CAB301 Assignment 1

Empirical Analysis of an Algorithm: Bubble Sort (Efficient Version)

Student name: n9602976

Date submitted: 22nd April 2016

Summary

This report is a summary of the empirical analysis of the Bubble Sort Algorithm (Efficient Version). The report includes the average result of several experiments conducted to prove that the efficient version of the Bubble Sort Algorithm sorts an array of different lengths filled with ‘random’ values better than a normal Bubble Sort Algorithm. The algorithm is implemented using a C++ Console Application, with both the number of basic operations performed and the execution time of the program being measured. The experimental results show that the improved version of the Bubble Sort performs only **slightly** better than a normal Bubble Sort.

1. Description of the Algorithm

The basic mechanic of this Bubble Sort algorithm is exchange elements which are adjacent to each other in an array if they are out of order either by ascending or descending [1, pp. 100] and exit the loop when the array is fully sorted. This process begins with a while loop which checks a ‘swap flag’ for **true** before proceeding to a for loop which iterates through the array and comparing the current iterating element in an array with the element after it. When the element after the current is smaller than the former, it indicates that the current element is out of place. The current element switches place with the element after it. The ‘swap flag’ is set **true** to indicate that elements have been swapped in the array. This process is repeated where the biggest element will be moved and finally be placed in the last position of the array. Once the biggest element is placed, a new loop will reset the ‘swap flag’ back to **false** and will iterate until before the largest element. This is repeated where the second largest element will be placed at the second last in the array, the third largest element being placed at the third last in the array and so forth, until the swap flag is still remain **false** after the looping [1, pp. 100].

2. Theoretical Analysis of the Algorithm

This section describes the algorithm's predicted (theoretical) average-case efficiency from a theoretical perspective.

2.1 Identifying the Algorithm's Basic Operation

The Basic Operation of the algorithm is identified by observing the part of the algorithm that has the largest effect on the execution time of the program.

The basic operation of the normal Bubble Sort algorithm in Figure 1 has one basic operation which one is situated at **condition c**, where a basic operation is counted every time the inner loop iterates.

From Figure 2 which is the efficient Bubble Sort algorithm, a basic operation is identified at **condition f** which will increment whenever the loop iterates.

Since this report is about comparing the improved version of Bubble Sort versus the normal version of the Bubble Sort, any other operations of the efficient Bubble Sort algorithms within the while loop at **condition d** such as *sflag* ← false at **condition e**, **if** $A[j + 1] < A[j]$ at **condition g**, *swap* $A[j]$ and $A[j+1]$ at **condition h**, *sflag* ← true at **condition i**, and *count* ← *count* – 1 at **condition j** in Figure 2 will be left out since the operations are equivalent to the nested for loop at **condition b and c**, and the comparison of adjacent elements at **condition d** in Figure 1 with the exception of 'sflag', as these operations are insignificant in the complexity analysis.

2.2 Average-Case Efficiency

A normal Bubble Sort basically has an outer loop which iterates through the array of n elements from the element (0) to the second last element (n-2) and an inner loop which iterates through the array from element (0) to the element of (n-2-outer loop current element). It does not matter if the small elements are placed at the end of the list. This is because the loop will iterate **n** times due to how the algorithm lack of the 'swap flag' which stop the iteration when all elements are arranged properly. According to Levitin [1, pp. 101], the algorithm has a **theoretical** average-case efficiency of:

Normal Bubble Sort Inner loop:

$$\sum_{j=0}^{n-2-i} 1 = 1.\big((n-2-i)-j+1\big) = 1.(n-i-1)$$

Normal Bubble Sort Outer loop:

$$\sum_{i=0}^{n-2} 1.(\mathrm{n}-\mathrm{i}-1) = 1.[(n-1)+(n-2)+\cdots+1] = 1.(\frac{(n-1)\big((n-1)+1\big)}{2})$$

$$= 1.\frac{(n-1)n}{2} = \frac{(n-1)n}{2}$$

Theoretical Normal Bubble Sort Algorithm's average case efficiency:

$$C_{avg}(n) = \frac{(n-1)n}{2} \approx \frac{n^2}{2} \ \epsilon\ \theta(n^2)$$

For the efficient Bubble Sort, it has an additional function which is when the looping pass through the array and no occurrence of swapping elements, thus indicating that the array list has already been sorted, and this function is done with a 'swap flag' [2].

Theoretically, the efficient Bubble Sort, in its average case, will still be expected to have some small elements placed at the end of the array list. Due to only the largest unsorted elements gets placed in each proper index in each iteration, with the small elements being situated at the near end of the array, these small elements will only be able to reach the front at approximately slightly less than **n** iterations [3]. This results in the algorithm to have a **theoretical** average-case efficiency of:

Efficient Bubble Sort **for** loop, where *count* = *n*-1 and assume *i* increment by 1 every time is equivalent to *count* decrease by 1 each time **for** loop iterates whole loop:

$$\sum_{j=0}^{n-1-1-i} 1 = (n-1-1-i) - j + 1 = n - i - 1$$

Efficient Bubble Sort While loop, in average case scenario, where 'sflag' turns false when the array is sorted:

$$\sum_{i=0}^{n-2} (\mathrm{n} - \mathrm{i} - 1) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)((n-1)+1)}{2} = \frac{(n-1)n}{2}$$

Theoretical Efficient Bubble Sort Algorithm's average case efficiency:

$$C_{avg}(n) = \frac{(n-1)n}{2} \approx \frac{n^2}{2} \ \epsilon\ \theta(n^2)$$

2.3 Order of Growth

Based on the average case efficiency equation in Section 2.2 above suggests that the sorting of an array of size n using a normal Bubble Sort will result in an efficiency of n^2. The improved Bubble Sort also has an efficiency of n^2, but will still do slightly better than the normal Bubble Sort which can be proven at Figure 5. Thus, the order of growth of both Bubble Sort algorithms will be $\theta(n^2)$. The time to sort an array for both Bubble Sort will be a 'quadratic growth' with the size of the array.

3. Methodology, Tools and Techniques

This section is a summary about the choice of computing environment and methods used to produce test data for the experiment.

1. C++ programming language is used to implement both the algorithm and the test experiment as it is a simple to understand and read the language.

2. The experiments were performed on a Windows 10 HP Pavilion laptop computer.

3. A website name Learn C++ [4, Ch. 6.9a] is used as a reference to create a dynamic array. While i/o file function is referred from cplusplus.com as a guide to creating text file to save numbers of basic operations and time taken to execute the Bubble Sort algorithm [5].

4. The Bubble Sort algorithm is written separately as a function to make sure that the algorithm is written as similar as possible for the Efficient Bubble Algorithm given in the question paper and the Normal Bubble Algorithm from Levitin's book [1, pp.100]. Moreover, its makes the recording of the execution time of the algorithms more accurate.

5. The array is filled with random numbers with the help of srand () function which is used in conjunction with rand () [6].

6. Basic Operations and Execution Time are recorded in text files by the program. Average of Basic Operations, Average of Execution Time of the algorithms and Graphs of the experimental results were produced using Microsoft Excel. Data is manually copied from the text files to the Excel file, calculated and graphed using Excel. Figure 3 to 10 is generated from Excel.

7. 20 array sizes are used to test both algorithms in the experiment 10 times. The number of basic operations and execution time is recorded for each size of array. Results can be referred at Figure 5.

4. Experimental Results

This section describes the outcomes of the experiments and compares the results with the theoretical predictions from Section 2. The programming language implementation of the algorithm from Figure 1 and Figure 2 are shown in Appendix A.

4.1 Functional Testing

The program is tested to ensure the algorithm implemented in Appendix A is ready for testing. The functional test code described in Appendix B was used. Basically, the test program creates an array of random numbers in random indexes which the Bubble Sort will sort the array in ascending order.

Several different tests were performed on both algorithms and shown in Figure 3 and Figure 4 to make sure that the algorithm can sort the array properly.

For instance, in Figure 3 and Figure 4 both array produces an output where the elements in the array are placed orderly after the sort.

4.2 Average-Case Number of Basic Operations

Both the Normal Bubble Sort and the Efficient Bubble Sort used the code in Appendix C to calculate the number of basic Operations and the average is calculated using Excel.

From Figure 5 and Figure 6, it can be observed that total basic operations increase at a quadratic growth which fulfills the order of growth $\theta(n^2)$. When calculating the algorithm's **theoretical** average case basic operation:

$$C_{avg}(n) = \frac{(n-1)n}{2} \approx \frac{n^2}{2} \ \epsilon \ \theta(n^2)$$

E.g. Let *n* be 20000,

$$C_{avg}(n) = \frac{(20000-1)20000}{2} = 199990000 \ basic \ operations$$

Which actually equals exactly to the average number of Normal Bubble Sort basic operations when *n* = 20000 in the experiment.

Now, the Efficient Bubble Sort is assumed to do better than the Normal Bubble Sort. But when the Efficient Bubble Sort sorts the same array size *n* of 20000, its average total basic operation is 199950181.4, which does not outperform the normal one significantly.

The graph of average basic operations can be viewed in Figure 6 and 7.

4.3 Average-Case Execution Time

Both the Normal Bubble Sort and the Efficient Bubble Sort used the code in Appendix D to calculate the execution time of the algorithm and the average is calculated using Excel. The graph of both Normal and Efficient Bubble Sort Average Execution Time are shown in Figure 8 and 9.

Overall, the execution time results are consistent, the ratio of the number of Basic Operation versus the execution time are similar as can be seen in Figure 10.

From Figure 10, when comparing the ratio of basic operations to the execution time, the Efficient Bubble Sort does better than the Normal Bubble Sort. But again, the performance of the Efficient version is not significant enough.

4.4 Conclusion

Overall, the average-case efficiency of both Bubble Sort is similar, with the Efficient Bubble Sort performing only slightly better than the Normal Bubble Sort.

$$C_{avg}(n) = \frac{(n-1)n}{2} \approx \frac{n^2}{2} \; \epsilon \; \theta(n^2)$$

The conclusion is, the Efficient version of the Bubble Sort **does not significantly sort an array of different lengths filled with ‘random’ values faster** than the Normal version of the Bubble Sort.

Reference

[1] A. Levitin, *Introduction to the design & analysis of algorithms*. Boston: Pearson, 2012. ISBN-13: 978-0132316811

[2] D. Roberts, "Arrays in C++ - Bubble Sort", *Mathbits.com*, 2016. [Online]. Available: http://mathbits.com/MathBits/CompSci/Arrays/Bubble.htm. [Accessed: 22- Mar- 2016].

[3] "SparkNotes: Bubble Sort: The Bubble Sort Algorithm", Sparknotes.com, 2016. [Online]. Available: http://www.sparknotes.com/cs/sorting/bubble/section1.rhtml. [Accessed: 24- Mar- 2016].

[4] "6.9a — Dynamically allocating arrays", Learn C++, 2015. [Online]. Available: http://www.learncpp.com/cpp-tutorial/6-9a-dynamically-allocating-arrays/. [Accessed: 24- Mar- 2016].

[5] "Input/output with files - C++ Tutorials", Cplusplus.com, 2016. [Online]. Available: http://www.cplusplus.com/doc/tutorial/files/. [Accessed: 24- Mar- 2016].

[6] "srand - C++ Reference", *Cplusplus.com*, 2016. [Online]. Available: http://www.cplusplus.com/reference/cstdlib/srand/. [Accessed: 24- Mar- 2016].

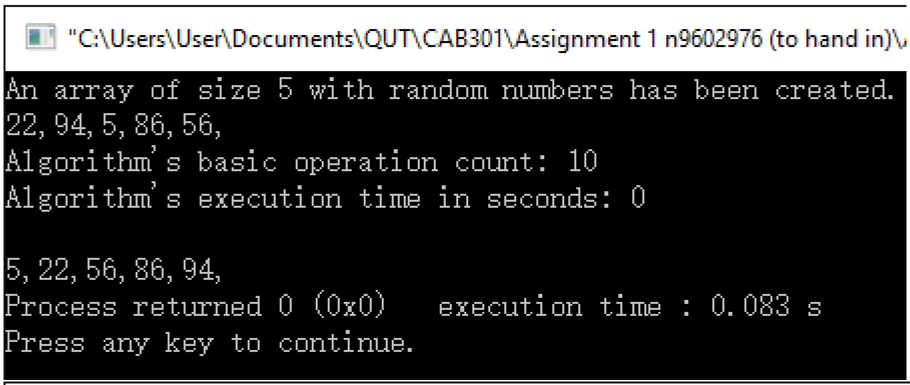
[7] "clock - C++ Reference", Cplusplus.com, 2016. [Online]. Available: http://www.cplusplus.com/reference/ctime/clock/. [Accessed: 25- Mar- 2016].

```
a. Algorithm BubbleSort (A[0..n − 1])
   //Sorts a given array by bubble sort
   //Input: An array A[0..n − 1] of orderable elements
   //Output: Array A[0..n − 1] sorted in nondecreasing order
b. for i ←0 to n − 2 do
c.     for j ←0 to n − 2 − i do
d.         if A[j + 1]<A[j ] swap A[j ] and A[j + 1]
```

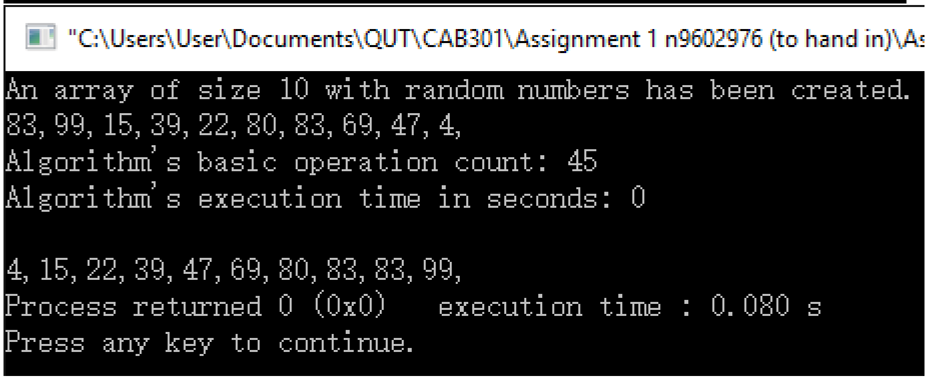
Figure 1: Normal Bubble Sort (Inefficient Version) algorithm [1, pp. 100] to be compared with the efficient version.

```
a. Algorithm BetterBubbleSort (A[0..n – 1])
   // The algorithm sorts array A[0..n – 1] by improved bubble sort
   // Input: An array A[0..n – 1] of orderable elements
   // Output: Array A[0..n – 1] sorted in ascending order
b. count ← n – 1 // number of adjacent pairs to be compared
c. sflag ← true   // swap flag
d. while sflag do
e.         sflag ← false
f.         for j ← 0 to count – 1 do
g.             if A[j + 1] < A[j]
h.                 swap A[j] and A[j+1]
i.                 sflag ← true
j.         count ← count – 1
```

Figure 2: Efficient Bubble Sort (Efficient Version) algorithm to be analyzed. Let *count* be the number of times the for loop need to loop. Let *swap* be the swapping of 2 values.

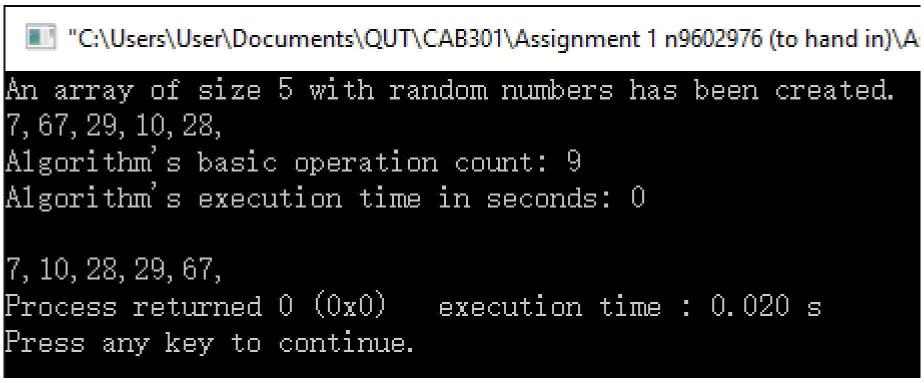


```
"C:\Users\User\Documents\QUT\CAB301\Assignment 1 n9602976 (to hand in)\
An array of size 5 with random numbers has been created.
22,94,5,86,56,
Algorithm's basic operation count: 10
Algorithm's execution time in seconds: 0

5,22,56,86,94,
Process returned 0 (0x0)   execution time : 0.083 s
Press any key to continue.
```

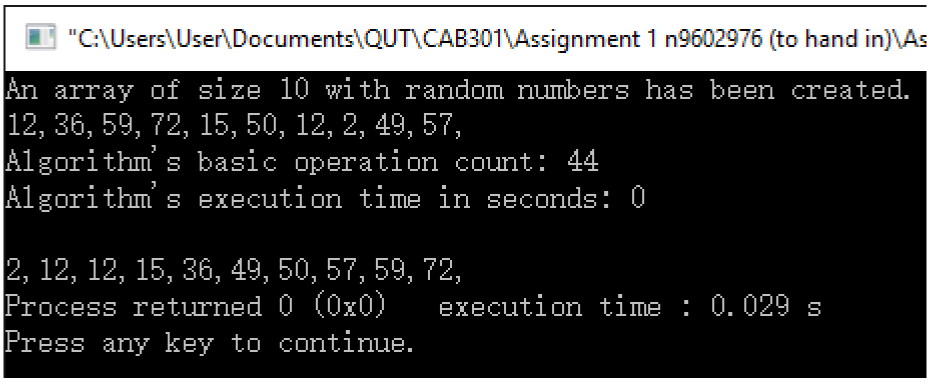


```
"C:\Users\User\Documents\QUT\CAB301\Assignment 1 n9602976 (to hand in)\A
An array of size 10 with random numbers has been created.
83,99,15,39,22,80,83,69,47,4,
Algorithm's basic operation count: 45
Algorithm's execution time in seconds: 0

4,15,22,39,47,69,80,83,83,99,
Process returned 0 (0x0)   execution time : 0.080 s
Press any key to continue.
```

Figure 3: Functional Testing of the Bubble Sort (Inefficient Version) algorithm. This is done to ensure the Bubble Sort can sort correctly. Here is an array of size 5 and size 10.



```
"C:\Users\User\Documents\QUT\CAB301\Assignment 1 n9602976 (to hand in)\A
An array of size 5 with random numbers has been created.
7,67,29,10,28,
Algorithm's basic operation count: 9
Algorithm's execution time in seconds: 0

7,10,28,29,67,
Process returned 0 (0x0)   execution time : 0.020 s
Press any key to continue.
```



```
"C:\Users\User\Documents\QUT\CAB301\Assignment 1 n9602976 (to hand in)\A
An array of size 10 with random numbers has been created.
12,36,59,72,15,50,12,2,49,57,
Algorithm's basic operation count: 44
Algorithm's execution time in seconds: 0

2,12,12,15,36,49,50,57,59,72,
Process returned 0 (0x0)   execution time : 0.029 s
Press any key to continue.
```

Figure 4: Functional Testing of the Bubble Sort (Efficient Version) algorithm. This is done to ensure the Bubble Sort can sort correctly. Here is an array of size 5 and size 10.

	Average number of operations		Average execution time	
Size of Array	Normal **Basic**	Efficient **Basic**	Normal **Time** (s)	Efficient **Time** (s)
2000	1999000	1996448.9	0.0266	0.0265
4000	7998000	7992304.6	0.0323	0.059
6000	17997000	17989082.1	0.1411	0.1217
8000	31996000	31986042.8	0.2347	0.2122
10000	49995000	49976841.5	0.3566	0.3359
12000	71994000	71965787.7	0.5057	0.4859
14000	97993000	97960018.7	0.6893	0.678
16000	127992000	127969758.2	0.8977	0.8734
18000	161991000	161951297.2	1.1438	1.1157
20000	199990000	199950181.4	1.4305	1.4028
22000	241989000	241933712.3	1.7209	1.6662
24000	287988000	287918364.9	2.062	1.9922
26000	337987000	337928469.3	2.4388	2.4029
28000	391986000	391923959.7	2.9528	2.8627
30000	449985000	449913790	3.3269	3.331
32000	511984000	511895560.3	3.8585	3.8095
34000	577983000	577863547.1	4.3723	4.3874
36000	647982000	647878374	4.9145	4.8185
38000	721981000	721878577.2	5.4876	5.286
40000	799980000	799846590.8	6.009	5.8611

Figure 5: DataTables of both Bubble Sort algorithms.

Basic stands for the total number of basic operations done to sort an array of a given size, **Time** stands for the total execution time taken in seconds (s) to complete the sorting of an array of a given size.

- Notice that the Normal and Efficient Bubble Swap did not really show any significant difference between each other both in terms of average execution time and average of basic operations. Overall, the Efficient Bubble Sort executes just a little faster than the Normal Bubble Sort.

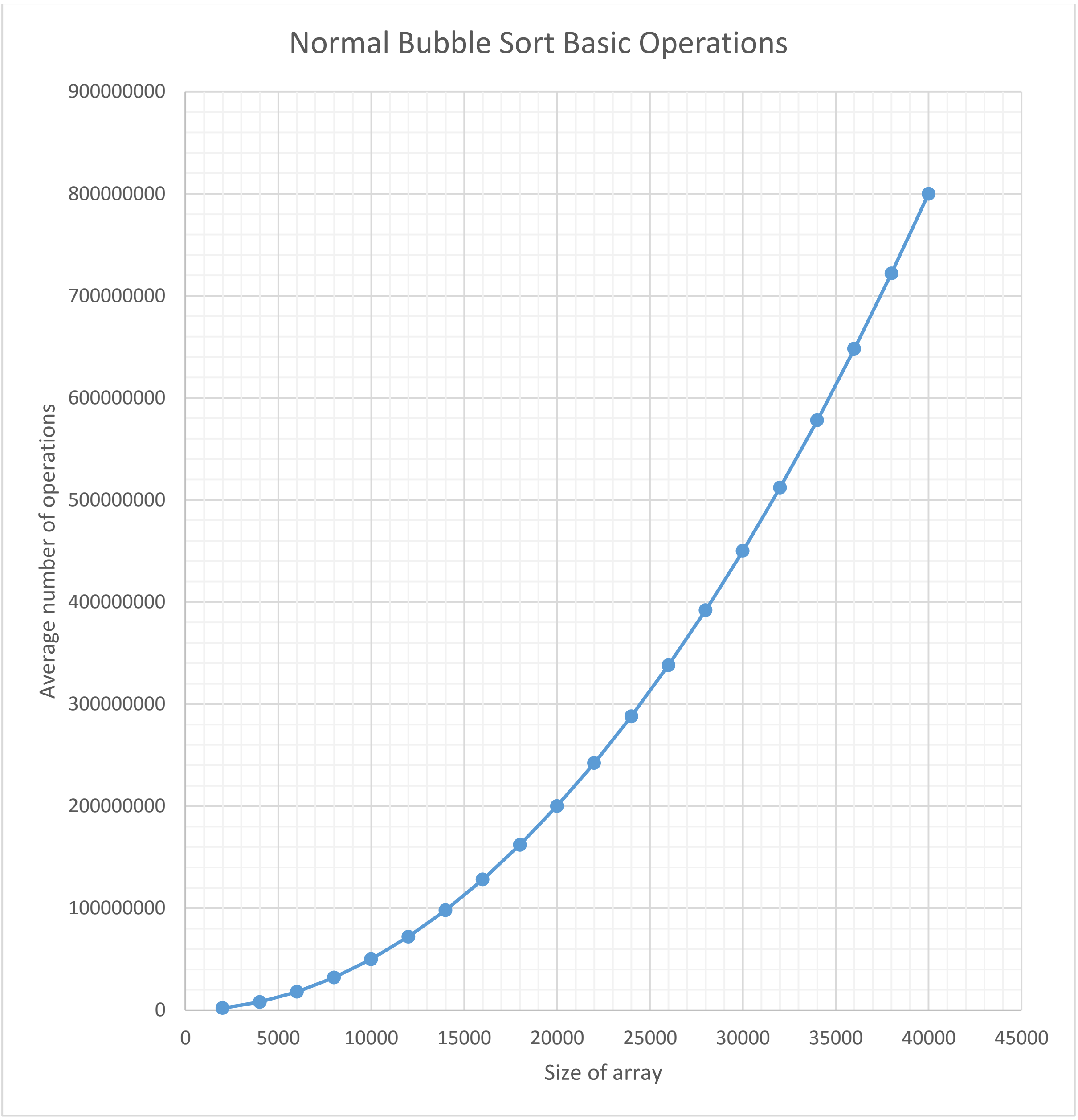


Figure 6: Average number of Basic Operations of Normal Bubble Sort.
This graph shows the measured number of basic operations required to sort arrays of different size.

20 data points are shown for each line, each representing the total number of basic operations required to sort an array of a given size (2000, 4000, 6000, 8000, 10000, 12000, 14000, 16000, 18000, 20000, 22000, 24000, 26000, 28000, 30000, 32000, 34000, 36000, 38000, and 40000 respectively).

The result confirms that the average number of basic operations grows quadratically $\theta(n^2)$ with the size of the array.

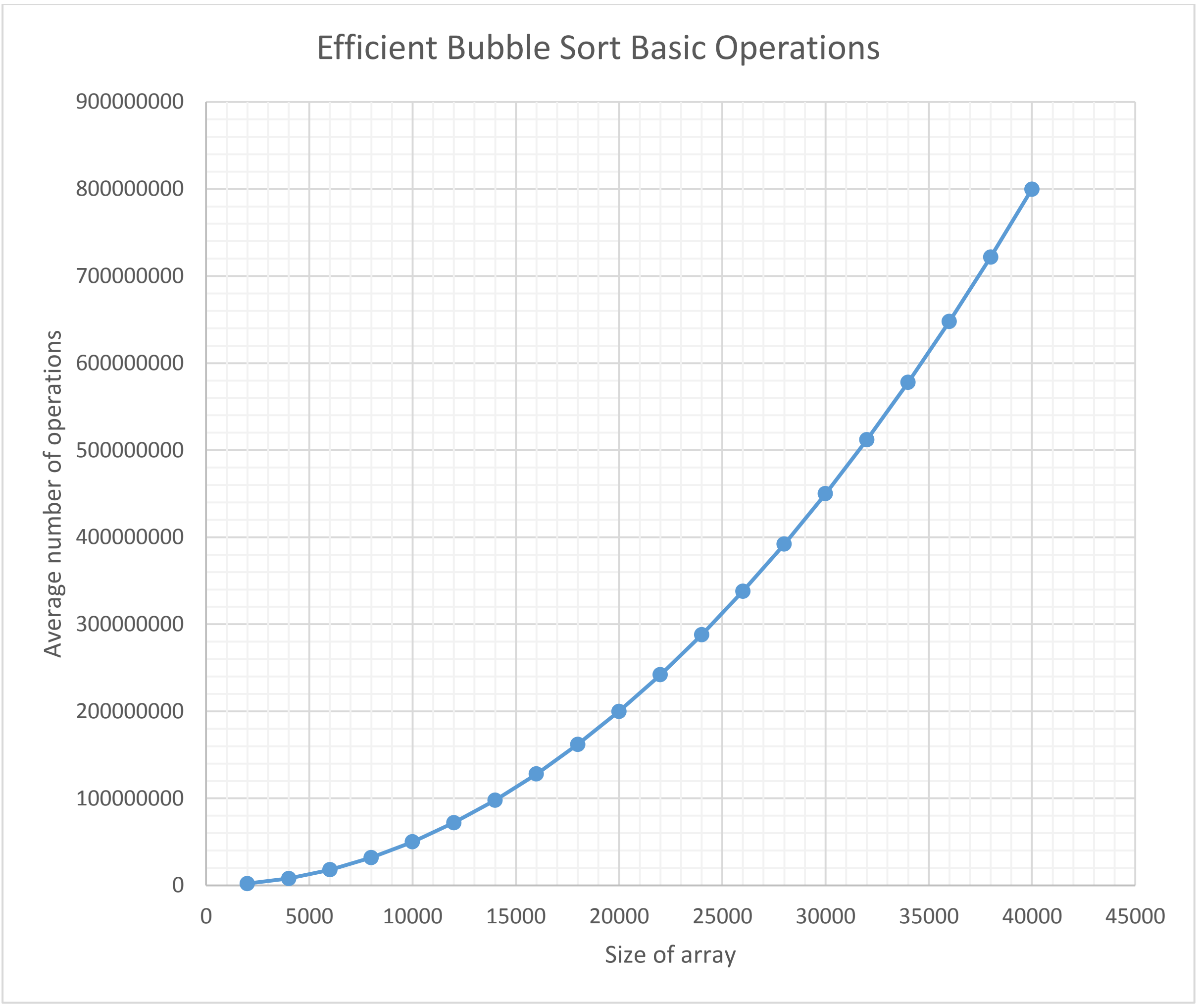


Figure 7: Average number of Basic Operations of Efficient Bubble Sort.
This graph shows the measured number of basic operations required to sort array of different size.

20 data points are shown for each line, each representing the total number of basic operations required to sort an array of a given size (2000, 4000, 6000, 8000, 10000, 12000, 14000, 16000, 18000, 20000, 22000, 24000, 26000, 28000, 30000, 32000, 34000, 36000, 38000, and 40000 respectively).

The result confirms that the average number of basic operations grows quadratically $\theta(n^2)$ with the size of the array.

- Also, notice that the Efficient Bubble Sort did not really show any significant difference to the Normal Bubble Sort as both algorithm's order of growth is the same at $\theta(n^2)$, with the Efficient Bubble Sort doing only a diminish less amount of basic operation compare to the Normal Bubble Sort.

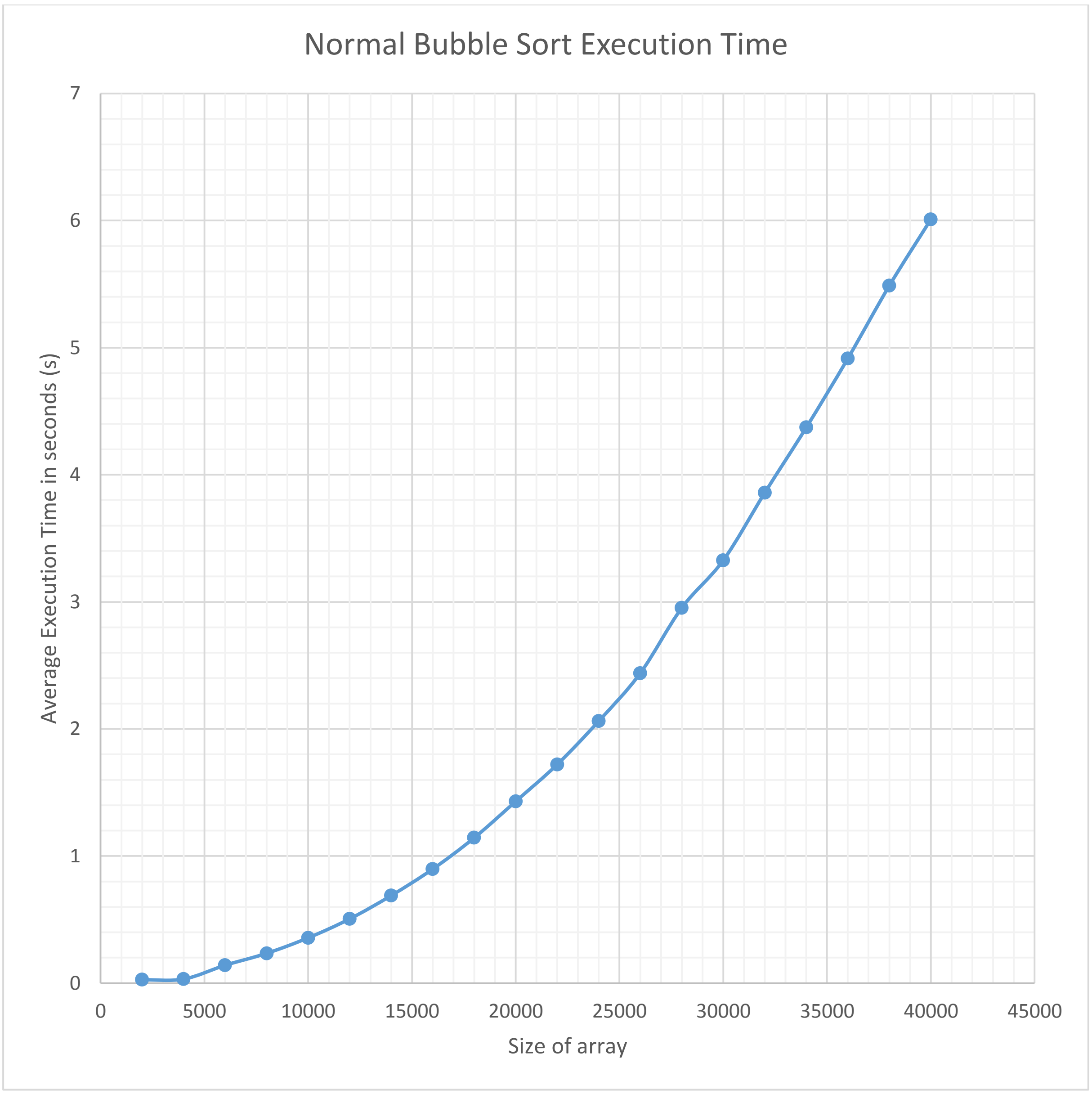


Figure 8: Average execution time of Normal Bubble Sort.
This graph shows the measured average total execution times for sorting arrays of different sizes.

20 data points are shown for each line, each representing the total elapsed execution time to sort an array of a given size (2000, 4000, 6000, 8000, 10000, 12000, 14000, 16000, 18000, 20000, 22000, 24000, 26000, 28000, 30000, 32000, 34000, 36000, 38000, and 40000 respectively). From the graph, it can be concluded that the execution time has a quadratic growth, $\theta(n^2)$, as was predicted similarly in the graph of the number of basic operations.

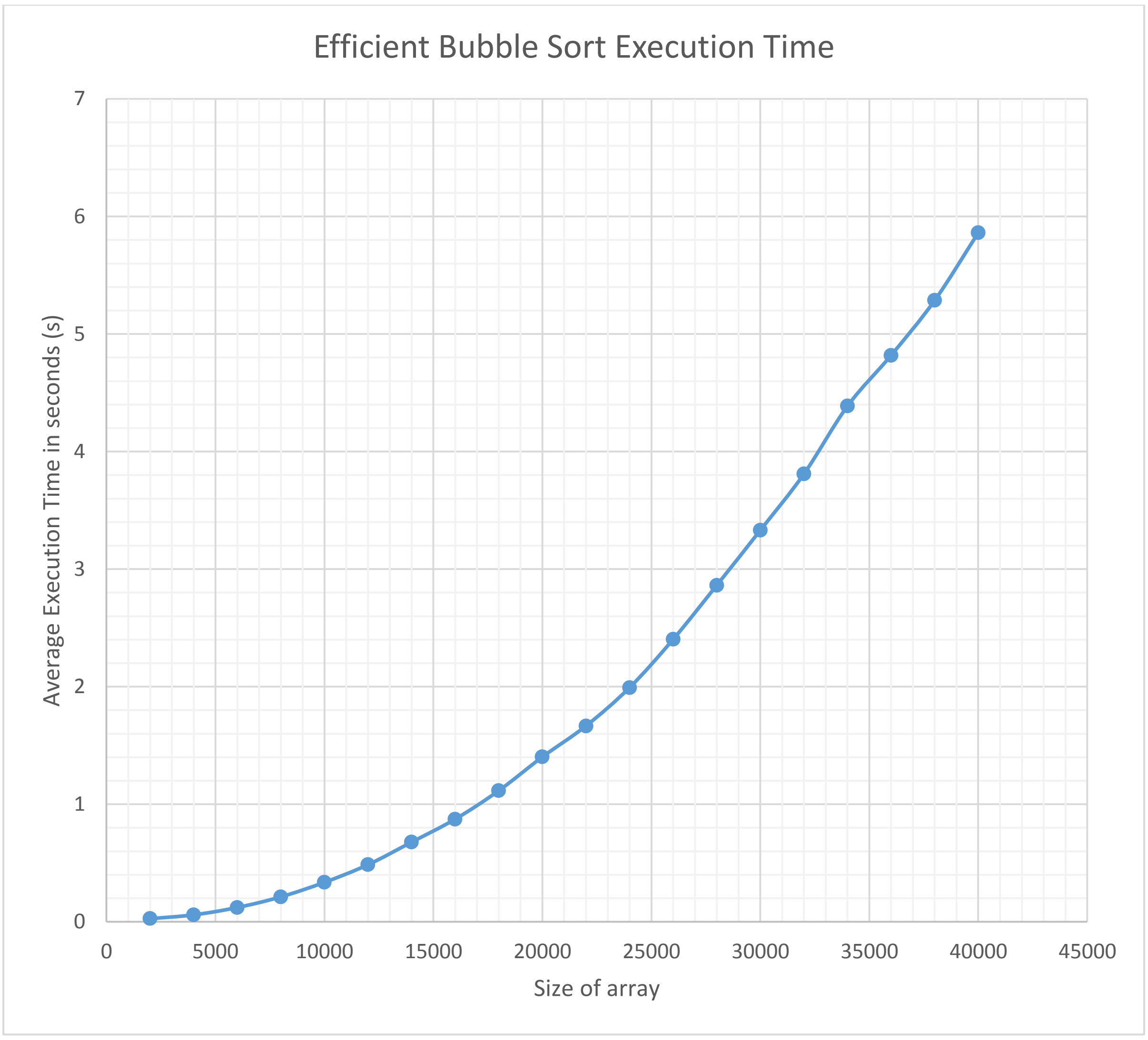


Figure 9: Average execution time of Efficient Bubble Sort.
This graph shows the measured average total execution times for sorting arrays of different sizes.

20 data points are shown for each line, each representing the total elapsed execution time to sort an array of a given size (2000, 4000, 6000, 8000, 10000, 12000, 14000, 16000, 18000, 20000, 22000, 24000, 26000, 28000, 30000, 32000, 34000, 36000, 38000, and 40000 respectively).

From the graph, it can be concluded that the execution time has a quadratic growth, $\theta(n^2)$, as was predicted similarly in the graph of the number of basic operations.

- Also, notice that the Efficient Bubble Sort did not really show any significant difference to the Normal Bubble Sort as both algorithm's execution time are similar at $\theta(n^2)$, with the Efficient Bubble Sort sorts a little bit faster.

	Ratio	
Size of Array	Normal Basic to Normal Time	Efficient Basic to Efficient Time
2000	75150375.94	75337694.34
4000	247616099.1	135462789.8
6000	127547838.4	147814972.1
8000	136327226.2	150735357.2
10000	140199102.6	148784880.9
12000	142365038.6	148108227.4
14000	142163064	144483803.4
16000	142577698.6	146519072.8
18000	141625284.1	145156670.4
20000	139804264.2	142536485.2
22000	140617700	145200883.6
24000	139664403.5	144522821.5
26000	138587420	140633596.6
28000	132750609.6	136907101.6
30000	135256545.1	135068685.1
32000	132689905.4	134373424.4
34000	132191981.3	131709793.3
36000	131851053	134456443.7
38000	131565894	136564240.9
40000	133130304.5	136466975.6

Figure 10: Ratio Table of Basic Operations to Execution Time of both Bubble Sort algorithms.

- Note that the ratio stabilizes at size of Array, *n* >= 10000.

Appendix

Appendix A Code for the Algorithm

Normal Bubble Sort

```
1.  long long int NormalBubbleSort(int *A, int size)
2.  {
3.      //Basic operation count
4.      long long int basic = 0;

5.      //Loop through the array
6.      for (int i = 0; i <= size - 2; i++)
7.      {
8.              for (int j = 0; j <= size - 2 - i; j++)
9.              {
10.                     //Increment basic
11.                     basic++;
12.                     //If element after current is smaller than current,
13.                     //it means the array is not ascending
14.                     if (A[j + 1] < A[j])
15.                     {
16.                             //Swap places
17.                             int temp = A[j + 1];
18.                             A[j + 1] = A[j];
19.                             A[j] = temp;
20.                     }
21.             }
22.     }
23.     return basic;
24. }
```

This appendix presents the C++ code written to implement the Normal Bubble algorithm in Figure 1. Basically, the function accepts a dynamic array and the size of the array, *n* (line 1).

A **for** loop then iterates from 0 to *size* -2 through the array (line 6). Its inner loop iterates from 0 to *size* – 2 – *i* (line 8). The algorithm will then compare an element with its adjacent element (line 14). If the current element compared is bigger than the succeeding adjacent element, it will swap places with the succeeding element (line 17-19). The process repeats until the largest unsorted element is placed at the end of the list. Then the inner loop goes back to element [0] (line 8) and the process is repeated for the second largest unsorted element to be placed at the second last place in the array, and so on. Even if the array is sorted, the loop will still proceed to loop through the whole array until *i* = *size* -2 and *j* = *size* -2 - *i*.

Efficient Bubble Sort

```
1.  long long int EfficientBubbleSort(int *A, int size)
2.  {
3.      //Basic operation count
4.      long long int basic = 0;
5.      //Counter for number of adjacent pairs to be compared
6.      int count = size - 1;
7.      //Swap flag
8.      bool sflag = true;

9.      //Loop through the array
10.     while (sflag)
11.     {
12.             //Swap flag to false
13.             sflag = false;
14.             for (int j = 0; j <= count - 1; j++)
15.             {
16.                     //Increment basic
17.                     basic++;
18.                     //If element after current is smaller than current,
19.                     //it means the array is not ascending
20.                     if (A[j + 1] < A[j])
21.                     {
22.                             //Swap places
23.                             int temp = A[j + 1];
24.                             A[j + 1] = A[j];
25.                             A[j] = temp;

26.                             //Swap flag to true
27.                             sflag = true;
28.                     }
29.             }
30.             //Decrement counter
31.             count--;
32.     }
33.     return basic;
34. }
```

This appendix presents the C++ code written to implement the Efficient Bubble algorithm in Figure 2. Basically, the function accepts a dynamic array and the size of the array, n and an operation counter (line 1). A variable *count* is initialized to *size* -1 (line 6). A **swap** flag is also initialized as *true* (line 8). Meanwhile, the **while** loop will check the **swap** flag for *true* (line 10) before proceeding to switch the **swap** flag to *false* (line 13) and enter the **for** loop (line 14). Once inside the **for** loop, the loop will iterate from 0 to *count* -1 through the array (line 14). The algorithm will then compare an element with its adjacent element. If the current element compared is bigger than the succeeding adjacent element, it will swap places with the succeeding element (line 23-25) while switching **swap** flag to *true*. The process repeats until the largest unsorted element is placed at the end of the list. Then the **for** loop ends and the *count* is decremented by 1 (line 31). The **while** loop checks the **swap** flag and goes back to element [0] (line 14) and the process is repeated for the second largest unsorted element to be placed at the second last place in the array, and so on. When all the elements in the array are sorted, **swap** flag will remain false, and the **while** loop will be stopped, thus no more sorting will be done.

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.

```
1.  //Create dynamic array
2.  size = 10;
3.  int *A = new int[size];

4.  //Create
5.  //Set srand seed
6.  srand(time(nullptr));

7.  //Insert random numbers from 1-100 into the array
8.  for (int j = 0; j < size; j++)
9.  {
10.     A[j] = rand() % 100 + 1;
11. }
12. cout << "An array of size " << size << " with random numbers has been created." <<
13. endl
14. //Display the array (for checking)
15. for (int k = 0; k < size; k++)
16. {
17.     cout << A[k] << ",";
18. }
19. cout << endl;

20. //Start timer
21. start = clock();

22. /* Algorithm execution begins */

23. //Returns total number of basic operation
24. basic = BubbleSortEfficient(A, size);
25. /* Algorithm execution ends */
..
..
26. cout << "Algorithm's basic operation count: " << basic << endl;
27. cout << "Algorithm's execution time in seconds: " << duration << endl;
28. cout << endl;
29. //Display the array (for checking)
30. for (int l = 0; l < size; l++)
31. {
32.      cout << A[l] << ",";
33. }
```

The program creates a dynamic array (line 3) [4], based on the size (line 2). A timer is started (line 21) and the BubbleSort function runs (line 24).

The results of the count of *basic* operation and the execution time are displayed. (line 26-27). From the array display (line 30-33), the algorithm correctness can be checked. Different size such as 5, and 10 were performed by changing the size (line 2).

Appendix C Code for Counting the Number of Basic Operations

Normal Bubble Sort

```
1.  long long int NormalBubbleSort(int *A, int size)
2.  {
3.      //Basic operation count
4.      long long int basic = 0;

5.      //Loop through the array
6.      for (int i = 0; i <= size - 2; i++)
7.      {
8.              for (int j = 0; j <= size - 2 - i; j++)
9.              {
10.                     //Increment basic
11.                     basic++;
12.                     //If element after current is smaller than current,
13.                     //it means the array is not ascending
14.                     if (A[j + 1] < A[j])
15.                     {
16.                             //Swap places
17.                             int temp = A[j + 1];
18.                             A[j + 1] = A[j];
19.                             A[j] = temp;
20.                     }
21.             }
22.     }
23.     return basic;
24. }
```

- Basic operations are underlined.
- To measure the number of basic operations (Section 2.1) performed by the algorithm a counter is incremented when a basic operation is executed. A variable *basic* is initialised as zero (line 4), is incremented each time the basic operation identified in Section 2.1 is performed (lines 11), and is returned when the method terminates (line 23).

Efficient Bubble Sort

```
1.  long long int EfficientBubbleSort(int *A, int size)
2.  {
3.      //Basic operation count
4.      long long int basic = 0;
5.      //Counter for number of adjacent pairs to be compared
6.      int count = size - 1;
7.      //Swap flag
8.      bool sflag = true;

9.      //Loop through the array
10.     while (sflag)
11.     {
12.             //Swap flag to false
13.             sflag = false;
14.             for (int j = 0; j <= count - 1; j++)
15.             {
16.                     //Increment basic
17.                     basic++;
18.                     //If element after current is smaller than current,
19.                     //it means the array is not ascending
20.                     if (A[j + 1] < A[j])
21.                     {
22.                             //Swap places
23.                             int temp = A[j + 1];
24.                             A[j + 1] = A[j];
25.                             A[j] = temp;

26.                             //Swap flag to true
27.                             sflag = true;
28.                     }
29.             }
30.             //Decrement counter
31.             count--;
32.     }
33.     return basic;
34. }
```

- Basic operations are underlined.
- To measure the number of basic operations (Section 2.1) performed by the algorithm a counter is incremented when a basic operation is executed. A variable *basic* is initialised as zero (line 4), is incremented each time the basic operation identified in Section 2.1 is performed (lines 17), and is returned when the method terminates (line 33).

Appendix D Code for Measuring Execution Times for sorting an array of random number using Bubble Sort

The code is used to measure how long it takes to sort an array with random numbers using the algorithm in Figure 1 and 2.

```
1.  //Timer
2.  clock_t start;
3.  long double duration; //Duration of the execution of the algorithm
4.  ..
5.  ..
6.  //Start timer
7.  start = clock();

8.  /* Algorithm execution begins */
9.  //Returns total number of basic operation
10. basic = NormalBubbleSort(A, size);
11. /* Algorithm execution ends */

12. //End timer
13. duration = (clock() - start) / long double(CLOCKS_PER_SEC);
14. ..
15. ..
16. //Open file
17. myfile.open("BubbleNormal.txt", ios::out | ios::ate | ios::app);
18. //Check is file is open
19. if (myfile.is_open())
20. {
21. myfile << basic << "\t\t\t\t" << duration << endl;
22. myfile.close();
23. }
24. else cout << "Unable to open file";
25. ..
26. ..
```

The implementation is referred from Cplusplus.com [7]. Firstly, the *timer* is declared (line 2) to record the starting time (line 7), then the algorithm runs and the *timer* records the finishing time of the sorting when the method *NormalBubbleSort* is terminated, followed by calculating the difference between the two (line 13) to determine the elapsed execution time. The recorded duration is then saved in a text file with the basic operations (line 19-24).