# CAB301 Assignment 1
# Empirical Analysis of an Algorithm
# for Finding Smallest and Largest Numbers in an Array.

Student Name: John Layson

Student No. n9823239

Date Submitted: 15/04/2018

## Summary:

This report analyses an algorithm, determines its average case efficiency and order of growth. This is done by performing tests with various size of inputs that counts the basic operation in each execution and recording multiple runtimes to get the average execution times. The algorithm is implemented in Java. The results from the test cases matches the theoretical prediction of the algorithm.

## 1 Description of the Algorithm:

The algorithm's purpose is to find the smallest and largest value in an array. This algorithm ends when it comes to a decision that *x* and *y* is the smallest and largest amongst every other value which only happens after it has scanned every element in the array.

The algorithm starts by scanning the array from the index 1 in this case, and simultaneously updating the 2 variables *(minValue & maxValue)* for each time the algorithm finds a value larger than itself, this is done by creating a for loop.

The for loop starts at index 1 **for** $i \leftarrow 1$ **to** $n - 1$ **do** , therefore it is running until n(array.length) – 1 is met. Because of the for loop start point, there was a need to make sure *index 0* is scanned therefore the *A[0]* is assigned to *minValue* and *maxValue*. Also, the algorithm moves on to the next index by incrementing the count each time the for loop is ran which is done by doing: *i*++.

The purpose of the if statements is to **compare** the value of the current index in the array to the *maxValue*. If the current index value is greater than the stored *maxValue*, the algorithm then replaces the stored value to the current index value. Likewise, for *minValue*, if the current index value is less than the stored *minValue,* the algorithm then replaces it to the current index value. This ensures that the *maxValue* and *minValue* is always updated to the largest and smallest value in the array.

## 2 Theoretical Analysis of the Algorithm:

Just by looking at the algorithm, I believe it is a *Brute Force* AKA Exhaustive search or Linear search therefore it is a ***O(n)***. *n* being the size of input. This means that the running time **increases** depending on the size of *n*. For example, if a machine takes 1 second to run the algorithm with 100,000 inputs, then it will also approximately take 2 seconds to run the same algorithm with 200,000 inputs. There are however a lot of external factors that could change the time taken, such as programming language, hardware/software and machine system load. Just by looking at the pseudocode, the first comparison (if statement) is for the maxValue, therefore the number of basic operation executions would be dependent on where the largest value is in the array. If the largest value in the array is at the end, then it would be the best-case scenario because the else statement will never be executed.

### 2.1 Identifying the Algorithm's Basic Operation:

Typically, the basic operation of an algorithm is the line that is executed the most in a *worst-case* scenario. As I have mentioned above, the purpose of the algorithm being analysed is to find minimum and maximum value in an array. I have concluded that the basic operation is the ***if A[i] > MaxValue***. I chose this as the basic operation for 3 reasons: Firstly, because it is a *key comparison* between 2 variables (Dunne, n.d.). Secondly, Due to the nature of the linear search algorithms, the fact that it goes through the entire array before exiting the loop, this if statement is checked equal to the amount of inputs. I chose this over the *assignment* lines like *MaxValue ← A[i]* because these lines are skipped over when the if condition is unsatisfied. Lastly, I chose this over the other *else statement* because the first *if* has the main comparison for the algorithm regardless of the *else* and it also has the most effect on the running time. If the first if statement is true, which is good, then the else will never be ran therefore reducing the runtime and increasing efficiency. In this case, it is better to have a high amount of basic operations because it means that the ***first if-statement*** is executed the most and the algorithm does not have to go to the ***second if-statement.***

### 2.2 Average Case Efficiency:

Calculating the efficiency can be done by identifying the amount of times a basic operation in the algorithm is ran/executed. There are several factors which can affect the running time of an algorithm, some of which are:

- Computer processor (and/or its load)
- Programming language

- Operating System
- Size of Input

While identifying the average case efficiency, it is assumed that the array is not empty, the min and max values can differ from numbers 1-1000(random generated), and that the min and max values can be either at the start/end/middle of an array. The average case in this algorithm is quite simple.

Each time the algorithm is executed, I also count the amount of times the basic operation is executed. The basic operation is executed when it meets the first if statement however due to the algorithm being linear the running time increases linearly with the size of input (Time Complexity, 2018). We can conclude that the average number of basic operations to find the minimum or maximum value (or a specific element in an array) is $C_{avg}(n) = (n+1)/2$ for the min-max algorithm. This means that if the maximum value is around the middle of the array, the basic operations count will only be executed half the times of the array input. However, regardless of where the minimum and maximum input is in the array, the algorithm will keep running until it reads the entire output hence that it's growth is a ***O(n)***.

## 2.3 Order of Growth:

Having a linear growth rate means that the amount of time algorithm will execute is proportional to the size of input. For example, if we assume that the PC1 can run program at the exact same speed, size input of 10 will run at 0.10ms, size input of 20 will run at 0.20ms (MHA, 2003), to put it in numbers: (*n*) is the size of array *(See Figure 1)*

***O(n)***

Constant(*n*) = 10(10) = 100

Constant(*n*) = 10(100) = 1,000

Constant(*n*) = 10(1000) = 100,000

Constant(*n*) = 10(100000) = 1,000,000

This means that with a constant growth, the runtime will increase in proportion of the length of the array.

## 3. Methodology Tools and Techniques:

The algorithm implementation and test cases were created in Java. Java is the most popular programming language. It is used in mobile phones, laptops, data centres, video games and many more (Oracle, n.d.).

The test cases were performed on an Acer Aspire S3 laptop running a Windows 10. It contains an i3-3217U CPU, 4GB RAM and 64-bit OS.The program Eclipse was used to implement the code in Java. I used Java's *java.util.Random* (Oracle, 2017) to populate an array with random numbers. However, when an array isn't random, if all values are specified and is unsorted/sorted, I used Text Mechanic. Text Mechanic is a browser-based text manipulation tool, it allowed me to generate numbers with specific range, separate them with certain characters like line breaks, commas, and spaces and reverse the order of the numbers. Calculating the time for the algorithm was done by using Java's *System.nanoTime()* (Oracle, 2017). When testing the time elapsed for the algorithm, background apps were closed for less load on the machine which produces results better accuracy.

The tables and graphs used to display results are from Microsoft 365 Pro Plus Excel. I used this program to create tables and generate line graphs with the data.

## 3.1 Testing methodology

A testing method called Test() is created to call the algorithm and display an output relevant to the test. For the functionality test, println() only consisted of printing the minimum and maximum value. For the basic operation test, the size of the array and basic operation count are displayed. And lastly for the execution time tests, only the time and array size are displayed. *See Appendix 4,5 and 6.*

## 4. Experimental Results

The implementation of the given pseudocode in Appendix 1 was done in Java language as similar as possible which is shown in Appendix 2. The first line in the pseudocode is the method name with the input parameter ***A[0..n-1]*** meaning an array that doesn't contain negatives and therefore starts at a 0 or a positive number. The next step for the algorithm is to assign the output parameters to the first index of the array by doing *minValue = A[0]* and *maxValue = A[0]*. Next is the for loop, ***for i ← 1 to n − 1 do,*** this means that the for-loop index should be assigned the value 1 and effectively reading the array starting from index 1. As for *n-1 do,* this is telling us to make sure not to access an index that does not exist as the last index is one less than the length because the index starts at 0 and do the rest of the code until then. The next step is the ***if A[i] > MaxValue***
***MaxValue ← A[i].*** The *if-statement* is the line that compares the current maxValue to the current array index. The condition is if the current value of the index array is greater than ***(>)*** the maxValue, and if this condition is met, then the algorithm updates the maxValue to the current index value as it is proven to be a larger value. The last lines in the algorithm are ***else***

***if A[i] < MinValue***
***MinValue ← A[i].*** Very similar to the last 2 lines, but one thing to note is the else statement. *Else* is only entered if the first if statement is false, the reason for this order of line is because if the current index isn't bigger than the current *maxValue* then it could also be smaller than the *minValue*. When the *else-if* statement is true, the algorithm then assigns the current index value to the *minValue* as it has proven smaller than the current *minValue*.

There are some lines that aren't shown in the pseudocode that was added later for functionality and/or testing purposes.
***return new int[] {minValue, maxValue};*** This line allows the algorithm to return the minValue and maxValue from one method. Java typically doesn't allow methods to return more than one variable. Returning a new integer array is the work-around that limitation. Because of this return line, I also had to change the return type to ***int[]*** which is shown left of the method name.
***basicOperation = 0;*** is inserted at the start of the method call so that the count is reset to 0 each time the method is called. Otherwise, if 2 arrays are tested at the same time in the main() function, the basicOperation count will add on to each other.
***basicOperation++;*** is inserted at the start of the for-loop, this ensures that the counter is incremented by 1 each time the for-loop is executed.

NOTE: All pseudocode-programming code implementation is presented side-by-side in Appendix 3.

## 4.1 Functional Testing to Prove Correctness

A series of tests were created and performed to prove the algorithm's implementation functionality. I made 6 arrays, all of them have the same length (5 inputs), however 5 of them are manually made and the other is randomly filled. I used the testing method called Test() *See Appendix 4.*

If the array consists of numbers {2,3,2,4,5}, the algorithm will return 2 as minValue and 5 as maxValue. On the other hand, if the array consists of {1,2,3,4,5} or {5,4,3,2,1} the algorithm will return 1 and 5. The order of the numbers does not matter because the algorithm will go through the entire array.

I created string representations of the returned values, arrays and basic operation count. *See Figure 2 and 3 for more details.*

## 5 Counting the Number of Basic Operations

Counting the number of basic operation is executed is another way of telling us the algorithm's efficiency. We can test an algorithm repeatedly but by counting the basic operation theoretically gives us an estimate of what the efficiency is without having perform exhaustive testing.

The algorithm analysed has a simple counting of basic operations. The amount of basic operations was counted by adding a **counter variable** I mentioned in *Section 4*. I also added a counter in the method to count how many times the else-if statement is executed. Every time the for-loop is entered, the basic operation is also executed because it is the first if-statement. Therefore, having the counter variable increment each time the if-statement is entered gives us the number of basic operations performed.

The number of basic operations reflects the efficiency of the algorithm. As I have mentioned in Section 2.1, the best-case scenario would be the one with higher number of basic operation count where the max value is at the last index. The worst-case scenario would be if the max value is at the first index. The average-case scenario would be if the max value is around the middle index.

I created 3 arrays as test cases to demonstrate this theory.

For example:

| Array lengths ***(n)*** | No. of BO ***(x)*** | No. of BO differences ***(y)*** |
|---|---|---|
| 10 sorted (best case) | 9 | 0 |
| 10 max value in middle (Avg. case) | 5 | 4 |
| 10 max value first (worst case) | 0 | 9 |

Notice how the ***x*** value in the first array is 9 and not 10, this is since the for-loop is starting at index 1 and not index 0. It is unnecessary to start at index 0 because they are assigned to minValue and maxValue therefore we do not need to compare them to themselves to find the true max and min values. No. of BO differences ***(y)*** in this table will represent the amount of times else-if was executed.

There are 2 conclusions that became apparent after these tests.

1. The theoretical analysis about the number of basic operations was correct. In the first array the maximum value is at the end therefore there were 9 basic operation executions which means that the else statement was *never* entered/executed. In the third scenario, the third array had the least basic operation count because the max value was found in the first index which means that the else-statement was *always* executed.
2. The second scenario is meant to be the average case, however if we take note of its ***x*** and **y** value, we notice that if we add them together we get the same amount **x** value as the best-case scenario. This means that Berman and Paul were correct about how little the average scenario is better than the worst scenario.

The average scenario executed the if statement **x** (5) times and else-statement, **y** (4) times. While the worst scenario executed the else-statement **y** (9) times. This means that regardless of where max value is, if the basic operation is not executing the else statement is, and vice versa. This results to the average case running if and else-if statements 9 times in total. Whereas the worst case ran the else-if statement also 9 times in total. The algorithm is only truly at it's **best** efficiency if the array is sorted and ONLY the first if statement is always executed.

These tests were done using a testing method I mentioned in *Section 3. See Appendix 5.*

I also performed more tests to show the increase in basic operation count when the arrays are constantly higher or lower from one another. Results are discussed in the next section 5.1 Trends.

*See figures 4, 5 and 6 for constant growth in arrays resulting in equal basic operation differences/counts.*

## 5.1 Trends identified in Basic Operations – Prediction Match

Each **blue** data point in the graphs represent the number of basic operations, whilst each **orange** data points represent the difference in number of basic operation. In Figure 6, with consistent growth/differences in the array it reflects the count in basic operations. Because of this, all arrays have 20 size of input difference, each blue data point gap is equally separated from each other creating a diagonal straight line and the orange data points are on the horizontal straight line.

Regardless of the trends, these results match the predicted number of basic operation in Section 2. The number of operations is one less than the actual array size.

## 6 Execution Times

I measured the execution times by creating a method called Test()*(See Appendix 6).* It assigns the System.nanoTime() to a variable called *startTime*, calls the algorithm, assign the System.nanoTime() again to a variable called *stopTime*. The time taken is calculated by assigning stopTime minus startTime to the variable called *gap,* which is then displayed as the elapsed time.

The algorithm is called between startTime and stopTime to mark the start and end of the algorithm execution time.

I created 5 arrays that are the same in length but ordered differently. The purpose of testing these 5 arrays is to prove that the order of the elements does not affect the execution time.

| | |
|---|---|
| ArrayA (Length 1000) | Random numbers from 1-1000(Unsorted) |
| ArrayB (Length 1000) | Sorted numbers from 1-1000 |
| ArrayC (Length 1000) | Min and Max are at the start {1,1000,2,3,4,5….999} |
| ArrayD (Length 1000) | Min and Max are at the end {999,2,3,4….1,1000} |
| ArrayE (Length 1000) | Min and Max are at the middle {2,3,4…1,1000…999} |

I ran tested the execution times 10 times per array which also means that I ran the program 10 times for ArrayA, ArrayB and the others. This gives me a better time accuracy results.

The results were as expected, each array has a run time that is the ***exact*** same as another array *(See Figure 7)*. I have color-coded the run times that are the same or similar. Also, summing up all the arrays' averages, gives me a number that is very well within the range of each array's average. The lowest array average is 0.19ms and the highest is 0.29ms. The average of all averages is 0.25 which is only 0.04-0.06 difference from the others. *(See Figure 8 for a graphical representation of all averages).*

**Blue data points** represent each array's averages. Whilst **orange data points** represent the total average.

Furthermore, I wanted to prove that the algorithm has a linear time complexity. The running times will be dependent on the size of arrays. To do this, I created 9 more arrays that are distinctively different from each other.

| | |
|---|---|
| Array1(Length 100) | Random numbers from 1-1000(Unsorted) |
| Array2(Length 1000) | Random numbers from 1-1000(Unsorted) |
| Array3(Length 50000) | Random numbers from 1-1000(Unsorted) |
| Array4(Length 100000) | Random numbers from 1-1000(Unsorted) |
| Array5(Length 300000) | Random numbers from 1-1000(Unsorted) |
| Array6(Length 400000) | Random numbers from 1-1000(Unsorted) |
| Array7(Length 600000) | Random numbers from 1-1000(Unsorted) |
| Array8(length1000000) | Random numbers from 1-1000(Unsorted) |
| Array9(Length 5000000) | Random numbers from 1-1000(Unsorted) |

The results show that the size input affects the runtime of the algorithm. The array with 100 inputs has an average runtime of 0.018246ms whilst the array with 5 million inputs has an average runtime of 149.050448ms. *(See Figures 9,10,11 for arrays, all averages, and graph representation).*

## 6.1 Trends identified in Execution Times – Prediction Match

Each **blue** data point in the graph represent the execution times in milliseconds. In Figure 11, the graph shows that the execution times are higher with higher inputs. The results match the predicted increase in execution time in Section 2.

## References:

### References

MHA. (2003). *Computer Science*. Retrieved from MultiWingSpan: http://www.multiwingspan.co.uk/a23.php?page=compare

Oracle. (2017). *Class Random*. Retrieved from Oracle Java Documentation: https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

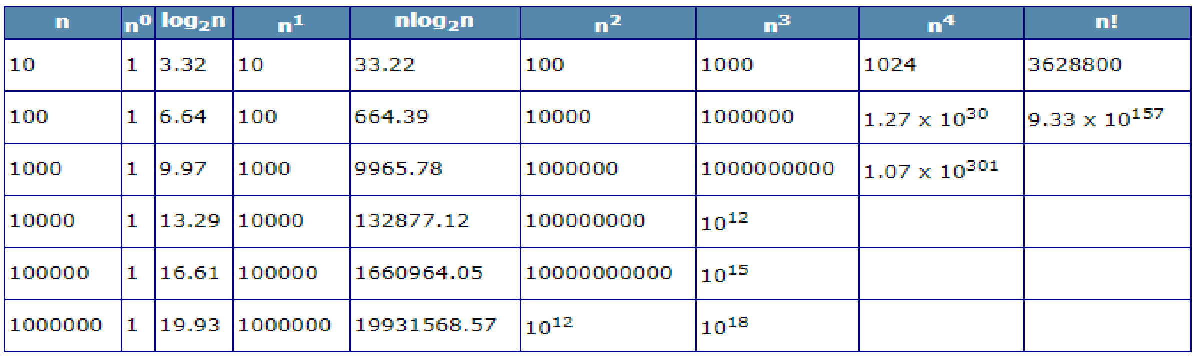
Oracle. (2017). *Class System*. Retrieved from Java Documentation: https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime--

Oracle. (n.d.). *What is Java technology and why do I need it?* Retrieved from Java: https://www.java.com/en/download/faq/whatis_java.xml

*Time Complexity*. (2018, March 30). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Time_complexity#Linear_time

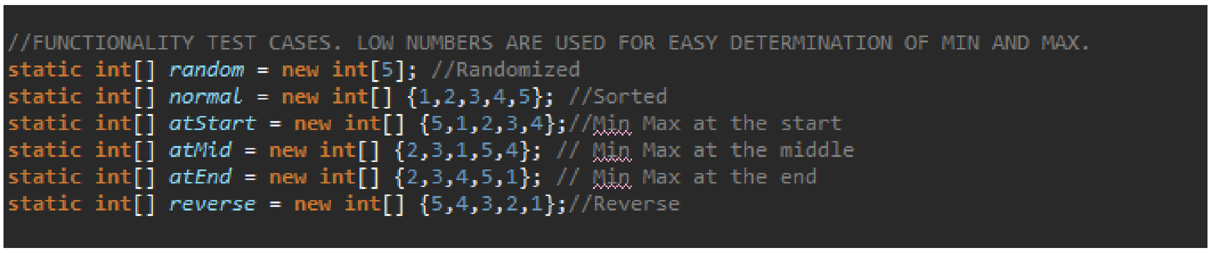
## Figures:

*Figure 1 – Linear Growth Rate – first column*



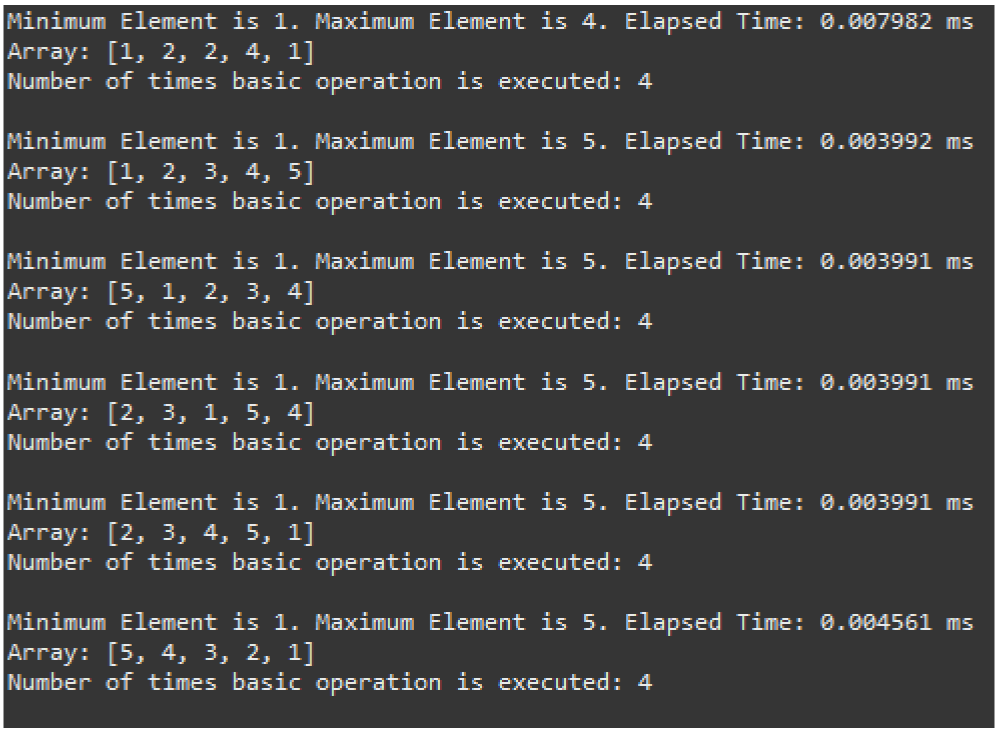
| n | $n^0$ | $\log_2 n$ | $n^1$ | $n\log_2 n$ | $n^2$ | $n^3$ | $n^4$ | n! |
|---|---|---|---|---|---|---|---|---|
| 10 | 1 | 3.32 | 10 | 33.22 | 100 | 1000 | 1024 | 3628800 |
| 100 | 1 | 6.64 | 100 | 664.39 | 10000 | 1000000 | $1.27 \times 10^{30}$ | $9.33 \times 10^{157}$ |
| 1000 | 1 | 9.97 | 1000 | 9965.78 | 1000000 | 1000000000 | $1.07 \times 10^{301}$ | |
| 10000 | 1 | 13.29 | 10000 | 132877.12 | 100000000 | $10^{12}$ | | |
| 100000 | 1 | 16.61 | 100000 | 1660964.05 | 10000000000 | $10^{15}$ | | |
| 1000000 | 1 | 19.93 | 1000000 | 19931568.57 | $10^{12}$ | $10^{18}$ | | |

*Figure 2 – Test Cases*



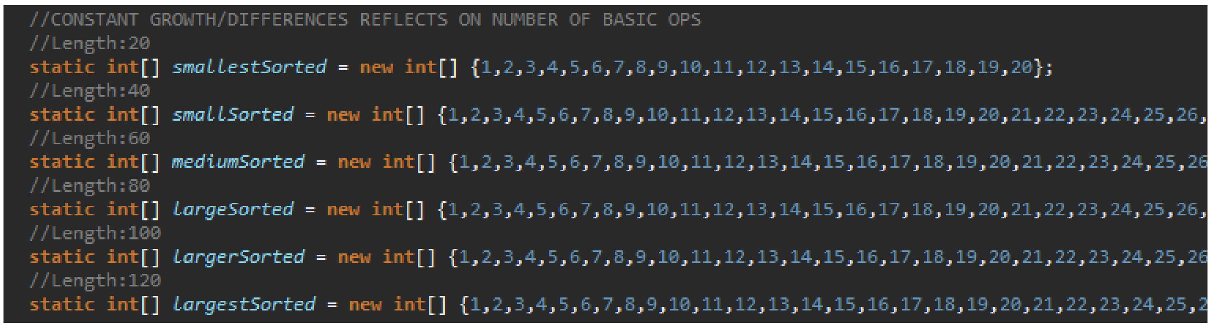
```
//FUNCTIONALITY TEST CASES. LOW NUMBERS ARE USED FOR EASY DETERMINATION OF MIN AND MAX.
static int[] random = new int[5]; //Randomized
static int[] normal = new int[] {1,2,3,4,5}; //Sorted
static int[] atStart = new int[] {5,1,2,3,4};//Min Max at the start
static int[] atMid = new int[] {2,3,1,5,4}; // Min Max at the middle
static int[] atEnd = new int[] {2,3,4,5,1}; // Min Max at the end
static int[] reverse = new int[] {5,4,3,2,1};//Reverse
```

*Figure 3 – Functionality Test Results*



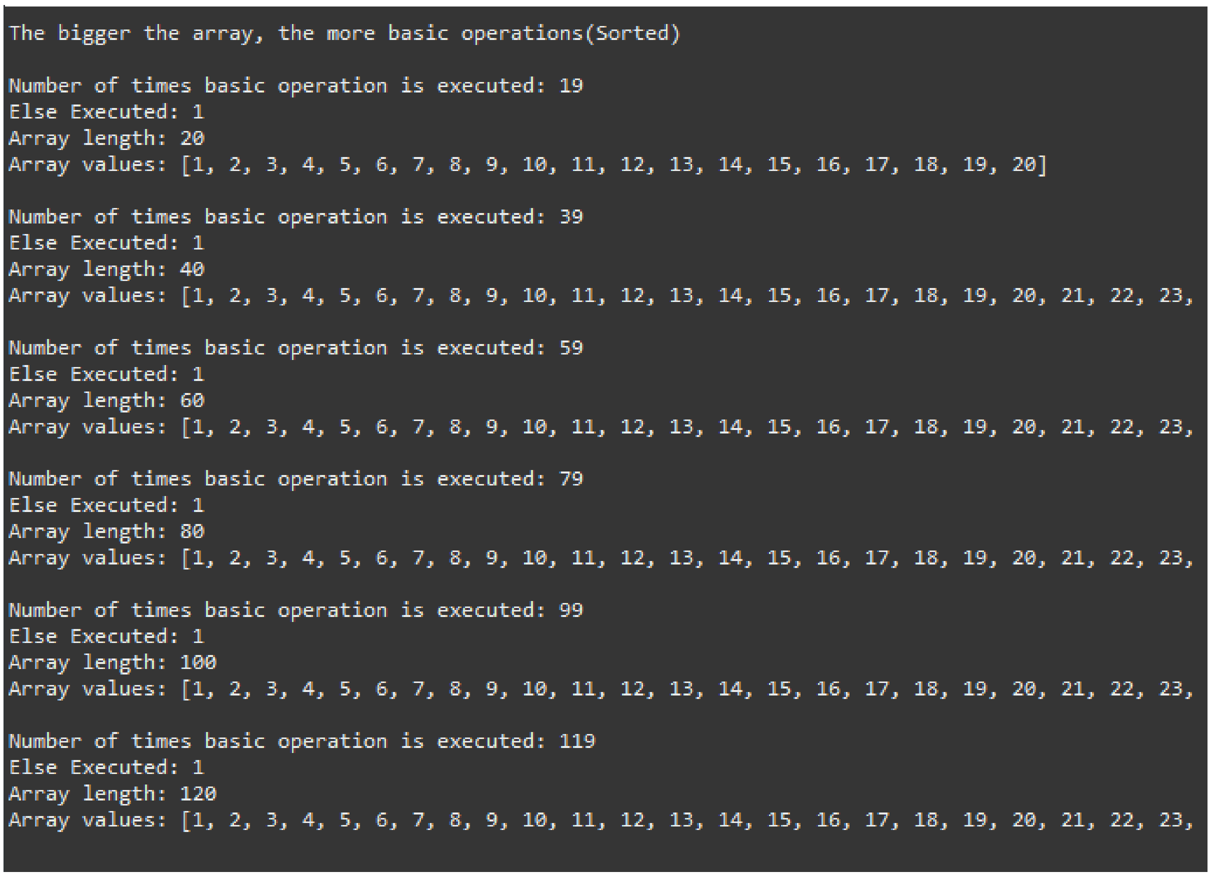
```
Minimum Element is 1. Maximum Element is 4. Elapsed Time: 0.007982 ms
Array: [1, 2, 2, 4, 1]
Number of times basic operation is executed: 4

Minimum Element is 1. Maximum Element is 5. Elapsed Time: 0.003992 ms
Array: [1, 2, 3, 4, 5]
Number of times basic operation is executed: 4

Minimum Element is 1. Maximum Element is 5. Elapsed Time: 0.003991 ms
Array: [5, 1, 2, 3, 4]
Number of times basic operation is executed: 4

Minimum Element is 1. Maximum Element is 5. Elapsed Time: 0.003991 ms
Array: [2, 3, 1, 5, 4]
Number of times basic operation is executed: 4

Minimum Element is 1. Maximum Element is 5. Elapsed Time: 0.003991 ms
Array: [2, 3, 4, 5, 1]
Number of times basic operation is executed: 4

Minimum Element is 1. Maximum Element is 5. Elapsed Time: 0.004561 ms
Array: [5, 4, 3, 2, 1]
Number of times basic operation is executed: 4
```

*Figure 4 – Arrays with constant difference.*



```
  //CONSTANT GROWTH/DIFFERENCES REFLECTS ON NUMBER OF BASIC OPS
  //Length:20
  static int[] smallestSorted = new int[] {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
  //Length:40
  static int[] smallSorted = new int[] {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
  //Length:60
  static int[] mediumSorted = new int[] {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26
  //Length:80
  static int[] largeSorted = new int[] {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
  //Length:100
  static int[] largerSorted = new int[] {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26
  //Length:120
  static int[] largestSorted = new int[] {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,2
```

*Figure 5 – Number of Basic Operation*



```
The bigger the array, the more basic operations(Sorted)

Number of times basic operation is executed: 19
Else Executed: 1
Array length: 20
Array values: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

Number of times basic operation is executed: 39
Else Executed: 1
Array length: 40
Array values: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,

Number of times basic operation is executed: 59
Else Executed: 1
Array length: 60
Array values: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,

Number of times basic operation is executed: 79
Else Executed: 1
Array length: 80
Array values: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,

Number of times basic operation is executed: 99
Else Executed: 1
Array length: 100
Array values: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,

Number of times basic operation is executed: 119
Else Executed: 1
Array length: 120
Array values: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
```

*Figure 6 – Number of basic operation and constant difference in graph*

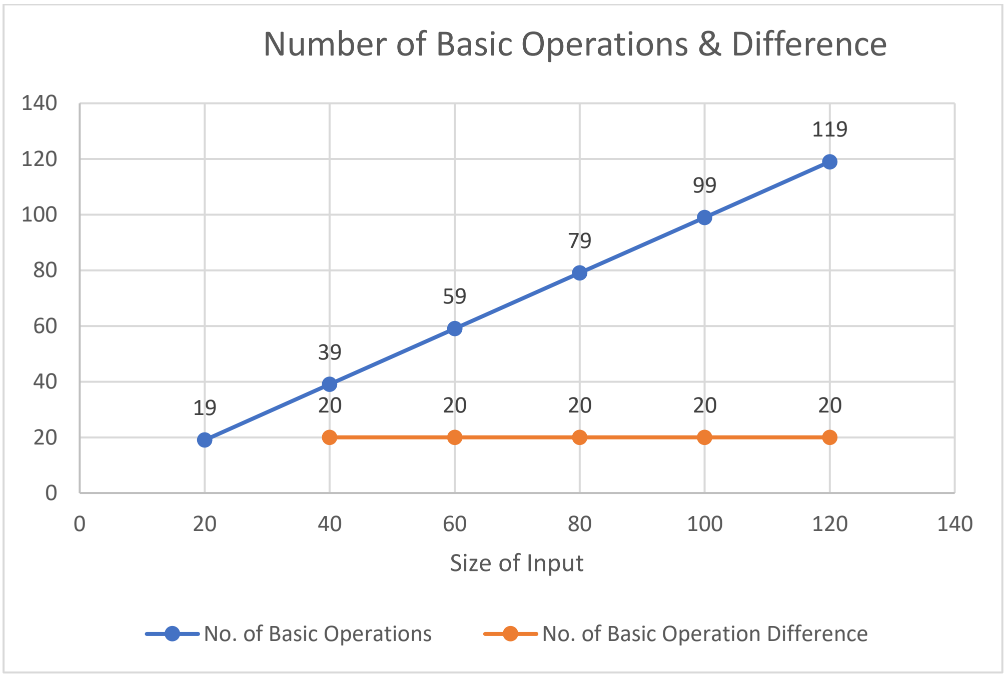
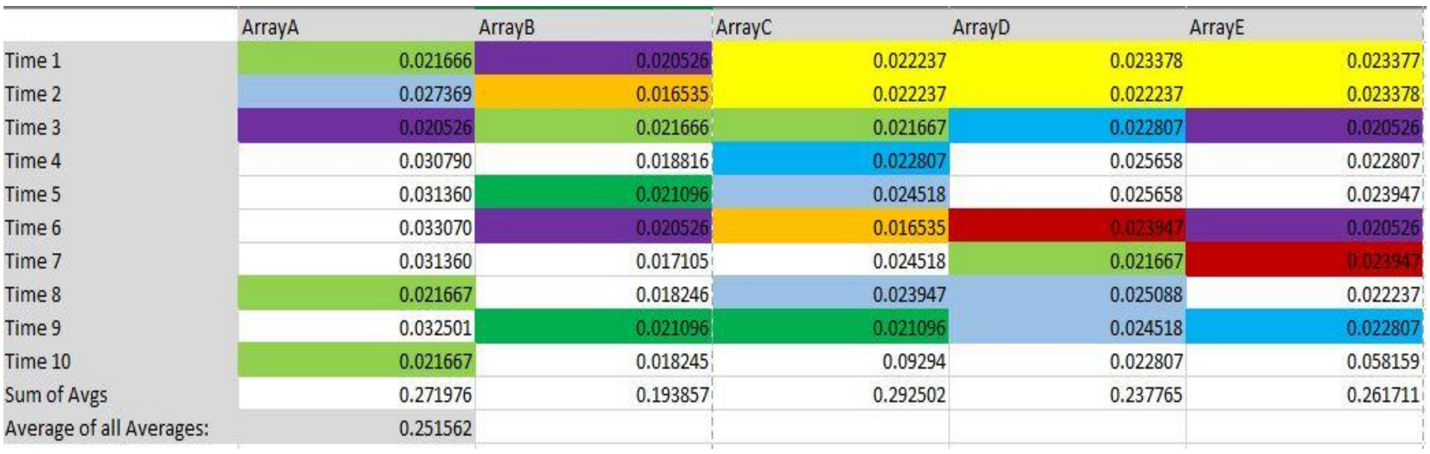


*Figure 7 – Time Results + Averages (in milliseconds) Equal array lengths*



| | ArrayA | ArrayB | ArrayC | ArrayD | ArrayE |
|---|---|---|---|---|---|
| Time 1 | 0.021666 | 0.020526 | 0.022237 | 0.023378 | 0.023377 |
| Time 2 | 0.027369 | 0.016535 | 0.022237 | 0.022237 | 0.023378 |
| Time 3 | 0.020526 | 0.021666 | 0.021667 | 0.022807 | 0.020526 |
| Time 4 | 0.030790 | 0.018816 | 0.022807 | 0.025658 | 0.022807 |
| Time 5 | 0.031360 | 0.021096 | 0.024518 | 0.025658 | 0.023947 |
| Time 6 | 0.033070 | 0.020526 | 0.016535 | 0.023947 | 0.020526 |
| Time 7 | 0.031360 | 0.017105 | 0.024518 | 0.021667 | 0.023947 |
| Time 8 | 0.021667 | 0.018246 | 0.023947 | 0.025088 | 0.022237 |
| Time 9 | 0.032501 | 0.021096 | 0.021096 | 0.024518 | 0.022807 |
| Time 10 | 0.021667 | 0.018245 | 0.09294 | 0.022807 | 0.058159 |
| Sum of Avgs | 0.271976 | 0.193857 | 0.292502 | 0.237765 | 0.261711 |
| Average of all Averages: | 0.251562 | | | | |

*Figure 8 – All averages and Total Average as a graph.*

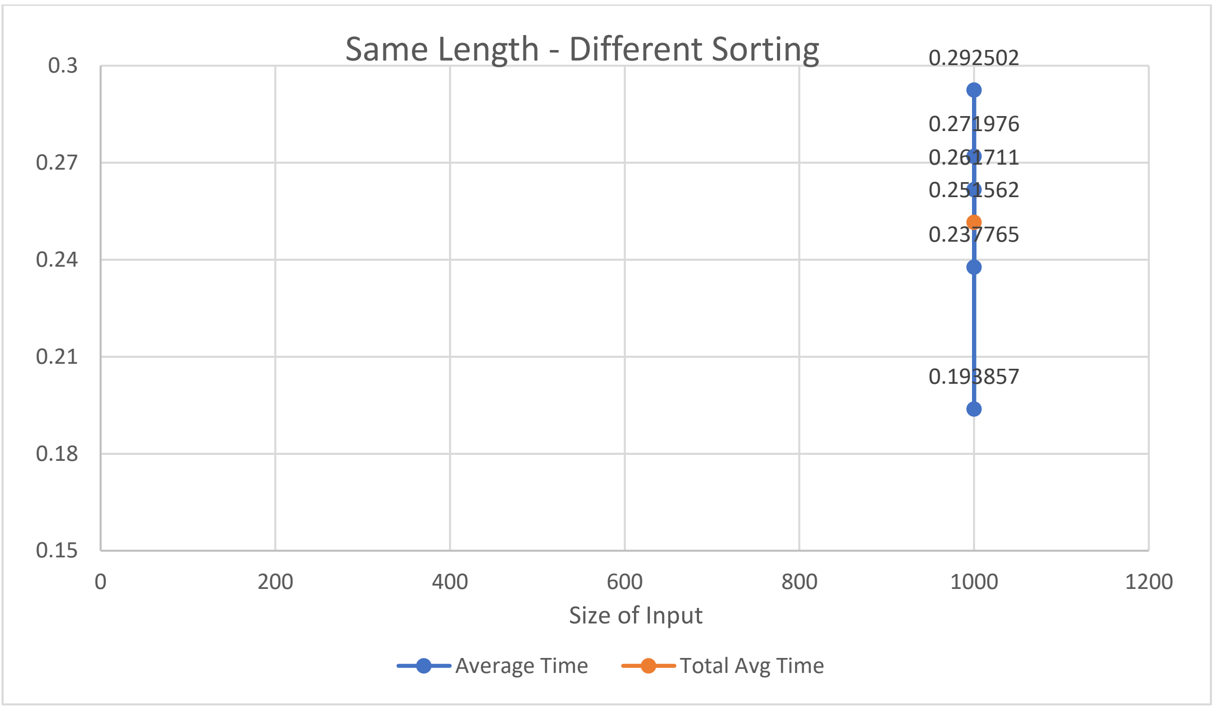
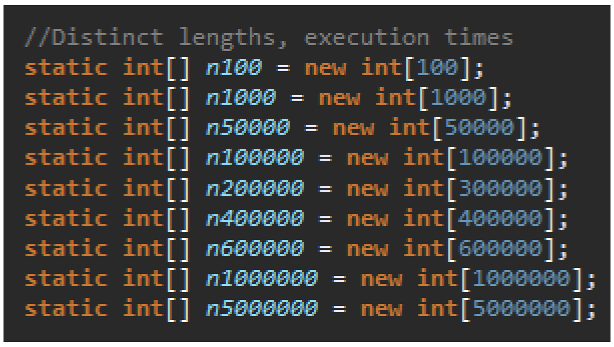


*Figure 9 – Arrays used to test elapsed time with distinct lengths.*

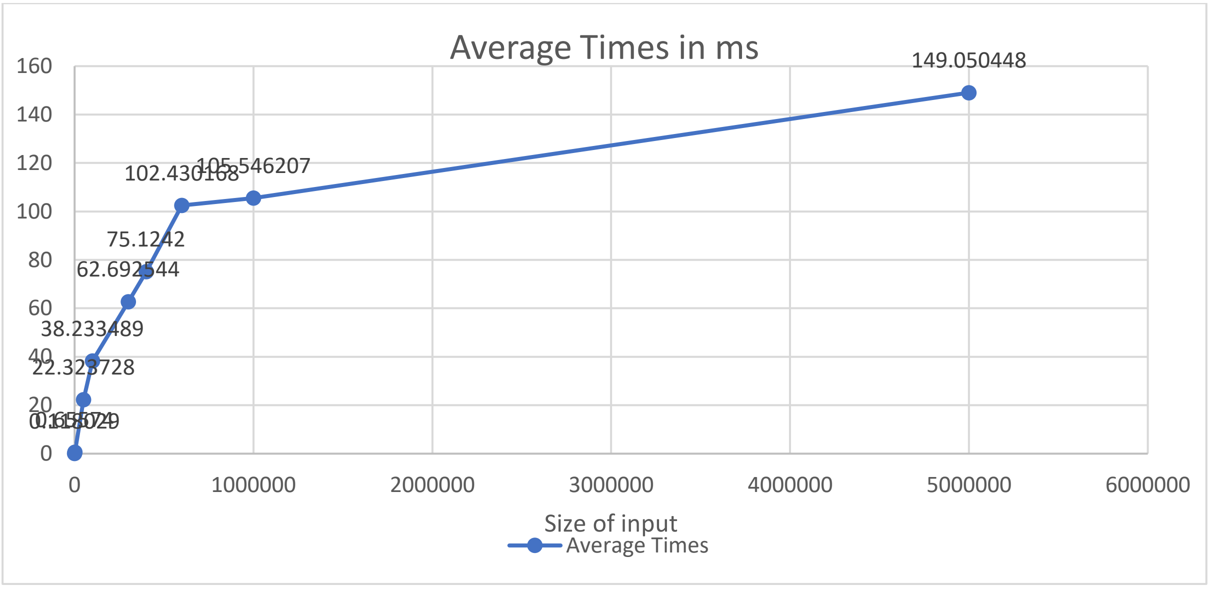


```
//Distinct lengths, execution times
static int[] n100 = new int[100];
static int[] n1000 = new int[1000];
static int[] n50000 = new int[50000];
static int[] n100000 = new int[100000];
static int[] n200000 = new int[300000];
static int[] n400000 = new int[400000];
static int[] n600000 = new int[600000];
static int[] n1000000 = new int[1000000];
static int[] n5000000 = new int[5000000];
```

*Figure 10 – Time results. Distinct array lengths*

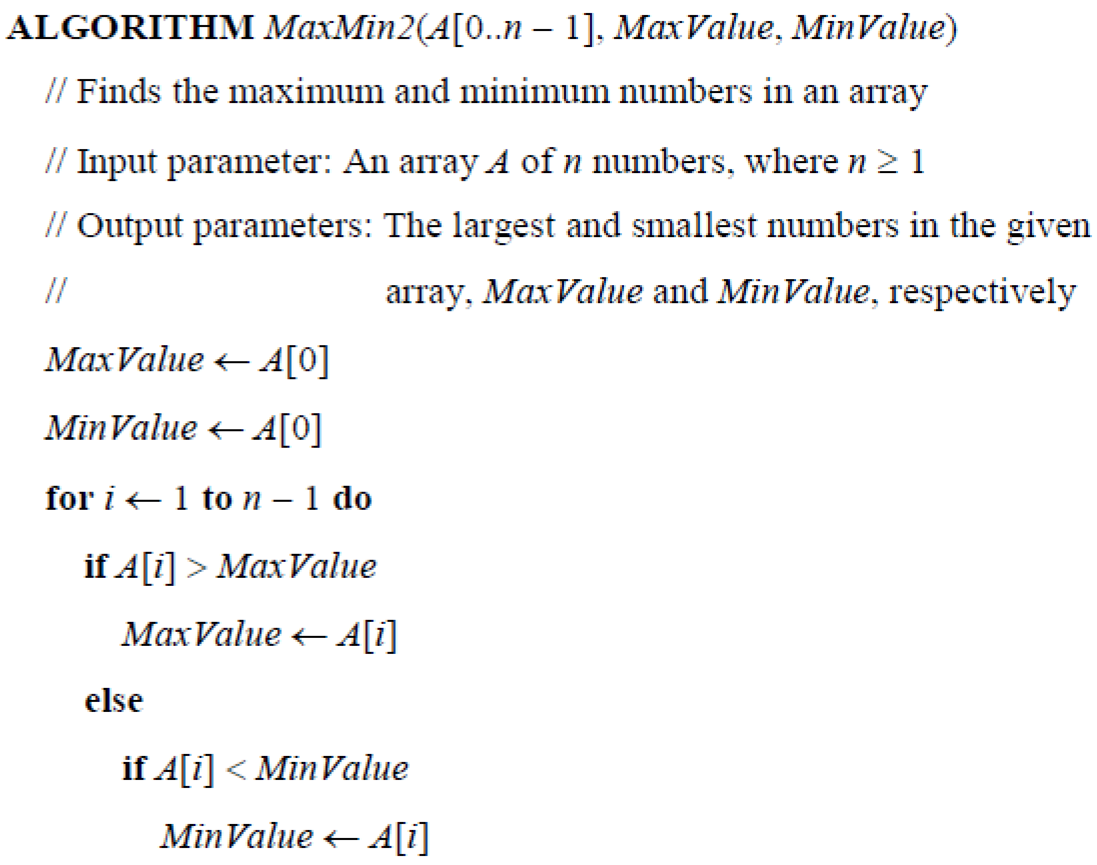
| Size of Input | Array n100 | Array n1000 | Array n50000 | Array n100000 | Array n300000 | Array n400000 | Array n600000 | Array n1000000 | Array n5000000 |
|---|---|---|---|---|---|---|---|---|---|
| Time 1 | 0.018246 | 0.075834 | 2.276733 | 3.564202 | 5.867163 | 7.009235 | 9.029387 | 12.074723 | 14.704399 |
| Time 2 | 0.011403 | 0.050175 | 2.17524 | 3.614947 | 5.787337 | 10.794667 | 10.734798 | 11.08717 | 15.163394 |
| Time 3 | 0.010834 | 0.050176 | 2.198048 | 3.77973 | 6.058744 | 7.27551 | 9.236363 | 11.858055 | 14.71124 |
| Time 4 | 0.010263 | 0.050176 | 2.168398 | 3.504333 | 5.994883 | 7.269238 | 15.167955 | 9.48211 | 15.208438 |
| Time 5 | 0.010834 | 0.052456 | 2.199188 | 4.995926 | 5.775363 | 7.107306 | 9.156537 | 9.409697 | 14.874882 |
| Time 6 | 0.010834 | 0.104913 | 2.17296 | 3.804818 | 7.335949 | 7.002963 | 12.863854 | 9.750665 | 14.93019 |
| Time 7 | 0.010833 | 0.066711 | 2.162127 | 3.863547 | 7.208229 | 7.060552 | 8.889692 | 12.870126 | 14.769398 |
| Time 8 | 0.011404 | 0.049606 | 2.403313 | 3.568763 | 6.397431 | 7.1495 | 8.602321 | 9.569918 | 14.915365 |
| Time 9 | 0.011404 | 0.077544 | 2.293839 | 3.893196 | 6.197298 | 7.470512 | 9.416539 | 10.265539 | 14.83611 |
| Time 10 | 0.011974 | 0.072983 | 2.273882 | 3.644027 | 6.070147 | 6.984717 | 9.332722 | 9.178204 | 14.937032 |
| Averages | 0.118029 | 0.650574 | 22.323728 | 38.233489 | 62.692544 | 75.1242 | 102.430168 | 105.546207 | 149.050448 |

*Figure 11 – Results in Graphs – Time Elapsed Growth*



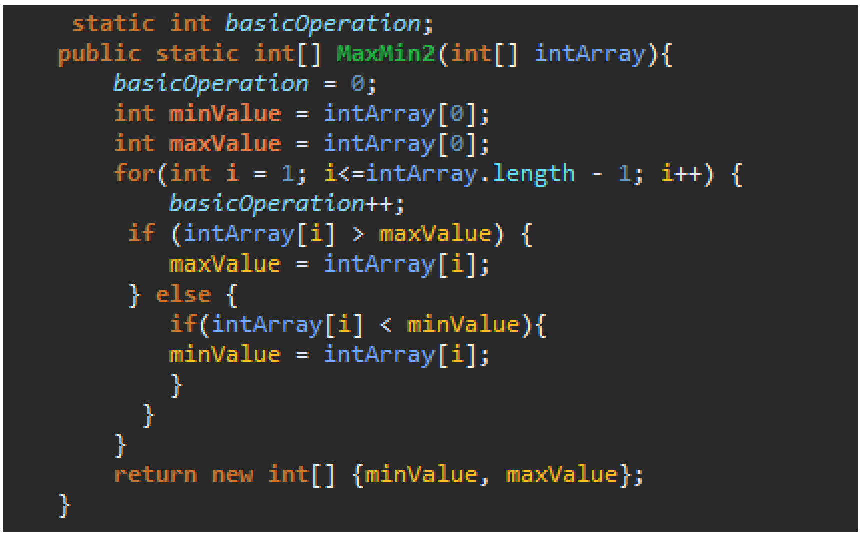
## Appendices:

*Appendix 1 – Algorithm to be analysed and converted from Pseudocode to programming language.*



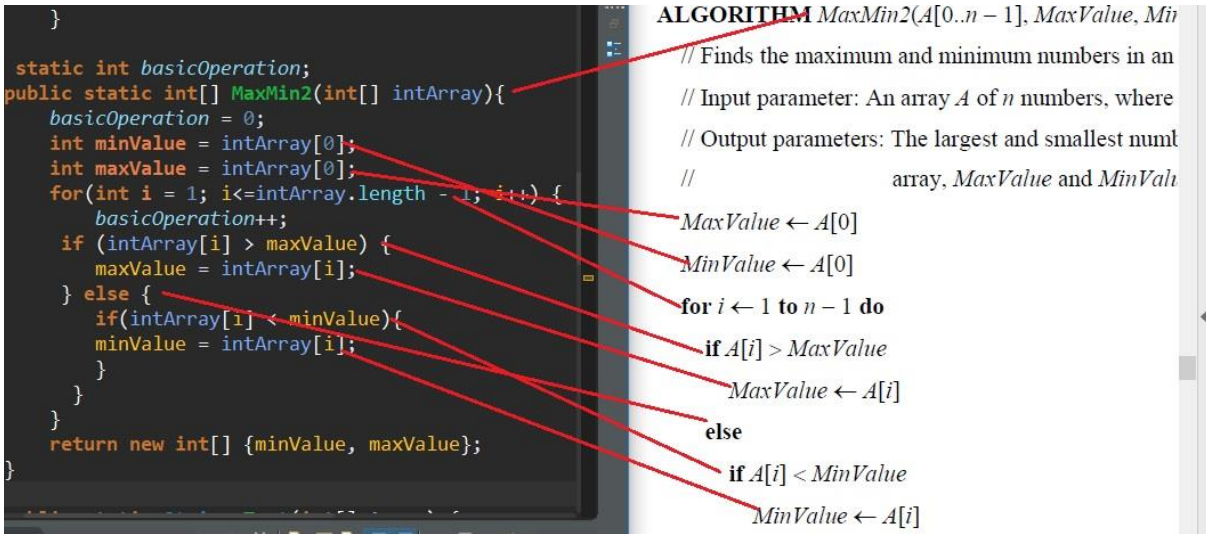
```
ALGORITHM MaxMin2($A[0..n-1]$, $MaxValue$, $MinValue$)
    // Finds the maximum and minimum numbers in an array
    // Input parameter: An array $A$ of $n$ numbers, where $n \geq 1$
    // Output parameters: The largest and smallest numbers in the given
    //                    array, $MaxValue$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

*Appendix 2 – Method implementation with Counter. NOTE: Counter is not used when recording time for less memory usage and more accuracy.*



```
  static int basicOperation;
 public static int[] MaxMin2(int[] intArray){
     basicOperation = 0;
     int minValue = intArray[0];
     int maxValue = intArray[0];
     for(int i = 1; i<=intArray.length - 1; i++) {
         basicOperation++;
      if (intArray[i] > maxValue) {
         maxValue = intArray[i];
      } else {
         if(intArray[i] < minValue){
         minValue = intArray[i];
         }
       }
     }
     return new int[] {minValue, maxValue};
 }
```

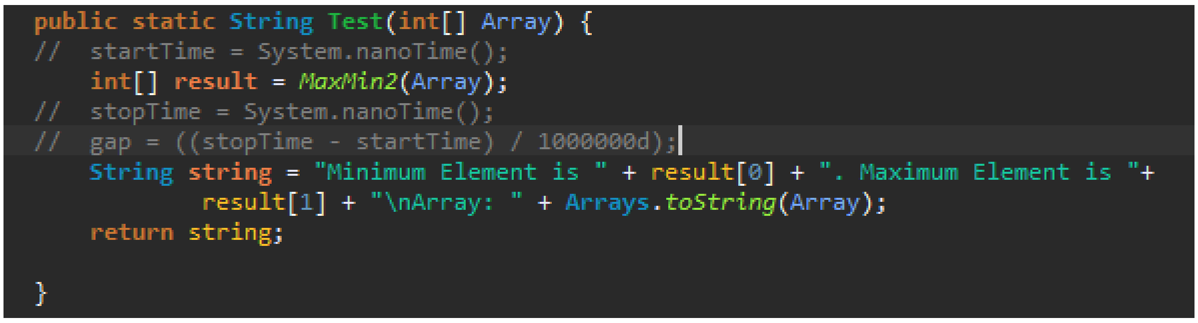
*Appendix 3 – implementation side by side.*



```
    }

 static int basicOperation;
public static int[] MaxMin2(int[] intArray){
    basicOperation = 0;
    int minValue = intArray[0];
    int maxValue = intArray[0];
    for(int i = 1; i<=intArray.length - 1; i++) {
        basicOperation++;
     if (intArray[i] > maxValue) {
        maxValue = intArray[i];
     } else {
        if(intArray[i] < minValue){
        minValue = intArray[i];
        }
      }
    }
    return new int[] {minValue, maxValue};
}
```

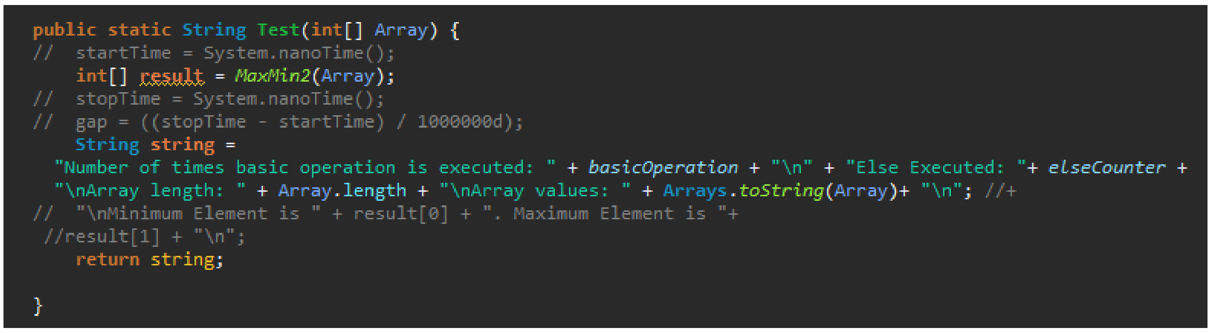
```
ALGORITHM MaxMin2(A[0..n − 1], MaxValue, Mi
  // Finds the maximum and minimum numbers in an
  // Input parameter: An array A of n numbers, where
  // Output parameters: The largest and smallest numb
  //                    array, MaxValue and MinVal
  MaxValue ← A[0]
  MinValue ← A[0]
  for i ← 1 to n − 1 do
    if A[i] > MaxValue
      MaxValue ← A[i]
    else
      if A[i] < MinValue
        MinValue ← A[i]
```

*Appendix 4 – Functionality Test Method*



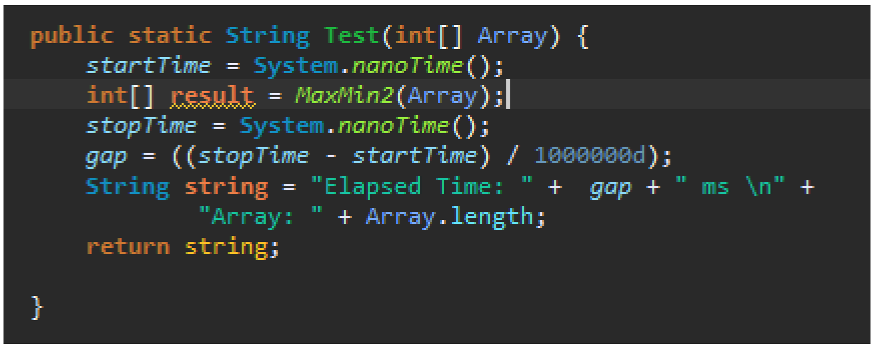
```
public static String Test(int[] Array) {
//   startTime = System.nanoTime();
     int[] result = MaxMin2(Array);
//   stopTime = System.nanoTime();
//   gap = ((stopTime - startTime) / 1000000d);
     String string = "Minimum Element is " + result[0] + ". Maximum Element is "+
             result[1] + "\nArray: " + Arrays.toString(Array);
     return string;

}
```

*Appendix 5 – Basic Operation Test Method*



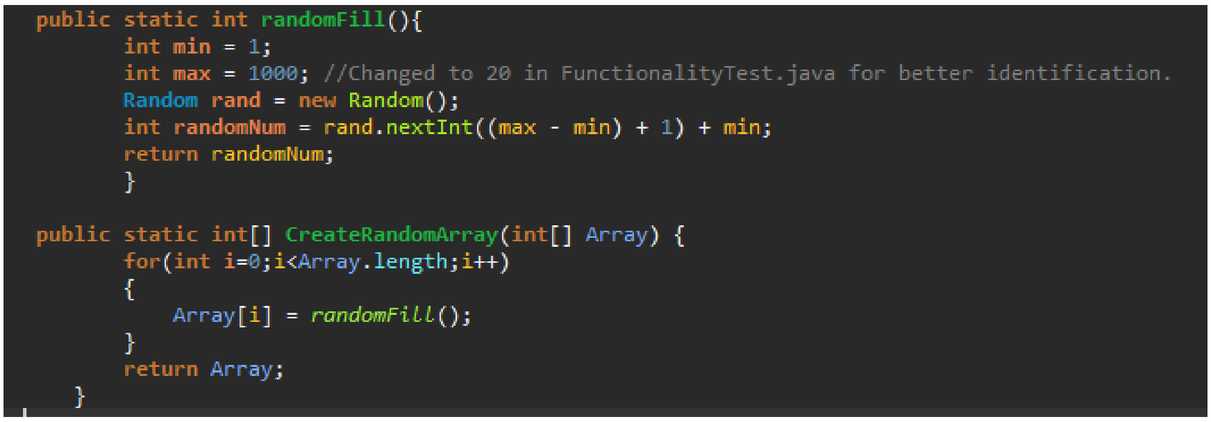
```
public static String Test(int[] Array) {
//   startTime = System.nanoTime();
     int[] result = MaxMin2(Array);
//   stopTime = System.nanoTime();
//   gap = ((stopTime - startTime) / 1000000d);
     String string =
   "Number of times basic operation is executed: " + basicOperation + "\n" + "Else Executed: "+ elseCounter +
   "\nArray length: " + Array.length + "\nArray values: " + Arrays.toString(Array)+ "\n"; //+
//   "\nMinimum Element is " + result[0] + ". Maximum Element is "+
 //result[1] + "\n";
     return string;

}
```

*Appendix 6 – Execution Time Test Method*



```
public static String Test(int[] Array) {
    startTime = System.nanoTime();
    int[] result = MaxMin2(Array);
    stopTime = System.nanoTime();
    gap = ((stopTime - startTime) / 1000000d);
    String string = "Elapsed Time: " +  gap + " ms \n" +
            "Array: " + Array.length;
    return string;

}
```

*Appendix 7 – Code for creating random Arrays*



```
 public static int randomFill(){
        int min = 1;
        int max = 1000; //Changed to 20 in FunctionalityTest.java for better identification.
        Random rand = new Random();
        int randomNum = rand.nextInt((max - min) + 1) + min;
        return randomNum;
        }

 public static int[] CreateRandomArray(int[] Array) {
        for(int i=0;i<Array.length;i++)
        {
            Array[i] = randomFill();
        }
        return Array;
    }
```