CAB301 Assignment 1

Average-Case Efficiency Analysis of an Insertion Sort Algorithm

Student Name: Michael Cartwright

Student Number: n8869871

Date Submitted: 22nd April 2016

**Summary**

Enter Summary

**1 Description of the Algorithm**

The algorithm of interest is called Insertion Sort. Insertion Sort takes any amount of numbers that are in any order, whether decreasing or mixed, and outputs those same numbers into a non-decreasing order. Insertion Sort takes an unsorted array and compares the nth element and the element before the nth element. Insertion Sort determines whether the nth element is greater than or less than the value of the element before the nth. If the nth element is greater than the value of the element before it, these numbers are already in a non-decreasing order and the next element is analysed. If the nth element’s value is less than the element before it, then the Insertion Sort algorithm switches these numbers around and continues to do so until the elements of the unsorted array being analysed are in a non-decreasing order. This then repeats until every element of the unsorted array has gone under the process of the Insertion Sort algorithm. When completed the array has its elements values sorted into a non-decreasing order. The Insertion Sort algorithm of interest is a simple Insertion Sort presented by Levitin [p. 161] and equivalent descriptions are also noted by Johnsonbaugh and Schaefer [p. 241]. This algorithm is exemplified to explain what the algorithm does and how it goes about this. This algorithm is presented as a textbook based question to be developed into a programming language of choice by the reader. For this particular algorithm Levitin [p. 162], Berman and Paul [Sect. 6.4] state that the inequality of is performed times the average-case analysis of this Insertion Sort algorithm.

**2 Theoretical Analysis of the Algorithm**

This section of the document explains the algorithm’s estimated time complexity from a theoretical perspective.

**2.1 Identifying the Algorithm’s Base Operation**

b. for loop has integer i being valued the 1st element of the Array (Not the 0th) and continues to increment the value of i until i is the value of the length of the array.

c. the integer v becomes the same value as the ith element value in the array.

d. the integer j becomes the element of the array before the ith element of the same array

e. while loop with the purpose of identifying if the value of the jth element of the array is greater than the value of the ith element of the same array. If the jth element’s value is greater than these two elements are not in a non-decreasing order and f. and g. lines of the algorithm are initialised. Also the value of integer j cannot be less than zero.

f. The integer value of the jth element of the array becomes the value of the jth element + 1 of the array. This shifts the jth elements value across to allow for a non-decreasing order.

g. The value of integer j decrements by one. This allows for all elements of the array before the ith element to be checked to ensure the array is fully arranged into a non-decreasing order.

h. The jth element + 1 of the array becomes the value of the integer v.

**2.2 Average-Case Efficiency**

The average-case scenario for the insertion sort algorithm is when half of the random numbers in the array have a value greater than the value of the element before it. In this situation, it is stated by Khan Academy, *“a call to insert on a subarray of elements would slide of them”* [1]. Considering that , this concludes that the average number of basic operations required to sort the variables of the array in an average-case scenario with an array of length is based on arithmetic sequence. It should be said that from analysis and research this is the same case for a worst-case efficiency scenario but the run-time for the average-case should be half of the result of a worst-case scenario but still follows . Using notation the run-time is . It should also be said that this occurs for a ‘random’ array where half of the variables so happen to have a number greater than the elements before it. In a scenario where the half the array is sorted, thus an *“almost sorted”* array, the case would be [1].

**2.3 Order of Growth**

Enter order of growth… Same as above?

**3 Methodology. Tools and Techniques**

This section of the document explains the computing environment used for the experiments.

1. The Insertion Sort algorithm and experiments were redeveloped in the programming language C++ utilising the Code::Blocks Integrated Development Environment (IDE) and the GNU Compiler Collection for Code::Blocks. C++ is an object oriented programming language and is being used as per the specifications of *“CAB301 Assignment 1 – Empirical Analysis of an Algorithm”*.

2. The experiments in this document were executed on a custom built desktop which utilises the Windows 10 Operating System (OS). Code::Blocks integrated execution time was used to measure the execution times. Insert how data was inserted into the array. Mention array length. Mention how many times tested. During the experimentation, other software applications running on the system had their performance reduced or removed for testing purposes.

3. Graphs of all results from the experiments were developed on MathWorks MATLAB R2015b. Figures Enter Graph Figure Numbers Here were all created in MATLAB. This document was prepared in Microsoft’s Word 2016.

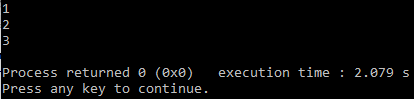
**4 Experimental Results**

This section of the document explains the results of the experiments and compares this to the theoretical predictions as stated in Section 2 of this document. The C++ program implementation can be seen in Appendix A or the C++ Code::Blocks file attached to the *“CAB301 Assignment 1”* Blackboard Assignment Submission.

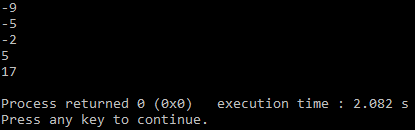
**4.1 Functional Testing**

To test if the C++ program in Appendix A functions correctly the variable int arrayInput was set with the following lengths and values based from the code in Appendix A.

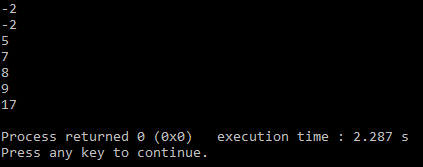
For the first test, arrayInput was given, in respective order from left to right, the values 3, 2 and 1. Below is the result of the test providing a non-decreasing order for all the values of the array. Therefore, the functionality of the program is successful.



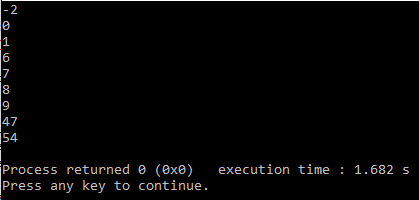
For the second test, arrayInput was given, in respective order from left to right, the values -2, -5, 17, -9 and 5. Below is the result of the test providing a non-decreasing order for all the values of the array. Therefore, the functionality of the program is successful.



For the fourth test, arrayInput was given, in respective order from left to right, the values -2, 17, 5, 7, 9, -2 and 8. Below is the result of the test providing a non-decreasing order for all the values of the array. Therefore, the functionality of the program is successful.



For the last functionality test, arrayInput was given, in respective order from left to right, the values 54, 7, 9, -2, 0, 1, 47, 6 and 8. Below is the result of the test providing a non-decreasing order for all the values of the array. Therefore, the functionality of the program is successful.



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

**4.3 Average-Case Execution Time**

**References**

[1] https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/analysis-of-insertion-sort

a. ALGORITHM InsertionSort(A[0..n – 1])

//Sorts a given Array by insertion sort

//Input: An array A[0..n – 1] of n orderable elements

//Output: Array A[0..n – 1] sorted in non-decreasing order

b. **for** i ← 1 to n – 1 **do**

c. v ← A[i]

d. j ← i – 1

e. **while** j ≥ 0 **and** A[j] > v **do**

f. A[j + 1] ← A[j]

g. j ← j – 1

h. A[j + 1] ← v

Figure 1: The algorithm provided based on my student number.

**Appendix A Code for the Algorithm**

This appendix shows the C++ code that was developed from the Insertion Sort algorithm in *Figure 1*. The Insertion Sort algorithm could be developed as two classes. Before the classes insertionSort, outputArrayToConsole and main; The following integers were written in the program and can be seen below. Lines 1 to 4 set the variables i, vand j as integers based from *Figure 1*. The integer sizeValue is also set to be used for the two classes and its purpose will be explained further in Appendix A. Line 5 sets an array of user defined length and elements.

1. int variableI;

2. int variable;

3. int variableJ;

4. int sizeValue;

5. int arrayInput [9] = {0, 1, 2, 6, 7, 8, 9, 10, 54};

The Insertion Sort algorithm was developed in the class insertionSort() which can be seen below. Line 2 sets the value of sizeValue to the length of arrayInput by found by using sizeof and divided by four due to the byte size of integers. Line 4 to 6 sets the values of integer variables to identify the ith element of interest, its value stored in the array and the jth element of the array which the before the ith element. Line 8 to 11 is computational logic that identifies whether the jth element’s value of the array is greater than the ith element’s value of the array. Line 9 and 11 sets the values of the array in a non-decreasing order.

1. int insertionSort() {

2. sizeValue = sizeof(arrayInput)/4;

3.

4. for (variableI = 1; variableI < sizeValue; variableI++) {

5. variableV = arrayInput[variableI];

6. variableJ = variableI - 1;

7.

8. while (variableJ >= 0 && arrayInput[variableJ] > variableV) {

9. arrayInput[variableJ + 1] = arrayInput[variableJ]; 10.

10. variableJ = variableJ - 1;

11. }

12. arrayInput[variableJ + 1] = variableV;

13. }

14. }

The variables of the array that print to the Code::Blocks Console was developed in the class outputArrayToConsole() which can be seen below. Line 2 sets the value of variableI. Lines 4 to 5 prints each value of arrayInput, to the respective value of variableI, on a new line.

1. int outputArrayToConsole() {

2. variableI = 0;

3.

4. for (variableI = 0; variableI < sizeValue; variableI++) {

5. std::cout << arrayInput[variableI] << "\n";

6. }

7. }