**CAB301 Assignment 1**

**Empirical Analysis of an Algorithm for Negatives before Positives (and Zeroes)**

**Student name: Saewoon Kim**

**(Student no. n10020772)**

**Date submitted: 15 April 2018**

# **Summary**

This report summarises the outcomes of several experiments conducted to measure the time complexity of an algorithm for Negatives before Positives (and Zeroes). The algorithm was implemented as a C++. Both the number of basic operations performed by the program and its execution time were measured. The experimental results were found to be consistent with the theoretical predictions for this algorithm.

# **Description of the Algorithm**

The description of the algorithm describes negative before positive and zeros. The basic operation will compare negative, positive and zeros numbers swapping so the negative numbers will stay on the left side while the positive number will stay on the right and then zero which is pivot stays in the middle. There is a version of well-known algorithm. It is called ‘Dutch national flag problem’. In this algorithm code it uses ‘while loop’ in statement c in figure 2. This is to arrange the negative numbers that are orderly sorted. In the code the ‘while loop’ will stop when variable { i }is less than{ j } or equal to variable { j }. The ‘if’ code in statement d in figure 2 checks whether the number of index { i }in a array element is negative or not. The ‘else’ statement f in figure 2 will swap the index of { i }when in a array element. Index { j } in a array will also swap if { i } is a negative number.

The algorithm required to do this is a common textbook example. Negative number is a [real number](https://en.wikipedia.org/wiki/Real_number) that is [less than](https://en.wikipedia.org/wiki/Inequality_(mathematics)) [zero](https://en.wikipedia.org/wiki/0_(number)). Positive number is a higher than zero (Hodgkin, 2005). As you can see, the numbers are arranged randomly. It will sort to negative before positive (and zeroes).

Before sort

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| -10 | 0 | 5 | 1 | 4 | -8 | 3 | -2 |

After sort

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| -10 | -8 | -2 | 0 | 1 | 3 | 4 | 5 |

The numbers greater than zero, are described as positive numbers. We do not put a plus sign (+) in front of them because we do not need to since the general understanding is that numbers without a sign are positive. Numbers that are less than zero are known as negative numbers. These have a minus sign (−) in front of them to indicate that they are less than zero (for example Figuer1) (Hodgkin, 2005).

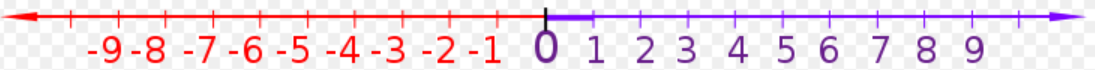


Figure1 number line.

# **Theoretical Analysis of the Algorithm**

This section describes the algorithm’s anticipated time complexity from a theoretical perspective.

## **Identifying the Algorithm’s Basic Operation**

## Identifying the Algorithm of Basic Operation as shown a figure 2 explains about execution time from an array to sort out in orders that is all negative numbers. In basic operation the algorithm array elements swap and compare with one another. In figure 1 the code ‘if-else’ is a statement of d and f. In this code the basic operation is performed in each iteration of while loop which is controlled by variable I and variable j.

## 

## **Average-Case Efficiency**

“Although we could not find a description of this specific algorithm’s average-case efficiency in the literature, it is clear that the problem of finding a particular place in an ordered linked list is analogous to searching sequentially for a particular item in an array. They conclude that the average number of basic operations required to find an item in an array size *n* is (*n* -1)*/*4. Therefore, we similarly assume that *Cavg*(*n*) = (*n* -1)*/*4 for the set insertion algorithm” (McMaster, 1978).

**a^2 = n(n - 1)/4a**

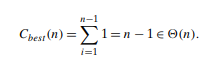


## **Order of Growth**

Input elements in an array according to their value is the order or O(n logn) based on its average-case efficiency (Levitin, 2011). Size of n grows when the number is larger. Numbers that are in array will allow negative numbers to occur first of order O (n) according to average-case efficiency used in this experiment. In order to grow for size of the array ‘straight line’ is expected to see linear.

Order of Growth sorting out the length of order in the table below:

N = C best(n) = n − 1 ∈ (n)



# **Methodology, Tools and Techniques**

This section briefly summarises the computing environment used for the experiments.

1. The algorithm and the experiments were implemented in C++ programming language. C++ was developed based on C language and retains C as a subset. This language is a general purpose high level language with both procedural and object-oriented language feature (Savitch, 2002).
2. The experiments were performed on Surface pro3 with Intel Core i5-4300U, CPU 1.90GHZ 2.50GHZ and 8GB RAM, running the Windows 10 PRO operating system. Visual studio Community 2017 as used to measure execution times and basic operation.
3. Graphs of the experimental results were produced using Microsoft Excel. The experimental program recoded all the data in a text files, it is easy to copy and paste to Excel. The data was selected and used to generated graphs in Excel. Figures 0 and 0 were all prepared in Excel, and exported into the document as well.

# **Experimental Results**

This section describes the outcomes of the experiments and compares the results with the theoretical predictions from Section 2. The C++ programming language implementation of the algorithm from Figure 2 is shown in Appendix A, B, C, D and E.

## **Functional Testing**

The test program described in Appendix B, it has 5 array tested. The results from the 5 tests are shown. The below:

1. The Functional Testing code is tested in test1 which the array numbers are {-10, 0, 5, 1, 4, -8, 3, -2}

The array element number is are {-10, 0, 5, 1, 4,-8, 3, -2}

The set show array is { 8 }

The result show {-10, -8, -2, 0, 1, 3, 4, 5 }

2. The second code is tested in test2 which the array numbers is { -1 }

The array element number is {-1}

The set show array is { 1 }

The result show {-1}

3. The Functional Testing code is tested in test3 which the array numbers are {0, -1}

The array element number is are {0, -1}

The set show array is { 2 }

The result show {-1, 0}

4. The Functional Testing code is tested in test4 which the array numbers are {-1, -1, -1}

The array element number is are {-1, -1, -1}

The set show array is { 3 }

The result show {-1, -1, -1}

5. The Functional Testing code is tested in test5 which the array numbers are {-10, -9, -8, -7, -6, -5, -4, -3, -2, -1 }

The array element number is are {-10, -9, -8, -7, -6, -5, -4, -3, -2, -1 }

The set show array is { 10 }

The result show {-10, -9, -8, -7, -6, -5, -4, -3, -2, -1 }

## **Average-Case Number of Basic Operations**

The average-case number of basic operation range is from 100-5000 and the size will increase every 100 array. The test { i } setting random number .The Basic operation value range from 0 to 450. Coding is in (Appendix E Line) which will explain how the basic operation works.

|  |  |  |
| --- | --- | --- |
| Size |  | Basic Operation |
| 100 | == | 8 |
| 1300 | == | 109 |
| 3700 | == | 312 |
| 4900 | == | 413 |

## **Average-Case Execution Time**

## The average-case number of execution Time range is from 100-5000 and the size will increase every 100 array. The test { i } setting random number. Coding is in (Appendix G Line) which will explain how the execution time works

|  |  |
| --- | --- |
| Size | Execution Time |
| 100 | 0.000000443 (4.43E-07) |
| 1300 | 0.00000316 (3.16E-06) |
| 3700 | 0.00000909 (9.09E-06) |
| 4900 | 0.0000118 (1.18E-05) |

**References**

1. Levitin. A. (2011) *Introduction to the Design and Analysis of Algorithms. Addison-Wesley*, third edition. ISBN 10: 0-13-231681-1.
2. Savitch, W. (2002). *Absolute C++.* Addison-Wesley.
3. [Stewart, James B.](https://en.wikipedia.org/wiki/James_Stewart_(mathematician)); Redlin, Lothar; Watson, Saleem (2008). College Algebra (5th ed.). [Brooks Cole](https://en.wikipedia.org/wiki/Brooks_Cole). pp. 13–19. [ISBN](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [0-495-56521-0](https://en.wikipedia.org/wiki/Special:BookSources/0-495-56521-0).
4. Hodgkin, L. (2005). [*A History of Mathematics: From Mesopotamia to Modernity: From Mesopotamia to Modernity*](https://books.google.com/books?id=f6HlhlBuQUgC&pg=PA88). Oxford University Press. p. 88. [ISBN](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [978-0-19-152383-0](https://en.wikipedia.org/wiki/Special:BookSources/978-0-19-152383-0).

1. McMaster, C. (1978). *An Analysis of Algorithms for the Dutch National Flag Problem*, Communications of the ACM, 21(10):842–846.

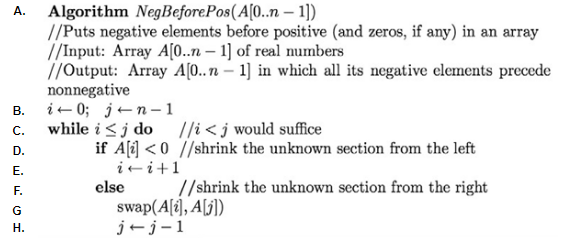


Figure 2 Negative Before Positive (and zeroes)

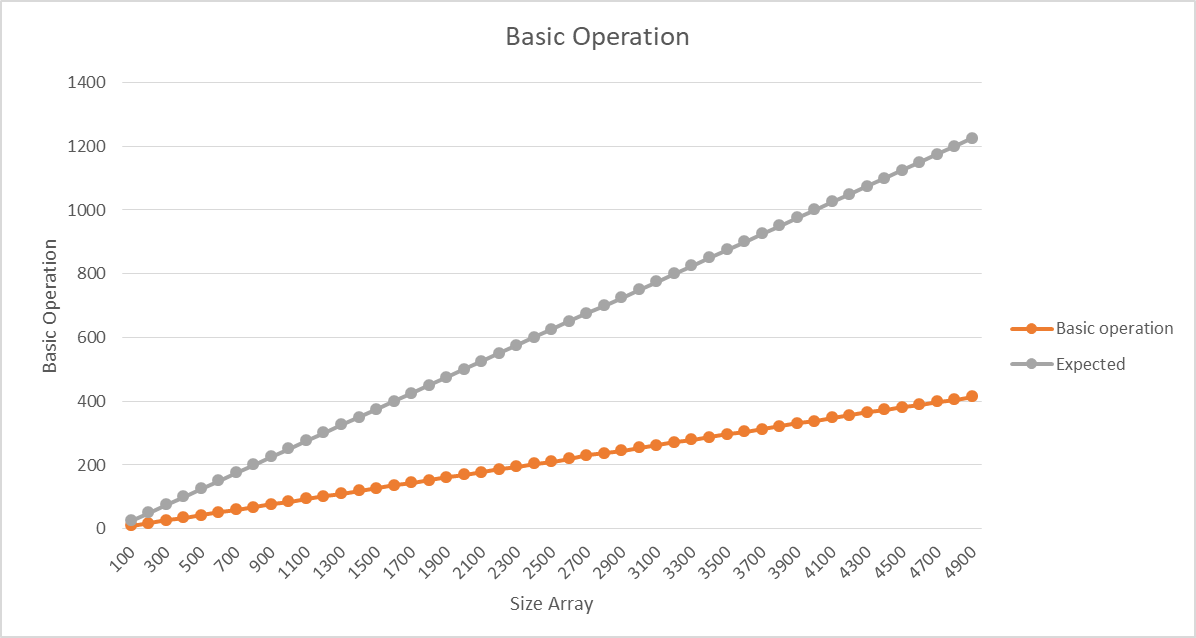


Figure 3 Basic Operation

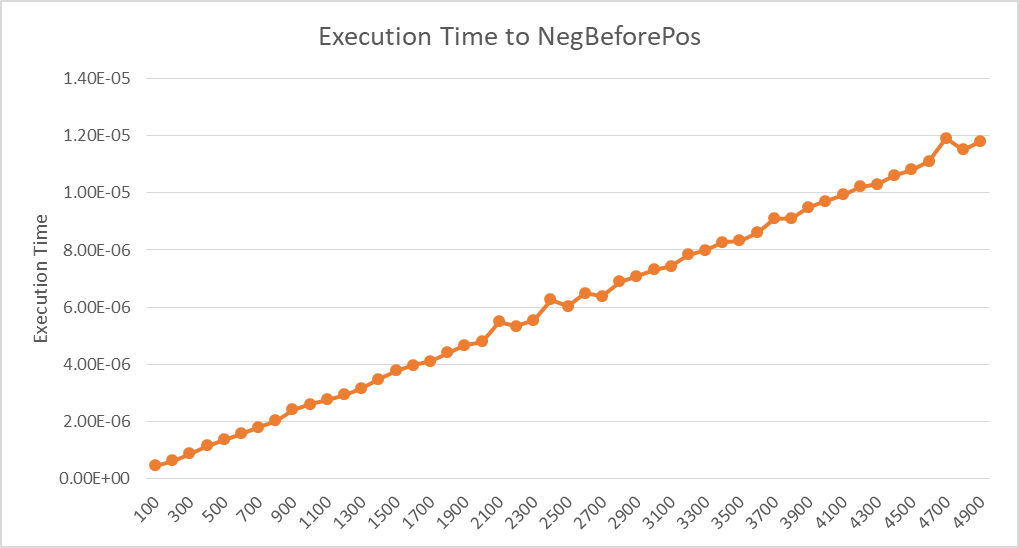
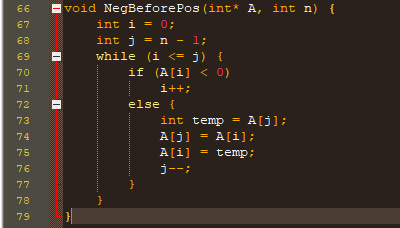


Figure 4 Execution Time

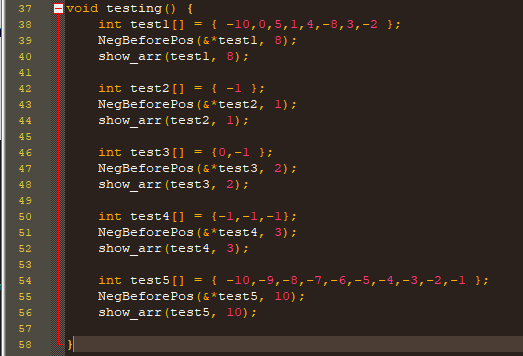
# **Appendix A Code for the Algorithm**

This appendix presents the C++ code written to implement the algorithm in Figure 2. The NegativeBeforePositive method method leads to the result that all negative numbers in an array occurs from left side and positive number being right side and zero is pivot. The two parameters in the method are an array (int\* A) and the size of the array (int n) line 73 -75 running swap function. There is no A[ J ] > 0. It will not swap A{ i } and A { j }.



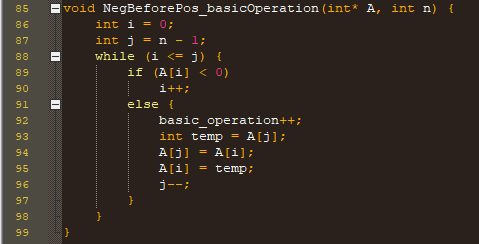
# **Appendix B Code for Functional Testing**

The following code was used to test the functional correctness of the algorithm’s implementation. This is the program that produced the output described in Section 4.1. All test arrays are rearranged, with all negative numbers being occurred left side and positive number being right side and zero is pivot.



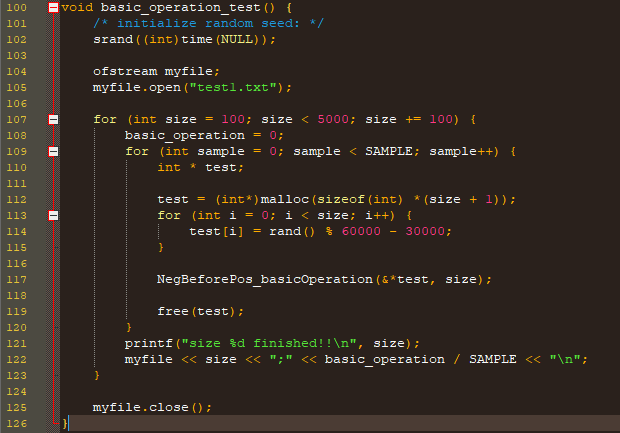
# **Appendix C Code for Counting the Number of Basic Operations**

To measure the number of basic operations (Section 2.1) performed by the algorithm it was first necessary to modify procedure ‘basic\_operation++’ (line 92) so that it increments a counter whenever a basic operation is executed. The ‘temp’ part is same as swap.



# **Appendix D Code for Averaging the Number of Basic Operations**

The following program was used to count the average number of basic operations needed to insert numbers into sets of different sizes. This was the program used to produce the results shown in section 4.2 and text1.txt file. The program is inserts an item into each set in the array at each iteration, counting the total number of basic operations for all the insertions (lines 108–121), and then prints the average number of basic operations to the results file (lines 105 – 106 and 123- 126).



# **Appendix E Code for Measuring Execution Times and average for Negative before Positive and Zeroes**

The following program was used to measure how long it takes to insert a randomly-generated number into a set using the algorithm in Figure 2. The program uses C++ clock procedure [138] to record the starting and finishing times for insertions and then calculates the difference between the two to determine the elapsed execution time. This is the program used to produce the results shown in section 4.3 and text2.txt file. For each set in the given range of sizes (lines 136), records the starting time (line 138 - 155), records the finishing time (line 155), and prints the average number of execution times to the results file (lines 133 – 134 and 154- 158).



