Queensland University Of Technology

# Empirical Analysis of an Algorithm for finding the median value in a given data set.

Daniel Huffer
n9493671
April 12, 2017

## Summary

This report details the process involved with experimentally determining the average case efficiency of an algorithm designed to determine the median value within an array of integers. The algorithm is coded and implemented in C++ where the count of basic operations performed by the program alongside it's execution time are both experimentally documented. The measured execution time is then used to investigate additional overheads or possible optimisations not accounted for in the prior theoretical analysis.

# Contents

**Contents** 2

**1 Description of the Algorithm** 3

**2 Theoretical Analysis of the Algorithm** 4
- 2.1 Identifying the Algorithms Basic Operation . . . . 4
- 2.2 Best Case Efficiency . . . . 4
- 2.3 Worst Case Efficiency . . . . 4
- 2.4 Average Case Efficiency . . . . 5

**3 Tools, Methodology and Techniques** 6
- 3.1 Tools . . . . 6
  - 3.1.1 Programs . . . . 6
  - 3.1.2 Environment . . . . 6
- 3.2 Methodology and Techniques of Test Experiments . . . . 7

**4 Code Implementation** 7
- 4.1 Testing Code Implementation . . . . 7

**5 Basic Operation Counts** 8
- 5.1 Experiment One . . . . 8
- 5.2 Experiment Two . . . . 9
- 5.3 Experiment Three . . . . 11

**6 Execution Times** 12
- 6.1 Test One . . . . 12
- 6.2 Test Two . . . . 12

**7 Results and Conclusions** 13

**8 Appendix A Counting the Basic Operations** 14

**9 Appendix B Code for Experiment One** 15

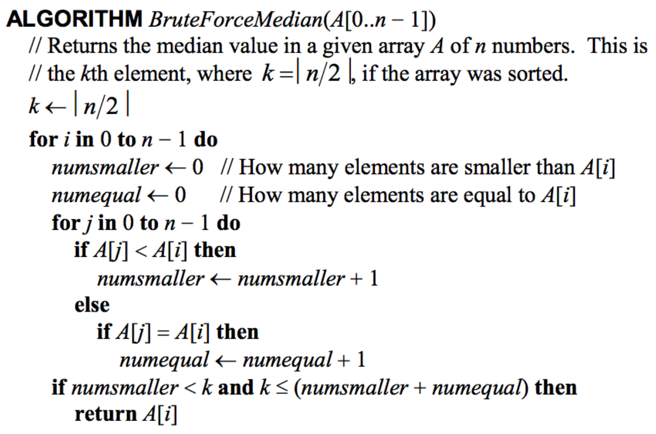
**10 Appendix C Plotting the Theoretical Average Efficiency** 16

**11 Appendix D Code for Experiment Two** 17

**12 Appendix E Code for Execution Test One** 18

**13 References** 19

## 1 Description of the Algorithm



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

Figure 1.1: Pseudocode

The algorithm under inspection in Figure 1.1 takes one given argument, an array ( A[0..n-1] ) of type integer with the objective of returning the median value. The median is deemed to be element "k" where "k" is the arrays length divided by two and rounded up ( $k \Leftarrow \lceil \frac{n}{2} \rceil$ ). Throughout this report the length of arrays will be referred to as "n"; To find "k" a brute force approach is implemented through the use of a nested loop setup. In the outer loop which iterates over the length of the array there are two local integer variables "numsmaller" and "numequal" which are both assigned to zero each iteration of the outer loop. The purpose of these two local variables becomes evident as we enter the inner loop also iterating the length of the array. Inside this nested loop is where a vast majority of the computational time occurs due to two Boolean expressions that are evaluated. The first of the two is an if statement checking if ( A[ j ] $<$ A[ i ] ) where j is the inner loops variable literator and i the outer loops variable literator. If this statement is true then the previously declared local variable numsmaller is incremented. If the expression is false then the second Boolean statement, an if else block will run checking (A[ j ] == A[ i ] ) in which case if true numequal is incremented. Once the inner loop's literator is no longer less than n-1 the loop breaks in which another Boolean expression is checked. If ( numsmaller $<$ k and k $\leq$ (numsmaller + numequal) then Return A[ i ] If this expression evaluates to true then the median value of the array has been found and the function returns the value. If a false Boolean value is returned the outer loop will iterate to the next value where the entire process described will begin again.

# 2 Theoretical Analysis of the Algorithm

This section covers the theoretical average running times and expected time efficiency of the algorithm.

## 2.1 Identifying the Algorithms Basic Operation

The basic operation of any algorithm is defined to be the operation that is computationally the most expensive resulting in a majority of the algorithms execution time. In this case the basic operation is defined as the Boolean expressions within the nested loop, "if ( A[ j ] < A[ i ] )" and "else if ( A[ j ] == A[ i ] )". These two expressions are the most evaluated pieces of code within the algorithm. "if ( A[ j ] < A[ i ] )" is evaluated every single iteration within the nested loop and similarly "else if ( A[ j ] == A[ i ] )" is called every time the previous if statement returns false.

## 2.2 Best Case Efficiency

The best case scenario for this algorithm is when the median value is the first element of the array, this will result in the outer loop running only a single iteration and the inner loop running a total of n iterations. Following the exiting of the inner loop the Boolean expression " If ( numsmaller < k and k ≤ (numsmaller + numequal) then Return A[ i ] " will evaluate true and the arrays median value is returned. A mathematical function to model the amount of basic operations for any input value n in a best case scenario is:

$$C_{best(n)} = \sum_{j=0}^{n-1} 2 - \left\lceil \frac{n}{2} \right\rceil + 1$$

$$= 2n - \left\lceil \frac{n}{2} \right\rceil + 1$$

$$\in \Omega n$$

This formula is generated from the fact that when the median is the first value in the array then $\frac{n}{2} - 1$ will be single counts of basic operations in even arrays. Similarly $\lceil \frac{n}{2} \rceil - 1$ for odd arrays; these are the total count of instances where only the first Boolean expression is evaluated. Therefore when $\lceil \frac{n}{2} \rceil$ is deducted an additional constant one must added back in the equation.

## 2.3 Worst Case Efficiency

The worst case running time of this algorithm is when the median value is situated as the last element of the array. The outter loop must then perform n iterations, resulting in $n^2$ iterations of the nested loop before a result is returned from the function. A mathmatical equation to model the amount of basic operations for any input value n in the worst case scenario is:

$$C_{worst_{(n)}} = (\sum_{i=0}^{n-1} 1)(\sum_{j=0}^{n-1} 2) - (\sum_{c=1}^{n-1} c)$$

$$= 2n^2 - (\frac{n^2}{2} - \frac{n}{2})$$

$$= \frac{3n^2}{2} + \frac{n}{2}$$

$$\in On^2$$

To come up with an equation for the worst case scenario it is easier to work backward as such, first lets assume it was possible that both Boolean expressions could be evaluated every iteration. The basic operation count would then be equivalent to the two summations $(\sum_{i=0}^{n-1} 1)(\sum_{j=0}^{n-1} 2)$ or $\equiv$2n$^2$. All that needs to be done now is take back all the counts where there was only actually one basic operation performed. This becomes easy as notably there is a trend with the basic operation count for example, an array of four integers the worst case basic operation count will consist of the following counts of operations in a varying order:
2,2,2,2
1,2,2,2
1,1,2,2
1,1,1,2
Following the trend the actual count can be be found by subtracting a summation $\sum_{c=1}^{n-1} c$; This logic applies to all input sizes's (n).

## 2.4 Average Case Efficiency

For some applications especially when we wish to aggregate the cost of running an algorithm many times with varying inputs, worst and best cases are not often an accurate representation of an algorithms performance. It's much nicer to have an average case efficiency in order to understand the *typical* behavior of the algorithm. Unfortunately though acquiring the average efficiency is most of the time increasingly more difficult than determining the best and worst cases [1] and in some instances not always possible at all [2]. If we are to work out an average case efficiency we first have to understand how the actual inputs and the costs associated are distributed with respect to a set of all possible inputs. Let's begin with considering a data set with strictly unique values, the average efficiency can be calculated as a probability distribution of the median value's location within the array. To elaborate the probability of the median been in any value of the array is $\frac{1}{n}$ for each element within the array. Therefore we can take the mean average of the best and worst case efficiency to obtain an average efficiency model. This is calculated as a summation from $C_{best}$ to $C_{worst}$ then division by ($C_{worst}$ - $C_{best}$), The complete mathematical formula can be seen below.

$$C_{avg_{(n)}} \approx \sum_{c=2n-\lceil \frac{n}{2} \rceil+1}^{\frac{3n^2}{2}+\frac{n}{2}} c/((\frac{3n^2}{2}+\frac{n}{2})-(2n-\left\lceil \frac{n}{2} \right\rceil+1))$$

$$\in \theta n^2$$

## 3 Tools, Methodology and Techniques

This section details the computing environment and programs used for all experiments conducted.

### 3.1 Tools

#### 3.1.1 Programs

- CLion
  - A powerful cross platform C and C++ IDE
- Grapher
  - A native app bundled with the macOS that is able to create 2D and 3D graphs
- MacTeX
  - A type setting environment which is a redistribution of TeX Live

#### 3.1.2 Environment

- All experiments were conducted on the following machine:
  - Model Name: MacBook Pro
  - Model Identifier: MacBookPro13,1
  - Operating System: macOS Sierra (version 10.12)
  - Processor Name: Intel Core i5
  - Processor Speed: 2 GHz
  - Number of Processors: 1
  - Total Number of Cores: 2
  - L2 Cache (per Core): 256 KB
  - L3 Cache: 4 MB
  - Memory: 8 GB

### 3.2 Methodology and Techniques of Test Experiments

To understand the true average efficiency of the algorithm and how it compares to the theoretical analysis three in-depth experiments are conducted. The experiments are designed to show in deep detail the behavior of the algorithm under a range of varying circumstances. To do this random data sets of varying sizes are produced and tested multiple times to determine average results.

## 4 Code Implementation

The implementation of the pseudocode into C++ is basically identical, one minor discrepancy is the division of $\frac{n}{2}$. The C++ ceil() function must be imported from the math library and used on the fraction due to the fact that C++ automatically floors integer division. Additionally "n" was added as an input parameter as apposed to having it declared in the global scope.

Listing 1: Algorithm Implementation

```

int BruteForceMedian(int A[], int n){
    // Returns the median value in a given array A of n numbers. This is
    // the kth element, where k = |n/2|
    int k = ceil(n/2.0);
    for(int i = 0; i < n; i++){
        int numsmaller = 0; // How many elements are smaller than A[i]
        int numequal = 0; // How many elements are equal to A[i]
        for(int j = 0; j < n; j++){
            if(A[j] < A[i]){
                numsmaller++;
            }
            else if(A[j] == A[i]){
                numequal++;
            }
        }
        if((numsmaller < k) && (k<= (numsmaller + numequal))){
            return A[i];
        }
    }
    return 0;
}
```

### 4.1 Testing Code Implementation

To confirm the coded implementation of the pseudocode was working as intended, a few varying sized arrays were populated with integers where the median values were output to the console and compared against expected values.

Listing 2: Testing implementation

```
 if((numsmaller < k) && (k<= (numsmaller + numequal))){
    std::cout << A[i];
    return A[i];
}
```

## 5 Basic Operation Counts

This section documents the findings and results of three experiments involving basic operation counts. To count the basic operations during the experiments a counter variable was added into the algorithm. Every iteration of the inner loop the counter is incremented by two, then on occurrences where the first boolean expression A[j] < A[i] equates to true, the counter is decremented by one to account for the fact the following if else statement was not executed. The coded implementation can be seen in Appendix A - Counting the Basic Operations.

### 5.1 Experiment One

Once the algorithm had been modified to track basic operations a few varying sized inputs where permuted where each instance was ran through the algorithm. The count of basic operations were divided by n! and output to the console to display the average count value. The results produced for various inputs can be seen below, and the coded implementation can be seen in Appendix B - Code for Experiment One.

- Average count of operations for array permute:
  - Array size: 6 = 33
  - Array size: 7 = 44
  - Array size: 8 = 57
  - Array size: 9 = 70

Using a coded version of the average efficiency function in section 2.4 we can generate the exact same outputs.

Listing 3: Average Efficiency

```
int main() {
    double nMax = 10, sum = 0;
    for(double n = 2; n <= nMax; n+=1){
        sum = 0;
        for(double i = (2*n - ceil(n/2) + 1); i <=
                (((3*n*n)/2) + (n/2)); i++){
            sum += i;
        }
        double result = sum / ((((3*(n*n))/2) + n/2) - (2*n - ceil(n/2) +
            1));
        std::cout << result << "\n";
    }
    return 0;
}
```

As an example in the code sample above, an input value n of 9 will return 70, 8 will return 57, 7 will return 44 etc. This proves that for any input n as long as your data set is strictly unique values you can predict the average cost exactly.

## 5.2 Experiment Two

Testing for experiment two involved testing array sizes 50 to 3000 in increments of 50. Each varying size was tested 500 times, where for each test the array was re populated with pseudorandom generated numbers from 0 to 10,000. Basic operation counts for the 500 tests were totalled into a global variable, upon reaching 500 tests the total was averaged and recorded into a text file on the local disk. The global variable was then re-assigned back to zero to repeat the same procedure with the incremented array size. Results of the experiment showed that the average basic operation count varied slightly from the theoretical average efficiency. The trend visible is that as the data set accumulates higher probability for duplicate values due to size increasing, the accuracy of the theoretical average reduces. This is due to the fact that the first instance of the median value will result in the function returning. Basic operation counts for the theoretical average at array size 3000 were 6,753,000, while the actual average basic operation count was 6,099,042. Despite the discrepancy in the counts, it can be seen in Figure 5.1 that the theoretical average calculation still remains a good approximation for estimating the average efficiency of the algorithm with a data set that has low probability for duplicate values.

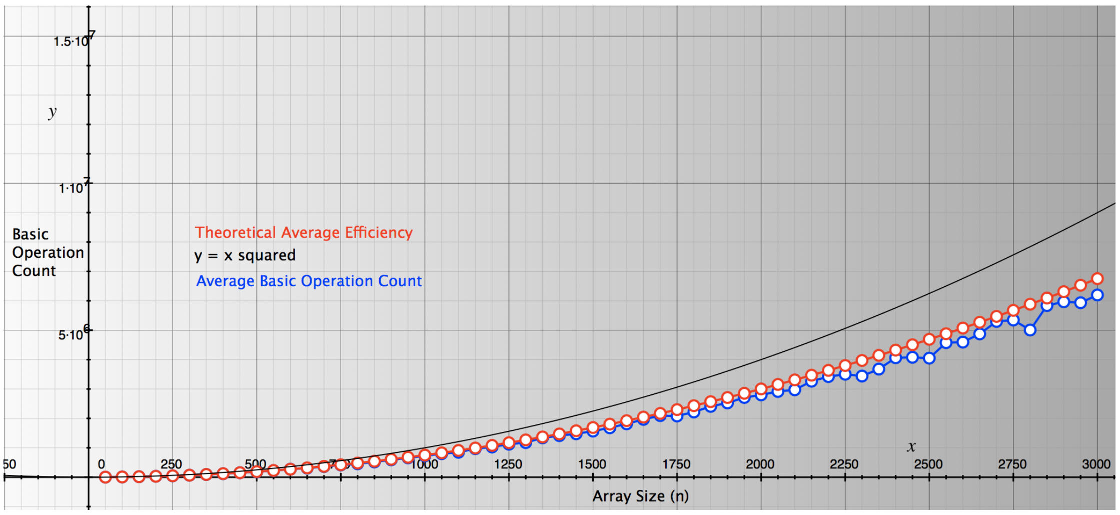


Figure 5.1: Experiment Two Results

The theoretical average efficiency plot in Figure 5.1 was generated by using the coded version of the average efficiency calculation outlined in section 2.4. Details of the coded implementation for the theoretical average efficiency plot can be viewed in Appendix C - Plotting the Theoretical Average Efficiency. Furthermore details for the coded implementation of experiment two can be viewed in Appendix D - Code for Experiment Two.

## 5.3 Experiment Three

Testing for experiment three was conducted exactly the same as experiment two except for a change in the range of randomly generated values. The range of randomly generated numbers was drastically reduced to values between 0 and 1000. The consistency in the two experiments was kept to further investigate the trend identified in experiment two.

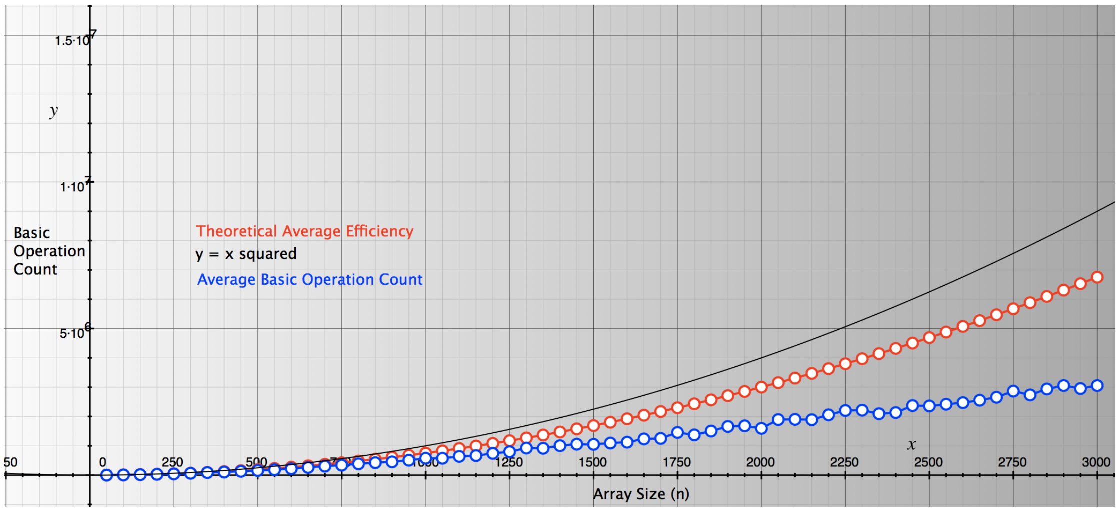


Figure 5.2: Experiment Three Results

Results of experiment three further verify the trend that the algorithm is finding the median value much faster due to the exceedingly higher probability of the median value occuring earlier on within the array. With a data set of 0-1000 the probability of finding the median value is $\frac{1}{1001}$ as zero is included in the randomly generated numbers. This means that for an array size of 3000 the median value statistically should be found within the first 1001 elements. This is why for the max array size of 3000 the average basic operation count was 3,264,760, under half that of the predicted average. Code used for experiment three was the same as experiment two bar the change in randomly generated values.

## 6 Execution Times

To measure the execution time of the algorithm the <time.h> library was imported where the clock function was used. The overall time was divided by clocks per second to get the CPU execution time required. The test implementations were kept the same as the basic operation count experiments two and three for comparison purposes. The BruteForceMedian algorithm was stripped back to the raw implementation and all background processes on the computer were closed to ensure proper testing.

### 6.1 Test One

Test one emulates the basic operation experiment two except execution time is analyzed instead of basic operation counts. Results were produced in the same manner where just a few minor adjustments were made to the code. The updated code can be seen below in Appendix E - Code for Execution Test One.

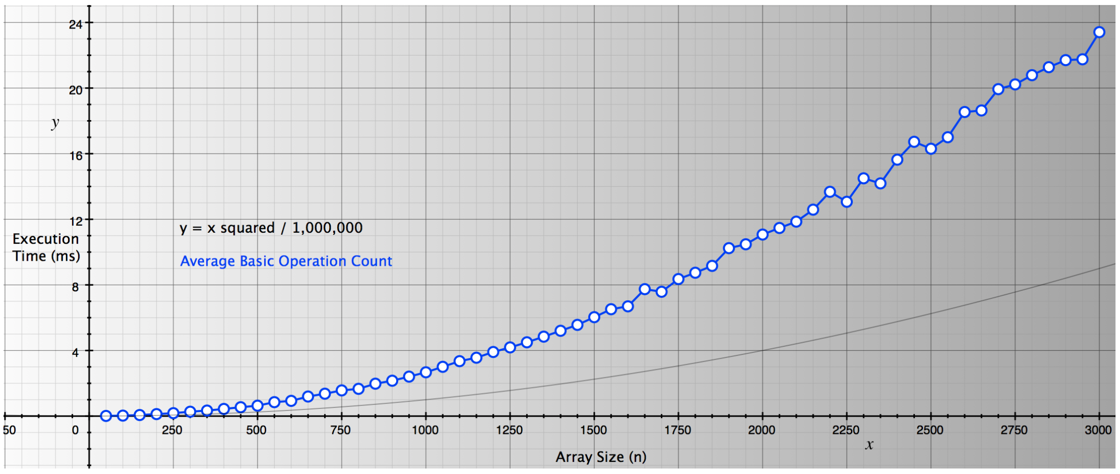


Figure 6.1: Execution Time - Test One

The graph Figure 6.1 represents 60 different array sizes ranging from 50 to 3000 on the x axis and average execution time on the y axis. The equation

$$y = \frac{x^2}{1000000}$$

is plotted as a reference point for the two tests. Looking at the graph it's evident the execution times with the growing array size follow the predicted order of growth seen in the basic operation experiments. The rate of growth follows a constant multiple of $n^2$ where the asymptotic notation $\theta n^2$ holds true.

### 6.2 Test Two

Test two is conducted in the same fashion except the range of random numbers was dropped to 0 to 1,000, replicating the basic operation count experiment three.

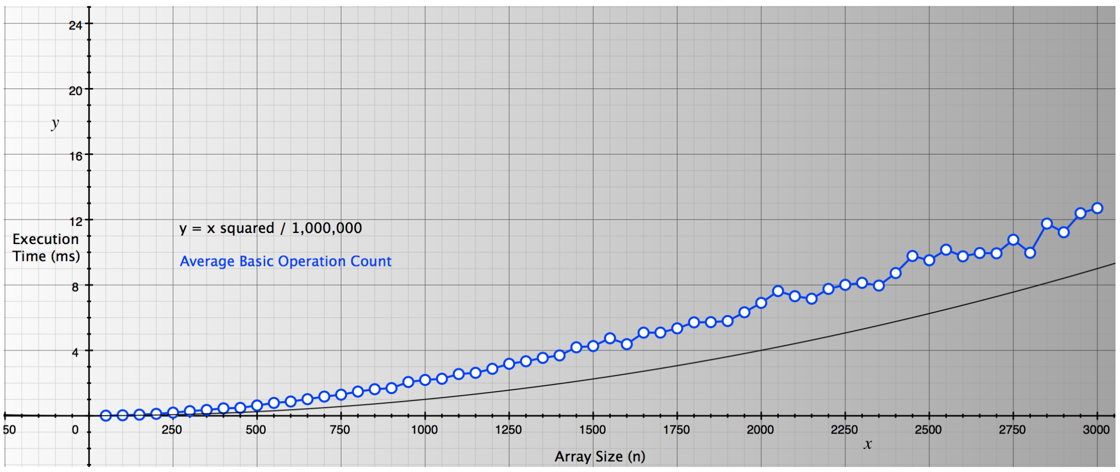


Figure 6.2: Execution Time - Test Two

As was visible with the basic operation counts the average execution time for the arrays with the tighter range of values is roughly half that of the corresponding test using values ranging from 0 to 10,000. This is evidence that the algorithm is following it's expected order of growth but due to the algorithms behavior with duplicate values, the algorithm is running quicker than a predicted average using unique values.

## 7 Results and Conclusions

Due to the similarities in the basic operation theoretical experiment results and the execution time test results; I can conclude that there doesn't appear to be any additional overheads or possible optimisations not accounted for. If there is they are only minor as the algorithm appears to follow the same rate of growth whilst maintaining the same behaviour in both the theoretical experiments and the execution time tests.

# 8 Appendix A Counting the Basic Operations

Listing 4: Basic Operation Counter

```
 1  int BruteForceMedian(int A[], int n){
 2      int k = ceil(n/2.0);
 3      counter = 0;
 4      for(int i = 0; i < n; i++){
 5          int numsmaller = 0, numequal = 0;
 6          for(int j = 0; j < n; j++){
 7              counter+=2;
 8              if(A[j] < A[i]){
 9                  numsmaller++;
10                  counter-=1;
11              }
12              else if(A[j] == A[i]){
13                  numequal++;
14              }
15          }
16          if((numsmaller < k) && (k<= (numsmaller + numequal))){
17              return A[i];
18          }
19      }
20      return 0;
21  }
```

As explained in section 5, the implementation of the counter can be seen in lines 3, 7 and 10.

## 9 Appendix B Code for Experiment One

Listing 5: Calculate Factorial

```
int factorial(int n)
{
    if(n > 1)
        return n * factorial(n - 1);
    else
        return 1;
}
```

Small recursive function to calculate the factorial value of n.

Listing 6: Array permute

```
void swapElement(int x, int y, int A[]){
    int temp = A[x];
    A[x]=A[y];
    A[y]=temp;
}

void permute(int k, int size, int A[]){
    int i;
    if (k == 0){
        BruteForceMedian(A, size);
    }
    else{
        for (i = k-1; i >= 0 ; i--){
            swapElement(i, k-1, A);
            permute(k-1, size, A);
            swapElement(i, k-1, A);
        }
    }
    return;
}
```

Two functions used in conjunction to permute the array, each permutation input into the BruteForceMedian function.

Listing 7: Main

```
int main() {
    int* A;
    for(int i = 0; i < ARRAY_SIZE; i++){
        A[i] = i + 1;
    }
    permute(ARRAY_SIZE, ARRAY_SIZE, A);
    cout << "Average = " << basicOperations / factorial(ARRAY_SIZE);
}
```

Inside the main function call where array size is a constant integer and basic operations is a global variable that was used to sum the total count for the n! calls of BruteForceMedian.

# 10 Appendix C Plotting the Theoretical Average Efficiency

Listing 8: Plotting Theoretical Average

```
int main() {
    ofstream avg ("predictionAverages.txt");
    double nMax = 3000, sum = 0;
    for(double n = 50; n <= nMax; n+=50){
        sum = 0;
        for(double i = (2*n - ceil(n/2) + 1); i <=
                (((3*n*n)/2) + (n/2)); i++){
            sum += i;
        }
        double result = sum / ((((3*(n*n))/2) + n/2) - (2*n - ceil(n/2) +
            1));
        std::cout << result << "\n";
        avg << n << " " << result << "\n";
    }
    avg.close();
    return 0;
}
```

The above code was used to calculate all the data used for plotting the theoretical average against the average basic operation counts of the experiments. The equation below was looped over for the array size's of n = 50 to 3000 in increments of 50. Results were streamed into a text file on the local disk for easy importing into graphing software.

$$C_{avg_{(n)}} \approx \sum_{c=2n-\lceil \frac{n}{2} \rceil+1}^{\frac{3n^2}{2}+\frac{n}{2}} c/((\frac{3n^2}{2}+\frac{n}{2})-(2n-\left\lceil \frac{n}{2} \right\rceil+1))$$

# 11 Appendix D Code for Experiment Two

Listing 9: Main function call

```
 1 // Globals
 2 long avg = 0;
 3 
 4 // Constants
 5 int ARRAY_SIZE = 3000;
 6 int STEP_SIZE = 50;
 7 int NUM_TESTS = 500;
 8 
 9 int main() {
10     std::ofstream average("data.txt"); // Output file stream
11     srand(time(NULL));
12     int count = 0;
13     int A[ARRAY_SIZE]; // Declare array for 3k elements
14     for(int i = STEP_SIZE; i <= ARRAY_SIZE; i+=STEP_SIZE){ // Start array
           size at 50, increments of 50
15         for(int p = 0; p < NUM_TESTS; p++){ // Test each array size 500
               times
16             for(int k = 0; k < i; k++){ // Generate random numbers for
                   every test
17                 A[k] = rand() % 10000; // Random numbers 0-10,000
18             }
19             count++; // Count loop iterations
20             BruteForceMedian(A,i); // Find median algorithm
21             if(count % NUM_TESTS == 0){ // when 500 tests is reached
22                average  << i << " " << avg/NUM_TESTS << "\n"; // Write the
                      array size and avg count to file
23                 avg = 0; // Reset average total
24                 count = 0; // Reset count total
25             }
26         }
27     }
28     average.close(); // Close Stream
29 return 0;
30 }
```

Listing 10: Algorithm Modifications

```
1 
2           if((numsmaller < k) && (k<= (numsmaller + numequal))){
3               avg += counter;
4               return A[i];
5           }
```

Above in Listing 9, Line 3 shows the only modification made to the algorithm. Here avg is a global variable totaling the count of basic operations each test. After the 500 tests have been reached this value is reset, this can be seen in Listing 8, Line 23.

## 12 Appendix E Code for Execution Test One

```
  // Constants
int ARRAY_SIZE = 3000;
int STEP_SIZE = 50;
int NUM_TESTS = 500;

int main() {
    srand(time(NULL));
    int count = 0;
    double av = 0.0;

    int A[ARRAY_SIZE]; // Declare array for 3k elements
    for(int i = STEP_SIZE; i <= ARRAY_SIZE; i+=STEP_SIZE){ // Start array
        size at 50, increments of 50
        for(int p = 0; p < NUM_TESTS; p++){ // Test each array size 500
            times
            for(int k = 0; k < i; k++){ // Generate random numbers for
                every test
                A[k] = rand() % 10000; // Random numbers 0-10000
            }
            count++; // Count loop iterations for the average calc
            clock_t start = clock(); // Start the clock
            BruteForceMedian(A,i); // Find median algorithm
            clock_t end = clock(); // End the clock
            double time = (double) (end-start) / CLOCKS_PER_SEC*1000; //
                Record running time in ms
            av += time;
            if(count % NUM_TESTS == 0){ //when 500 tests is reached
                avg << i << " " << av/NUM_TESTS << "\n"; //write the
                    average to file
                av = 0; //reset average total
            }
        }
    }
    avg.close();
    return 0;
}
```

## 13 References

[1]A. Levitin, *Introduction to the design & analysis of algorithms*, 1st ed. Boston: Pearson Education, 2012, pp. 41-96.

[2]C. Shaffer, *A practical introduction to data structures and algorithm analysis*, 3rd ed. London, Sydney, Toronto: Prentice Hall, 2009.