# CAB301 Algorithms and Complexity Assignment 1

Empirical Analysis of an Algorithm for Sorting Negatives Before Positives in a Set

Student name: Brendan Doncaster

(Student no. n9819363)

Date submitted: 10 April 2018

# Summary

This report summarises the results of experiments performed to measure the time complexity of an algorithm for sorting an array of numbers so that all negative numbers appear first. The program was implemented as a Java program which measured the execution time as well as the number of operations performed. The results of both experiments proved to be consistent with that of the theoretical prediction for this algorithm.

# Description of the Algorithm

The algorithm of interest sorts a set of numbers such that all negative numbers appear first in the set. The algorithm required to do this is a common textbook example, known as a two-colour Dutch flag solution. Levitin has this algorithm as a possible answer to a sorting challenge that he described (Levatin, 2006). The particular version of the algorithm used in these experiments is shown in figure 1. It works through the set from left to right and if the current number satisfies the criteria of being negative (less than 0), the algorithm removes it from the search area and moves onto to next number (line 3 and 4 in Figure 1). If the number does not satisfy being negative, the number is then swapped with that at the back of the search area of the set, and that number is removed from the search area(line 6 and 7 in Figure 1). The algorithm stops once the search area of the set only has a single number left in it.

# Theoretical Analysis of the Algorithm

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

## Identifying the Algorithm’s Basic Operation

The function within the algorithm in Figure 1 which would take the longest execution time would be the swapping of numbers. This function happens whenever a number that is 0 or higher is found at the front of the search area, unless it is the last number to sort. This function occurs on line 6 of Figure 1.

## Average-Case Efficiency

This precise version of the Dutch flag problem does not have it’s average-case efficiency described, however, a more efficient version by Colin McMaster is said to have an efficiency of (n-1)/4 (McMaster, 1978). The major difference between the two algorithms is that McMaster’s only swaps if the current element is positive and the last element is negative, whereas this version swaps whether the last element is negative or positive. Therefore, since the last element being negative or positive is purely random and therefore a 50% chance, the number of swaps would theoretically double. This means that this algorithm’s average number of basic operations would be ((n-1)/4) \* 2 or (n-1)/2.

## Order of Growth

With the average number of operations being (n-1)/2 it means that the sorting algorithm is of order Θ(n). Therefore, when measuring the experimental number of operations and execution time, a clear linear growth to the size of the set should be seen.

# Methodology, Tools and Techniques

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

1. The algorithm and the test experiments were implemented using the Java programming language.
2. The experiments were performed on a Windows Surface Pro 3, running the Windows 10 operating system. Java.util.Random was used to produce random test data and System.nanoTime was used to measure the execution time, with accuracy to the nanosecond. For experiments involving execution time measurements, the number of other software applications running concurrently with the tests was minimised.
3. Graphs and experiment results were produced using Microsoft Office 2016 Excel. The program can directly read tab separated text documents and format them as data points, therefore the program was designed to create a text file with such specifications. These data points can then be graphed using the in-built graphing feature. Figures 2 and 3 were produced using Excel and exported as images. This report was then prepared using Microsoft Word.

# Experimental Results

This section explains the experimental outcome of the algorithm being implemented compared to the theoretical predictions of it. The programmed implementation of the algorithm shown in Figure 1 is shown in Appendix A.

## Functional Testing

To test the functional correctness of the program in Appendix A, the test program shown in Appendix C was used. The test program creates an empty set of a decided length and fills it with random numbers with a given maximum and minimum number possible. It displays this random array, sorts the array, then displays the new sorted array. A basic example where it is given a size of 10, a maximum number of 100 and a minimum number of -100 returned the result:

Initial numbers: [-42, -13, 29, 8, 35, -77, 34, -97, -27, 63]

Sorted numbers: [-42, -13, -27, -97, -77, 34, 35, 8, 63, 29]

This confirms that the set of numbers have successfully been sorted to have all negatives first.

## Average-Case Number of Basic Operations

To test the algorithm’s average-case of basic operations, the test program in Appendix D was used. This program creates multiple arrays of random numbers where the array size increments by 10 until it reaches a predefined max size, which was set to 1000 for the test. Each size of array is sorted 1000 times with a new random set of numbers each time and then the number of operations was averaged over the 1000 sorts. The results are shown in Figure 2. The results managed to turn out extremely clean on the first attempt and clearly show that the number of basic operations grows at half the rate of the set’s size, just as predicted in the theoretical analysis of the average-case efficiency. The largest divergence from the predicted (n-1)/2 within the results was at size 60 with an average operation count of 29.7273 whereas the predicted value would be 29.5, which is still only a 0.77% increase and well within reason when considering the randomness of numbers being sorted.

## Average-Case Execution Time

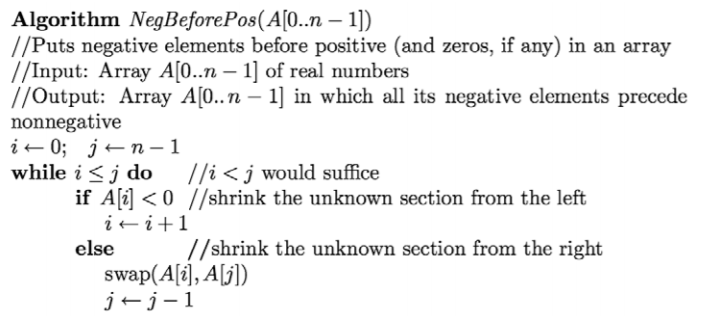
To test the algorithm’s average execution time, the test program in Appendix E was used. This program does the same as the average basic operations test, but times the section of the program that conducts the actual sorting algorithm. The results are shown in Figure 3. The results contained an extremely little amount of noise, likely due to each data point being averaged over 1000 tests, limiting randomness, as well as limiting all other programs running on the system as to reduce system congestions causing fluctuations in program speed. The graph shows a clear linear growth just as predicted in the theoretical order of growth.

# References

Levatin, A. (2006). Introduction to the Design and Analysis of Algorithms. 2nd ed.

McMaster, C. (1978). An Analysis of Algorithms for the Dutch National Flag Problem, Communications of the ACM

# Figures



1.  
2.  
3.  
4.  
5.  
6.  
7.

Figure 1: The algorithm to be analysed.

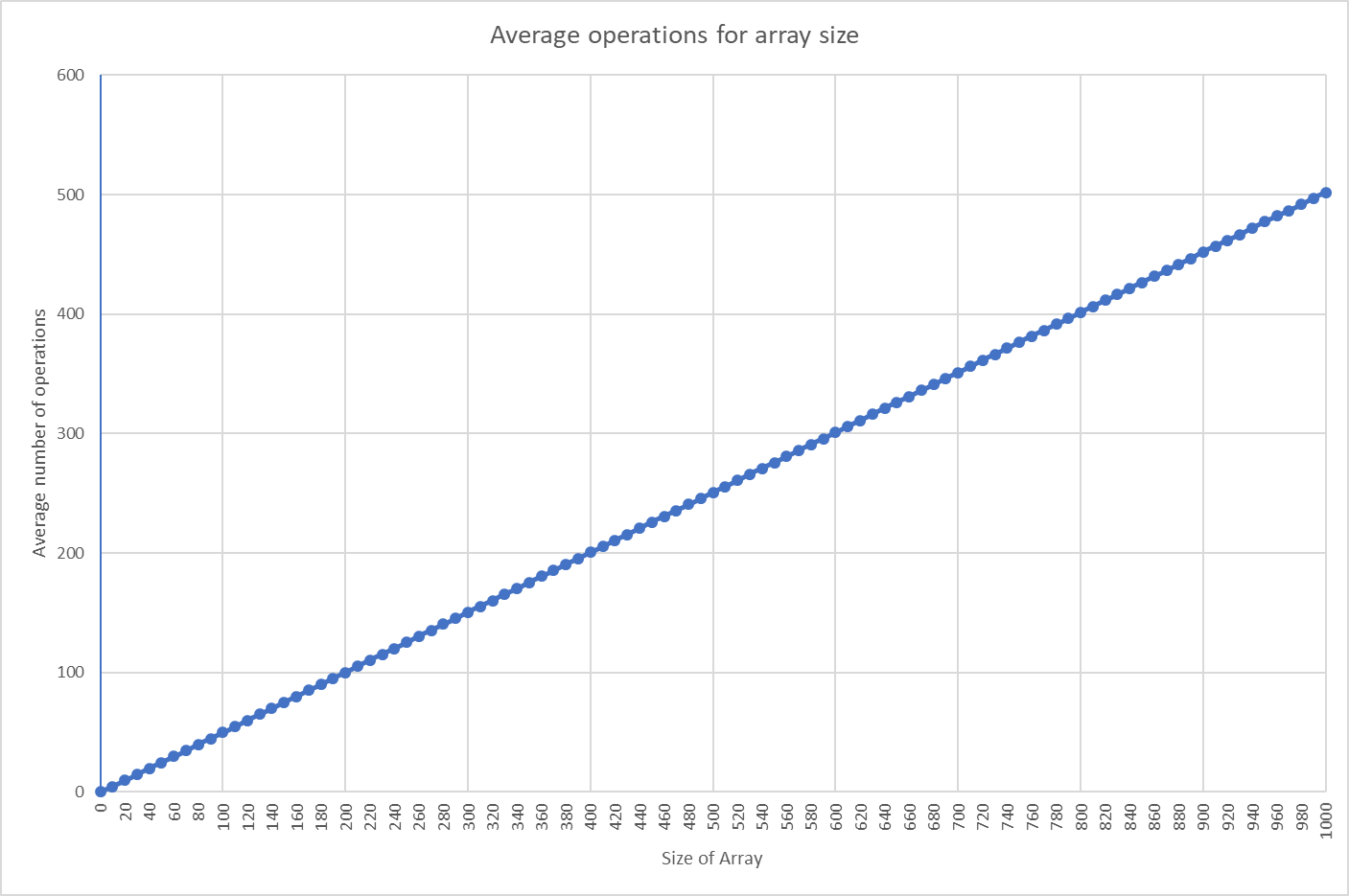


Figure 2: Average number of basic operations. This graph shows the measured number of basic operations required to perform sorting of random data (Appendix D). There are 101 data points, ranging from array size 0 to 1000 in increments of 10. Each point represents the average number of operations over 1000 tests. The data and graph confirm that the average operations taken to sort a random array increases linearly at (n-1)/2.

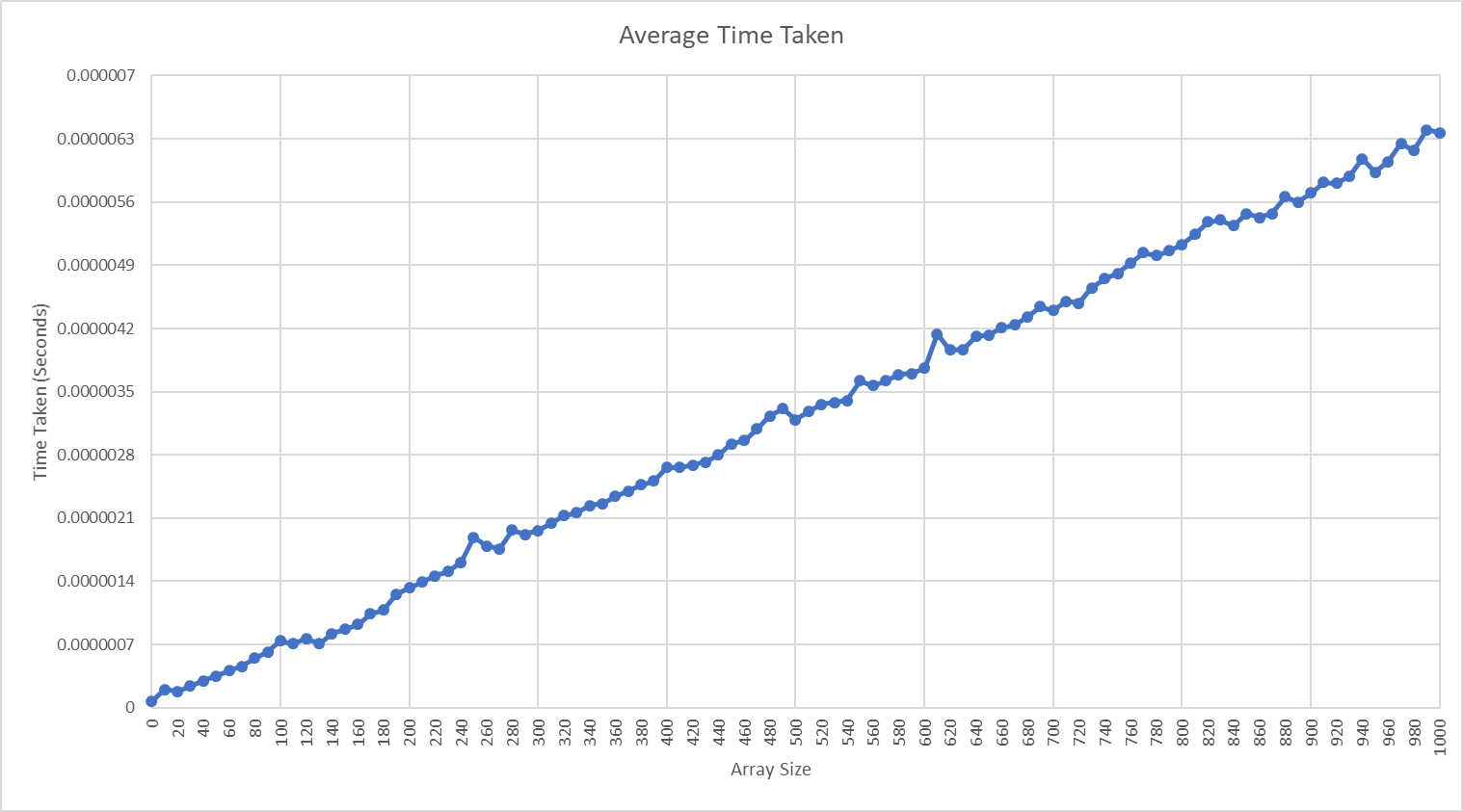


Figure 3: Average execution time for sorting an array. This graph shows the measured execution times for sorting an array of random numbers (Appendix E) . There are 101 data points, ranging from array size 0 to 1000 in increments of 10. Each point represents the average number of operations over 1000 tests. The result confirms that the execution time grows linearly with the size of the set.

# Appendices

## Appendix A: Code for the Algorithm

1. **public** **static** **void** NegBeforePos(**int**[] list){
2. **int** j = list.length - 1;
3. **int** i = 0;

6. **while**(i < j) {
7. **if** (list[i] < 0) {
8. i++;
9. }
11. **else** {
12. **int** temp = list[i];
13. list[i] = list[j];
14. list[j] = temp;
15. j--;
16. }
17. }
18. }

This appendix presents the Java code used to implement the algorithm in Figure 1. Basic arrays in Java do not have a swap function, therefore lines 13-15 create a temporary variable to set one of the elements to so that the two elements can swap positions.

## Appendix B: Setup code for Algorithm Use

1. **package** assignment1;
2. **import** java.io.FileNotFoundException;
3. **import** java.io.PrintWriter;
4. **import** java.text.DecimalFormat;
5. **import** java.util.Arrays;
6. **public** **class** Main {
7. //Set amount of runs to average
8. **private** **static** **int** *runs* = 10000;
9. //Set max size of Array
10. **private** **static** **int** *maxSize* = 1000;
11. //Set single size of Array
12. **private** **static** **int** *singleSize* = 10;
13. //Set highest possible value
14. **private** **static** **int** *maxNum* = 100;
15. //Set lowest possible value, must be less than maxNum
16. **private** **static** **int** *minNum* = -100;
17. //Set up names for each test
18. **private** **static** **enum** TestType {
19. ***AVERAGE***(),
20. ***TESTAVERAGE***(),
21. ***OPERATIONS***(),
22. }
23. //Set which test to run
24. **static** TestType *test* = TestType.***OPERATIONS***;
25. **public** **static** **void** main(String[] args) **throws** InterruptedException,

**36b.** FileNotFoundException {

2. //Initialize variables
3. **int**[] list;
4. String[] output;
5. String[] outputSize;
6. **int** currentRun;
7. **double** start;
8. **double** done;
9. **double** average;
10. **double** opAverage;
11. **double** operations;
13. //Sets output number as decimal not scientific notation
14. DecimalFormat df = **new** DecimalFormat("0");
15. df.setMaximumFractionDigits(8);
17. **switch**(*test*) {

This appendix presents the setup code for the 3 following tests in Appendices C, D and E. The choice of test is decided via an enumerator, created on line 26 and the test is chosen on line 33. Lines 10-23 decide the different variables used when initiating the tests. Lines 49 and 50 set a decimal format so that the files output with a decimal number for the time taken instead of a scientific notation which java would usually output. Line 52 opens the switch path for the enumerator so that the chosen test will be run.

## Appendix C: Code for Functional Testing

1. **case** ***TESTAVERAGE***:
2. //Print the initialization of a random array of specified size
4. list = **new** **int**[*singleSize*];
5. Algorithm.*RandFillArray*(list, *minNum*, *maxNum*);
6. System.***out***.println("Initial numbers: " + Arrays.*toString*(list));
8. //Print the transformation of the array
10. Algorithm.*NegBeforePos*(list);
11. System.***out***.println("Sorted numbers: " + Arrays.*toString*(list));
13. **break**;

This appendix presents the code to test the functional accuracy of the algorithm. Line 4 creates an array with the size determined during the code in Appendix B. Line 5 then uses the code in Appendix F to fill the array with random numbers. Line 10 performs the sorting of the array as of the code in Appendix A, and line 6 and 10 display the array before and after the sorting.

## Appendix D: Code for Counting the Number of Basic Operations

This appendix presents both the code for counting the number of basic operations, as well as the code for the test that outputs the average number of basic operations.

1. **public** **static** **int** NegBeforePosOps(**int**[] list){
2. **int** j = list.length - 1;
3. **int** i = 0;
4. **int** numOperations = 0;

7. **while**(i < j) {
8. **if** (list[i] < 0) {
9. i++;
10. }
11. **else** {
12. **int** temp = list[i];
13. list[i] = list[j];
14. list[j] = temp;
15. numOperations++; //For swap operation
16. j--;
17. }
18. }
19. **return** numOperations;
20. }

The only difference between this and Appendix A is line 16 which counts each time the swap function runs, and line 20 with outputs the final count of swaps performed.

1. **case** ***OPERATIONS***:
3. output = **new** String[1+(*maxSize*/10)];
4. outputSize = **new** String[1+(*maxSize*/10)];
5. currentRun = 0;
7. **for**(**int** s = 0 ; s <= *maxSize*; s += 10) {
8. opAverage = 0.0;
9. operations = 0.0;
10. list = **new** **int**[s];
12. //Run the algorithm for 'runs' amount of times with different inputs
13. **for** (**int** i = 0; i < *runs*; i++) {
14. Algorithm.*RandFillArray*(list, *minNum*, *maxNum*);
15. operations = Algorithm.*NegBeforePosOps*(list);
16. opAverage += operations;
17. }
18. //Average the number of operations
19. opAverage = opAverage / *runs*;
20. output[currentRun] = df.format(opAverage);
21. outputSize[currentRun] = Integer.*toString*(s);
22. currentRun++;
23. }
24. **try** (PrintWriter averageOut = **new** PrintWriter("Operations.xls")) {
25. averageOut.println("Size\tAverage operations");
26. **for**(**int** i = 0; i < output.length; i++) {
27. averageOut.println(outputSize[i] + "\t" + output[i]);
28. }
29. }
30. **break**;

This code creates a tab separated text file called Operations.xls with all the data used to create Figure 2. Line 3 and 4 create the arrays which will hold all the data to be output. Line 7 makes the program run for every size array up until the determined max size from line 14 in Appendix B. Lines 13-17 makes the array size have a new random array be sorted for the amount of times defined in line 11 of Appendix B, which is then averaged in line 19. Lines 24-29 output the data into the created file, with a tab between the size of the array and the average operations for the array, then a new line for each new array size.

## Appendix E Code for Measuring Execution Times

1. **case** ***AVERAGE***:
3. output = **new** String[1+(*maxSize*/10)];
4. outputSize = **new** String[1+(*maxSize*/10)];
5. currentRun = 0;
7. **for**(**int** s = 0 ; s <= *maxSize*; s += 10) {
8. average = 0.0;
9. list = **new** **int**[s];
11. //Run the algorithm for 'runs' amount of times with different inputs
12. **for** (**int** i = 0; i < *runs*; i++) {
13. Algorithm.*RandFillArray*(list, *minNum*, *maxNum*);
14. //Time sorting the array
15. start = System.*nanoTime*();
16. Algorithm.*NegBeforePos*(list);
17. done = (System.*nanoTime*() - start)/1000000000;
18. //Add all the times together
19. average += done;
20. }
21. average = average / *runs*;
22. output[currentRun] = df.format(average);
23. outputSize[currentRun] = Integer.*toString*(s);
24. currentRun++;
25. }
26. **try** (PrintWriter averageOut = **new** PrintWriter("Average.xls")) {
27. averageOut.println("Size\tAverage time taken");
28. **for**(**int** i = 0; i < output.length; i++) {
29. averageOut.println(outputSize[i] + "\t" + output[i]);
30. }
31. }
32. **break**;

This appendix presents code that creates a tab separated text file called Operations.xls with all the data used to create Figure 3. The main difference between this and Appendix D is lines 15-17. Lines 15 and 17 are used to time the sorting algorithm, and line 16 uses the algorithm code from Appendix A where the operations are not counted. The output variable created on line 3 is also storing the average execution time instead of the average operations.

## Appendix F: Code for Filling an Array

1. **static** Random *randNum* = **new** Random();
2. **public** **static** **int**[] RandFillArray(**int**[] list, **int** minNum, **int** maxNum) {
3. **for**(**int** j = 0; j < list.length; j++) {
4. list[j] = (*randNum*.nextInt((maxNum - minNum + 1)) + minNum);
5. }
6. **return** list;
7. }

This appendix presents code that fills a given array with random numbers between a given minimum and maximum number. The code implements java.util.Random in line 4 to create random integers and then they are transformed to abide by the max and min boundaries.