CAB301 Algorithms and Complexity

Assignment – Empirical Analysis of an Algorithm

Due: Friday, 12th April 2019

Weight: 30%

Student Name: Kwun Hyo Lee

Student Number: N9748482

Due Date: 12/04/2019

Table of Contents

1.0 Introduction

This report will analyse the *BruteForceMedian*(*A*[*0..n-1*]) algorithm (refer to appendix 1) to determine the time complexity of the given algorithm. The report will discuss the choice of basic operation and the input size, methodology, programming language, and will present an empirical analysis of the algorithm followed by testing.

2.0 Algorithm Description

The *BruteForceMedian*(*A*[*0..n-1*]) algorithm takes any array of integers as an input and returns the median value of the given array. The algorithm searches each item of the collection until the median is found. This algorithm would return the *k*th element, where $k = \lfloor n / 2 \rfloor$ if the array was sorted.

The algorithm pseudocode is as given below:

```
ALGORITHM BruteForceMedian(A[0..n - 1])

  //Returns the median value in a given array A of n numbers. This is
  // the kth element, where k = |n/2|, if the array was sorted.

  k ← |n/2|
  for i in 0 to n − 1 do
    numsmaller ← 0      // How many elements are smaller than A[i]
    numequal ← 0        // How many elements are equal to A[i]
    for j in 0 to n − 1 do
      if A[j] < A[i]then
        numsmaller ← numsmaller + 1
      else
        if A[j] = A[i] then
          numequal ← numequal + 1
    if numsmaller < k and k ≤ (numsmaller + numequal) then
      return A[i]
    else
      return -1
```

Figure 1: *BruteForceMedian*(*A*[*0..n* - 1]) Algorithm Pseudocode

The algorithm assigns *k* to the absolute value of half the length of the array, *n*. The variable *k* also refers to the index of the median number of the array if the array was sorted in ascending order.

The algorithm then uses a nested for loop to evaluate how many elements are smaller, and how many are equal to *A*[*i*]. Doing so proves whether the current inspected value is the median of the array. The number of values smaller than *A*[*i*] must be less than *k* as the median value is defined by the number separating the higher half from the lower half in a set of integers. Thus, the sum of the number of elements smaller than *A*[*i*] and equal to *A*[*i*], must be lesser than or equal to *k*.

In the case of an array containing an odd number of integers, the algorithm simply returns the median value. When the array contains an even number of integers, the algorithm will return the *k*th index of the array if the array was sorted in ascending order. This is the last element of the first half of the data set. If the given array is empty, the **for** loops would be disregarded and the algorithm would simply return -1.

3.0 Choice of Basic Operation and Input Size

The basic operation of a given algorithm is defined by the operation which best characterises the efficiency of the algorithm of interest (Tang, 2019). For the *BruteForceMedian*(*A*[0..*n* - 1]) algorithm, the efficiency will be defined by time complexity. Thus, the chosen basic operation will be the conditional statements within the nested **for** loop as the two comparisons take the largest toll on the execution time.

```
if A[j] < A[i] then
    numsmaller ← numsmaller + 1
else
    if A[j] = A[i] then
        numequal ← numequal + 1
```

Figure 2: *BruteForceMedian*(*A*[*0..n* - 1]) Algorithm Basic Operation(s)

The two statements lie within the nested **for** loop as shown in figure 1. Thus, this operation adopts a quadratic efficiency of $\theta(n^2)$ (refer to appendix 2), where *n* is based on the length of the given array.

3.1 Best-Case Efficiency

The best-case efficiency for a non-empty array *A* is where the first element in the array is the median value or if the array only contains a single element. If either of these requirements is met, then the algorithm will only execute the **if** condition statement *n* number of times. The **else** condition statement will be executed $\lceil n/2 \rceil$ times where *n* is an odd number, and executed $\lceil n/2 \rceil + 1$ times where *n* is an even number.

3.2 Worst-Case Efficiency

The worst-case efficiency for the given algorithm is when the median is the last element of the array. If so, the algorithm will always execute the **if** condition statement n^2 times. The **else** condition will be executed $n \cdot \lceil n/2 \rceil$ times where *n* is an odd number, and executed $n \cdot \lceil n/2 \rceil + n/2$ where *n* is an even number.

3.3 Average-Case Efficiency

The average-case efficiency of the *BruteForceMedian*(*A*[0..*n* - 1]) algorithm is dependant on the fact that any of the given elements within the array can potentially be the median. Appendix 3 represents the probability for the median to appear in each element in an average-case efficiency. The calculations below show an example of the quadratic efficiency of the algorithm,

$$Let\ n\ =\ 100,$$

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

$$c_{avg}(100) = \left(\frac{100^2 + 100}{2}\right)$$

$$= 5050$$

$$Let\ n\ =\ 10\ 000,$$

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

$$c_{avg}(10\ 000) = \left(\frac{10\ 000^2 + 10\ 000}{2}\right)$$

$$= 50\ 005\ 000$$

Figure 3: Quadratic Property of Average-Case Efficiency for *BruteForceAlgorithm*(*A*[0..*n* - 1])

3.4 Order of Growth

As the algorithm was determined to have a quadratic efficiency of $\theta(n^2)$, the expected order of growth of the algorithm will be quadratic, with the efficiency being dependant on the length of the given array, *n*.

4.0 Algorithm Analysis Methodology, Tools, and Techniques Selection

1. The algorithm analysis and testing were implemented with the C# programming language. C# is a general-purpose, multi-paradigm programming language encompassing strong typing, lexically scoped, imperative, declarative, functional, generic, object-oriented, and component-oriented programming disciplines (Introduction to the C# Language and the .NET Framework, 2015).
2. The testing was performed on an Acer Spin SP513-51 laptop, running Microsoft Windows 10 Home, with an Intel Core i7-7500U CPU @ 2.70GHz. The randomness of the testing was achieved via the Next() function of the Random class, seeded based on the current time (DateTime.Now.Ticks). The Stopwatch class was also used to measure an accurate execution time of the algorithm in milliseconds. While measuring execution times, counters were removed from the *BruteForceMedian*(*A*[0..*n* - 1]) algorithm to avoid any unnecessary operations that would risk the validity of the end results. The number of processes open during the execution time testing was also minimised to further ensure accuracy.
3. The final results of the algorithm testing were generated from the average of 100 permutations of unique arrays where *n* had ranged from 0 to 20 000. The integers within the arrays had ranged from 0 to Int32.MaxValue (2 147 483 647). For testing, the data was collected in increments of 1000.

5.0 Algorithm Implementation in C#

Figure 4 shows the implementation of the *BruteForceMedian*(*A*[0..*n* - 1]) algorithm in C#.

```
// 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.
public int BruteForceMedian(int[] A)
{
	double k = Math.Ceiling((double)A.Length / 2);

	for (int i = 0; i <= A.Length - 1; 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 <= A.Length - 1; j++) {
			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];
		}
	}
	return -1;  // Will only return -1 in case of an empty array
}
```

Figure 4: C# Implementation of the *BruteForceMedian*(*A*[0..*n* - 1]) Algorithm

6.0 Algorithm Empirical Analysis

The figure below represents the hypothetical empirical analysis of the *BruteForceMedian*(*A*[0..*n* - 1]) algorithm. As calculated in appendix 2 and 3, the graph shows the quadratic efficiency of the algorithm, where the x-axis represents the length of the array *n*, and the y-axis represents the number of times the basic operation is performed.

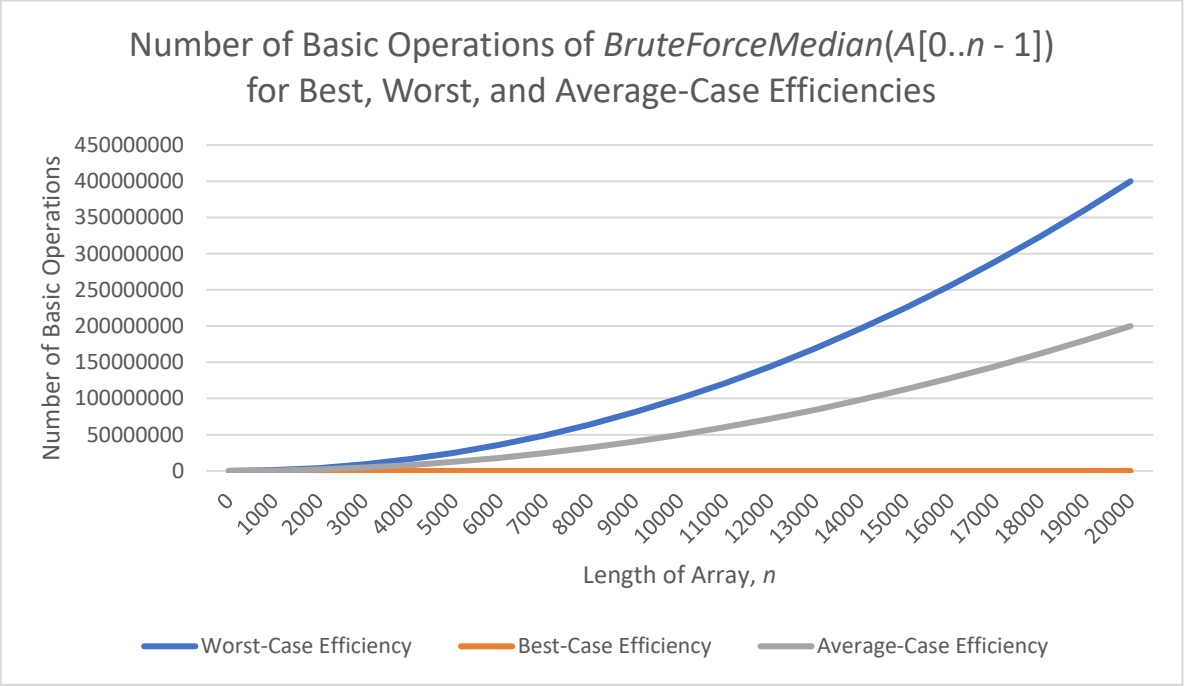


Figure 5: *BruteForceMedian*(*A*[0..*n* - 1]) Algorithm Best-Case, Worst-Case, and Average-Case Efficiencies

In figure 5, an average efficiency of $c_{avg}(n) = \frac{n^2+n}{2}$ was calculated (refer to appendix 3). However, this is only an estimate and the actual efficiency will deviate only slightly.

Figure 5 also shows a comparison between the Average-Case, Best-Case, and Worst-Case efficiency. It is notable that the Best-Case takes almost no operations to complete the algorithm while the Worst-Case takes almost twice the amount of operations of the Average-Case efficiency.

7.0 Algorithm Testing

The *BruteForceMedian*(*A*[0..*n* - 1]) algorithm in appendix 1 was tested with both, an incrementing counter for every basic operation performed, and execution time. These testings were conducted with *n* = 0 to 20 000, with the testing specifications detailed in section 4.0.

7.1 Testing *BruteForceMedian*(*A*[0..*n* - 1]) Functionality

To test the functionality of the *BruteForceMedian*(*A*[0..*n* - 1]) algorithm, several unique arrays were used to observe the outcome. The unique array compositions included arrays of even and odd numbers of elements, sorted and unsorted arrays, reversed arrays, and arrays of equal values. The testing results can be seen in appendix 5.

The table below contains the test case, a test instance, expected output, actual output, and test result of appendix 5.

Test Case	Test Instance	Expected Output	Actual Output	Test Result
Empty Array	A = { }	-1	-1	Pass
Single Element Array	A = { 1 }	1	1	Pass
Sorted Array	A = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }	5	5	Pass
Unsorted Array	A = { 4, 2, 6, 3, 8, 1, 7, 10, 5, 9 }	5	5	Pass
Reversed Array	A = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 }	5	5	Pass
Equal Array	A = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 }	1	1	Pass
Mostly Equal Array	A = { 1, 1, 1, 1, 1, 1, 2, 3, 3, 3 }	1	1	Pass
Odd Number of Elements Array	A = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }	5	5	Pass
Even Number of Elements Array	A = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }	5	5	Pass

Figure 6: *BruteForceMedian*(*A*[0..*n* - 1]) Algorithm Functionality Test Results Table

An empty array was expected to output -1 as the **for** loop would not execute and would simply return -1 (refer to figure 4). For a single element array, the basic operation would run once, then exit the loop, returning the only integer in the array. The algorithm would work similarly for the sorted array, unsorted array, reversed array, equal array, mostly equal array, and the odd number of elements array as it would simply follow the algorithm as designed. However, when an array with an even number of elements is passed into the *BruteForceMedian*(*A*[0..*n* - 1]) algorithm, the median returned will always be the *k*th element if the array was sorted. Therefore, in a sorted array containing ten elements, the algorithm will return the 5^{th} element.

As the testing shown in figure 6 had met the expected results, it can be confirmed that the given pseudocode of the *BruteForceMedian*(*A*[0..*n*-1]) algorithm in figure 4 has been implemented correctly.

7.2 Analysis by Basic Operation Count

To analyse the average-case efficiency of the *BruteForceMedian*(*A*[0..*n* - 1]) algorithm, a counter was used to measure the number of times the basic operation was performed. The counter was placed in the **for** loop as shown in appendix 6. The graph below was achieved by calculating the average number of the basic operation performed in 100 permutations of an array of *n* = 0 to 20 000.

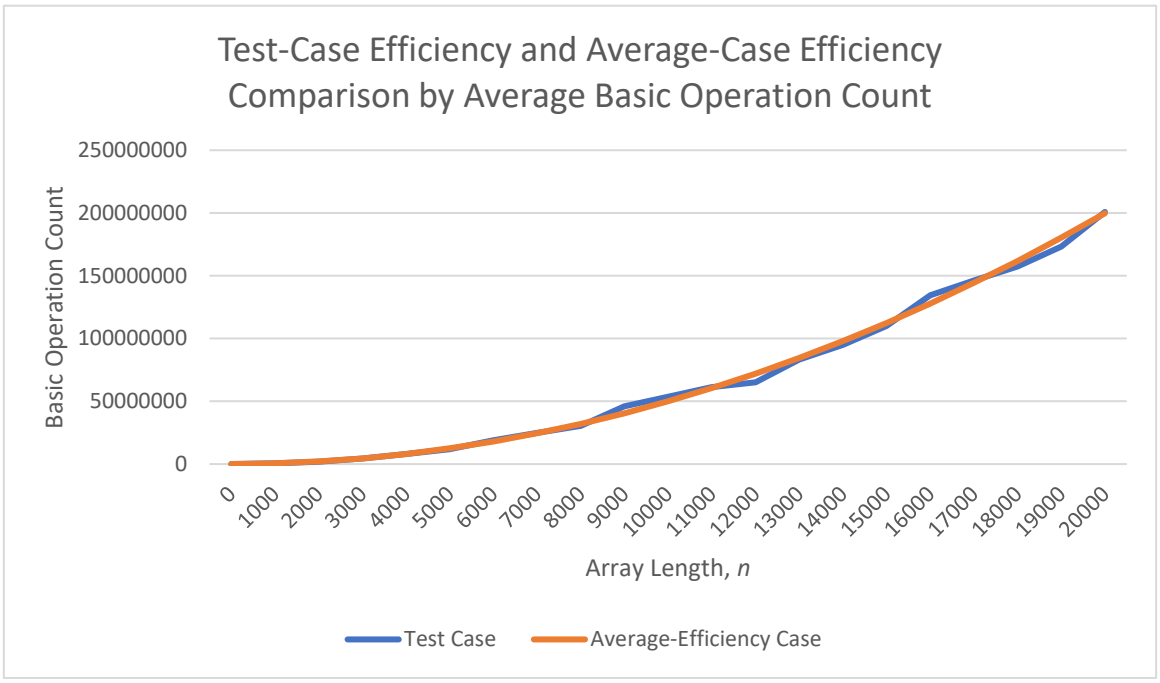


Figure 7: *BruteForceMedian*(*A*[0..*n* - 1]) Algorithm Test-Case and Average-Case Efficiency Comparison via Basic Operation Count

The Test-Case within figure 7 depicts the analysis achieved in appendix 7. The average-case efficiency was then calculated by hypothesising a likely efficiency class from the generated data. The calculations are as shown below,

Assuming an efficiency class of n^2,

$$\text{When } n = 1000, \quad operationCount = 501\,950,$$

$$cn^2 = operationCount$$

$$c = \frac{operationCount}{n^2}$$

$$c = \frac{501\,950}{1000^2}$$

$$= 0.50195$$

Thus,

$$\in \theta(0.50195n^2)$$

Figure 8: *BruteForceMedian*(*A*[0..*n* - 1]) Algorithm Average-Case Efficiency Calculation for *n* = 1000

Once an efficiency class was calculated for *n* = 1000, the mean efficiency class was calculated by finding the average of all efficiency classes from *n* = 1000 to 20 000, as shown below,

From $n = 1000\ to\ 20\ 000$,

$$c_{mean} = \left(\sum_{\substack{i=1000 \\ Step\ by\ 1000}}^{20\ 000} c_i \right) / numOfSteps$$

$$c_{mean} = 0.499475$$

Figure 9: *BruteForceMedian*(*A*[0..*n* - 1]) Algorithm Mean Average-Case Efficiency Class for Basic Operation Count Analysis

Where c_{avg} is the mean c value calculated as shown in figure 8 and c_i is the c value of the current array permutation.

As an efficiency class of $\theta(0.499475n^2)$ was calculated for the average-case efficiency, it was possible to estimate the number of basic operations performed for all given array sizes, as shown below,

When $n = 10\ 000$, $operationCount = 53\ 621\ 800$,

$$operationCount = cn^2$$
$$= 0.499475 \cdot 10\ 000^2$$
$$= 49\ 947\ 500$$
$$\approx 53\ 621\ 800$$

When $n = 15\ 000$, $operationCount = 109\ 878\ 600$,

$$operationCount = cn^2$$
$$= 0.499475 \cdot 15\ 000^2$$
$$= 112\ 381\ 875$$
$$\approx 109\ 878\ 600$$

When $n = 20\ 000$, $operationCount = 200\ 831\ 200$,

$$operationCount = cn^2$$
$$= 0.499475 \cdot 20\ 000^2$$
$$= 199\ 790\ 000$$
$$\approx 200\ 831\ 200$$

Figure 10: Estimated Basic Operation Count for Arrays of *n* = 10 000, 15 000, and 20 000

It can be observed in figure 7 that the Test-Case calculated in appendix 7 shows a quadratic rate of growth. While the Test-Case slightly deviates from the Average-Case Efficiency the line remains almost exactly accurate. The slight deviation may simply be due to the random property of the arrays.

It can also be noted that figure 7 is a more accurate representation of the algorithm's efficiency than figure 5, as figure 5 only took approximations given the architecture of the algorithm. Figure 7 was achieved through thorough testing and calculations (refer to figure 8-10).

7.3 Analysis by Measuring Execution Time

To analyse the average-case efficiency of the *BruteForceMedian*(*A*[0..*n* - 1]) algorithm, the execution time of the algorithm was also used. The graph below was achieved by calculating the average execution time of the algorithm performed in 100 permutations of an array of *n* = 0 to 20 000.

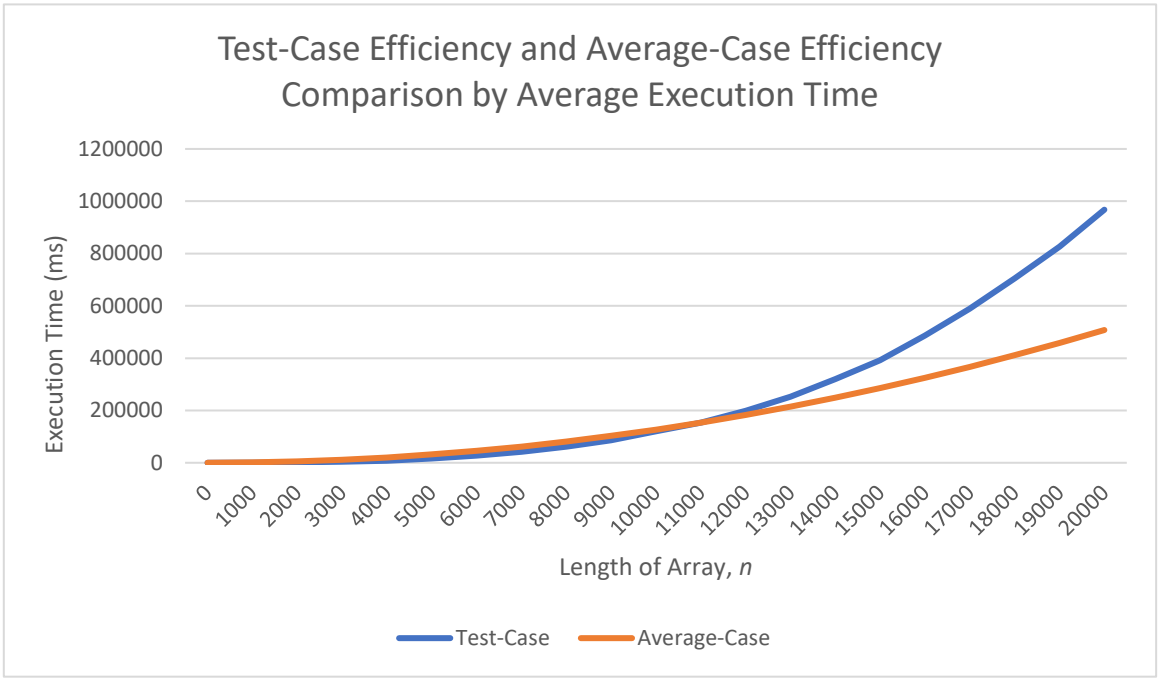


Figure 11: *BruteForceMedian*(*A*[0..*n* - 1]) Algorithm Test-Case Efficiency via Execution Time

From $n = 1000\ to\ 20\ 000$,

$$c_{mean} = \left(\sum_{\substack{i=1000 \\ Step\ by\ 1000}}^{20\ 000} c_i \right) / \ numOfSteps$$

$$c_{mean} = 0.001269$$

Figure 12: *BruteForceMedian*(*A*[0..*n* - 1]) Algorithm Mean Average-Case Efficiency Class for Execution Time Analysis

As an efficiency class of $\theta(0.001269n^2)$ was calculated for the average-case efficiency, it was possible to estimate the mean execution times for all given array sizes.

The Test-Case within figure 11 depicts the execution time analysis in appendix 9. This was achieved by calculating the average execution time from 100 array permutations from where *n* = 0 to 20 000.

By observation, the graph demonstrates the predicted quadratic rate of growth as *n* increases. However, while the Test-Case appears to look accurate by itself, it does not match with the Average-Case.

The difference between both efficiencies most significantly occur when *n* exceeds 11 000. The Test-Case efficiency shows a steeper slope than the Average-Case efficiency. This can be attributed to many potential factors. Such factors can include CPU utilisation, and language choice. The algorithm was implemented in C# which can be slower than a lower-level language such as C.

By observation of the Test-Case in figure 11, it can be stated that the efficiency of the algorithm decreases even further than expected as the length of the array, *n*, increases, providing a skewed data set.

However, despite the given inaccuracies in figure 11, the graph still shows a predicted quadratic efficiency. The basic operation count analysis also shows an accurate representation of the calculated quadratic efficiency. Thus, it can be said that the testing and calculations confirm the accuracy of the chosen basic operation, input size, and analysis methodology.

9.0 References

Introduction to the C# Language and the .NET Framework. (2015, 07 20). Retrieved from Microsoft: https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework

Tang, M. (2019). *CAB301 Lecture 2.*

8.0 Appendices

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 1: *BruteForceMedian*(*A*[0..*n* - 1]) Pseudocode

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1$$

$$Using \sum_{i=l}^{u} u - l + 1, \quad for\ integers\ l \leq u,$$

$$= \sum_{i=0}^{n-1} (n-1) - 0 + 1$$

$$= \sum_{i=0}^{n-1} n$$

$$= n \cdot \sum_{i=0}^{n-1} 1$$

$$= n \cdot ((n-1) - 0 + 1)$$

$$= n^2$$

Appendix 2: Summation Equation *M*(n) for the *BruteForceMedian*(*A*[0..*n* - 1]) Algorithm

$$c_{avg}(n) = \left(n \cdot \frac{p}{n} + 2n \cdot \frac{p}{n} + \cdots + n^2 \cdot \frac{p}{n}\right)$$

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

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

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

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

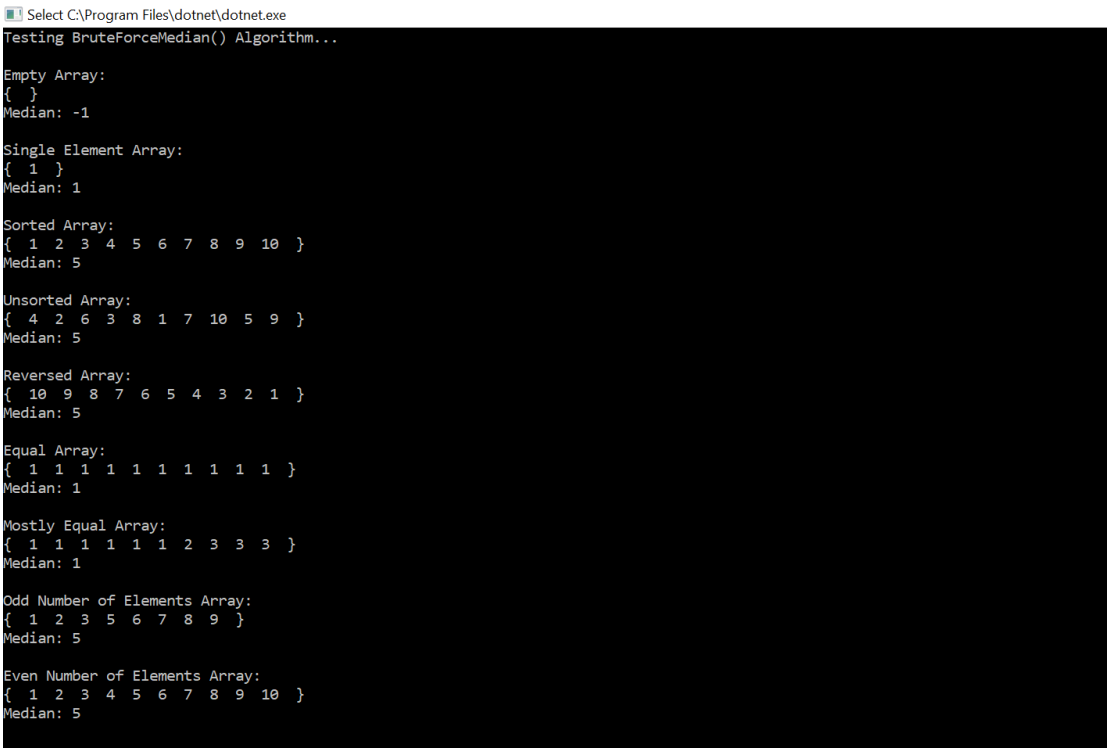
As there is always a median in a given list of integers, $p = 1$ in all cases. Thus,

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

Appendix 3: Probability of Median in Average-Case Efficiency in the *BruteForceMedian*(*A*[0..*n* - 1]) Algorithm

```
public static void functionalBruteForceMedian()
{
        // Empty Array
        int[] emptyArray = { };

        // Single Element Array
        int[] singleElementArray = { 1 };

        // Sorted Array
        int[] sortedArray = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

              ... // REFER TO THE CODE SOLUTION FOR THE REST OF THE C# CODE

        // TESTING
        Console.WriteLine("Testing BruteForceMedian() Algorithm...");
        Console.WriteLine();

        // Printing Arrays
        // EMPTY
        Console.WriteLine("Empty Array: ");
        Console.Write("{  ");
        for (int i = 0; i < emptyArray.Length; i++)
        {
               Console.Write(emptyArray[i] + "  ");
        }
        Console.Write("}");
        Console.WriteLine();
        Console.WriteLine("Median: " + test.BruteForceMedian(emptyArray));
        Console.WriteLine();

              ... // REFER TO THE CODE SOLUTION FOR THE REST OF THE C# CODE
}
```

Appendix 4: C# Implementation of the *BruteForceMedian*(*A*[0..*n* - 1]) Algorithm for Functional Testing (Uses Code from Figure 4)

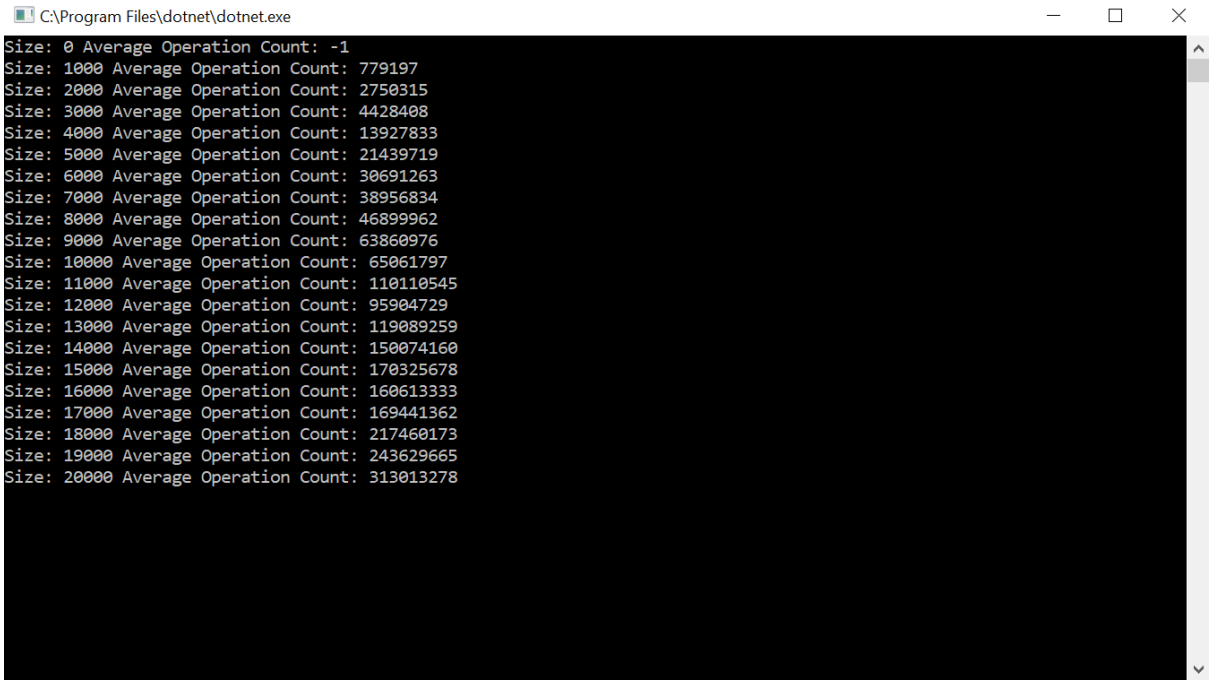


Appendix 5: Functional Testing for the *BruteForceMedian*(*A*[0..*n* - 1]) Algorithm

```
public int CountBruteForceMedian(int[] A)
{
	int count = 0;
	double k = Math.Ceiling((double)A.Length / 2);

	for (int i = 0; i <= A.Length - 1; i++)
	{
		int numsmaller = 0;
		int numequal = 0;

		for (int j = 0; j <= A.Length - 1; j++) {
			count++;
			if (A[j] < A[i])
			{
				numsmaller = numsmaller + 1;
			} else
			{
				if (A[j] == A[i])
				{
					numequal = numequal + 1;
				}
			}
		}
		if ((numsmaller < k) && (k <= (numsmaller + numequal)))
		{
			return count;
		}
	}
	return 0;
}
```

```
public static void countBruteForceMedian()
{
	for (int size = 0; size < 20000 + 1; size += 1000)
	{
		long c_total = 0;
		double c_average = 0;
		for (int i = 0; i < numOfArrays; i++)
		{
			int[] A = test.GenerateRandomArray(size);
			int count = test.CountBruteForceMedian(A);
			c_total = c_total + count;

		}
		c_average = c_total / numOfArrays;
		Console.WriteLine("Size: " + size + " " + "Average Operation Count: " + c_average);
	}

	Console.ReadKey();
}
```

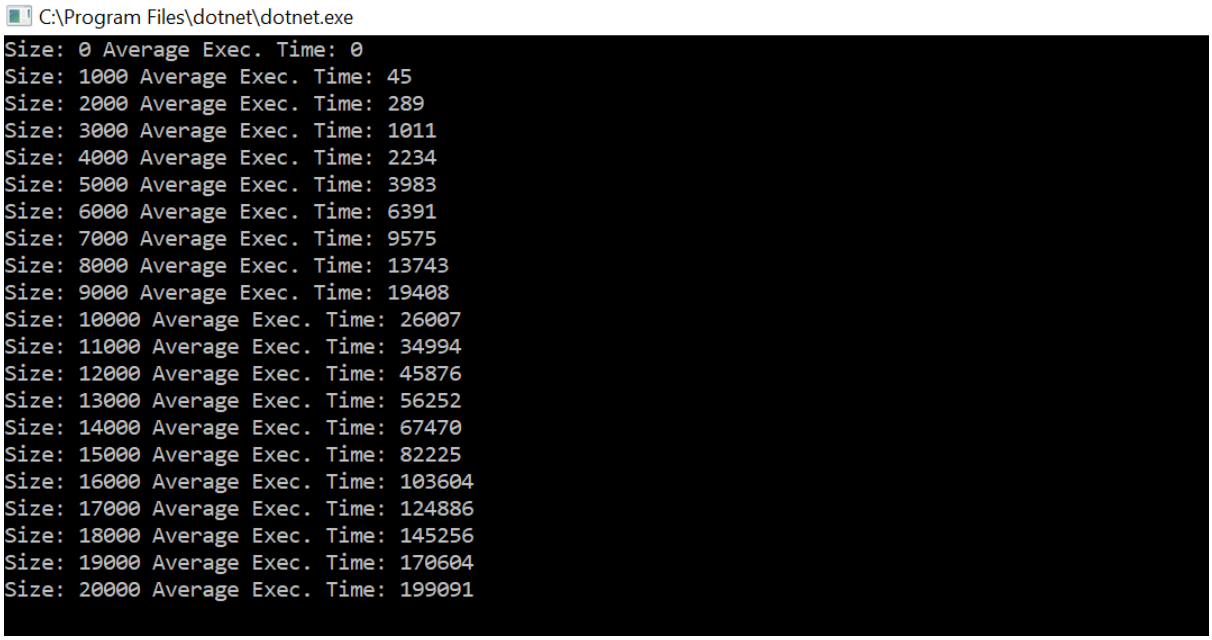
Appendix 6: C# Implementation of the *BruteForceMedian*(*A*[0..*n* - 1]) Algorithm for Basic Operation Count Testing



Appendix 7: Basic Operation Count Testing for the *BruteForceMedian*(*A*[0..*n* - 1]) Algorithm

```
public static void execTimeBruteForceMedian()
{
	Stopwatch timer = new Stopwatch();

	for (int size = 0; size < 20000 + 1; size += 1000)
	{
		long t_total = 0;
		double t_average = 0;
		for (int i = 0; i < numOfArrays; i++)
		{
			int[] A = test.GenerateRandomArray(size);
			timer.Start();
			test.BruteForceMedian(A);
			timer.Stop();
			long t_elapsed = timer.ElapsedMilliseconds;

			t_total = t_total + t_elapsed;
		}
		t_average = t_total / numOfArrays;
		Console.WriteLine("Size: " + size + " " + "Average Exec. Time: " + t_average);
	}
	     Console.ReadKey();
}
```

Appendix 8: C# Implementation of the *BruteForceMedian*(*A*[0..*n* - 1]) Algorithm for Execution Time Testing (Uses Code from Figure 4)



Appendix 9: Execution Time Testing for the *BruteForceMedian*(*A*[0..*n* - 1]) Algorithm