Queensland University of Technology

CAB301 - Algorithms and Complexity

Assignment 1

Efficiency Analysis of Negative Before Positive Sorting Algorithm

---

Author:
Alice Hendicott
n9366164

April 15, 2018

1 Algorithm Description

This algorithm takes an input of an array of negative and positive numbers and sorts the array so that all negative numbers precede those that are positive. It is a two colour version of the dutch flag problem which involves sorting colours according to their order on the dutch flag [1]. However, instead of colours, we are using negative and positive numbers. The algorithm can be found in Appendix 1 on page

To sort the positive and negative numbers, there are two counters which to start, are set to the index of 0 (i) and n-1 (j) (where n is the length of the array). The algorithm will then check if the value in the array at the index of i is less than 0. If it is, the algorithm will move onto the next index for i (i = 1) as the value at i = 0 is in the correct place. Otherwise, the value will be swapped with the one at the index j and moves the index to j = j-1 because the value at j is now in the correct place. The algorithm continues in this way until the index for i is greater than or equal to j at which point all values have been sorted to the correct positions.

Essentially, this sorting algorithm splits the array into three sections. First we have the section $A[0....i-1]$ which contains values which are less than 0 (negative). The middle section of values $A[i....j]$ filled with unknown values of which the sorting algorithm is yet to check and sort. Finally, the third section $A[j+1...n]$ which consists of value which are greater than 0 (positive). The algorithm will continue until the middle section does not contain any values, meaning the array is completely sorted.

2 Theorised Algorithm Analysis

This section defines the basic operation and uses this to predict the average case efficiency as well as the order of growth.

2.1 Choice of Basic Operation

It is clear that the operation which will have the greatest influence on the time efficiency of the Negative Before Positive sorting algorithm is the action of swapping values in the array once they are found to be in the incorrect position. As seen in Appendix 1 displaying the algorithm, this operation is performed on line 7, in the **else** section of the **if...else** statement.

2.2 Predicted Average Case Efficiency

To find the average case efficiency we must first look at the best and worst case efficiency. The best case efficiency occurs when the inputted array is already correctly sorted (i.e. all negative numbers appear before those that are positive). When this is the case, the algorithm will only ever enter the first **if** statement within the while loop, meaning the basic operation will never occur. Therefore the minimum number of operations for this algorithm is 0.

The worst case efficiency occurs when the array inputted contains only positive numbers. In this scenario, every number in the array will need to be swapped. Hence, the maximum number of basic

[1] Rosettacode.org. (2018). Dutch national flag problem - Rosetta Code. https://rosettacode.org/wiki/Dutch_national_flag_problem

operations for this algorithm is n. Hence the average case efficiency will be $C_{avg}(n) = (0+n)/2 = n/2$.

A particular solution to the two-colour Dutch flag problem was claimed to require $(n-1)/4$ swaps in the average case [Colin McMaster, An Analysis of Algorithms for the Dutch National Flag Problem, Communications of the ACM, 21(10):842-846, October 1978]. Based on this we can predict that the Negative Before Positive Algorithm is not as efficient as this because $n/2 > (n-1)/4$. However, they both belong to the same average case efficiency class of $\Theta(n)$, meaning the difference in average case efficiency is not that great.

2.3 Order of Growth

The order of growth can be given in big O notation and represents the upper-bound of the algorithm case efficiency. As found above in section 3.3, the average case efficiency is $\Theta(n)$. An upper bound of this can be anything above n (i.e. n^2, n^3, 2^n, etc.). The order of growth needs to be the tightest possible upper-bound for the algorithm average case efficiency, and hence it is predicted that the order of growth can be represented as $O(n^2)$.

3 Implementation

This section outlines the implementation of the Negative Before Positive Algorithm using C# in Visual Studio. The function created to carry out the algorithm can be found in Appendix 2.

3.1 Computing Environment

The algorithm and all testing were created and implemented in Visual Studio as a console application using the programming language C#. Visual Studio is an integrated development environment (IDE). C# is an object-oriented language which has a syntax that is simple and easy to learn and will be recognisable to those who are familiar with C, C++ or Java [2].

The computer used to carry out all experiments was an Apple Macintosh fourth generation MacBook Pro laptop computer, running on the UNIX-based Mac OS X operating system. The test data for the experiments was produced using the .NET Random Class pseudo-random number generator. To calculate elapsed time of the experiments, the .NET Timer Class was used. To maximise the accuracy of this timer, the number of program executing simultaneously during experiments requiring timing was minimised.

Any graphs created were done so using Microsoft Office Excel. Excel is a spreadsheet software that can be utilised for calculations, graphing and much more.

3.2 Proof of Correctness through testing

To confirm the implementation in C#, a program was produced to test several different cases. This program takes a given array, applies the NegBeforePos sorting algorithm and outputs the sorted array. This will then be compared with original array to confirm this implementation correctly sorts the given array according to the aim of this algorithm. The program used can be seen and is further explained in Appendix 3.

[2] Microsoft. (2018). Introduction to the C# Language and the .NET Framework. https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework

Three main tests were executed which represent the extreme possibilities of an array input. First a random array of unsorted values was tested. Arrays in the correct and reverse orders were then tested and to conclude, arrays which were entirely positive and entirely negative were tested.

For the first test, the array: $[8, 2, -3, 6, -2, -5, 6, 10, -3, -7, 0]$ was inputted. The output of this, seen below, shows clearly that this array was sorted correctly with all negative numbers preceding those that are positive. In this algorithm 0 is distinguished as a positive number as determined in the **if** statement (**if** $A[i] < 0$) stating that all negative numbers are those less than zero and not including 0.

```
The original array: [8, 2, -3, 6, -2, -5, 6, 10, -3, -7]
The sorted array: [-7, -3, -3, -5, -2, 6, 10, 6, 2, 8]
```

For the second test, an array in reverse order (all positive numbers before negative numbers): $[2, 5, 6, 10, -3, -8, -4, 6]$. As it can be seen from the output of this test below, the algorithm once again has correctly sorted this array.

```
The original array: [2, 6, 5, 10, -3, -8, -4, -6]
The sorted array: [-6, -4, -8, -3, 10, 5, 6, 2]
```

Similarly, a test was conducted in which the array was already sorted: $[-5, -10, -12, -3, 7, 4, 8, 2, 15]$. From the output below, it can be seen that the correct result was produced.

```
The original array: [-5, -10, -12, -3, 7, 4, 8, 2, 15]
The sorted array: [-5, -10, -12, -3, 4, 8, 2, 15, 7]
```

4 Average Case Efficiency Analysis

This section first investigates the theoretical time complexity of the algorithm and then compares this to the calculated time efficiency based on the implementation of this algorithm in C#.

4.1 Counting the Basic Operation

To count the number of times the basic operation is performed within the algorithm, a counter is added to the algorithm. See the programming for this in Appendix 4. Each time the basic operation of swapping two values in the array is performed, the counter is incremented by 1. To check if the counter has been implemented correctly test cases for arrays of small n were tested and confirmed.

For instance, a test was performed on the array of numbers: [5, -2, -8, 3, 7, -4, 8] which returned the output below stating that the basic operation was performed 3 times. To check this, the algorithm is stepped out by hand on this array.

Iteration 1: 5 is swapped with 8 to produce [8, -2, -8, 3, 7, -4, 5], unchecked numbers: 8, -2, -8, 3, 7, -4
Iteration 2: 8 is swapped with -4 to produce [-4, -2, -8, 3, 7, 8, 5], unchecked numbers: -4, -2, -8, 3, 7
Iteration 3: -4 is in correct position, no swap is made, unchecked numbers: -2, -8, 3, 7
Iteration 4: -2 is in correct position, no swap is made, unchecked numbers: -8, 3, 7
Iteration 5: -8 is in correct position, no swap is made unchecked numbers: 3, 7
Iteration 6: 3 is swapped with 7 to produce [-4, -2, -8, 7, 3, 8, 5], unchecked numbers: 7
Iteration 7: 7 is swapped with 3 to produce [-4, -2, -8, 3, 7, 8, 5], array completely checked.

Therefore, the output below is correct that the total number of swaps is 4 for this array.

```
The original array: [5, -2, -8, 3, 7, -4, 8]
The sorted array: [-4, -2, -8, 7, 3, 8, 5]
The basic operation was performed: 4 times
```

4.2 Average Case Number of Basic Operations

To produce an experiment to find the average number of basic operations for a particular size, the program in Appendix 5 was used. In this experiment, a set size was given, for example an array of size 1000, for which 10 random arrays were created and sorted. The count of basic operations for each of these arrays is then averaged. This process is then repeated for several different array sizes until a clear trend is produced. The graph constructed to represent the data created in this experiment can be found in Figure 1 in Appendix 7.

It is clear from this graph that the total number of basic operations increases by half of the current size of the array. For example, when the array size is 1400, the total number of basic operations is 696 and when the size is 10000, the basic operation count is 4994. As a result of this, it can be concluded that the predicted average case efficiency suggested in Section 3.2 of $C_{avg}(n) = n/2$. Which can be represented as $\Theta(n)$. We can also see this from the linear relationship appearing on the graph.

Another graph (Figure 2 in Appendix 7) was produced to further investigate the difference in efficiency from the particular average case efficiency of the two-colour dutch flag algorithm, mentioned in section 2.2, with the Negative Before Positive Algorithm. On this graph, the blue line represents the experiment results, the red dotted line sitting over this represents the predicted average case efficiency of $n/2$. The grey dotted line represents the average case efficiency of $(n-1)/4$. Clearly, it can be seen that the growth of the experimental results is much greater than that of the two-colour dutch problem algorithm. This confirms our original prediction in section 2.2 that the Negative Before Positive Algorithm is less efficient that the particular two-colour dutch problem.

5 Execution Time Analysis

To find the execution time of the algorithm, a program was produced to measure the average execution time across several different sizes of n. The program runs the sorting program multiple times for increasing sizes of n, calculating the execution time using the Stopwatch function and averaging the times for each value of n. This program can be seen in Appendix 6.

A graph was then produced to represent the results found (Figure 3 in Appendix 7). From this graph, a clear trend appeared suggesting that the execution time can be represented by $n^2/10000000$. For example, when the array was of size 11000, the execution time was found to be 12.7 ($11000^2/10000000 = 12.1$), and when $n = 20000$, the execution time was 40.9 ($20000^2/10000000 = 12.1$).

A comparison to this formula can be seen in Figure 4 of Appendix 7. It can clearly be seen that the experimental results follow this trend. There is a slight variation between the data and this formula but this is likely due to the time efficiency of the computing environment and other such factors. This confirms the predicted order of growth $O(n^2)$ as suggested in Section 3.3.

6 Appendix 1

Below is the algorithm analysed in this report. The basic operation which will have the most influence on the algorithm's time efficiency can be seen on line 6 (swapping the values A[i] and A[j].

```
   Algorithm NegBeforePos(A[0..n − 1])
   //Puts negative elements before positive (and zeros, if any) in an array
   //Input: Array A[0..n − 1] of real numbers
   //Output: Array A[0..n − 1] in which all its negative elements precede
   nonnegative
1  i ← 0;   j ← n − 1
2  while i ≤ j do     //i < j would suffice
3          if A[i] < 0  //shrink the unknown section from the left
4              i ← i + 1
5          else         //shrink the unknown section from the right
6              swap(A[i], A[j])
7              j ← j − 1
```

7 Appendix 2

This appendix shows the function created using C# in Visual Studio to carry out the algorithm. This program follows the algorithm as closely as possible in order to generate the most accurate results for a time efficiency analysis for the Negative Before Positive Algorithm. The only difference is that no such function exists in C# which allows two values in an array to be swapped and hence this has been spread out over further lines, however the process is the same and hence should not affect the time efficiency.

```
/// <summary>
/// Sorts the an array to have all negative numbers preceeding the positive
    numbers
/// </summary>
/// <param name="Array">The inputted array to be sorted</param>
static void NegBeforePos(int[] Array) {
  //Set both counters - i to the first value of the array and j to
  //the final value
  int i = 0;
  int j = Array.Length - 1;

  //begin the while loop which will end once i is less than or equal to j
  while (i <= j) {
    //check if the value in the array at index of i is less than
    //0
    if (Array[i] < 0) {
      //increment i because the value at i is in the correct
      //position
      i++;
    } else {
      //otherwise, swap the values at index of i and j
      int firstValue = Array[i];
      Array[i] = Array[j];
      Array[j] = firstValue;

      //decrease the value of j as the value at this index is
                        //now in the correct position
      j--;
    }
  }
}
```

8 Appendix 3

This Appendix displays the program used to test the implementation of the Negative Before Positive algorithm.

```
/// <summary>
/// Prints an array on the console
/// </summary>
/// <param name="A">The array to be printed</param>
static void printArray(int[] A) {
  Console.Write("[");
  for (int i = 0; i < A.Length; i++) {
    if (i == 0) {
      Console.Write(A[i]);
    } else {
      Console.Write(", " + A[i]);
    }
  }
  Console.Write("]");
  Console.WriteLine();
}

static void Main(string[] args) {
  //CASE 1: sort array [-5, -10, -12, -3, 7, 4, 8, 2, 15]
  int[] A = { -5, -10, -12, -3, 7, 4, 8, 2, 15};

  //Output this array to the console.
  Console.Write("The original array: ");
  printArray(A);

  //sort the array according to the NegBeforePos algorithm
  NegBeforePos(A);

  //Output the array to the screen
  Console.Write("The sorted array: ");
  printArray(A);
  Console.WriteLine();
}
```

9 Appendix 4

This appendix shows the altered algorithm used to count the number of times the basic operation occurs while sorted an Array. Note that the changes to the code in Appendix 2 can be seen on lines 12, 29 an 37. On line 12 a counter is initialised as an integer starting at 0. The counter is then incremented on line 29 (every time the basic operation is performed). Finally, the final count is returned on line 37.

```
1  /// <summary>
2  /// Sorts the an array to have all negative numbers preceeding the positive
       numbers
3  /// </summary>
4  /// <returns>An integer value representing number of times the basic operation
5  /// has been carried out</returns>
6  /// <param name="Array">The inputted array to be sorted</param>
7  static int NegBeforePos(int[] Array) {
8    //Set both counters - i to the first value of the array and j to
9    //the final value
10   int i = 0;
11   int j = Array.Length - 1;
12   int count = 0;
13
14   //begin the while loop which will end once i is less than or equal to j
15   while (i <= j) {
16     //check if the value in the array at index of i is less than
17     //0
18     if (Array[i] < 0) {
19       //increment i because the value at i is in the correct
20       //position
21       i++;
22     } else {
23       //otherwise, swap the values at index of i and j
24       int firstValue = Array[i];
25       Array[i] = Array[j];
26       Array[j] = firstValue;
27
28       //increase basic operation counter by 1
29       count++;
30
31       //decrease the value of j as the value at this index is
32       //now in the correct position
33       j--;
34     }
35   }
36   //return the sorted array
37   return count;
38 }
```

10 Appendix 5

This appendix shows the program used to calculate the average number of basic operations to occur for arrays of size 100 to 2000 incrementing by 100 after each test. The results from this experiment are graph which can be found in Figure 1 in Appendix 7.

```
/// <summary>
/// Generate a random array of length n
/// </summary>
/// <returns>The array</returns>
/// <param name="n">Integer which represents the length of the array to be
    generated</param>
static int[] GenerateArray(int n) {
  Random rnd = new Random();
  int[] A = new int[n];
  for (int i = 0; i < n; i++) {
    A[i] = rnd.Next(100) - 50;
  }
  return A;
}

static void Main(string[] args) {
  //create a variable to represent the number of tests to average for each size n
  int numTestsPerSize = 20;

  //create an array to store the average values
  double [] average = new double[20];

  //go through each test
  for (int tests = 1; tests <= 20; tests++) {
    //generate size depending on which test is occuring
    int n = tests*100;

    //set up an array to store the counter values for each test
    int[] counter = new int[numTestsPerSize];

    //for each test
    for (int i = 0; i < numTestsPerSize; i++) {

      //generate a random array of size n
      int[] A = GenerateArray(n);

                        //sort the array according to the NegBeforePos algorithm
       counter[i] = NegBeforePos(A);
                }

                //find the sum of all counter values
    int sum = 0;
    for (int i = 0; i < counter.Length; i++){
      sum += counter[i];
    }

    //average the counter values and output the results
    average[tests - 1] = sum / numTestsPerSize;
    Console.WriteLine("The average running time for arrays of size " + n + " is: "
        + average[tests-1]);
  }
}
```

11 Appendix 6

This Appendix shows the program used to test the execution time of the Negative Before Positive Algorithm. The stopwatch class was used to measure this time taken to complete the function representing the algorithm. The data produced from this experiment are represented in a graph (Figure 2 in Appendix 7).

```
static void Main(string[] args) {
  //create a variable to represent the number of tests to average for each size n
  int numTestsPerSize = 20;

  //set up a Stopwatch variable
  Stopwatch sw = new Stopwatch();

  //Step through each size n
  for (int n = 1000; n <= 20000; n=n+1000) {
    double averageMilliSecs = 0;
    long totalMilliSecs = 0;

    //Step through each test for the current size n
    for (int i = 0; i < numTestsPerSize; i++) {
      long milliSecs = 0;

      //Generate a random array of size n
      int[] A = GenerateArray(n);

      //Start the stopwatch
      sw.Start();

      //sort the array using the NegBeforePos algorithm
      NegBeforePos(A);

      //Stop the stopwatch
      sw.Stop();

      milliSecs = sw.ElapsedMilliseconds;
      totalMilliSecs = totalMilliSecs + milliSecs;
    }

    //find the average time in milliseconds and output the results
    averageMilliSecs = totalMilliSecs * 1.0 / numTestsPerSize;
    Console.WriteLine("The average running time for arrays of size " + n + " is: "
        + averageMilliSecs);
  }
}
```

12 Appendix 7

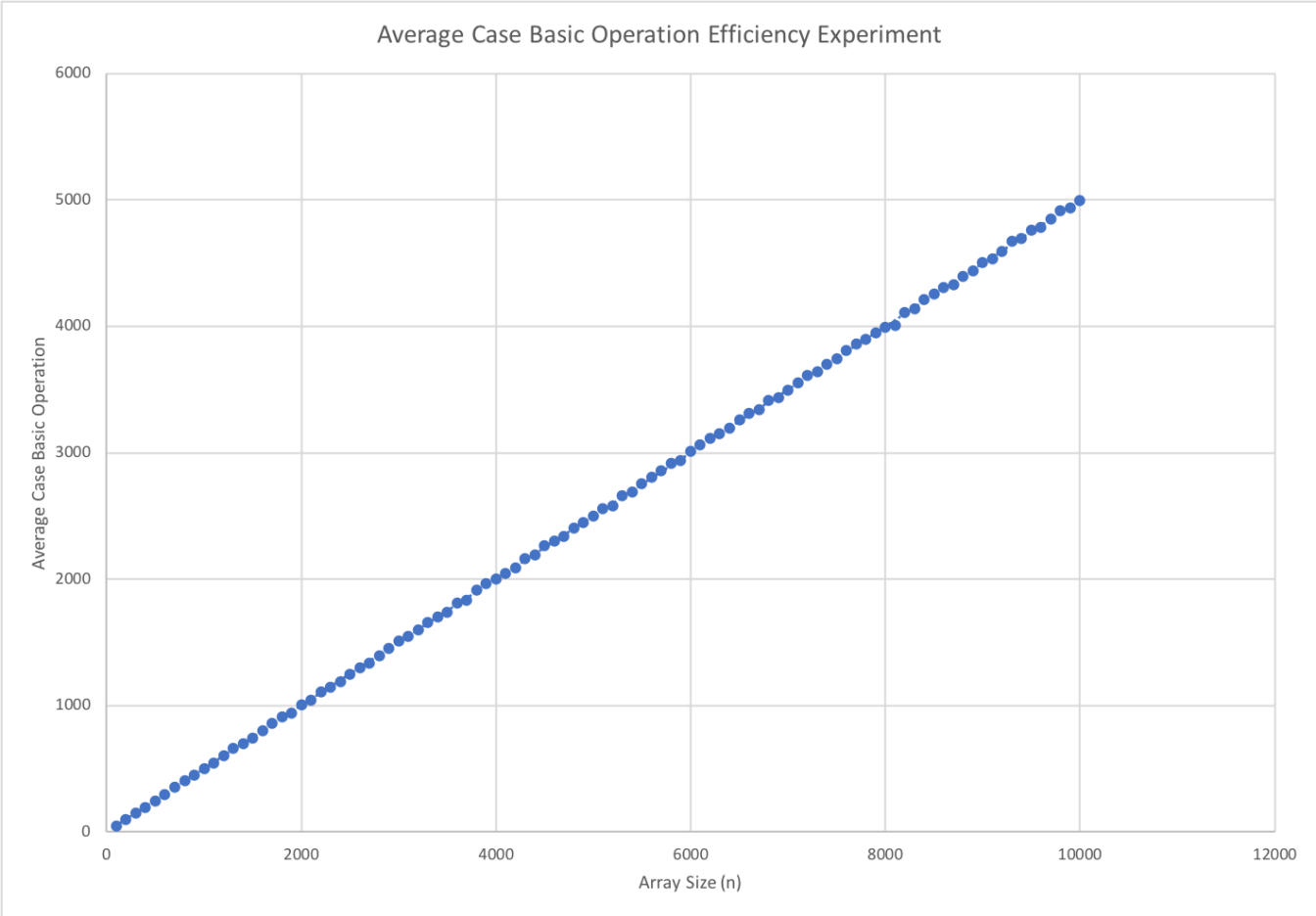

Figure 1: This graph presents the results of the experiment in section 4.2 It can clearly be seem from this graph that the average case efficiency is calculated by $C_{avg}(n) = n/2$ as suggested in Section 2.2. The trend in the graph is noticeable linear and hence the average case efficiency class is proven to be $\Theta(n)$, as theorised in also in Section 2.2. The line produced in the graph above is based on 100 data points evaluated using the program defined in Appendix 5.

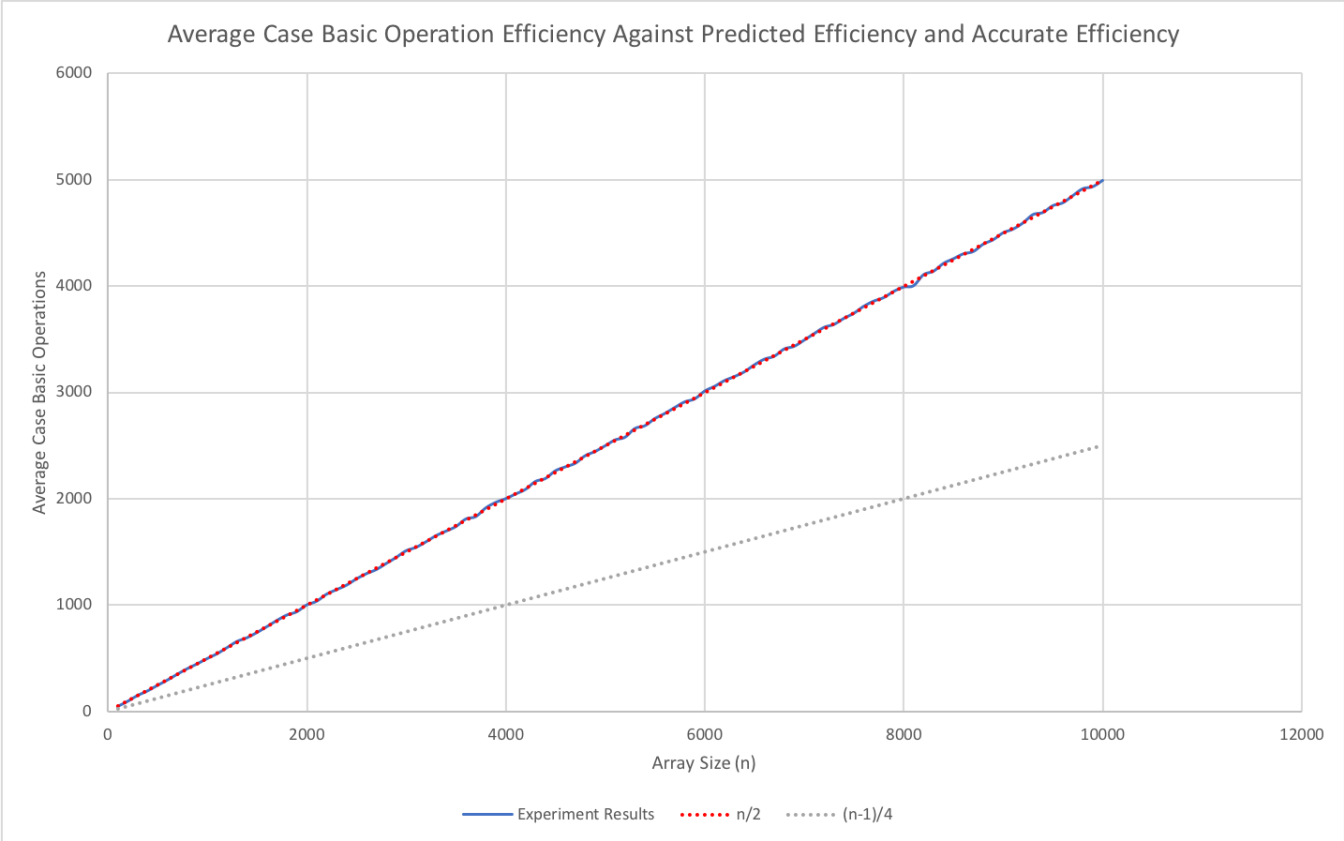

Figure 2: This figure compares the average case experimental results from Section 4.2 with the two average case efficiency formulas, $n/2$ and $(n-1)/4$. Clearly the more accurate formula is $n/2$ for the experimental results. Hence, the algorithm is less efficient then $(n-1)/4$. These 100 data points were produced using the program in Appendix 5.

Figure 3: This figure shows the average time efficiency experiment results from Section 5. It is clear the shape of this graph suggests a order of growth of $O(n^2)$ which matches with the prediction in Section 2.3. The formula which represents this data is $n^2/10000000$. These 20 data points were produced used the program in Appendix 6.

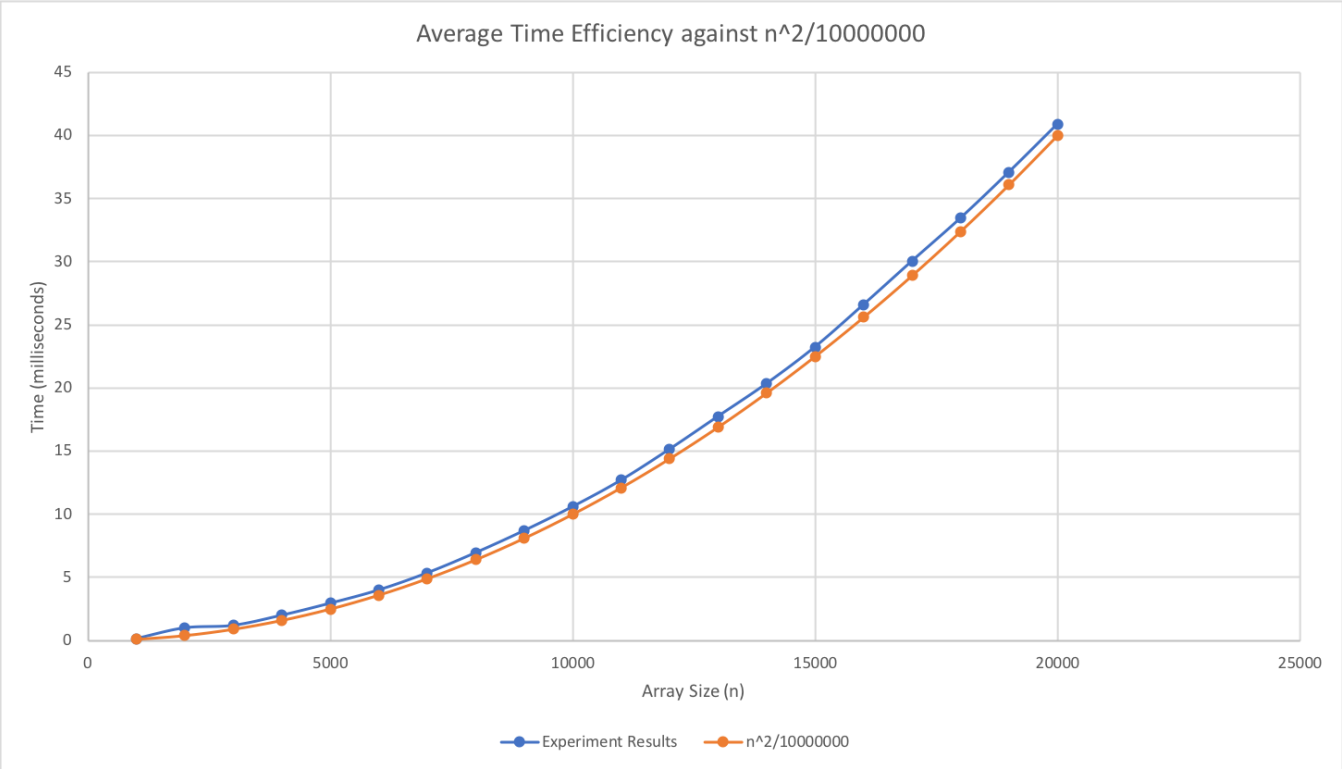

Figure 4: This figure shows the average time efficiency experiment results from Section 5 compared with the suggested formula $n^2/10000000$ which models the time efficiency of the algorithm. The slight variation between the two lines is likely due to the time efficiency of the computer environment.