Empirical Analysis of an Algorithm – MaxMin2

Cab301 – Algorithms & Complexity

Rick Mccasker – n9449949

2018

Table of Contents

[Part 1: Description of the Algorithm 2](#_Toc509668921)

[Part 2: Predicted efficiency 2](#_Toc509668922)

[Basic Operation 2](#_Toc509668923)

[Avg Case 2](#_Toc509668924)

[Order of growth 3](#_Toc509668925)

[Part 3: Methodology, Tools & Techniques 3](#_Toc509668926)

[Part 4: Implementation 3](#_Toc509668927)

[Part 5: Basic Operations - Practical 4](#_Toc509668928)

[Part 6: Execution Time - Practical 4](#_Toc509668929)

[References 5](#_Toc509668930)

[Appendix A: MaxMin2 Program 9](#_Toc509668931)

[Appendix B: MaxMin2 Timer 10](#_Toc509668932)

[Appendix C: MaxMin2 Basic Counter 11](#_Toc509668933)

[Appendix D: Set generator 11](#_Toc509668934)

# Part 1: Description of the Algorithm

The *MaxMin2* algorithm proposed by Bernam and Paul (2005) finds the smallest and largest values in an array, unsorted or otherwise, which is passed in as the sole parameter. Given that the array is larger than 0, it iterates over the entirety of the array and returns the values determined to be the maximum and minimum. Initially the algorithm will assign the 0th value in the array to both the *MaxValue* and *MinValue* variables. It then proceeds to iterate the array starting from the 1th element, comparing it to the initial *MaxValue* variable and changing it to the compared array value if it is larger than the current *MaxValue* variable. If the array value is larger and the comparison fails, a nested if statement will trigger and the current element in the array will be compared to the initial *MinValue* and change its value if it is smaller. This process will loop until the entirety of the array is iterated over.

# Part 2: Predicted efficiency

## **Basic Operation**

The two main areas of the algorithm that affect execution time are comparison and reassignment of variables, however given that both cases of reassignment are nested within comparison blocks, the comparisons make for the best instances of basic operations. In the algorithm shown in Figure 1,the two comparison cases are contained within a for loop that runs for n-1 times given that n is the length of the array and larger than 1. The first comparison represents an if statement comparing the *MaxValue* to the current array index value. The second comparison is contained within the else statement, executed if the first basic operation fails, and compares the *MinValue* to the current array index value.

## **Average Case**

The average case efficiency for this algorithm was solved by Berman and Paul (2005) and was concluded to make comparisons, where ln represents the natural logarithm. Further analysis of the algorithm proved that it didn’t improve efficiency and instead caused it to align closer to the worst case efficiency, demonstrated in Figure 2.

## **Order of growth**

The expected order of growth of this algorithm is , derived from the solution to the average case that Berman and Paul (2005) provide. This can be supported by finding the Big O and Big Theta values of this algorithm and finding if it lies within the space expected.

In conclusion, based on the equations above, since both and are represented by the order of growth n it is safe to assume that is correct.

# Part 3: Methodology, Tools & Techniques

The algorithm and experiments on it were performed using C# within Microsoft Visual Studio Community. The computer used to perform the experiments and implement the algorithm was a desktop computer containing an i5-4670k and 8gb of RAM using the Windows 10 OS.

The arrays used for testing in this analysis were generated using the Random class provided by C#. Contained within the method ‘*SetGenerator’* (Appendix D) it would iterate over an array of a given size, then provide a random number for each iteration. The default bounds used for each number are -10000 and 10000.

The stopwatch utility provided by C# was used to measure the execution time of the program. By placing before and after execution it can gather an accurate representation of seconds elapsed.

Graphs produced where created within Microsoft excel. The program has additional methods to take data and update the relevant fields in an excel document which would then map it on a graph.

# Part 4: Implementation

This algorithm was relatively simple to translate into a program. The overall structure has not been changed however variable names were altered to make the code easier to read. The max and min variables were changed to snake case to maintain consistency in coding style and the array(*A*) and array length(*n*) variable names were altered to improve readability, being changed to *input*\_*arr* and *arr\_length* accordingly. One alteration that isn’t made clear by the algorithm is whether the array’s length should be checked every loop or be pre-assigned, in the case of the program being used in the report, the array length was preassigned.

To prove that the program runs as expected, several tests were run, all of which are contained within the ‘*ProgramTests’* class. By using the unit testing suite provided by visual studio combined with hard-coded arrays to test most possible outcomes, it was concluded that the program would run as expected, given that the array passed the precondition of n > 0.

# Part 5: Basic Operations - Practical

As shown in Figure 3the basic operation counters, represented by the *basic\_counter* variable, are both located just above the comparison statements, which were identified as the basic operations earlier in the report. The reason for this is that in any event that the counter is triggered, so will the comparison and therefore the counter will match comparisons made.

To demonstrate that the algorithm plotted as expected, 100 tests were run and the average was taken, which then was plotted for each 100th number starting from 100 up to 10000. As shown by Figure 4the graphic produced a linear line with few deviations, which matches what was expected. The plots, or the number of basic operations was almost exactly as expected with few to no deviations.

Further analysis of the basic operations counted proved that it aligned with what was expected in the theoretical analysis. By subbing any given value of n into the number calculated should be the same as the average value provided by the 5 tests. In most cases this was successful but in some cases, it wasn’t. This can be assumed to be because of the randomized values being skewed in certain directions, which could not be avoided given the nature of the experiment. Even given these deviations all values were at least similar and overall the formula verifies the legitimacy of results in the graph.

# Part 6: Execution Time - Practical

Execution times was measured by using the stopwatch function as mentioned above. As seen in Figure 5the stopwatch calls were placed just outside of where the algorithm starts and ends to produce the best result. The result is then accessed and sent back to the main method for processing. Initially the stopwatch was to test for milliseconds however the program ran too fast, the solution was to grab the entirety of what had elapsed during the timeframe, including seconds, minutes and hours and snip the unnecessary data from the string before sending it to main.

By again using 100 tests and gathering the averages of each of them, a graph was produced that matched the expected order of growth that was concluded above, as seen in Figure 6. The line steadily grows without any apparent exponential increase as n increases, matching observations made when analysing the order of growth. Some deviations occurred randomly, either causing an execution to take longer than expected or slightly less time than expected. This can be attributed to physical components of the software and computer’s processing abilities, and is not allowed for in the theoretical analysis as well as the fact that the array was generated randomly.

# References

Berman, K. and Paul, J. Algorithms: Sequential, Parallel, and Distributed. Boston: Thomson. 2005. ISBN  978-0534420574

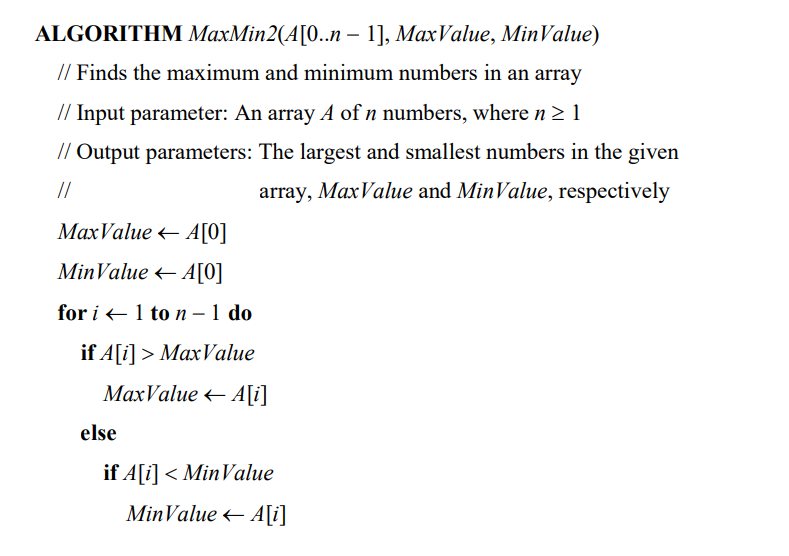
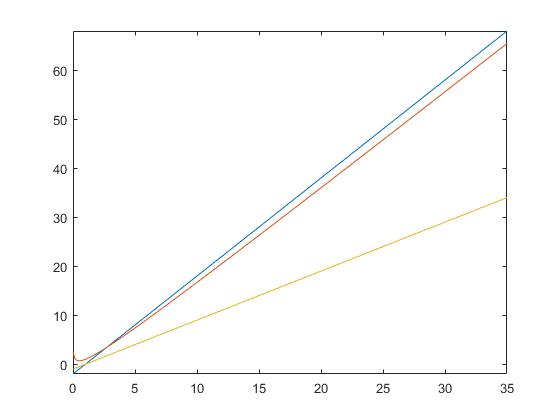


Figure 1: The algorithm devised by Berman and Paul to be converted and analysed.

Figure 2: A graph representing the upper bound (blue), lower bound (yellow) and predicted order of growth/average case (red). This should roughly map the future findings regarding this algorithm.

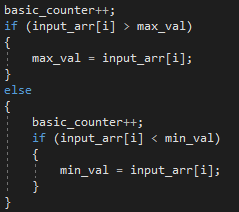


Figure 3: A snippet of the basic counter version of the program. It demonstrates how the basic operations are counted.

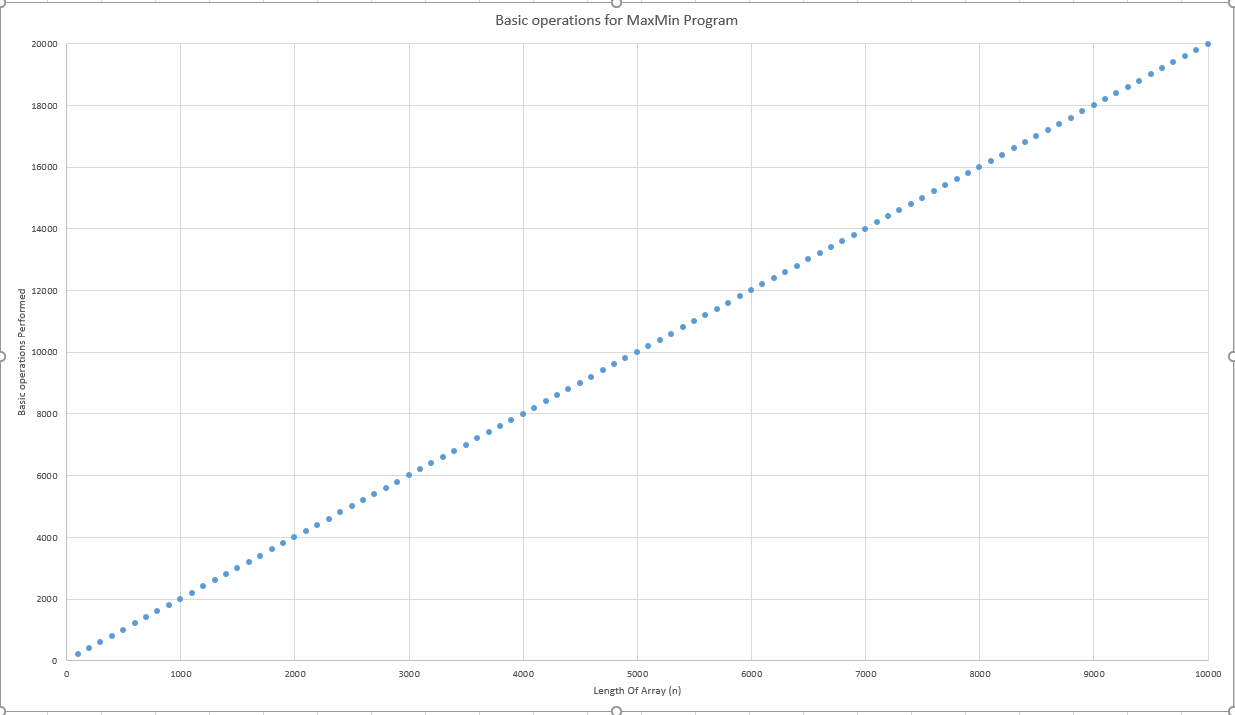


Figure 4: A relatively linear line represented by plot points, where each of which in turn represent and average of basic operations counted from multiple runs of the program. The data used was a random set so some deviation should be expected. A copy of this graph can be found in the ‘Test rsults.xlsx’ file within the report directory.

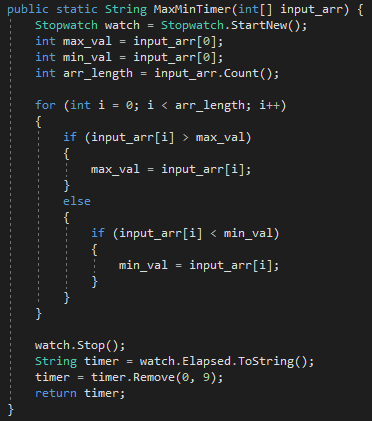


Figure 5: The method used to count the second passed from algorithm start to finish. It demonstrates the positioning and usage of the stopwatch command and how it was implemented so as not to intrude on the original algorithm but still provide results.

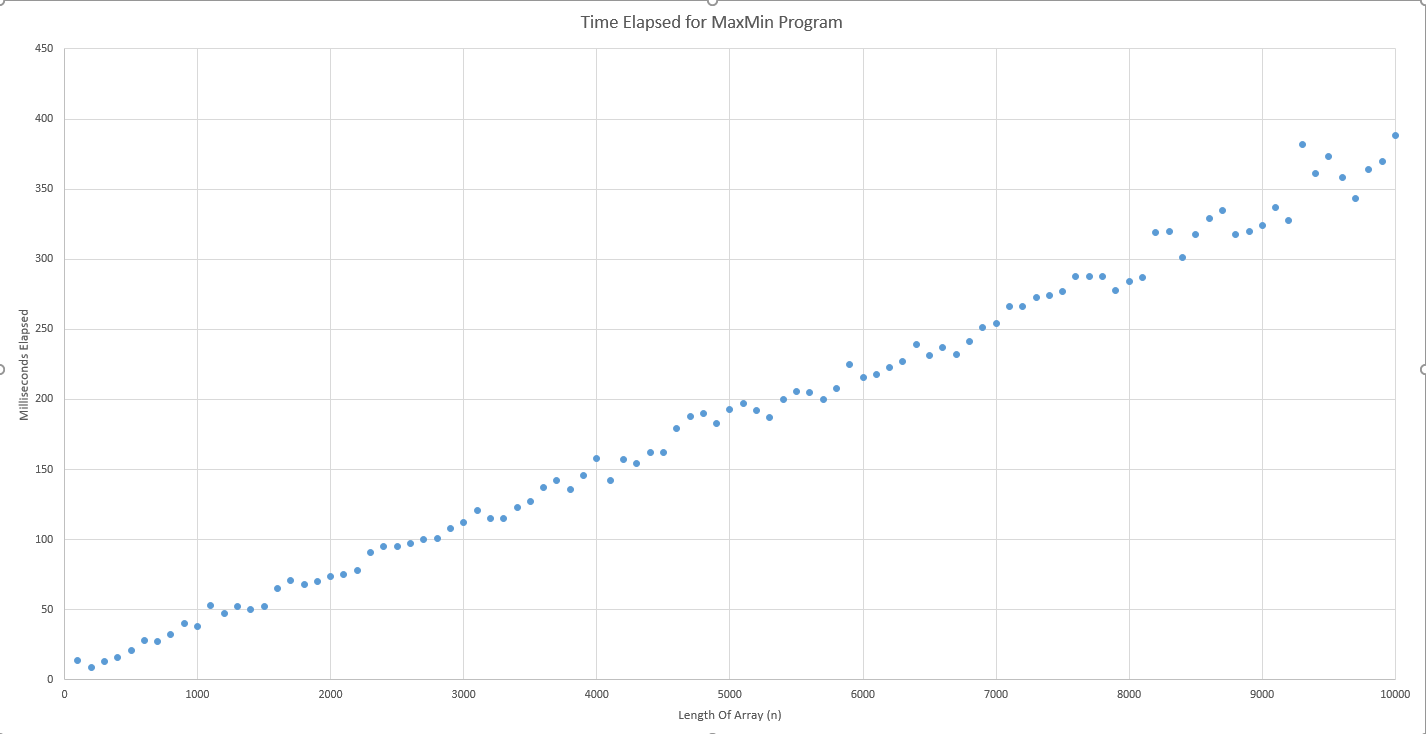
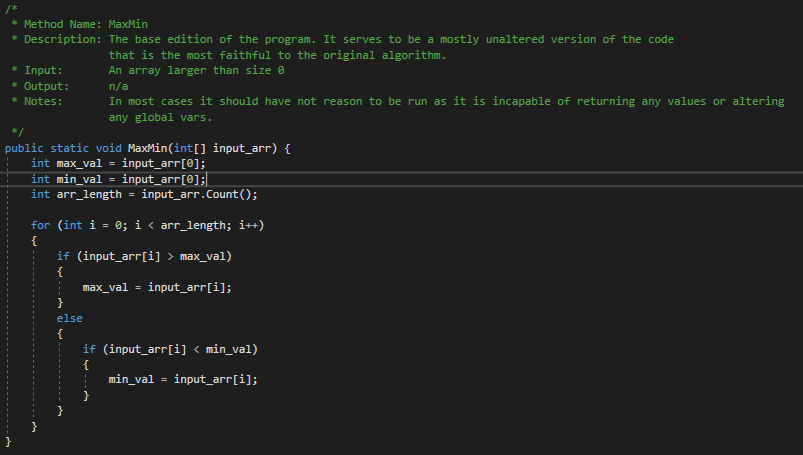


Figure 6: A plot graph where each plot represents the average of 100 tests where each test was the execution time of the program for an array of length n. The results confirm the order of growth ignoring the major deviations.

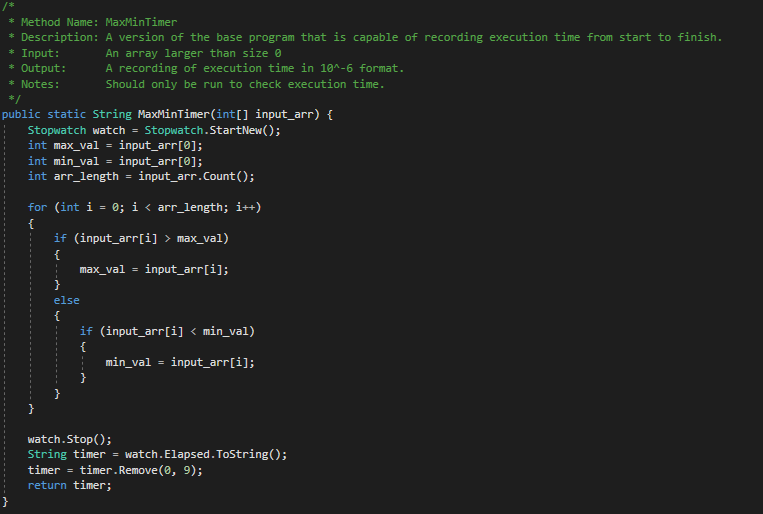
# Appendix A: MaxMin2 Program

The version of the program created as faithfully to the original algorithm as possible. Comparisons between the two has been outlined within the report.



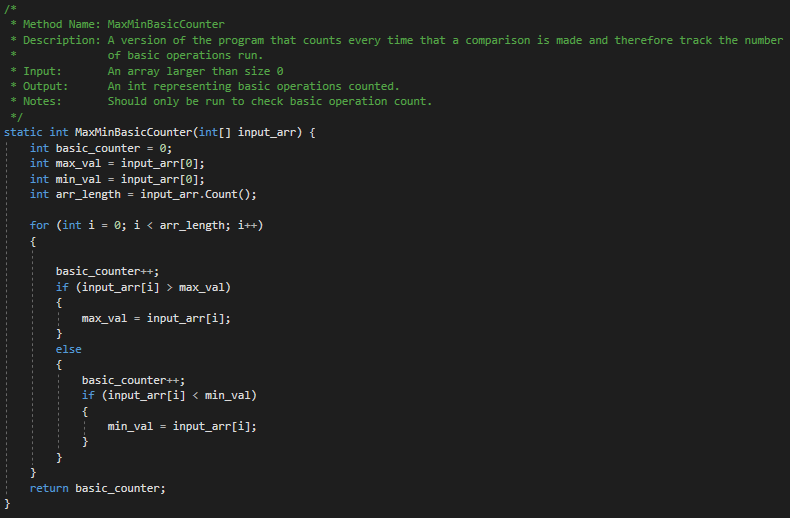
# Appendix B: MaxMin2 Timer

A version of the algorithm that records the time elapsed since the start of execution to the end using the stopwatch feature.



# Appendix C: MaxMin2 Basic Counter

A version of the algorithm that features a counter that increments every time that a basic operation in run.



# Appendix D: Set generator

The code used to generate a set of a specified size (or default to 10) using random numbers in the range of -10000 to 10000. While this may not be perfectly random, pseudo random in programs is often a sufficient replacement and has been implemented here as a result.

