, 16, 14, 20, 18,
This array's Median is: 10

Reversed
20, 18, 16, 14, 12, 10, 8, 6, 4, 2,
This array's Median is: 10

Random
12, 19, 13, 10, 12, 5, 19, 6, 4, 8,
This array's Median is: 10

Few Unique
6, 6, 6, 6, 8, 10, 10, 10, 13, 20,
This array's Median is: 8

Odd Length
6, 6, 6, 6, 8, 10, 10, 10, 13,
This array's Median is: 8


****************************************
```

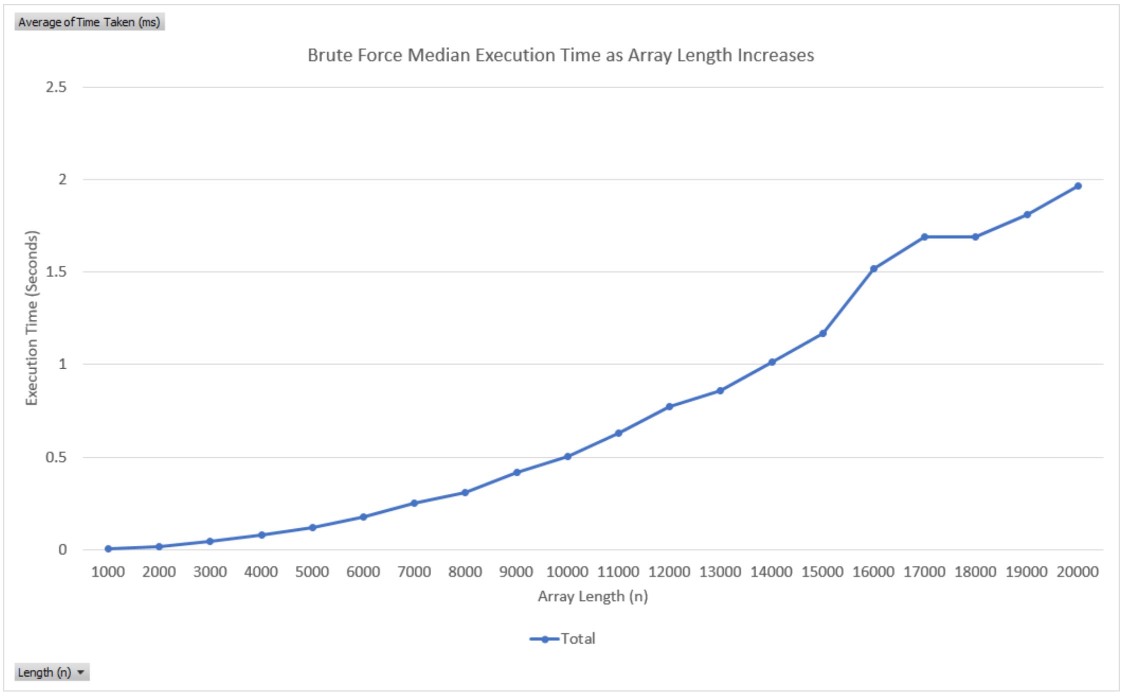
APPENDIX G

The tested average number of basic operations for each array length, compared to the best, worst, and average case number of basic operations after passing through the brute force median algorithm. Average based on 200 data points for each array length.



APPENDIX H

The tested average case execution time for each array length after passing through the brute force median algorithm. Average based on 200 data points for each array length.


Average of Time Taken (ms)
Brute Force Median Execution Time as Array Length Increases
Execution Time (Seconds)
Array Length (n)
Total
Length (n)