



---

# EMPIRICAL ANALYSIS OF TWO ALGORITHMS

---

CAB301 ALGORITHMS & COMPLEXITY



MAY 21, 2018  
DOUGLAS BRENNAN  
N7326645

## Summary

The following is an in-depth analysis of two algorithms that both find the median value from a supplied array of integers.

### 1 - Description of the Algorithms

The first algorithm (Figure 1) is named the Brute Force Algorithm. To begin, it assigns value 'k' to be equal to the length of the array divided by 2. Then it uses two for-loops to analyse the array elements against one another. The analysis determines how many of the integers are smaller or equal to the other integer elements in the array. The function later uses these counts to return the median value. This is achieved by assessing two statements in parallel, the first is whether the count of smaller values is less than k, the second is whether k's value is less than or equal to the number of detected smaller values, added to the number of detected equal values. Only when these two statements are true the median value is outputted.

```
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 – Brute Force Median)

The second algorithm (Figure 2) is named Median, it finds the median by incorporating the use of two external algorithms Select (Figure 3) and Partition (Figure 4).

```
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  $Select(A, 0, \lfloor n/2 \rfloor, n-1)$  // NB: The third argument is rounded down
```

(Figure 2 - Median)

The Select algorithm returns an index value from a supplied array if the array was sorted in nondecreasing order. To achieve such an order the Partition algorithm is used.

**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.  
 $pos \leftarrow \text{Partition}(A, l, h)$   
**if**  $pos = m$  **then**  
     **return**  $A[pos]$   
**if**  $pos > m$  **then**  
     **return** *Select*( $A, l, m, pos - 1$ )  
**if**  $pos < m$  **then**  
     **return** *Select*( $A, pos + 1, m, h$ )

(Figure 3 - Select)

Parsed into this algorithm is an array, a partition start index value and a partition end index value. The algorithm assesses each value in the array, values which are smaller than the partition start index value are swapped with those that are larger than the partition end index value. This process is repeated until the array is sorted in nondecreasing order, once completed the algorithm returns the value of the sorted array at the partition start index value. The value is parsed back into the Select algorithm, the algorithm uses the value to return the desired median value back to the Median algorithm.

**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.  
 $pivotval \leftarrow A[l]$  // Choose first value in slice as pivot value  
 $pivotloc \leftarrow l$  // Location to insert pivot value  
**for**  $j$  **in**  $l + 1$  **to**  $h$  **do**  
     **if**  $A[j] < pivotval$  **then**  
          $pivotloc \leftarrow pivotloc + 1$   
          $\text{swap}(A[pivotloc], A[j])$  // Swap elements around pivot  
 $\text{swap}(A[l], A[pivotloc])$  // Put pivot element in place  
**return**  $pivotloc$

(Figure 4 - Partition)

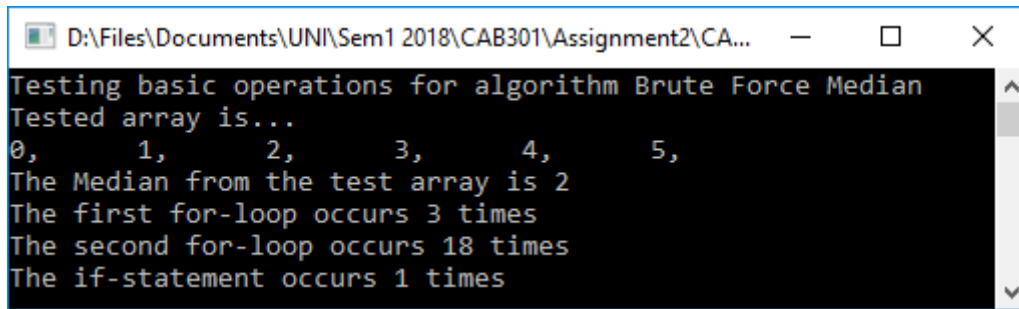
## 2 - Algorithm Analysis

This segment goes into detail about the complexity of the algorithms and their efficiencies in relation to execution times.

### 2a – Basic Operation Identification

According to (Levitin, 2012), a basic operation is the most important task of an algorithm, it is the operation that contributes the most towards the total running time.

For the first algorithm (Figure 1) the basic operation after testing is recognised (Figure 5) as the second for-loop for it occurs more than any other operation in the algorithm.



```
D:\Files\Documents\UNI\Sem1 2018\CAB301\Assignment2\CA...
Testing basic operations for algorithm Brute Force Median
Tested array is...
0, 1, 2, 3, 4, 5,
The Median from the test array is 2
The first for-loop occurs 3 times
The second for-loop occurs 18 times
The if-statement occurs 1 times
```

(Figure 5 – Brute Force Median Basic Operations Test)

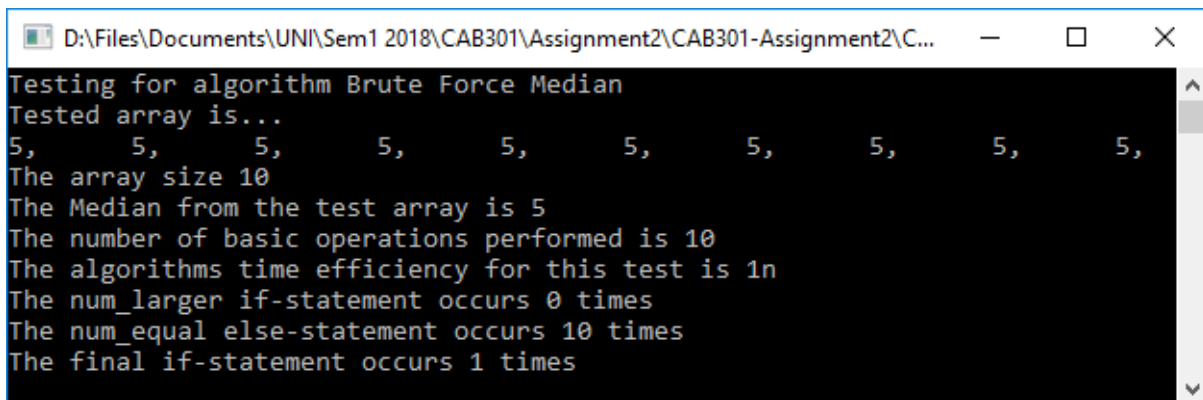
For the second algorithm (Figures 2 –4) the basic operation is recognised as the for loop which occurs in the Partition sub-algorithm for it is the only for-loop which occurs.

## 2c – Algorithms Time Efficiencies – Best & Worst Case

The algorithms time efficiencies ( $C(n)$ ) for both algorithms are affected by the size of the inputs ( $n$ ) and the value matrices. These algorithms can perform different numbers of basic operations at each iteration, so their best-case, worst-case and average-case efficiencies are unique (Paul, 2005).

Due to the input array size having the largest effect on time efficiency, testing of this type has been left out intentionally.

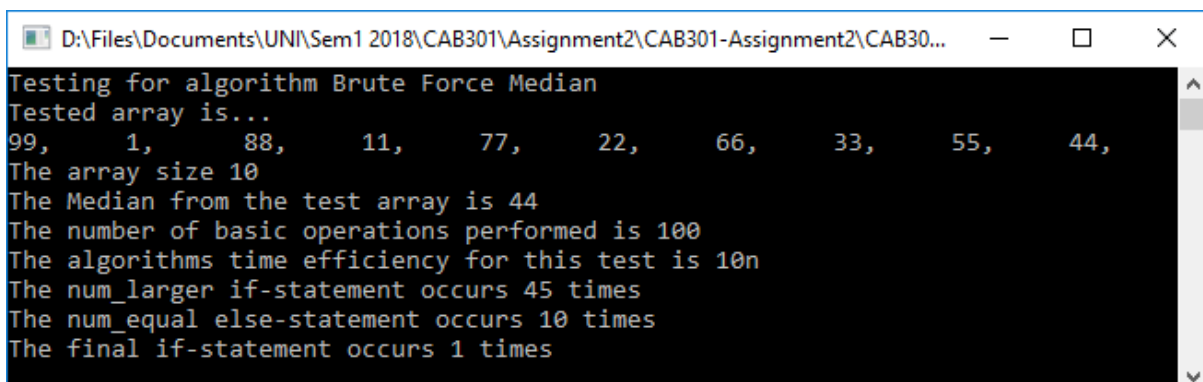
The best-case scenario for the Brute Force Median algorithm (Figure 6) is when each number in the supplied array is the same, the function iterates through each array instance so the best-case efficiency for this algorithm is linear  $\Theta(n)$ .



```
D:\Files\Documents\UNI\Sem1 2018\CAB301\Assignment2\CAB301-Assignment2\C...
Testing for algorithm Brute Force Median
Tested array is...
5, 5, 5, 5, 5, 5, 5, 5, 5,
The array size 10
The Median from the test array is 5
The number of basic operations performed is 10
The algorithms time efficiency for this test is 1n
The num_larger if-statement occurs 0 times
The num_equal else-statement occurs 10 times
The final if-statement occurs 1 times
```

(Figure 6 – Brute Force Median Best-Case)

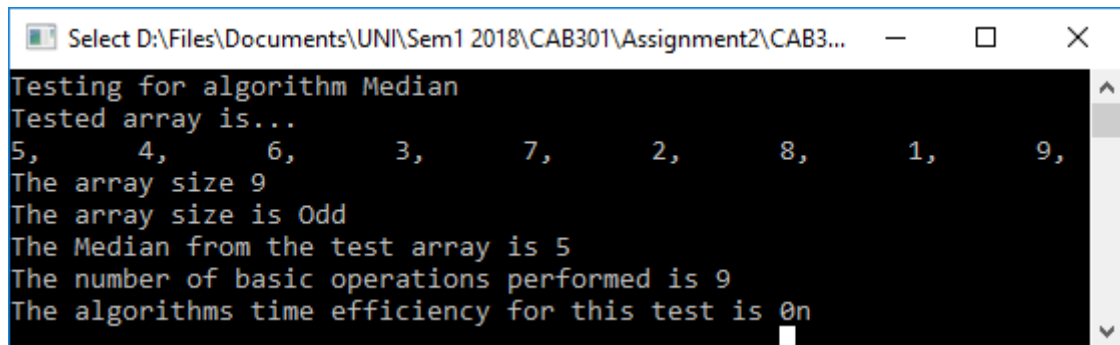
The worst-case scenario for the Brute Force Median algorithm is when each consecutive number's integer gap is largest it can be (Figure 7), causing a quadratic time efficiency of  $\Theta(n^2)$ .



```
D:\Files\Documents\UNI\Sem1 2018\CAB301\Assignment2\CAB301-Assignment2\CAB30...
Testing for algorithm Brute Force Median
Tested array is...
99, 1, 88, 11, 77, 22, 66, 33, 55, 44,
The array size 10
The Median from the test array is 44
The number of basic operations performed is 100
The algorithms time efficiency for this test is 10n
The num_larger if-statement occurs 45 times
The num_equal else-statement occurs 10 times
The final if-statement occurs 1 times
```

(Figure 7 – Brute Force Median Worst-Case)

The Median algorithm's best-case time efficiency is recognised as linear by (Levitin, 2012), "Partitioning an  $n$ -element array always requires  $n - 1$  key comparisons. If it produces the split that solves the selection problem without requiring more iterations, then for this best case we obtain  $C_{best}(n) = n - 1 \in \Theta(n)$ ."



```
Select D:\Files\Documents\UNI\Sem1 2018\CAB301\Assignment2\CAB3...
Testing for algorithm Median
Tested array is...
5, 4, 6, 3, 7, 2, 8, 1, 9,
The array size 9
The array size is Odd
The Median from the test array is 5
The number of basic operations performed is 9
The algorithms time efficiency for this test is 0n
```

(Figure 8 – Median Best Case)

The Median algorithms worst-case time efficiency is recognised as quadratic by (Levitin, 2012), "Unfortunately, the algorithm can produce an extremely unbalanced partition of a given array, with one part being empty and the other containing  $n - 1$  elements. In the worst case, this can happen on each of the  $n - 1$  iterations."

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

(Figure 9 – Median Worst Case)

## 2d – Algorithms Time Efficiencies – Average Case

An algorithms average-case efficiency best characterises the relationship between the number of basic operations performed and the input size (Levitin, 2012).

The results from section 4 show that the Brute Force Median algorithm has varying average-case efficiencies depending on the parsed integer value matrix, the number of basic operations performed vary from being linear as  $\Theta(n)$  to being quadratic  $\Theta(n^2)$ . This results in the equation not having a fixed order of growth.

The results also show that the Median algorithm has a constant linear average case efficiency of  $3n$ , this means the order of growth for this function is linear  $\Theta(n)$ .

Functions classed as  $\Theta(n)$  have linear growth for they are asymptotically bound. According to (University of Nebraska-Lincoln), "As  $n$  increases,  $f(n)$  grows at the same rate as  $g(n)$ . In other words,  $g(n)$  is an asymptotically tight bound on  $f(n)$ ."

## 3 – Methodology, Tools & Techniques

This section summarises the computing environment for data collection and the implementation of the assessed algorithms.

### 3a – Programming Environment

**Environment:** Visual Studio 2017 was used as the environment to conduct the testing and results were exported to Microsoft Excel for data collection. A console application was chosen within Visual Studio, and the script language was C#.

**System:** The tests were performed using an Acer Aspire laptop with the following specifications;

- Operating System: Windows 10 Home Edition - 64-bit
- Random Access Memory: 16gb
- Processor: Intel® Core™ i7-7700HQ CPU @ 2.80GHZ

**Test Data:** To generate test data various techniques were implemented to calculate large variances of results.

- Functional Tests: Arrays containing varying fixed values were tested to prove the algorithms functionalities and time efficiencies
- Ascending Array Test: An array containing 10 ascending integers from 1-10 was tested to show varying time efficiencies
- Descending Array Test: An array containing 10 descending integers from 10-1 was tested to show varying time efficiencies
- Random Array Test: To properly test the average-case scenarios, arrays of the same length were tested that contain random variables. A random number generator was implemented to ensure the value matrix of the tested arrays were unique.

### 3b – Implementation of the Algorithms

The below images are cropped screenshots of the algorithm implementations. They have been implemented with reference to the original algorithms (Figures 1-4). As mentioned in section 2a the number of basic operations are linked to the for-loops, the counters are named 'basic\_operations', this is the same for both algorithm implementations. Small additions have been made to each algorithm so that data could be successfully captured, this exists below the 'Export to Excel' comments.

```
private static int BruteForceMedian(int[] array) {
    // 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.
    worksheet = (Excel.Worksheet)excel.ActiveSheet;
    int k = array.Length / 2;
    for (int i = 0; i <= array.Length; i++) {
        int numsmaller = 0;
        int numequal = 0;
        for (int j = 0; j <= array.Length-1; j++) {
            basic_operations++;
            if (array[j] < array[i]) {
                numsmaller++;
            } else {
                if (array[j] == array[i]) {
                    numequal++;
                }
            }
        }
        if ((numsmaller < k && k <= (numsmaller + numequal))) {
            time_efficiency = basic_operations / array_size;
            // Export to Excel
            worksheet.Cells[1, 1] = "Basic Operations";
            worksheet.Cells[1, 2] = "Time Efficiency";
            worksheet.Cells[1, 3] = "Array Length";
            worksheet.Cells[data_count, 1] = basic_operations;
            worksheet.Cells[data_count, 2] = time_efficiency;
            worksheet.Cells[data_count, 3] = array_size;
            data_count++;
            return array[i];
        }
    }
    return 0;
}
```

(Figure 10- Brute Force Median)



```

private static int Median(int[] function_array) {
    // Returns the median value in a given array A of n numbers
    int output;
    worksheet = (Excel.Worksheet)excel.ActiveSheet;
    if (function_array.Length == 1) {
        return function_array[0];
    } else {
        output = Select(function_array, 0, (function_array.Length / 2), function_array.Length - 1);
        time_efficiency = basic_operations / array_size;
        // Export to Excel
        worksheet.Cells[1, 1] = "Basic Operations";
        worksheet.Cells[1, 2] = "Time Efficiency";
        worksheet.Cells[1, 3] = "Array Length";
        worksheet.Cells[data_count, 1] = basic_operations;
        worksheet.Cells[data_count, 2] = time_efficiency;
        worksheet.Cells[data_count, 3] = array_size;
        data_count++;
        return output;
    }
}

private static int Select(int[] function_array, int l, int m, int h) {
    // Returns the value at index m in array slice A[l..h], if the slice
    // were sorted into nondecreasing order.
    int pos = Partition(function_array, l, h);
    if (pos == m) {
        return function_array[pos];
    } else if (pos > m) {
        return Select(function_array, l, m, pos - 1);
    } else if (pos < m) {
        return Select(function_array, pos + 1, m, h);
    } else {
        return 0;
    }
}

```

(Figure 11 – Median & Select)

```

private static int Partition(int[] function_array, int l, int 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.
    int pivotval = function_array[l];
    int pivotloc = l;
    for (int j = l + 1; j <= h; j++) {
        basic_operations++;
        if (function_array[j] < pivotval) {
            pivotloc++;
            //Swap
            int temp = function_array[pivotloc];
            function_array[pivotloc] = function_array[j];
            function_array[j] = temp;
        }
    }
    //Swap
    int temp_2 = function_array[pivotloc];
    function_array[pivotloc] = function_array[l];
    function_array[l] = temp_2;
    return pivotloc;
}

```

(Figure 12 – Partition)

### 3c – Testing Method Implementations

The below images are cropped screenshots detailing how the tests (outlined in section 3a) were programmed. See section 5 for the implementation of these methods.

```
// Creates array with random values
private static int[] CreateRandomArray(int arraySize) {
    rnd = new Random();
    int[] temp_array = new int[arraySize];
    for (int i = 0; i < temp_array.Length; i++) {
        temp_array[i] = rnd.Next(1, 100);
    }
    return temp_array;
}

// Creates array with ascending values
private static int[] CreateAscendingArray(int arraySize) {
    int[] temp_array = new int[arraySize];
    int counter = 1;
    for (int i = 0; i < temp_array.Length; i++) {
        temp_array[i] = counter++;
    }
    return temp_array;
}

// Creates array with descending values
private static int[] CreateDescendingArray(int arraySize) {
    int[] temp_array = new int[arraySize];
    int counter = arraySize;
    for (int i = 0; i < temp_array.Length; i++) {
        temp_array[i] = counter--;
    }
    return temp_array;
}
```

(Figure 13 – Test Implementations)

```
// Creates array with set values
private static int[] CreateFunctionalTestArray() {
    int[] temp_array = {5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5};
    return temp_array;
}
```

(Figure 14 – Test Implementations Continued)

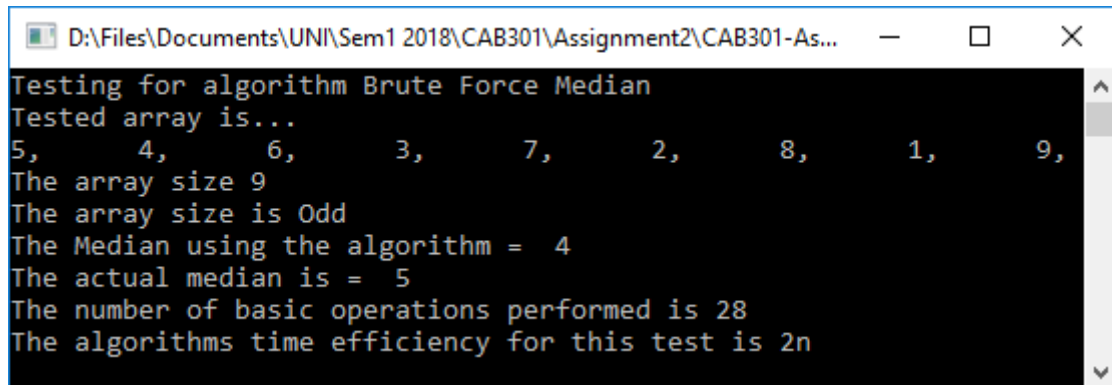


## 4 – Test Results

This section describes the results of the tests environments described in section 3a and their implementations 3c.

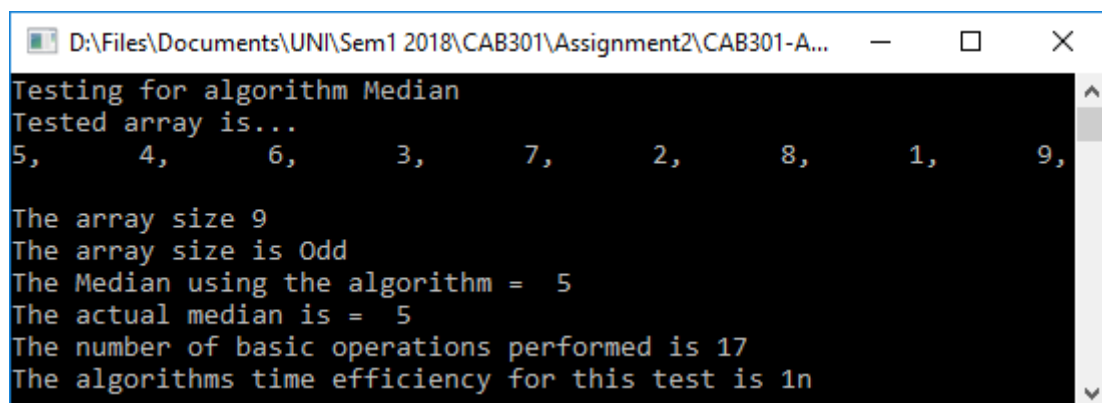
### 4a – Functional Testing

To ensure that the implementations of the algorithms function correctly, an array containing 9 fixed values were parsed that would cause both algorithms processes to trigger successfully. The parsed array and corresponding results are as follows;



```
D:\Files\Documents\UNI\Sem1 2018\CAB301\Assignment2\CAB301-As...
Testing for algorithm Brute Force Median
Tested array is...
5, 4, 6, 3, 7, 2, 8, 1, 9,
The array size 9
The array size is Odd
The Median using the algorithm = 4
The actual median is = 5
The number of basic operations performed is 28
The algorithms time efficiency for this test is 2n
```

(Figure 15 - Functional Test Brute Force Median)

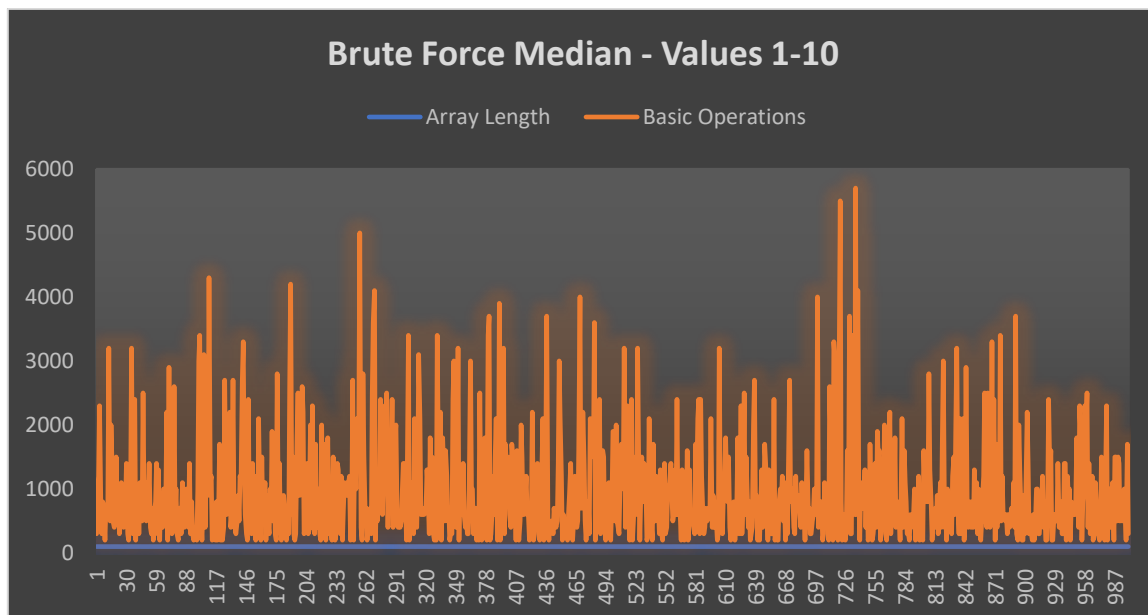


```
D:\Files\Documents\UNI\Sem1 2018\CAB301\Assignment2\CAB301-A...
Testing for algorithm Median
Tested array is...
5, 4, 6, 3, 7, 2, 8, 1, 9,
The array size 9
The array size is Odd
The Median using the algorithm = 5
The actual median is = 5
The number of basic operations performed is 17
The algorithms time efficiency for this test is 1n
```

(Figure 16 - Functional Test Median)

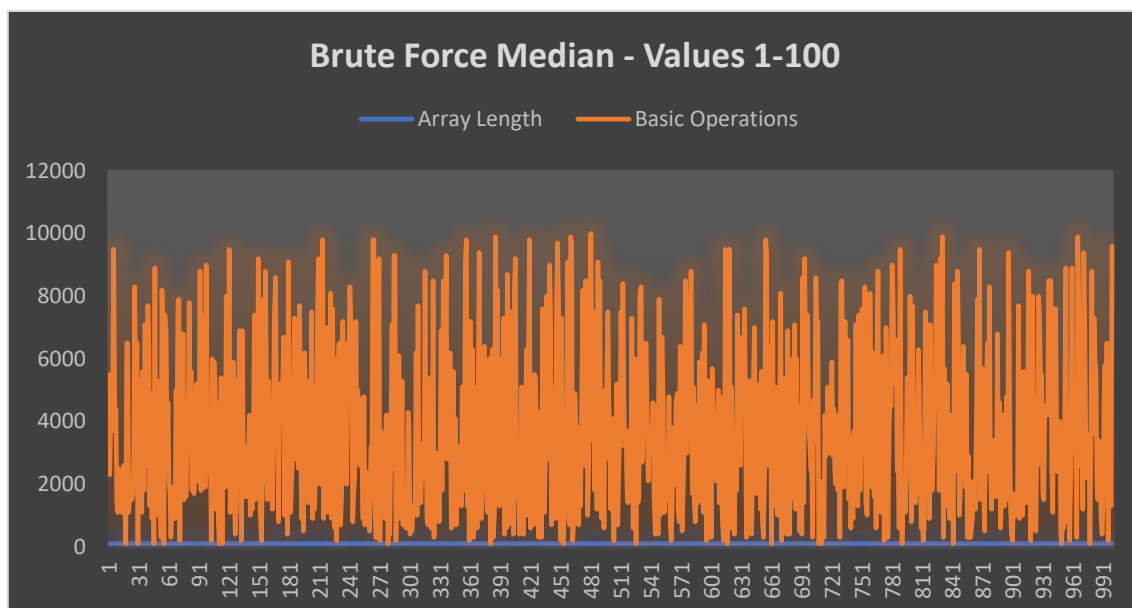
## 4b – Average-Case Scenarios

To calculate the average case efficiency of an algorithm, tests using the same size input are performed that contain random values. To achieve calculative accuracy the test was performed 1000 times for the following integer ranges, 1-10, 1-100 and 1-1000 for arrays of size 100.



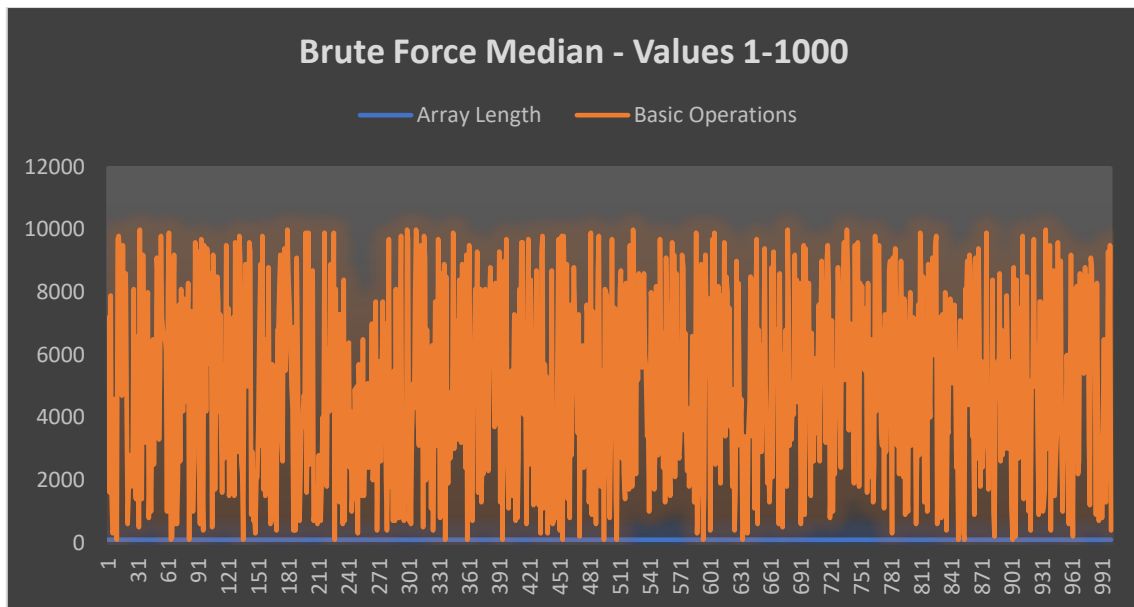
(Figure 17 – Brute Force Median – Values 1-10)

Based off the 1000 tests run with values between 1-10 on arrays of size 100, the average number of basic operations performed was 869.9. Divided by the array size results in a time efficiency of 8.699(n).



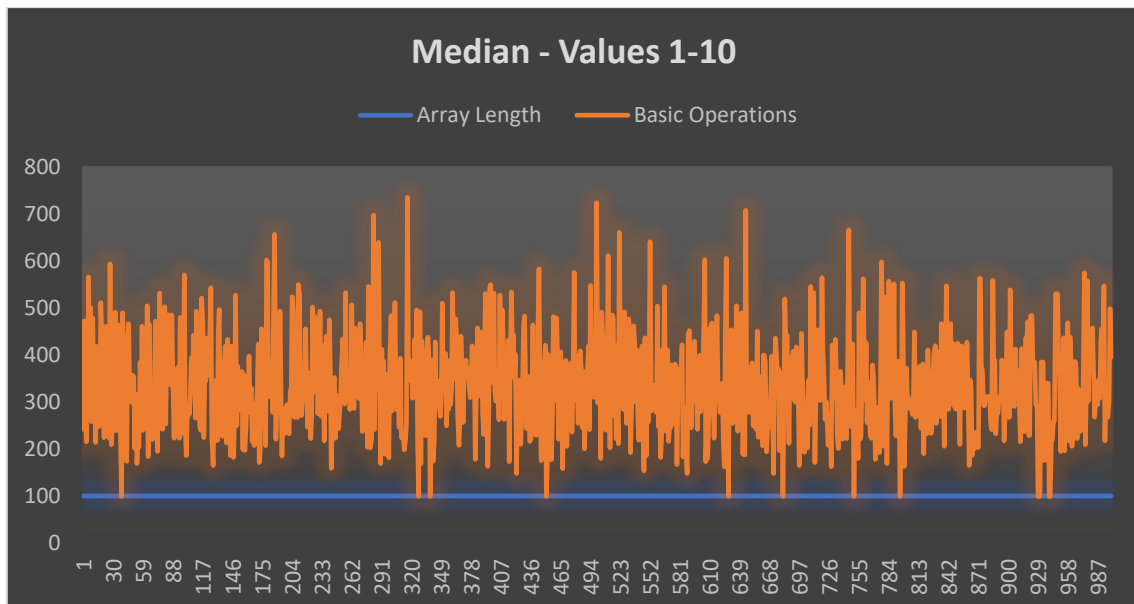
(Figure 18 – Brute Force Median – Value 1-100)

Based off the 1000 tests run with values between 1-100 on arrays of size 100, the average number of basic operations performed was 3,663.3. Divided by the array size results in a time efficiency of 36.633(n).



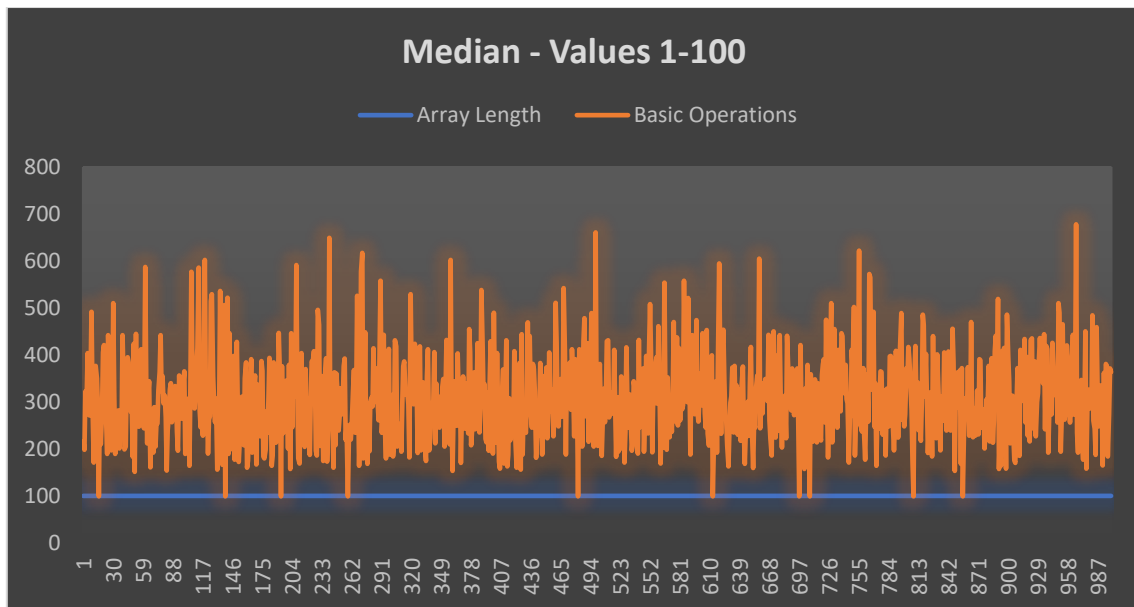
(Figure 19 – Brute Force Median – Value 1-1000)

Based off the 1000 tests run with values between 1-1000 on arrays of size 100, the average number of basic operations performed was 4,992.3. Divided by the array size results in a time efficiency of  $49.923(n)$ .



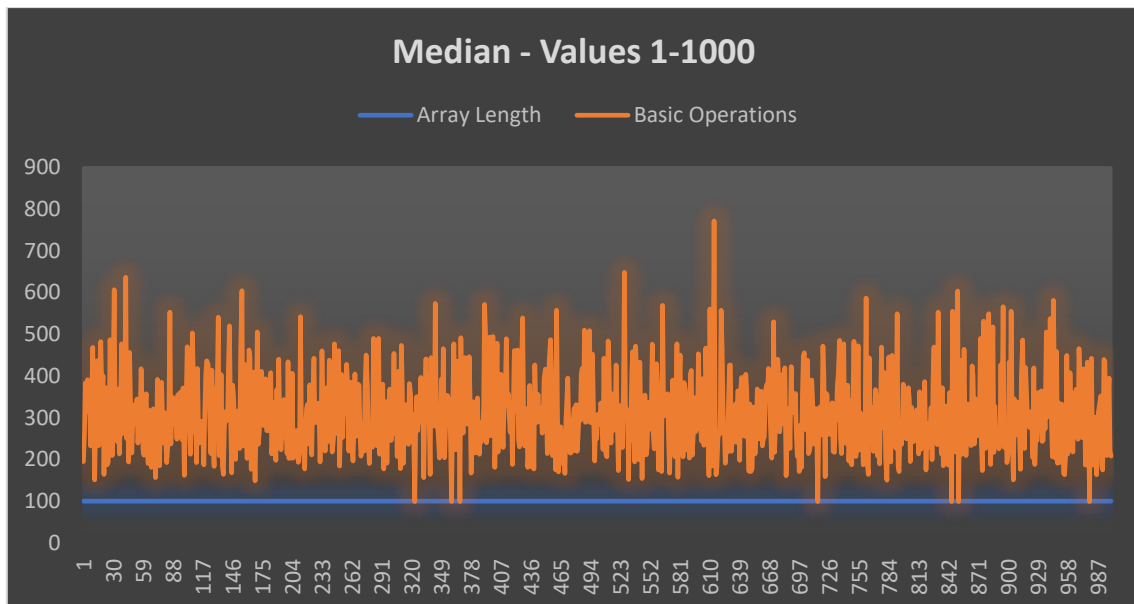
(Figure 20 – Median – Value 1-10)

Based off the 1000 tests run with values between 1-10 on arrays of size 100, the average number of basic operations performed was 335.135. Divided by the array size results in a time efficiency of  $3.351(n)$ .



(Figure 21 – Median – Value 1-100)

Based off the 1000 tests run with values between 1-100 on arrays of size 100, the average number of basic operations performed was 308.009. Divided by the array size results in a time efficiency of  $3.080(n)$ .



(Figure 22 – Median – Value 1-1000)

Based off the 1000 tests run with values between 1-1000 on arrays of size 100, the average number of basic operations performed was 307.334. Divided by the array size results in a time efficiency of  $3.073(n)$ .

## 5 – Results & Conclusion

The conducted test results in section 4 show the mean number of basic operations performed when given random integers of varying integer ranges.

The Brute Force Median algorithms average time efficiency varied greatly depending on the integer range. A range of 1-10 resulted in an efficiency of  $8.699(n)$ , a range of 1-100 resulted in an efficiency of  $36.633(n)$ , and a range of 1-1000 resulted in an efficiency of  $49.923(n)$ . The range of 1-1000 showed multiple occurrences where the basic operation count was quadratic in relation to the input size.

The Median algorithms average time efficiency did not vary like the Brute Force Median did, contrarily this algorithm performed better when supplied with greater integer ranges. A range of 1-10 resulted in an efficiency of  $3.351(n)$ , a range of 1-100 resulted in an efficiency of  $3.080(n)$ , and a range of 1-1000 resulted in an efficiency of  $3.073(n)$ . This evidence of time efficiency stability shows the growth rate of the Median algorithm to be often linear  $\Theta(n)$ .

The test results detailed in this report clearly prove that the Median algorithm outperforms the Brute Force Median algorithm in every result, especially when parsed with arrays containing largely varying value matrices.



## 6 – Further Results and Code Snippets

```
D:\Files\Documents\UNI\Sem1 2018\CAB301\Assignment2\CAB301-Assignment2\CA...
Testing for algorithm Brute Force Median
Tested array is...
1,    2,    3,    4,    5,    6,    7,    8,    9,    10,
The array size 10
The Median from the test array is 5
The number of basic operations performed is 50
The algorithms time efficiency for this test is 5n
The num_larger if-statement occurs 10 times
The num_equal else-statement occurs 5 times
The final if-statement occurs 1 times
```

```
D:\Files\Documents\UNI\Sem1 2018\CAB301\Assignment2\CAB301-Assignment2\C...
Testing for algorithm Brute Force Median
Tested array is...
10,    9,    8,    7,    6,    5,    4,    3,    2,    1,
The array size 10
The Median from the test array is 5
The number of basic operations performed is 60
The algorithms time efficiency for this test is 6n
The num_larger if-statement occurs 39 times
The num_equal else-statement occurs 6 times
The final if-statement occurs 1 times
```

```
D:\Files\Documents\UNI\Sem1 2018\CAB301\Assignment2\CAB301-Assignment2\CA...
Testing for algorithm Brute Force Median
Tested array is...
9,    7,    4,    6,    6,    1,    4,    7,    1,    4,
The array size 10
The Median from the test array is 4
The number of basic operations performed is 30
The algorithms time efficiency for this test is 3n
The num_larger if-statement occurs 18 times
The num_equal else-statement occurs 6 times
The final if-statement occurs 1 times
```

```
D:\Files\Documents\UNI\Sem1 2018\CAB301\Assignment2\CAB301-Assignment2\C...
Testing for algorithm Brute Force Median
Tested array is...
5,    84,    32,    8,    49,    14,    72,    30,    23,    39,
The array size 10
The Median from the test array is 30
The number of basic operations performed is 80
The algorithms time efficiency for this test is 8n
The num_larger if-statement occurs 36 times
The num_equal else-statement occurs 8 times
The final if-statement occurs 1 times
```

```
D:\Files\Documents\UNI\Sem1 2018\CAB301\Assignment2\CAB301-Assignment2\C...
Testing for algorithm Median
Tested array is...
1,    2,    3,    4,    5,    6,    7,    8,    9,    10,
The array size 10
The array size is Even
The Median from the test array is 5.5
The number of basic operations performed is 39
The algorithms time efficiency for this test is 3n
```

```
D:\Files\Documents\UNI\Sem1 2018\CAB301\Assignment2\CAB301-Assignment2\CAB...
Testing for algorithm Median
Tested array is...
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
The array size 10
The array size is Even
The Median from the test array is 5.5
The number of basic operations performed is 45
The algorithms time efficiency for this test is 4n
```

```
D:\Files\Documents\UNI\Sem1 2018\CAB301\Assignment2\CAB301-Assignment2\C...
Testing for algorithm Median
Tested array is...
9, 1, 8, 1, 7, 2, 6, 3, 5, 4,
The array size 10
The array size is Even
The Median from the test array is 4.5
The number of basic operations performed is 22
The algorithms time efficiency for this test is 2n
```

```
D:\Files\Documents\UNI\Sem1 2018\CAB301\Assignment2\CAB301-Assignment2\C...
Testing for algorithm Median
Tested array is...
5, 84, 32, 8, 49, 14, 72, 30, 23, 39,
The array size 10
The array size is Even
The Median from the test array is 31
The number of basic operations performed is 29
The algorithms time efficiency for this test is 2n
```

```
D:\Files\Documents\UNI\Sem1 2018\CAB301\Assignment2\CAB301-Assignment2\...
Testing for algorithm Median
Tested array is...
5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
The array size 10
The array size is Even
The Median from the test array is 5
The number of basic operations performed is 39
The algorithms time efficiency for this test is 3n
```

```
D:\Files\Documents\UNI\Sem1 2018\CAB301\Assignment2\CAB301-Assignment2\...
Testing for algorithm Median
Tested array is...
9, 7, 4, 6, 6, 1, 4, 7, 1, 4,
The array size 10
The array size is Even
The Median from the test array is 5
The number of basic operations performed is 22
The algorithms time efficiency for this test is 2n
```

```

namespace CAB301_Assignment2 {
    class Program {
        public static Random rnd = new Random();
        public static int for_loop_counter1 = 0;
        public static int for_loop_counter2 = 0;
        public static int if_loop_counter1 = 0;
        public static int num_smaller_counter = 0;
        public static int num_equal_counter = 0;
        public static int basic_operations = 0;
        public static int array_size = 100;
        public static int data_count = 2;
        public static int time_efficiency = 0;
        public static int test_size = 1000;
        public static int test_count = 0;
        public static float median = 0;
        public static bool brute_force = true;

        public static int[] array = { };
        public static Excel.Worksheet worksheet;
        public static Excel.Application excel;
    }
}

```

```

static void Main(string[] args) {
    excel = new Excel.Application();
    CreateExcelData();
    //RefreshData();

    // Runs the function a set amount of times
    for (int i = 1; i <= test_size; i++) {
        rnd = new Random();
        array = CreateRandomArray(array_size);
        //array = CreateFunctionalTestArray();
        //array = CreateDescendingArray(array_size);
        //array = CreateAscendingArray(array_size);

        // Increases the test count and resets variables
        int[] array_copy = new int[array_size];
        array.CopyTo(array_copy, 0);
        //median = BruteForceMedian(array_copy);
        median = Median(array_copy);
        if (test_count < test_size) {
            //array_size++;
            test_count++;
            RefreshData();
        }
    }
}

```

```

//Output
List<int> list = array.ToList();
if (brute_force == true) {
    Console.WriteLine("Testing for algorithm Brute Force Median");
} else {
    Console.WriteLine("Testing for algorithm Median");
}
Console.WriteLine("Tested array is... ");
list.ForEach(o => Console.Write("{0}\t", o + ", "));
Console.WriteLine(System.Environment.NewLine + "The array size " + array.Length);
if (array.Length % 2 == 0) {
    Console.WriteLine("The array size is Even ");
    Array.Sort(array);
    float a = array[array.Length / 2 - 1];
    float b = array[array.Length / 2];
    float c = (a + b) / 2;
    Console.WriteLine("The Median using the algorithm = " + median);
    Console.WriteLine("The actual median is = " + c);
} else {
    Console.WriteLine("The array size is Odd ");
    Console.WriteLine("The Median using the algorithm = " + median);
    Console.WriteLine("The actual median is = " + Median(array));
}
Console.WriteLine("The number of basic operations performed is " + basic_operations);
Console.WriteLine("The algorithms time efficiency for this test is " + time_efficiency + "n");
Console.ReadKey();

```

## Bibliography

Levitin, A. (2012). *Introduction to The Design & Analysis of Algorithms*. Retrieved from vgloop:  
[http://www.vgloop.com/\\_files/1394454921-126688.pdf](http://www.vgloop.com/_files/1394454921-126688.pdf)

Paul, K. A. (2005). *Algorithms: Sequential, Parallel and*. Thomson.

University of Nebraska-Lincoln. (n.d.). *University of Nebraska-Lincoln* . Retrieved from Growth of Functions and:  
<http://cse.unl.edu/~sscott/teach/Classes/cse156S06/Notes/AsymptoticNotation.pdf>