CAB301 Assignment 1

Empirical Analysis of an Algorithm finding the Maximum and Minimum Number in an Array

Student name: John Santias
Student no. n9983244
Date submitted: 15 April 2018

Summary

This report analyses the average-case efficiency of the maximum and minimum algorithm. The algorithm was implemented in a C# program to perform experiments. The experiments count the basic operations and measures the execution time and were compared to the theoretical efficiency predictions. The results were found to be identical to the theoretical predictions of the algorithm.

1 Description of the Algorithm

The algorithm takes three parameters, an array or set of numbers, a maximum and a minimum number. By default, max and min are passed through the parameters as zero [see figure 1 on page 6]. The algorithm then assigns the first element of the array as the maximum and minimum numbers. Afterwards, the loop inspects each number of the given set and compares it with the minimum and maximum numbers that were found [1, Section 2]. If the number is larger or smaller, it updates the current minimum or maximum value. The loop repeats this process until it reaches the end of the array. In the algorithm, we cannot assume the last element in the array is always the maximum and the first element is always the minimum number as it can either be sorted or unsorted.

2 Theoretical Analysis of the Algorithm

This section describes the time efficiency/complexity of the algorithm in a theoretical perspective.

2.1 Identifying the Algorithm's Basic Operation

There are four operations inside the loop. The first two are comparisons A[i]>MinValue and A[i]>MaxValue, the other two are assignments if either of the comparisons is true MinValue←A[i] and MaxValue ←A[i] [3, Ch1.1]. These comparisons and the assignments are the basic operations. The for loop repeats itself until it has reached the end of the array [2, Ch2.3] giving us the result of the minimum and maximum values found during the operation [2, Ch2.3].

2.2 Average-case efficiency

The average-case efficiency of an algorithm can be found with the average number of operations on the size of the array n. Levitin analyses the MaxElement algorithm [2, ch2.3] which has the identical operation as our algorithm but it only has one comparison. Levitin finds that the efficiency of his algorithm is:

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

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \, \epsilon \, \theta(n)$$

In Levitin's analysis, he states that the number of comparisons is always one because his algorithm is only finding the maximum value. Therefore, it is not required to determine the worst, best and the average case. We can assume his equation above is the best-case efficiency as it always compares numbers with only one if statement, finding the maximum number. Levitin's max algorithm has the same operation as our MaxMin algorithm. When we add another comparison (add minimum comparison) to Levitin's MaxElement algorithm, we can find the worst-case:

$$C_{worst}(n) = \sum_{i=1}^{n-1} 2 * 1$$

$$C_{worst}(n) = \ 2 \sum_{i=1}^{n-1} 1 = 2(n-1)$$

$$C_{worst}(n) = \ 2n - 1 \, \epsilon \, \theta(n)$$

We can determine the average-case efficiency by getting the best-case and the worst-case efficiencies and divide it by two. However, finding the average by using that method will not give us an accurate calculation. To get a more accurate result, we combine both the best-case and worst-case to form the following equation:

$$C_{avg}(n) = 2n - \log_e n - 1 \, \epsilon \, \theta(n)$$

2.3 Time efficiency

The average time efficiency increases as the input array size grows [5, Ch1.1.2]. The time required for the algorithm to get the minimum and maximum value in an array can be determined by the use of average-case equation from Section 2.2. For example, a given array with the size of 1000 should take around 1992.092245 basic operations. Another example is

when the size of the array is 123456, the algorithm should take around 246899.2764 basic operations.

2.4 Order of growth

As explained in section 2.3, the bigger the array size, the bigger the time efficiency is. Thus, we expect to see a linear growth on a graph as the size of the array gradually increases.

3 Methodology, Tools, and Techniques

This section briefly summarises the methodology, tools and techniques used to perform experiments.

1. The minimum and maximum algorithm were implemented on the C# programming language and was used to perform experiments. The Visual Studio Community software was used to run the programming language. C# is an object-oriented programming language which can develop multiple types of applications such as a console, windows, web, mobile applications and much more [4].
2. The experiments were performed on a personally built Windows operating system with 16Gb of RAM, i7-2600 CPU running at 3.40GHz and enough free storage space to provide the best performance possible as well as the most accurate results to compare with the theoretical predictions. Other software running was minimised during the tests to get an accurate measurement for the execution time.
3. The results were recorded on two separate excel spreadsheets to record fifty tests of the basic operations and the execution times. Using the results, we were able to produce graphs using the built-in line graph function to view and analyse the results.

4 Experimental Results

The performed experiments are described in this section and were compared with the theoretical predictions described from section 2.

4.1 Functional Testing

The following tests were fun multiple times to approve its functionality. The tests were implemented in the algorithm after the for loop [Appendix A]. Testing the functionality is important to get the most accurate results and making sure the program did not produce any bugs or errors. The first test [Appendix B] was to check if the program provided the correct array size. The test produced the following output, confirming that the program did not cause any errors.

```
The array [12, 43, 63, 12, 77, 45, 50, 71, 38, 74] has 10 items
```

Another test was creating an array of different sizes and numbers, to confirm that the program picked out the correct minimum and maximum number in the random array. We confirmed that there were no errors.

```
The array [3, 82, 12, 67, 12, 9, 24, 84, 55] has the minimum number 3 and the
maximum number 84.

The array [2, 12, 93, 44, 14, 47, 65, 39, 72, 31, 10, 42, 67, 49] has the minimum
number 2 and the maximum number 93.
```

Another test confirmed the program had no problems when it was looking for a minimum & maximum value in an array sorted in ascending and descending order.

```
The array [10, 20, 30, 40, 50, 60, 70, 80, 90, 100] has the minimum number 10 and
the maximum number 100.

The array [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] has the minimum number 1 and the maximum
number 10.
```

These tests were run so we can make sure that when using larger arrays, the result for the basic operations and execution time could give us an exact match to the predictions.

4.2 Average-Case Number of Basic Operations

We determined the number of basic operations by inserting a counter in the algorithm. As mentioned earlier in section 2.1, the basic operations are the two comparisons and the assignments. We created an empty counter called 'basicOp' outside the loop. Each time an if statement was executed, the counter added one to its value [Appendix C]. The counter added another one when there was an assignment to either the maximum or the minimum variable.

The basic operation experiment [Appendix C] were performed fifty times, producing the average basic operations. We produced the results for each array size starting with the size of one then one hundred to ten thousand, increasing each array size by one hundred starting from one hundred to ten thousand. We were able to retrieve the results by making the program print the value of the counter *basicOp* on a console window. The results did closely match the theoretical predictions. The graph [see figure 2 on page 7] shows a continuous increase in the basic operations when increasing the array size. Using the formula from section 2.2, a prediction was made that an array size 5000 would have 9995.3 basic operations. The experiment we produced an average 10004.38 [Table A] basic operations through fifty tests. The predicted basic operations for the array size of 10000 was 19995 but the experiments produced an average of 19804.2 [Table A] basic operations.

The results produced may not have been precisely the same as what was predicted. There are some factors that could have affected the experimental results. The random values made in the array could have been the reason for the experiment to produce a slightly bigger result. If the arrays were sorted in ascending or descending order, we may have obtained a much closer result, smaller or bigger results. Also, the array values were produced between the range of 1

and 1000. We would have produced a much closer result if we created a smaller range. For example, a range between 1 and 100 could produce lesser assignments. Few other factors such as using whole numbers or having same values in an array could have also affected the results.

4.3 Average-Case Execution Time

The execution time was measured by adding a stopwatch in the algorithm, starting before and stopping after the loop [Appendix D]. Each iteration of the loop would print out the execution time corresponding to the array size. The test was done fifty times to obtain average execution time for size. The results did show that the execution time continued to increase as array size was also increasing. These results can be seen in Figure 3, where each data point shows the time it took for the program to finish finding the minimum and maximum number in the given array. It was found that an array size of 5000 would take an average of 0.006832 milliseconds [Table B] to execute and an array size of 10000 would take an average of 0.08816 milliseconds [Table B] to execute.

The experimental results did match the order of growth predicted in section 2.4. The execution time does increase when the input size increases. The graph [see figure 3 on page 8] does show a linear line as predicted. There are high and low peaks with certain array sizes. This may have been caused by the computer's components such as the memory, CPU, HDD, operating system as it can slow down at times. To conclude, the execution time does match with our predictions in section 2.

References

1 – Stoimen's web log. 2018. stoimen's web log. [ONLINE] Available at: http://www.stoimen.com/blog/2012/05/21/computer-algorithms-minimum-and-maximum/. [Accessed 6 April 2018].

2 – Levitin, A., 2011. Introduction to the Design and Analysis of Algorithms. Addison-Wesley. ISBN 978-0-13-231681-1

3 – McConnell, J., 2007. Analysis of Algorithms. Jones & Barlett Learning. ISBN 978-0-7637-0782-8

4 – BillWagner. 2018. Introduction to the C# Language and the .NET Framework | Microsoft Docs. [ONLINE] Available at: https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework. [Accessed 7 April 2018].

5 – Dr. Der Naturwissenschaften. 2011. From Worst-Case to Average-Case Efficiency - Approximating Combinatorial Optimization Problems. [ONLINE] Available at: http://www.qucosa.de/fileadmin/data/qucosa/documents/6531/diss.pdf. [Accessed 7 April 2018].

```
ALGORITHM MaxMin2(A[0..n - 1], MaxValue, MinValue)
 // Finds the maximum and minimum numbers in an array
 // Input parameter: An array A of n numbers, where n = 1
 // Output parameters: The largest and smallest numbers in the given
 // array, MaxValue and MinValue, respectively
 MaxValue ? A[0]
 MinValue ? A[0]
 for i ? 1 to n - 1 do
      if A[i] > MaxValue
             MaxValue ? A[i]
      else
             if A[i] < MinValue
             MinValue ? A[i]
```

Figure 1: The analysed algorithm, the average basic operations and execution time of a given array.

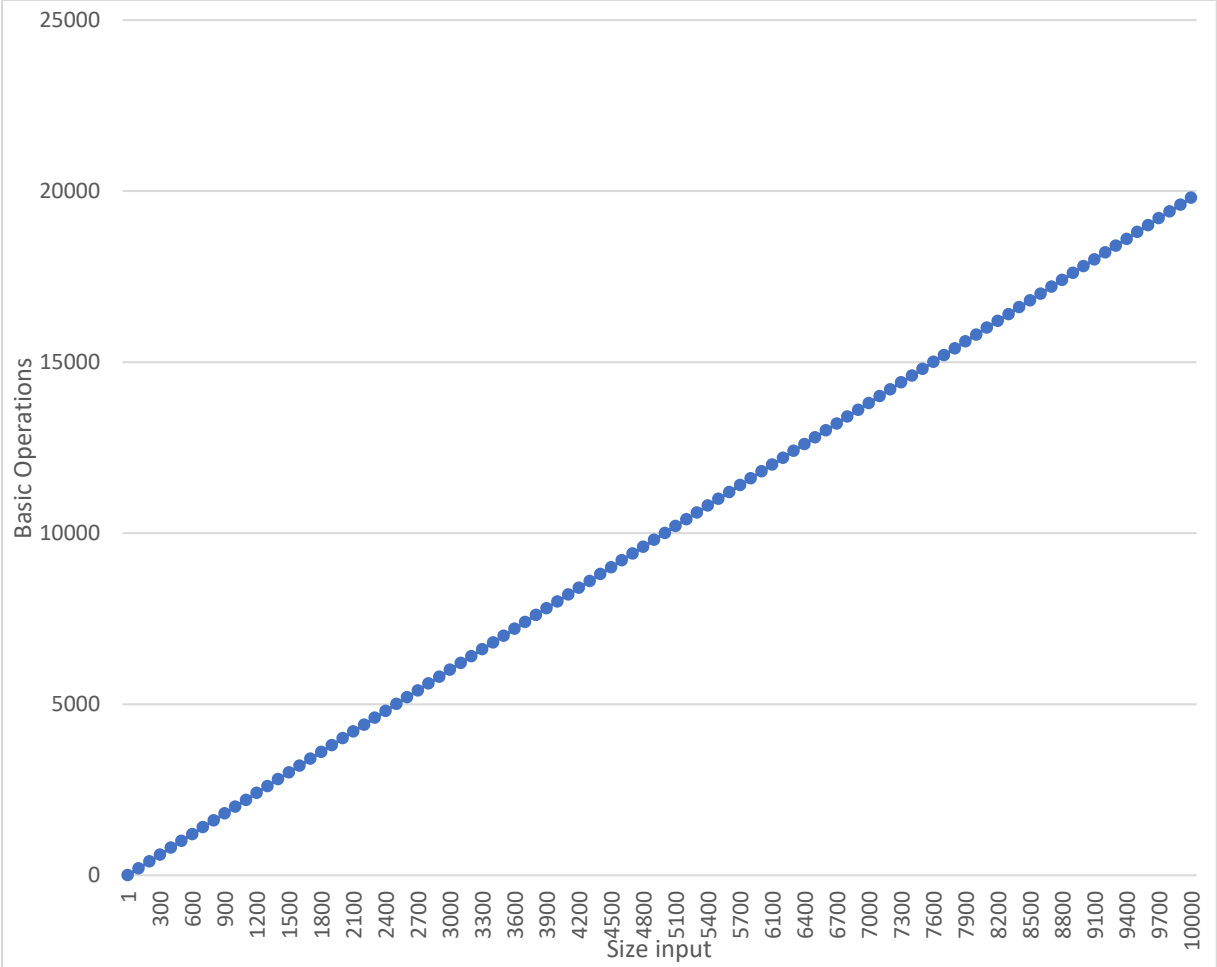

Figure 2: The average number of basic operations. There are one hundred data points, each representing an average of 50 tests. Each data point is the number of basic operations performed in regards to the array size input. This graph confirms that the basic operations consistently increases as the size input increases.

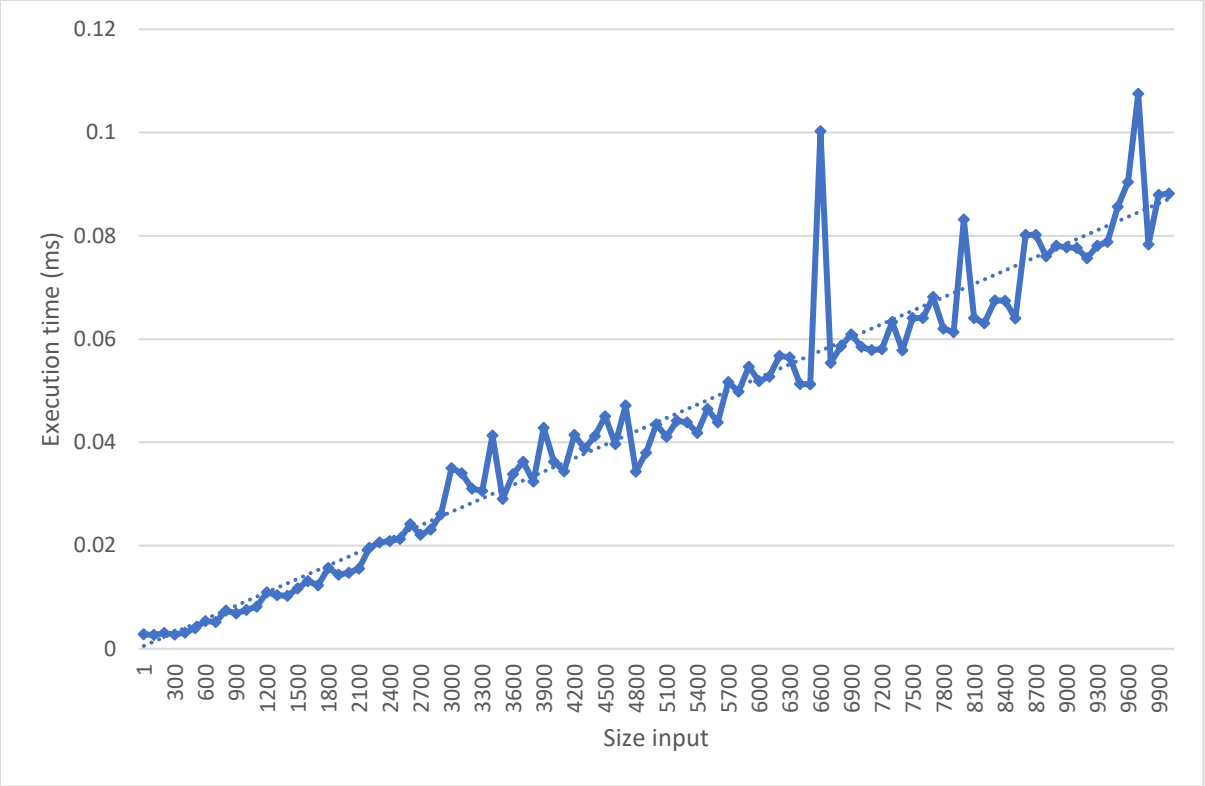

Figure 3: The execution time for finding the maximum and minimum number in an array. One hundred data points are shown, each one showing how long the program took to go through an array and finalise its results. The results do show that the execution time does increase linearly as the size input grows.

Table A Basic operations in respect to the size input

The results for the basic operations calculated in the algorithm. The experiment was done 50 times providing the average basic operation for each array size. The code in Appendix E was used to calculate the basic operations.

Input size	Predicted Basic Operations	Average Basic operation over 50 tests
1	1	0
100	197	202.22
200	396.69897	402.66
300	596.52288	602.64
400	796.39794	803.06
500	996.30103	1003.56
600	1196.2218	1203.8
700	1396.1549	1403.84
800	1596.0969	1603.78
900	1796.0458	1804.22
1000	1996	2004.04
1100	2195.9586	2203.78
1200	2395.9208	2403.96
1300	2595.8861	2603.8
1400	2795.8539	2804.14
1500	2995.8239	3004.32
1600	3195.7959	3204.74
1700	3395.7696	3404.4
1800	3595.7447	3604.06
1900	3795.7212	3804.32
2000	3995.699	4004.68
2100	4195.6778	4204.26
2200	4395.6576	4403.68
2300	4595.6383	4604.58
2400	4795.6198	4804.46
2500	4995.6021	5004.4
2600	5195.585	5204.48
2700	5395.5686	5404.16
2800	5595.5528	5604.82
2900	5795.5376	5804.2
3000	5995.5229	6004.84
3100	6195.5086	6204.7
3200	6395.4949	6404.38
3300	6595.4815	6604.74
3400	6795.4685	6804.52
3500	6995.4559	7004
3600	7195.4437	7204.22
3700	7395.4318	7404.82
3800	7595.4202	7604.62
3900	7795.4089	7804.14
4000	7995.3979	8004.44
4100	8195.3872	8204.16
4200	8395.3768	8404.34
4300	8595.3665	8604.18
4400	8795.3565	8804.44
4500	8995.3468	9004.32
4600	9195.3372	9204.96
4700	9395.3279	9405.1
4800	9595.3188	9604.68
4900	9795.3098	9804.9
5000	9995.301	10004.38
3800	7595.4202	7604.62
3900	7795.4089	7804.14
4000	7995.3979	8004.44

Input size	Predicted Basic Operations	Average Basic operation over 50 tests
4100	8195.3872	8204.16
4200	8395.3768	8404.34
4300	8595.3665	8604.18
4400	8795.3565	8804.44
4500	8995.3468	9004.32
4600	9195.3372	9204.96
4700	9395.3279	9405.1
4800	9595.3188	9604.68
4900	9795.3098	9804.9
5000	9995.301	10004.38
5100	10195.292	10204.92
5200	10395.284	10404.68
5300	10595.276	10604.46
5400	10795.268	10804.74
5500	10995.26	11004.84
5600	11195.252	11204.46
5700	11395.244	11404.76
5800	11595.237	11604.82
6000	11995.222	11804.84
6100	12195.215	12004.1
6200	12395.208	12204.74
6300	12595.201	12404.48
6400	12795.194	12604.24
6500	12995.187	12804.34
6600	13195.18	13004.96
6700	13395.174	13204.56
6800	13595.167	13404.6
6900	13795.161	13605.44
7000	13995.155	13804.22
7100	14195.149	14004.66
7200	14395.143	14204.5
7300	14595.137	14404.96
7400	14795.131	14605.14
7500	14995.125	14804.14
7600	15195.119	15004.38
7700	15395.114	15204.34
7800	15595.108	15404.64
7900	15795.102	15604.72
8000	15995.097	15804.24
8100	16195.092	16004.9
8200	16395.086	16204.42
8300	16595.081	16404.94
8400	16795.076	16604.76
8500	16995.071	16805.24
8600	17195.066	17004.16
8700	17395.06	17204.28
8800	17595.056	17404.46
8900	17795.051	17604.5
9000	17995.046	17804.04
9100	18195.041	18004.22
9200	18395.036	18204.96
9300	18595.032	18404.7
9400	18795.027	18604.52
9500	18995.022	18804.32

Input size	Predicted Basic Operations	Average Basic operation over 50 tests
9600	19195.018	19004.7
9700	19395.013	19205.02
9800	19595.009	19404.7
9900	19795.004	19604.8
10000	19995	19804.2

Table B Execution time in respect to the size input

The execution time for the program to run through the algorithm. The experiment was conducted 50 times to get the average execution time as the input size grew. The code in Appendix F was used to calculate the execution times.

Input size	Execution time over 50 tests
1	0.002832
100	0.002666
200	0.00308
300	0.002724
400	0.003136
500	0.003968
600	0.005348
700	0.00515
800	0.007416
900	0.006832
1000	0.007498
1100	0.008126
1200	0.01092
1300	0.010364
1400	0.01028
1500	0.011652
1600	0.013096
1700	0.01232
1800	0.015648
1900	0.014344
2000	0.014722
2100	0.015514
2200	0.019552
2300	0.020638
2400	0.020846
2500	0.021296
2600	0.024166
2700	0.02208
2800	0.023112
2900	0.026032
3000	0.034998
3100	0.034
3200	0.031046
3300	0.030576
3400	0.041338
3500	0.029016
3600	0.033788
3700	0.036214
3800	0.03236
3900	0.042828
4000	0.036208
4100	0.002832
4200	0.002666
4300	0.00308
4400	0.002724
4500	0.003136
4600	0.003968
4700	0.005348
4800	0.00515
4900	0.007416
5000	0.006832
3800	0.007498
3900	0.008126
4000	0.01092

Input size	Execution time over 50 tests
4100	0.03439
4200	0.041442
4300	0.038756
4400	0.04116
4500	0.045004
4600	0.039632
4700	0.04714
4800	0.034308
4900	0.037972
5000	0.043454
5100	0.041064
5200	0.044188
5300	0.04388
5400	0.041808
5500	0.04648
5600	0.04383
5700	0.051674
5800	0.049818
6000	0.054652
6100	0.051876
6200	0.052722
6300	0.056734
6400	0.056424
6500	0.051316
6600	0.051228
6700	0.100278
6800	0.055396
6900	0.058608
7000	0.060912
7100	0.058454
7200	0.05789
7300	0.058062
7400	0.0633
7500	0.057786
7600	0.064094
7700	0.064088
7800	0.068154
7900	0.062018
8000	0.061316
8100	0.083162
8200	0.064084
8300	0.063074
8400	0.067458
8500	0.067434
8600	0.064
8700	0.080184
8800	0.080166
8900	0.076002
9000	0.078056
9100	0.077758
9200	0.077648
9300	0.07564
9400	0.078078
9500	0.078842

Input size	Execution time over 50 tests
9600	0.090434
9700	0.10753
9800	0.078294
9900	0.0879
10000	0.08816

Appendix A Algorithm in chosen programming language

The code below is the max-min algorithm implemented in the C# programming language. The Main serves as the main controller of the whole program.

```
static void Main(string[] args)
{
	Random rnd = new Random(); //import random value
	int ArraySize = 1; //Start array size of one
	for (int i = 0; i < 101; i++)
	{
		int[] array = new int[ArraySize];
		for (int x = 0; x < ArraySize; x++)
		{
			array[x] = rnd.Next(1, 1000); //picks a value between 1 and 1000
		}
		MaxMin2(array, 0, 0);
		ArraySize += 100; //Adds 100 to the array size
		if (ArraySize == 101) //array size control
		{
			ArraySize--;
		}
	}
	Console.ReadKey();
}
static void MaxMin2(int[] A, int MaxValue, int MinValue)
{
	MaxValue = A[0]; //assign first array element to the variable MaxValue
	MinValue = A[0]; //assign first array element to the variable MinValue

	for (int i = 1; i <= A.Length - 1; i++)
	{
		if (A[i] > MaxValue)
		{
			MaxValue = A[i]; //Assigns array element to MaxValue
		}
		else
		{
			if (A[i] < MinValue)
			{
				MinValue = A[i]; //Assigns array element to MinValue
			}
		}
	}
	Console.Write("In the array, the minimum value is {0}, and the maximum value is
{1}.", MinValue, MaxValue);
}
```

Appendix B Code used to test functionality

The following code was used to test the algorithm and to determine if it has produced errors. Our first test started with using the code below to display the array's elements and print out its size. This helped us confirmed the program was processing the correct data.

```
int amount = 0;
foreach (int i in A)
{
amount++;
}
string arrayStr = string.Join(", ", A);
Console.Write("The array [{0}] has {1} items", arrayStr, amount);
```

The next test was to confirm that the program did pull out the correct min and max number from the given array. We were able to compare the results with the values that we found manually.

```
string arrayStr = string.Join(", ", A);
Console.Write("The array [{0}] has has the minimum number {1} and the maximum
number {2}", arrayStr, MinValue, MaxValue);
```

The last two codes were similar to the previous code. However, we tested the algorithm to see if the program still assigned the correct numbers when the array is sorted in order and in reverse order. We were able to approve that the program did not produce any errors nor any bugs.

```
Array.Sort(A);
Array.Reverse(A);
string arrayStr = string.Join(", ", A);
Console.Write("The array [{0}] has has the minimum number {1} and the maximum
number {2}", arrayStr, MinValue, MaxValue);
```

```
Array.Sort(A);
Array.Reverse(A);
string arrayStr = string.Join(", ", A);
Console.Write("The array [{0}] has has the minimum number {1} and the maximum
number {2}", arrayStr, MinValue, MaxValue);
```

Appendix C Code for calculating basic operations

The following code below is similar to Appendix C. However, a counter 'basicOP' is used to count the number of basic operations of a given array. This counter was helpful for us to compare the results with the theoretical predictions. The counter counted the basic operations for each array increased by one hundred.

```
static void Main(string[] args)
{
	Random rnd = new Random(); //import random value
	int ArraySize = 1; //Start array size of one
	for (int i = 0; i < 101; i++)
	{
		int[] array = new int[ArraySize];
		for (int x = 0; x < ArraySize; x++)
		{
			array[x] = rnd.Next(1, 1000); //picks a value between 1 and 1000
		}
		MaxMin2(array, 0, 0);
		ArraySize += 100; //Adds 100 to the array size
		if (ArraySize == 101) //array size control
		{
			ArraySize--;
		}
	}
	Console.ReadKey();
}

static void MaxMin2(int[] A, int MaxValue, int MinValue)
{
	MaxValue = A[0]; //assign first array element to the variable MaxValue
	MinValue = A[0]; //assign first array element to the variable MinValue
	int basicOp = 0; //start counter at zero

	for (int i = 1; i <= A.Length - 1; i++)
	{
		basicOp++; //Adds one to counter
		if (A[i] > MaxValue)
		{
			MaxValue = A[i]; //Assigns array element to MaxValue
			basicOp++; //Adds one to counter
		}
		else
		{
			basicOp++; //Adds one to counter
			if (A[i] < MinValue)
			{
				MinValue = A[i]; //Assigns array element to MinValue
				basicOp++; //Adds one to counter
			}
		}
	}
	Console.WriteLine("{0}", basicOp);
}
```

Appendix D Code used to measure execution time

The following code is similar to the code in Appendix C and E. However, the code measures the time taken for the program to process the max-min algorithm. A stopwatch started at the beginning of the algorithm and stopped after the loop. This allowed us to compare our theoretical predictions stated in section 2.

```
static void Main(string[] args)
{
	Random rnd = new Random(); //import random value
	int ArraySize = 1; //Start array size of one
	for (int i = 0; i < 101; i++)
	{
		int[] array = new int[ArraySize];
		for (int x = 0; x < ArraySize; x++)
		{
			array[x] = rnd.Next(1, 1000); //picks a value between 1 and 1000
		}
		MaxMin2(array, 0, 0);
		ArraySize += 100; //Adds 100 to the array size
		if (ArraySize == 101) //array size control
		{
			ArraySize--;
		}
	}
	Console.ReadKey();
}

static void MaxMin2(int[] A, int MaxValue, int MinValue)
{
	var timer = System.Diagnostics.Stopwatch.StartNew(); //starts stopwatch
	MaxValue = A[0]; //assign first array element to the variable MaxValue
	MinValue = A[0]; //assign first array element to the variable MinValue

	for (int i = 1; i <= A.Length - 1; i++)
	{
		if (A[i] > MaxValue)
		{
			MaxValue = A[i]; //Assigns array element to MaxValue
		}
		else
		{
			if (A[i] < MinValue)
			{
				MinValue = A[i]; //Assigns array element to MinValue
			}
		}
	}
	timer.Stop(); //stops stopwatch
	Console.WriteLine("{0}", timer.Elapsed.TotalMilliseconds); //prints stopwatch
value
}
```