CAB301 Assignment 1

Empirical Analysis of an Algorithm for Ordering Numbers In An Array

Student Name:
Morgan Frearson

Student Number:
N9510761

Due Date:
15th April 2018

Summary

The following report establishes a detailed analysis of the *Negatives Before Positives (Zeros)* algorithm. The algorithm orders numbers within an array based on whether the value is negative or positive, placing negative numbers or zeros before any positives in the array. The algorithm was implemented in C#, its complexity was measured and analysed using execution times and the counting of the basic operations. The analysis confirms that the experimental results were consistent with that of the theoretical predictions for this algorithm.

1. Description of the Algorithm

The algorithm being analysed shown in Figure 1, shifts numbers within a supplied array so that the negative values or zeros are placed before positive values. *NegBeforePos* achieves its desired result by looking for values greater than zero (positive integers) and shifts these values to the right-hand side of the array. The main pointer *i* starts from the left-hand side of the array on the 0^{th} item, this allows for the algorithm to check values within the array from left to right, similar to that of a sequential search algorithm. A second pointer *j* starts from the right-hand side of the array and is used as a reference point to swap values between itself and pointer *i*'s location. For each instance that the *i* pointer returns true to finding a positive integer; values at point *i* and point *j* are swapped thus the positive integer is now on the right-hand side of the array while the unknown value from *j* has been moved to location *i* for inspection during the next succession. After each instance swap, *j* is shifted one location towards the left, it is now safe to assume that any values on the right of *j* will be positive and are not required to be proven any further.

Unlike other sort algorithms which use a pivot point this algorithm shifts inwards based on its returned proof. Assuming the algorithm is only fed numerical values, it will break only when the *i* pointer has been incremented and the *j* point has been decreased to the same value.

The current algorithm is based off Levitin's design, a similar version of this algorithm called the 'Dutch National Flag problem' (Figure 2) proposed by Edsger Dijkstra is said to have a basic operation of swapping items, with an average case of *(n-1)/4* (McMasters, 1978). The difference between Levitin's design and Dijkstra's design is Levitin does not check the *j* pointer value and potentially causes a lack in efficiency with unnecessary shifts. In the analysis below we find that our algorithm (Levitin's) is less efficient than Dijkstra.

2. Theoretical Analysis of the Algorithm

The following section focuses on the algorithms expected time efficiency and theoretical predicted basic operation.

2.1 Identifying the Algorithm's Basic Operation

Most Time Consuming Basic Operation:

Majority of the execution time for the algorithm (Figure 1) is expelled on verifying the positivity or negativity of the passed array elements. The '*if*' statement within the algorithm is used to check that the current value is less than zero, therefore making it negative. This is the only comparison run upon every iteration making it the largest consumer of processing time. Although this '*if*' loop is situated inside a '*while*' loop, the condition of the while loop is assumed true until otherwise.

Most Frequently Used Basic Operation:

The '*while*' loop is an essential basic operation that allows for the algorithm to not exceed it size limitations. According to OCaml in an '*if*' statement the "test" is first, the "actions" are second, in a '*while*' statement, the "actions" precede the "test". The actions of the '*while*' loop within this algorithm continue until the "test" $i \leq j$ returns false resulting in a completed sort. All the actions of both Levitin and Dijkstra algorithms are dependent on this '*while*' loop remaining true, making it the most frequent operation.

Needed to Solve the Problem Basic Operation:

The significant basic operation we will focus on in theoretical terms is the *swap(A[i], A[j])*. This operation is used to rearrange the array to its desired form where *i* is the positive value being swapped with *j*. When the initial '*if*' statement fails the algorithm moves forward and the swap is performed. The swap is a single assignment statement in theoretical terms but a dual assignment in experimental terms (See Section 4).

2.2 Average-Case Efficiency

Within the *swap(A[i], A[j])* it is possible to visually examine the best and worst-case efficiency scenarios; Best Case the array contains only negative values and the swap is never performed, Worst Case the array contains only positive values and the swap is always performed. It can be said that the swap itself will be performed for '*n – (number of negative values)*' times as the '*if*' statement protects the basic operation from performing if the number being checked is already negative. As for the average-case, if *n* is the number of iterations being performed before the entire array has been checked and we assume that the distribution of numbers within the array is uniform (probability of positive or negative is ½), then the average number of swaps will be *n/2*. This can be proven as the left-hand position (*i*) will be incremented for each negative value and the right-hand position (*j*) will be decreased for each positive value therefore only half the array will use the swap operation under the average-case $C_{avg}(n) = \frac{n}{2}$.

2.3 Order of Growth

$i \leftarrow 0; j \leftarrow 1$	→ O(1)
$while\ (i \leq j)\ do$	→ O(n/2)
$if\ A[i] < 0$	→ O(1)
$i \leftarrow i + 1$	→ O(1)
$else$	→ O(1)
$swap(A[i], A[j])$	→ O(1)
$j \leftarrow j - 1$	→ O(1)

Levitin's algorithm has an average-case order of growth of $\Theta\left(\frac{n}{2}\right)$, the highest power of *n* within this is 1. Hence the time complexity will be O(n), algorithms that use this order of growth process all the items within the sequence making them a linear equation independent of the input array size.

3. Methodology, Tools and Techniques

1) Using the C# language in the Xamarin Studio environment the algorithm and experiments were executed. C# is an extension (newer) of C++ and also seen as a competitor to the Java language. For a high-level programming language, its simplistic syntax made it the ideal choice to implement the algorithm as faithfully as possible.

2) The experimental version of the algorithm was executed on a 2014 MacBook Pro running macOS High Sierra (10.13.3), a UNIX based operation system. C#'s random class was used to produce data from a pseudo-random number generator where numbers are chosen with equal probability from a finite set. For experiments using execution time the .NET API Stopwatch Class was implemented. For accurate results the installed hardware/OS of the Mac includes a high-resolution performance counter which by default the Stopwatch Class use over the simple system timer. Processing power can skew the results of the Stopwatch class even when running on a multiprocessor system thus care was taken when measuring execution times by not running any applications simultaneously.

3) Graphs of the experimental results were generated in Microsoft Excel. Helper code stated in Appendix D allowed for the data to be printed to the console screen, from there it was copied and pasted into an excel spreadsheet. Excel's transpose button allowed data to be easily sorted into rows and columns where it was then simply graphed. Figures 3 and 4 show the resultant of an Excel Graph which was then exported as a .jpg using OSX's inbuilt Grab tool.

4. Experimental Results

The following section shows the implementation of the algorithm in Figure 1. Several experiments were performed to compare to the theoretical predictions and analysis.

4.1 Functional Testing

For the functional testing of the algorithm several test cases were created using the code in Appendix C. The test code in Appendix C demonstrates a general test case for the algorithm to run. The sorted algorithm for an input array of {-1, 2, -3, 4, 5, 6, -7, 8, 9} was printed to the console as follows:

```
Array Before: -1 2 -3 4 5 6 -7 8 9
Array After: -1 -7 -3 5 6 4 8 9 2
```

The output shows that the array has been correctly sorted with positives shifted to the right, this can be proved because the position in the array of '-1' and '-3' has not changed. As the program is only simple it can be assumed that no non-numerical values will be inputted to the array, this allows the algorithm to save on processing time as no handler is needed.

To check the program was able to handle obscure test cases, the following complex experiments were run. The first was to check that the algorithm could handle a large input array, a random array of size 50 with values between -100 to 100 was created the output was as follows:

```
Array Before: -51 -17 35 -26 -91 -85 67 66 -94 7
60 -90 -18 61 67 -80 43 1 -41 49 40 -25 -50 -43 -51
12 71 97 -24 -98 -30 72 -42 -83 -25 -81 -38 -55 -36
```

```
7 96 25 -88 -23 -39 35 -72 9 -29 -73
Array After: -51 -17 -73 -26 -91 -85 -29 -72 -94
-39 -23 -90 -18 -88 -36 -80 -55 -38 -41 -81 -25
-25 -50 -43 -51 -83 -42 -30 -24 -98
72 97 71 12 40 49 1 43 7 96 25
67 61 60 35 7 9 66 67 35
```

At the other extreme a test was performed to check the algorithm could handle a small array of size 2:

```
Array Before: 2 -2
Array After: -2 2
```

In the following three tests the algorithm was checked for its ability to handle a singular integer type of negatives, zeros and positives. This allows us to observe if the array is unnecessarily shifting values to different positions.

```
Array Before: -4 -2 -9 -3 -30 -83
Array After: -4 -2 -9 -3 -30 -83

Array Before: 0 0 0 0 0 0
Array After: 0 0 0 0 0 0

Array Before: 3 7 21 93 22 7 9
Array After: 7 21 93 22 7 9 3
```

As shown in the positives test the algorithm shifts all the values one position to the left. This demonstrates the main difference between our algorithm (Levitin's) and the Dutch Flag Algorithm of Dijkstra. Dijkstra uses an *else if* statement in Figure 2 to evaluate if the number on the right-hand side of the array is already positive, if so it does not move position. The lack of this feature in Levitin's process creates an efficiency loss as it iterates through all instances of the array and constantly performs a shift due to line 25-26 shown in Appendix A.

In the final test case the algorithm was compared against an already sorted array:

```
Array Before: -2 -30 -19 4 6 47
Array After: -2 -30 -19 6 47 4
```

The output result of this test creates the desired result however continues to demonstrate Levitin's lack of efficiency. As the algorithm examines the positive right-hand side it needlessly shifts values within that section.

4.2 Average-Case Number of Basic Operations

To measure the average number of basic operations the code was optimized so the program would be readable and any errors could be distinguished. Appendix D shows how the test cases were created by using a basic counter. To gain accurate results 100 test cases were produced, the first 85 cases linearly increase the array size by 10 each time, for the last 15 cases the array size was exponentially increased doubling the previous array size for each test. This gave the algorithm vast results from arrays with a length as little as 10 to as large as 13959166 ($1.4x10^7$). Figure 3 graphs the experimental basic operational behaviour of the predicted theoretical *swap(A[i], A[j])*. It is clear to see that the number of basic operations is fundamentally half the size of the set hence the linear regression verified in the graph. For example, the number of basic operations required for:

109054 Set Size: 53888 Operations
480 Set Size: 240 Operations
40 Set Size: 21 Operations

This confirms the previous theoretical analysis that $C_{avg}(n) = \frac{n}{2}$.

4.3 Average-Case Execution Time

Measuring execution times throughout the algorithm proved difficult. Even for large arrays certain assignments and comparisons in the algorithm ran too quickly to measure accurately.
To evaluate the theoretical analysis against experimental results the $while\ (i \leq j)\ do$ loop was focused on using array sets of 500000+.
Appendix E shows how the execution time was measured for the block of code operated by the *while* loop (Line 14-31). The test data was gathered from 50 arrays, each with an increasing size of 500000. Results of Figure 4 exhibit a clear linear growth as predicted in section 2.3. This can be confirmed by strength of $R^2 = 0.99$, a high probability of linear growth thus O(n).
It can be assumed that the Order of Growth for the comparisons and assignments within the *while* loop are of constant O(1), confirming the theoretical analysis further. If otherwise, Figure 4 would not display the linear growth it currently does but rather be altered by the inner Order of Growth value. For example, if there was a O(n) loop inside the current O(n) *while* loop then the resulting graph for the *while* loop would be of $O(n^2)$.

References

McMasters, C. L. (1978). An analysis of algorithms for the Dutch National Flag Problem. *Communications of the ACM*, 842-846.

OCaml. (2017). *If Statements, Loops and Recursion*. Retrieved April 5, 2018, from OCaml: https://ocaml.org/learn/tutorials/if_statements_loops_and_recursion.html

Worcester Polytechnic Institute. (2017). *Computer Science* . Retrieved April 5, 2018, from Worcester Polytechnic Institute: http://web.cs.wpi.edu/~mebalazs/cs507/slides02/slides.html

Appendix A: Code for the Algorithm

The following code is the implemented version of the algorithm in Figure 1. The code contains a single class 'Program' and two methods 'NegBeforePos' and 'main'.

The first method NegBeforePos shown below encapsulates the entire algorithm. The assumed variable type for the method is an integer (32 bits), this is because it allows for time efficiency to be focused on the algorithm itself and not to be lost moving larger data types such as floats (64 bits).

Line 12 creates an empty variable '*val*' used to store a single integer when required. '*val*' is not discussed in the theoretical analysis of the algorithm because it is not present in the pseudocode however it is required in the experimental analysis and implemented in the code for the algorithm to perform its sort correctly. Line 18 initialises '*val*' to hold the current value of the array being evaluated at that point in time (always *array[i]*). In line 25-26 the *swap(A[i], A[j])* has been turned into a dual assignment statement moving the value from *array[i]* which was stored as '*val'* into *array[j]* and simply replacing *array[i]* with *array[j]* prior.

C# has an inbuilt method .Length which when invoked on an array returns the length of said array in the form of an integer. This method has been utilised to turn the original pseudocode '*n*' variable into a real number.

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.Linq;
 4 using System.Text.RegularExpressions;
 5
 6 namespace CAB301
 7 {
 8     class Program
 9     {
10         static void NegBeforePos(int[] array)
11         {
12             int val;
13             int j = array.Length - 1;
14             int i = 0;
15
16             while (i <= j)
17             {
18                 val = array[i];
19                 if (array[i] <= 0)
20                 {
21                     i = i + 1;
22                 }
23                 else
24                 {
25                     array[i] = array[j];
26                     array[j] = val;
27                     j = j - 1;
28                 }
29             }
30         }
```

The second method below is 'main'. This is the entry point of the program; its only action is to call the NegBeforePos method and pass it an array *arr* (highlighted in red because currently it is not being passed anything, see Appendix B for functional testing example).

```
38         public static void Main(string[] args)
39         {
40             NegBeforePos(arr);
41         }
42     }
43 }
```

Appendix B: Further Test Code and Readability

The following helper method was an extension to increase functionality. *printArray* takes in an input of type array and iterates through each element. Each element is printed to the console screen which allows for those viewing to see the contents of the array from left to right.

```
30          static void printArray(int[] arr)
31          {
32              for (int i = 0; i < arr.Length; i++)
33                  Console.Write(arr[i] + " ");
34          }
35
```

The *main* method below shows an example and test case for the full code. It passes the array *arr* to the *printArray* method which allows the user to read the unsorted array. The *NegBeforePos* method takes in the array *arr* which is sorted then reprinted to the console screen for the user to compare.

```
36          public static void Main(string[] args)
37          {
38              int[] arr = { -1, 2, -3, 4, 5, 6, -7, 8, 9 };
39              Console.Write("\nArray Before: ");
40              printArray(arr);
41              NegBeforePos(arr);
42              Console.Write("\nArray After: ");
43              printArray(arr);
44          }
45      }
46  }
```

Appendix C: Code for Functional Testing

The following *main* method presents how the test cases in Section 4.1 were generated. For general tests line 50 (currently commented out) allows for a human input array to be passed through the algorithm. Line 39-48 is a random array number generator, it creates an array of size 50 (Line 42) and generates random numbers (Line 47) between the bounds of -100 and 100 (Line 39-40). This random generator is further required in Appendix F for gathering data swiftly.

```
36          public static void Main(string[] args)
37          {
38
39              int Min = -100;
40              int Max = 100;
41
42              int[] arr = new int[50];
43
44              Random randNum = new Random();
45              for (int i = 0; i < arr.Length; i++)
46              {
47                  arr[i] = randNum.Next(Min, Max);
48              }
49
50              //int[] arr = {-2, -30, -19, 4, 6, 47};
51              Console.Write("\nArray Before: ");
52              printArray(arr);
53              NegBeforePos(arr);
54              Console.Write("\nArray After: ");
55              printArray(arr);
56          }
57      }
58  }
59
```

Appendix D: Code for Counting the Number of Basic Operations

The following code measured the number of basic operations performed for the swapping of elements. The random number array generator in line 43 is now implemented as a helper method this removes hardcoding from the *main* method when producing multiple data points. The counter placed on line 24 increases after each basic operation for that array.
To return the number of basic operations a public variable *basicOp* (line 6) is updated after each test allowing for access through the *main*. This could have been approached on a more sophisticated level by changing the return type of the *NegBeforePos* but this would affect the faithfulness of the algorithm with respect to its pseudocode.

```
1  using System;
2  namespace CAB301
3  {
4      class Program
5      {
6          public static int basicOp;
7
8          static void NegBeforePos(int[] array)
9          {
10             int val;
11             int basicOperations = 0;
12             int j = array.Length - 1;
13             int i = 0;
14
15             while (i <= j)
16             {
17                 val = array[i];
18                 if (array[i] <= 0)
19                 {
20                     i = i + 1;
21                 }
22                 else
23                 {
24                     basicOperations++;
25                     array[i] = array[j];
26                     array[j] = val;
27                     j = j - 1;
28                 }
29             }
30
31             basicOp = basicOperations;
32             Console.WriteLine(basicOperations);
33
34         }
35
36
37         static void printArray(int[] arr)
38         {
39             for (int i = 0; i < arr.Length; i++)
40                 Console.Write(arr[i] + " ");
41         }
42
43         static int[] randomGen(int size){
44             int Min = -100;
45             int Max = 100;
46
47             int[] arr = new int[size];
48
49             Random randNum = new Random();
50             for (int i = 0; i < arr.Length; i++)
51             {
52                 arr[i] = randNum.Next(Min, Max);
53             }
54             return arr;
55         }
```

The *main* below is set out to help the user read and understand the results the algorithm has produced. The two similar '*for'* loops are discussed in Section 4.2, the first creates linear arrays increasing by a size 10 from 10 to 850 and the second creates arrays doubling the size of the previous array from 850 to 27918334. The final two lines print the output to the console screen for the user to copy and graph (Figure 3).

```
62      public static void Main(string[] args)
63      {
64          int sizeOfArray = 0; // Holds the size of the array being tested
65          int[] graphDataSize = new int[100]; // Stores Set Sizes
66          int[] graphDataResults = new int[100]; // Stores Basic Operations
67
68          for (int i = 1; i < 86; i++) // Create 85 test cases
69          {
70              sizeOfArray = i * 10; // Linear Growth
71              int [] arr = randomGen(sizeOfArray); // Creates a random array with a size increasing by 10 each iteration
72              Console.WriteLine("\nTest Case: " + i + " Set Size: " + sizeOfArray + " Number of Basic Operations: ");
73              NegBeforePos(arr); // Run the algorithm
74              graphDataSize[i-1] = sizeOfArray; // Storing Data
75              graphDataResults[i-1] = basicOp; // Storing Data
76          }
77
78          for (int i = 86; i < 101; i++) // Create 15 test cases
79          {
80              sizeOfArray = (sizeOfArray + 1) * 2; // Exponential Growth
81              int [] arr = randomGen(sizeOfArray); // Creates a random array with a size doubling for each iteration
82              Console.WriteLine("\nTest Case: " + i + " Set Size: " + sizeOfArray + " Number of Basic Operations: ");
83              NegBeforePos(arr); // Run the algorithm
84              graphDataSize[i-1] = sizeOfArray; // Storing Data
85              graphDataResults[i-1] = basicOp; // Storing Data
86          }
87
88          Console.WriteLine("Array Size : "+ string.Join(",", graphDataSize)); // Allows us to copy the sets sizes from the console
89          Console.WriteLine("Number of Basic Operations : "+ string.Join(",", graphDataResults)); // Allows us to copy the number of
90                                                                                                  // operations from the console basic
91
92
93      }
94  }
95
```

Appendix E: Code for Measuring Execution Time

The code below follows a similar structure to that of Appendix D however a single counter has been replaced with a timer between point A and B. Using C#'s Stopwatch class a new Stopwatch object was initialised on line 14. The *watch* object counts in milliseconds the time it takes for the *while* loop to perform all its operations before it is stopped in line 30. The elapsed time is then obtained using the Stopwatch class's .ElapsedMilliseconds method casting it to the public variable *elapsedTime*.

```
4     class ExecutionTime
5     {
6         public static long elapsedTime; //Store time for public access
7
8         static void NegBeforePos(int[] array)
9         {
10            int val;
11            int j = array.Length - 1;
12            int i = 0;
13
14            var watch = System.Diagnostics.Stopwatch.StartNew(); //Start stopwatch
15            while (i <= j)
16            {
17                val = array[i];
18                if (array[i] <= 0)
19                {
20
21                    i = i + 1;
22                }
23                else
24                {
25                    array[i] = array[j];
26                    array[j] = val;
27                    j = j - 1;
28                }
29            }
30            watch.Stop(); //Stop Stopwatch
31            elapsedTime = watch.ElapsedMilliseconds; //Save time to public variable
32        }
```

```
59        public static void Main(string[] args)
60        {
61            long[] graphSetSize = new long[50]; // Stores Set Sizes
62            long[] graphTime = new long[50]; // Stores Time in Milliseconds
63
64            for (int i = 0; i < 50; i++) // Create 50 test cases
65            {
66                int sizeOfArray = i * 500000; // Linear Growth
67                int [] arr = randomGen(sizeOfArray); // Creates a random array with a size increasing by 500000 each iteration
68                NegBeforePos(arr); // Run the algorithm
69                graphSetSize[i] = sizeOfArray; // Storing Data
70                graphTime[i] = elapsedTime; // Storing Data
71
72            }
73            Console.WriteLine("Array Size : "+ string.Join(",", graphSetSize)); // Allows us to copy the sets sizes from the console
74            Console.WriteLine ("Time in Milliseconds: "+ string.Join(",", graphTime)); // Allows us to copy execution time
75
76        }
```

```
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
i ← 0;   j ← n − 1
while i ≤ j do     //i < j would suffice
      if A[i] < 0  //shrink the unknown section from the left
         i ← i + 1
      else         //shrink the unknown section from the right
         swap(A[i], A[j])
         j ← j − 1
```

Figure 1: The algorithm to be analysed based on student number N9510761. Let *'n'* be the length of the array, where *'n-1'* is the computed length, as arrays start from 0. Assume the array will only contain numerical values.

```
procedure three-way-partition(A : array of values, mid : value):
    i ← 0
    j ← 0
    n ← size of A - 1

    while j ≤ n:
        if A[j] < mid:
            swap A[i] and A[j]
            i ← i + 1
            j ← j + 1
        else if A[j] > mid:
            swap A[j] and A[n]
            n ← n - 1
        else:
            j ← j + 1
```

Figure 2: Pseudo code for The Dutch National Flag problem

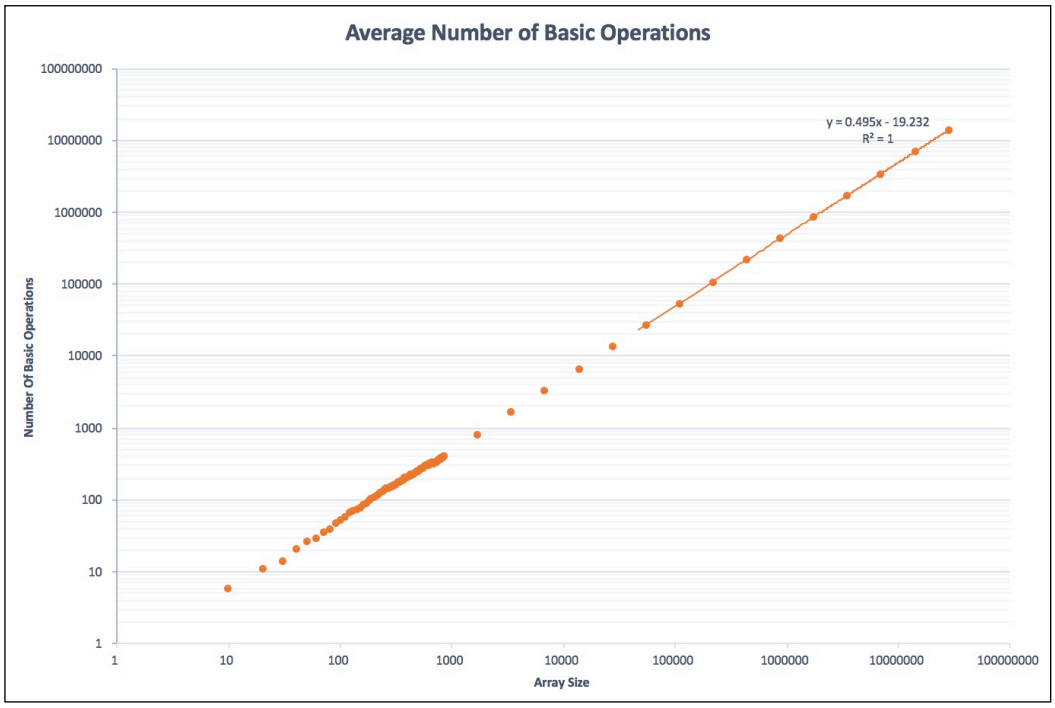

Figure 3: The number of basic operations performed verses the size of the input array. One hundred data points are displayed with an R^2 value of 1, verifying literality. The graph has a gradient of 0.495 supporting the theoretical average case of $\frac{n}{2}$ (0.5) with a negligible 1% standard error.

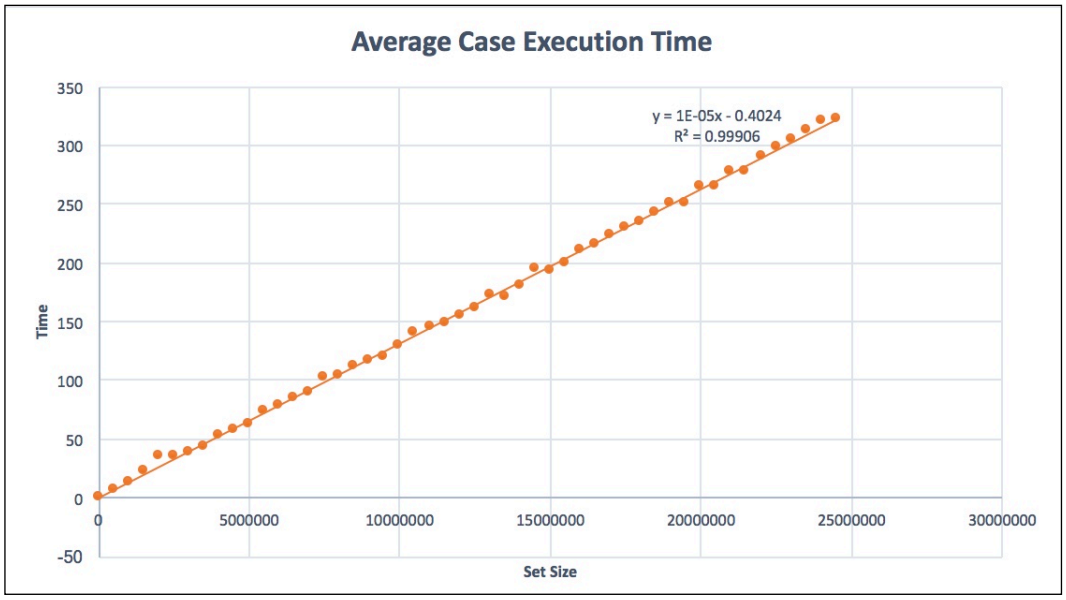

Figure 4: The execution time verses the size of the input array. Fifty data points are displayed with an R^2 value confirming a 99.9% linear growth relationship.