CAB301 Assignment 2

Empirical Analysis of an Two Algorithms

Student Name (Student Number):

Lachlan Cesca (n8580413)

Nathaniel Perkins (n7291035)

Due Date:

30th May 2016

# Summary

This report provides an in-depth analysis of the “Insertion Sort” algorithm by experimentally determining the average-case execution time and average-case number of basic operations. Using C++ to implement the algorithm, the results from the testing are compared against the theoretical to authenticate the validity of the algorithms theoretical predictions.

# - Description of the Algorithms

The insertion sort is a simple algorithm for sorting a list of numbers into numerical order (usually ascending but could also be modified for descending). It does this by maintaining an ordered sublist in the lower section of the array that grows as it compares the next value in the array with the all the previous. Starting at index 1 and iterating through to n-1 it compares the current value with all previous values in the sorted sublist as it shuffles the value down until it inserts itself into its respective position down the list, hence the name “Insertion Sort”. This particular algorithm implements array sorting but could easily be modified to other types of sets like linked list.

# - Theoretical Analysis of the Algorithm

This section briefly describes the algorithm under theoretical analysis before later comparing it to the experimental results.

## 2.1 - Identifying the Algorithms Basic operation

There are 3 main criteria to consider when identifying an algorithms basic operation, these criteria are as followed [1]:

* Is the operation needed to solve the problem.
* Is the operation the most time consuming.
* Is the operation the most frequently used.

From Figure 1 it can be seen that the check inside the while loop will occur the most often as it occurs multiples times for every value from 1 to n-1. Every other operation inside the function deals with storing and moving around variables and so are not considered basic operations.

Simply put, the comparative operation is considered to be the algorithms basic operation as it is necessary in sorting the array, it is the single operation of the algorithm that takes the most time and is also the operation that is most frequently executed.

## 2.2 – Average-Case efficiency

The initial assumption for the average-case efficiency provided is that number of comparisons is proportional to the square of the size of the input array for this particular algorithm. Specifically:

This was further found to be reinforced through other literature [2].

Simply stated that on average the number of comparisons or basic operations an insertion sort must perform is proportional to the size of the input array. For this case, as stated this proportionality is .

## 2.3 - Order of Growth

Once again referring to Figure 1 it can be seen that the order of growth is determined by the largest nesting of loops. In this particular case we have a while loop within a for loop, thus leading us to an order of growth .

It is also stated that the worst case scenario for an insertion sort is of [3]. But often is the situation that the worst case and the average case are closer together than that of the average case and the best case.

# 3.0 - Methodology, Tools and Techniques

1. The implementation of the algorithm was done in the C++ programming language through the eclipsed IDE. Eclipse is a great IDE with various debugging tools to facilitate code development. The project was managed in the common practice of keeping header files in an “include” folder and all other functional files in the “src” folder. The functions with distinctly different purposes were separated into their own files for better code management.
2. The experiments were performed on a Dell Inspiron 15-5521, with an Intel Core i5-3317U CPU running Windows 8.1 (64 bit version). As for the randomness of the arrays to be sorted, the C++ rand function was used with a seed value taken from the computers time with resolution of up to a microsecond. To combat any computations that would potentially take any less time than that, the range of random numbers would change on each iteration. As the values used in the array didn’t matter, it was only the randomness of the order and therefore ensuring the uniqueness of each array. Computation time was measured in the same way the seed was taken because of this the measured time had the same resolution of up to a microsecond.
3. All data was exported to CSV files, and then graphs Figure 2 and Figure 3 were then generated in Excel. The report was then written up in Microsoft Word.

# 4.0 - Experimental Results

This section details the experimental results from testing the insertion sort algorithm for both its average-case number of comparisons and its average-case execution time. Which are then both compared to the theoretical predictions in Section 2.0.

## 4.1 - Functional Testing

To test the functionality of the algorithm, three basic tests were performed. The first is sorting a random array of size 10, the second basic test performed was sorting a reverse ordered array of size 10 and lastly performing a sort on an array of size 10 that is already ordered.

From these three tests alone, the basic functionality of the algorithm can be determined for its validity. As expected all three tests performed correctly as seen in Figure 4.

## 4.2 - Average-Case Number of Basic Operations

The number of basic operations/comparisons was tested with a slightly modified version of the insertion sort function where the function would keep track of the comparisons and then the output of the function would return the number of comparisons made.

A sampling rate of 10 for the size of the array was used in order to reduce total run time. The overall results as seen in Figure 3 show a very clean smooth graph, which shows that the averaging of the number of comparisons worked out quite nicely. This is also noted by the R2 value of the trend line showing that the trend line models all the data perfectly (100%).

As theorised in Section 2.2, experimental testing of the data obtained verified this relationship as shown in Figure 3, where the equation for the trend line is y = 0.25x2 - 0.9908x + 355.11.

By taking the highest power of the polynomial and ignoring the rest, we are left with y = 0.25x2. It can be seen that the data gained verifies this relationship for the average-case efficiency.

## 4.3 - Average-Case Execution Time

The insertion sort function used in finding the average-case execution time is different than that used in average-case number of basic operations. In order to reduce overhead/extra execution time, the functionality of keep tracking and returning the number of comparisons were stripped into what would be a bare bones insertion sort.

Initial testing showed a fair bit of variance in execution time as the execution time is fairly dependant on the system in which its performed on, as there are other processes taken up by the CPU at the same time. To minimize the effect of other programs on the computation, all non-essential programs were shut down to improve data validity. The initial trend still showed that there was a quadratic increase in execution time respective to the size of the input as it has in Figure 2. In order to produce a cleaner graph, the sampling period was increased as see in the code in Section 6.7.2.

Cross referencing this with the data gathered from testing, as seen in Figure 2 it can be further reinforced that time it takes to execute the insertion sort algorithm is proportional to the square of the size the input array. Thus following a second order polynomial or as theorised in Section 2.3.

# 5.0 – References

# 6.0 – Appendix