Queensland University of Technology

CAB301

Assignment 1

An Empirical Analysis of the Insertion Sort Algorithm

Author:
Joshua Miles[n7176244]

Tutor:
Anthony Gough

22nd April 2016

Contents

1 Introduction

The purpose of this report is to perform an empirical analyses of the insertion-sort algorithm to confirm or deny whether Average Case Efficiency of the algorithm is indeed $\frac{n^2}{4}$. Initially the algorithm itself is detailed given what it does and how it does it. Different hypothesis will be made about characteristics of the algorithm based on investigation and what we currently understand about the algorithm. To find empirical evidence of the efficiency, a clear and logical methodology is outlined to demonstrate exactly how each experiment is performed and the advantages as well as disadvantages that are attributed to the methods. The results of the experiments will than be through to see how the assumptions aligned with reality.

2 Hypothesis

This section will provide predictions about the theoretically expected efficiency for both the system timed average case efficiency and the count of operations Average-case efficiency and reasons for those expectations.

2.1 Average-Case Efficiency

2.1.1 Basic Operations

Given appendix which contains an analysis of the average case efficiency of the Insertion Sort algorithm as well cases from literature such as Knuth [1, 82] expected average efficiency of this algorithms Basic operations is $\frac{n^2}{4}$.

2.1.2 System Time

The other units to measure the average-case efficiency in regards to time will be the units of seconds and milliseconds rather than the basic operations. It is understood that the experiments are limited to what the tools can be used as described by Kernighan [2, p. 185], the time taken for each algorithm can vary depending on the type of processor as well as what is running on the computer that the user doesn't have any knowledge of. This may create sporadic noise as a result skewing the function away from the expected .

2.2 Order of Growth

The expected order of growth is $\Theta(n^2)$ as Knuth [1, p.80] and Cormen et al [3, p.26] noted. This is also affirmed in the above section as it is the dominant factor in the Average Case efficiency for both the count of basic operation and the system time. Thus, if the Average Case efficiency turns out to be $\frac{n^2}{4}$ it is assumed that the $\frac{n^2}{4} \in \Theta(n^2)$ is true.

3 Theoretical Analysis of the Algorithm

3.1 Characteristics

The insertion sort algorithm, as discussed by Knuth [1, p.80], is similar to how a bridge player sorts their hand before they begin playing. The algorithm goes over a loop an initial loop and checks the current index, beginning at 0, is the smallest element in comparison to every element to the left of it. After this process the loop exits and than repeats until the list is in ascending order.

3.2 Time Complexities

The following sections will give an overview of how the time complexities will be approximated

3.2.1 Basic Operations

Using the general rule by Levtin [4, p.44] to find the basic operation which will be executed the most amount times is within the '**while**' loop on line 5 in Appendix 1. The while loop is not executed every time but only when the specific variable is out of order. When it does execute, the three operations on lines 6, 7, 8 from Appendix 1 are performed. Therefore this would give the most accurate representation of the basic operations in this Algorithm.

3.2.2 System Time

The operation will be performed enough times to get a statistical average than the mean of all the times at different inputs calculated. The results will than be all be graphed against what the expected Order of Growth is to see whether it follows the trend.

3.3 Input

Input into the Insertion Sort algorithm will be the determining factor as to how the time and basic operations will grow, as one of the major assumptions is that time and complexity are both a factor of the input. To find the average-case efficiency, the input needs to neither be completely sorted or unsorted but would be required to be in random order between unsorted and sorted. To generate the data to perform an average-case efficiency analysis, an array of different sizes proportional to the amount we want to test will be populated with random values across a range of 1 to 10,000 is used.

The danger of using this data is that it may be in almost sorted or almost unsorted which would skew any results, at this present point in time there is uncertainty as to whether or not there is a studied measurement for how un/sorted a certain set of data is without frivolous assumptions. This outcome of outliers can be alleviated, however, by running the tests a multitude of times (100) and than finding the average at each size of the random value array.

4 Methodology

This section will describe in detail exactly how all experiments and tests are performed as well as how certain possible discrepancies where taken into account.

4.1 The Code

4.1.1 Producing the data

The initial step in the process is the build the data to an appropriate standard that would create the average. A function is used, with the only (line 1 Appendix 5) parameter being the length of the array you require, and a pointer to the array that was constructed is returned. This method utilises the rand() function (line 4 Appendix 5) to produce integers in order to populate the array.

4.1.2 The Algorithm

Done to the exact specification of the pseudo code, the only difference is when the amount of basic operations need to be calculated which is described in the next section.

4.1.3 Testing

To ensure that the array is completely sorted after every test another test in linear time, although it is independent of the algorithm so it will not affect the results, will be used to check if the array is completely sorted.

4.1.4 Collecting the Data

There are two distinct methods being used to collect the data necessary, the basic operations which will be a counter in the inner while loop and the system time. As noted previously, being in the nested loop would increase the time complexity by a factor of n resulting in corruption in the time data collected. Therefore we will treat the basic operations and time independently. The basic operations will be found by incrementing the counter, made up of a long long integer, by one every time it travels through the inner while loop (line 10 Appendix B). There is than a getter method to get the particular basic operation at the end of every time the test is performed before it is again reset to 0 (line 3 Appendix B)
The time that it takes for the algorithm to be performed will be found by starting a timer using the standard libraries clock (line 3 of 3 Appendix 3), performing the algorithm and than taking the start time and take it away from the current time (line 5 Appendix 3).

4.1.5 Writing to CSV

To collect the data after it is performed the program takes the basic operations/time and writes each amount to a CSV (line 4 Appendix F) with its corresponding input.

4.1.6 Finding the Average and Graphing

In order to find the average of both the time and basic operations the CSV will be imported into Google Sheets and than the average will be found with

the using the AVERAGEIF() function. Once the average is found six graphs will be made:

- Represent the predicted amount complexity ($\frac{n^2}{4}$)
- To show the basic operations at different sizes of the array
- To show the calculated time at different sizes of the array
- The predicted complexity and the basic operations
- The predicted complexity and system time
- The total amount of basic operations to the predicted complexity

4.2 Tools

The computer used to process the algorithm will be a Macbook pro with a 2.7 GHz Intel core i5 and 8gb worth of RAM. Consistency throughout a number of the tests will be key to getting the most accurate results as such the processes that will be running when the algorithm runs will be kept to a minimum to not affect the running times generated.
The language used for the algorithm will be C++ with CLion as the IDE to compile and run the code for the algorithm.

5 Results

5.1 Average Time

From figure 11, it can be seen that as size of the array grows larger the the smoothness of the line decreased starting relatively small but growing somewhat indistinguishable from the actual predicted complexity which can be seen in figure 12. This can be attributed to what we hypothesised earlier, the interesting thing about these results are in fact how the time is differing from the predicted complexity. Between the 2400-3000 and 3000-4000 size of the array, the pattern makes a somewhat hump. Though there is lack of evidence, this hump may be attributed to the computer the algorithm was running on, there might have been an unexpected process that took up an enough CPU usage to differ the computational time of the algorithm. As the size of the array increase it is seen to get even worse relative to the expected complexity, this may be attributed to a number of factors but the predominant factor is the fact that the size is so sensitive to what the CPU

is currently doing. In this case it may slow down because of the algorithm itself and the amount of computing power it is taking up.

5.2 Average Number of Basic Operations

From figure 10, it can be seen that as the size of input array grew very smoothly in fact when graphed with the predicted complexity in figure 13 the two are almost indistinguishable. From figure 14 it is interesting to see that the minimum and maximum of the number of basic operations increases but the predicted complexity still remains in the centre of the outputs which really demonstrates the accuracy of the prediction. The results showed what we had hypothesised and found within research; that as the input grew, results followed a trend of being $\frac{n^2}{4}$ the size of the original input.

5.3 Comparison

As the results have clearly shown, the average amount of basic operations far outweighs the system time in terms of accuracy when trying to find the average complexity of an algorithm. This can be attributed to the basic operation being able to ignore the sensitivity of the CPU has to the different processes running on the current machine, in addition the method can ignore the machine entirely and just give pure unbiased results of the experiments on an algorithm. This was nothing new, however, as Levitin [4, p.44] describes the basic operations are referred to the time complexity because of how much more of an accurate representation it is in comparison to the alternative where Kernighan [**?**, 2,p.185]escribes that different processors take different cycles which also has an effect on the time complexity.

6 Conclusion

Therefore, the average efficiency of the Insertion Sort algorithm is seen to be truly represented by $\frac{n^2}{4}$ because of this it is also confirmed that $\frac{n^2}{4} \in \Theta(n^2)$ is true due to the Average Order of Growth just being the element to grow the largest over large inputs. The more interesting discovery was that the number of basic operations per size of input rather than the actual system time is a better representation of the average efficiency due to a number of

such as underlying CPU processes and different processors as well as other contributing factors.

References

[1] D. E. Knuth, *The art of computer programming: sorting and searching.* Pearson Education, 1998, vol. 3.

[2] B. W. Kernighan and R. Pike, *The practice of programming.* Addison-Wesley Professional, 1999.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, "Introduction to algorithms," 2009.

[4] A. Levitin, *Introduction to the design & analysis of algorithms*, 3rd ed. Addison-Wesley Reading, MA, 2007.

Appendices

A The Insertion Sort Algorithm

```
1 ALGORITHM InsertionSort(A[0...n-1])
2         for i <- to n-1 do
3                 v <- A[i]
4                 j <- i - 1
5                 while j >= 0 and A[j] > v do
6                         A[j+1] <- A[j]
7                         j <- j-1
8         A[j+1] <-v
```

Figure 1: The inefficient pseudo code for the Insertion Sort algorithm

B Coded Algorithm using Basic Operations

```
1  int *InsertionSortBasicOperations(int *unsortedArray,
     int lengthOfUnsortedArray) {
2          int j, temp;
3          basicOperations = 0;
4          for (int i = 1; i <= lengthOfUnsortedArray-1;
             ++i) {
5                  temp = unsortedArray[i];
6                  j = i-1;
7                  while (j >= 0 && unsortedArray[j]>temp)
                     {
8                          unsortedArray[j+1] =
                             unsortedArray[j];
9                          j--;
10                         basicOperations++;
11                 };
12         unsortedArray[j+1] = temp;
13         }
14
15         return unsortedArray;
16 }
```

Figure 2: The insertion sort algorithm in the language C++ which includes the basic operation

C Coded Algorithm using System Time

```
1  int *InsertionSortTime(int *unsortedArray, int
       lengthOfUnsortedArray) {
2          int j, temp;
3          for (int i = 1; i <= lengthOfUnsortedArray-1;
             ++i) {
4                  temp = unsortedArray[i];
5                  j = i-1;
6                  while (j >= 0 && unsortedArray[j]>temp)
                     {
7                          unsortedArray[j+1] =
                             unsortedArray[j];
8                          j--;
9                  };
10                 unsortedArray[j+1] = temp;
11         }
12         return unsortedArray;
13 }
```

Figure 3: The insertion sort algorithm in the language C++ which doesn't include the basic operation

D Running the Algorithm while Timed

```
1 double performAlgorithmAndTime(int sizeOfInput) {
2         int *randomArray = makeDataRandom(sizeOfInput);
3         start = std::clock();
4         InsertionSortTime(randomArray, sizeOfInput);
5         duration = (clock() - start) / (double) CLOCKS_PER_SEC;
6         return duration;
7 };
```

Figure 4: This function uses the C++ language to start a clock, perform an algorithm and than find out the time between the start time and the current time.

E Building the random Data

```
1 int* makeDataRandom(int lengthOfArray) {
2         int *arr = new int[lengthOfArray],v1;
3         for (int i = 0; i < lengthOfArray; ++i) {
4                 v1 = rand() % 1000000;
5                 arr[i] = v1;
6         }
7         return arr;
8 }
```

Figure 5: The C++ code related to making an array of size lenghtOfArray() filled with random integers of a range from 1-1000000.

F Writing to CSV format

```
1 void writeToCSVBasicOperations(int input, int
    basicOperations){
2         ofstream myfile;
3         myfile.open("../data/basicoperations.csv", ios
            ::app);
4         myfile << input << "," << basicOperations <<
            endl
5         myfile.close();
6 }
```

Figure 6: Writes the input and the basic operations in the format of a CSV.

```
1 void writeToCSVTime(int input, double timeTaken){
2         ofstream myfile;
3         myfile.open("../data/time.csv", ios::app);
4         myfile << input << "," << timeTaken <<endl ;
5         myfile.close();
6 }
```

Figure 7: Writes the input and the time taken in the format of a CSV.

G Average Case Efficiency

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

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

Because this is the worst case [3], it is presumed that the average case of the equation would be found by simply dividing by two:

$= \frac{n(n-1)}{4}$

$= \frac{n^2-n}{4}$

Looking into the actual average case, the -n becomes negligible therefore the average case efficiency is:

$= \frac{n^2}{4}$

Figure 8: This mathematics refers directly to Cormen et al. [3, p.26] The first sigma represents the for loop on line 2 of figure 1 and the second sigma refers to the while loop on line 5 of the same figure.

H Predicted Count

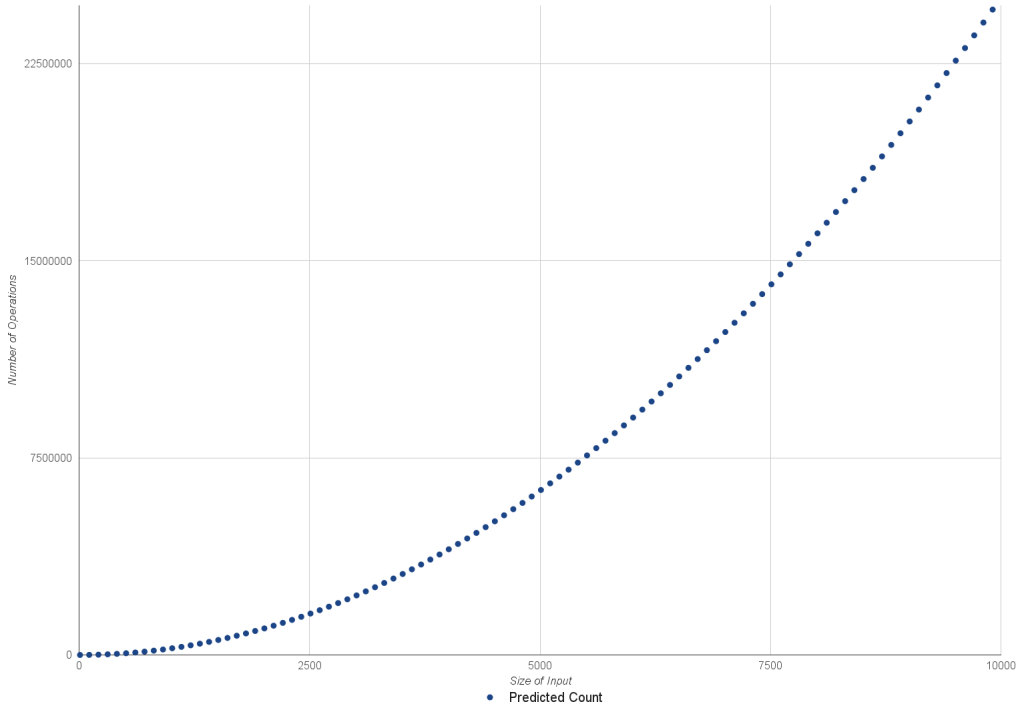

Figure 9: The blue circles are the count that represents the hypothesised number of operations given the size of the input ($\frac{n^2}{4}$).

I Average Basic Operations

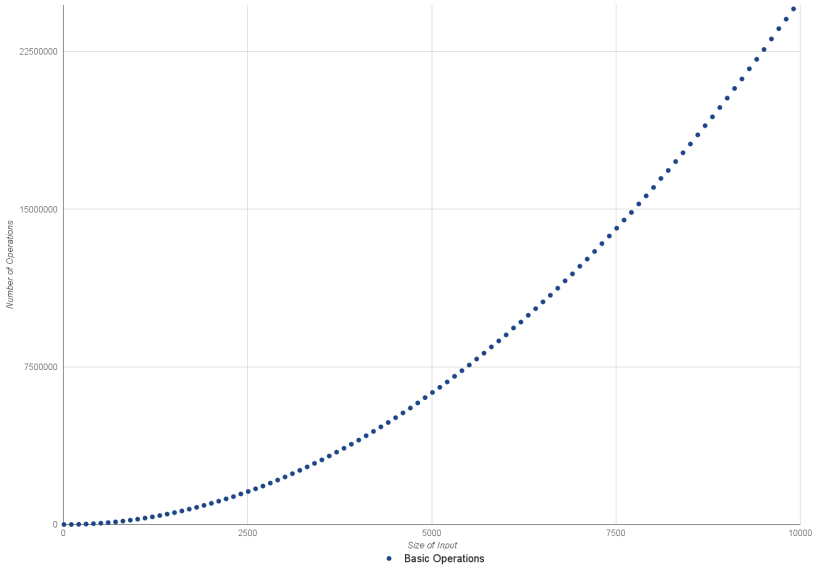

Figure 10: Each blue circle represents the average amount of basic operations it took over 100 iterations to sort an array, where the size of said array can be seen on the horizontal axis.

J Average Time

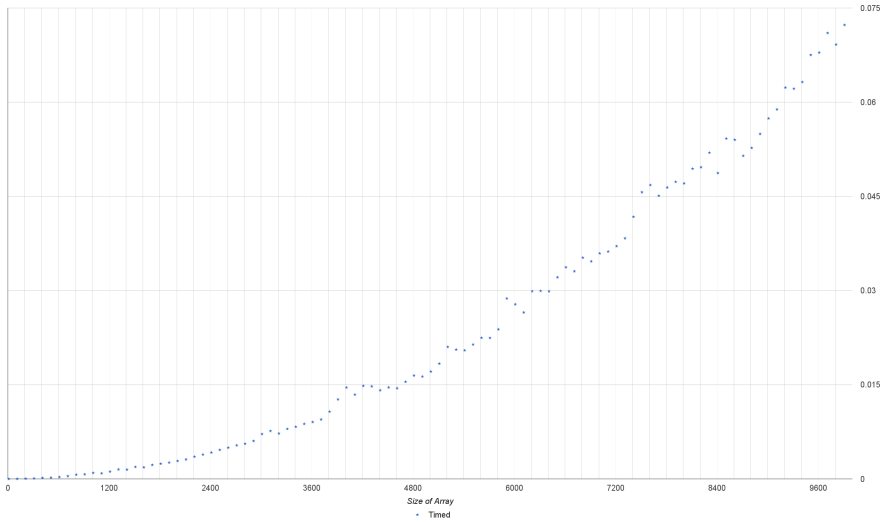

Figure 11: Each blue dot represents the average time it took over 100 iterations to sort an array, where the size of said array can be seen on the horizontal axis.

K Average Time vs Size of Array

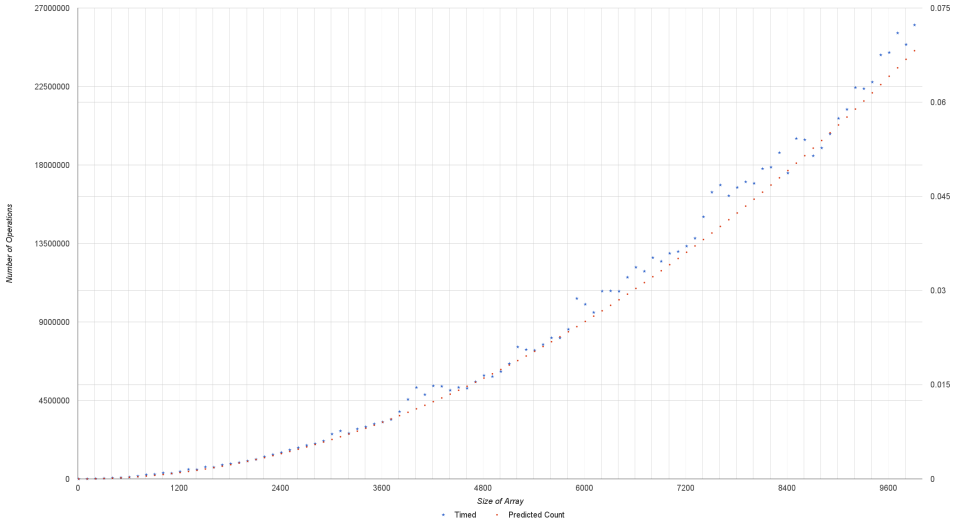

Figure 12: Each blue star represents the average time it took over 100 iterations to sort an array, where the size of said array can be seen on the horizontal axis. The red circles are the count that represents the hypothesised number of operations given the size of the input ($\frac{n^2}{4}$).

L Average Basic Operations Vs Predicted Count

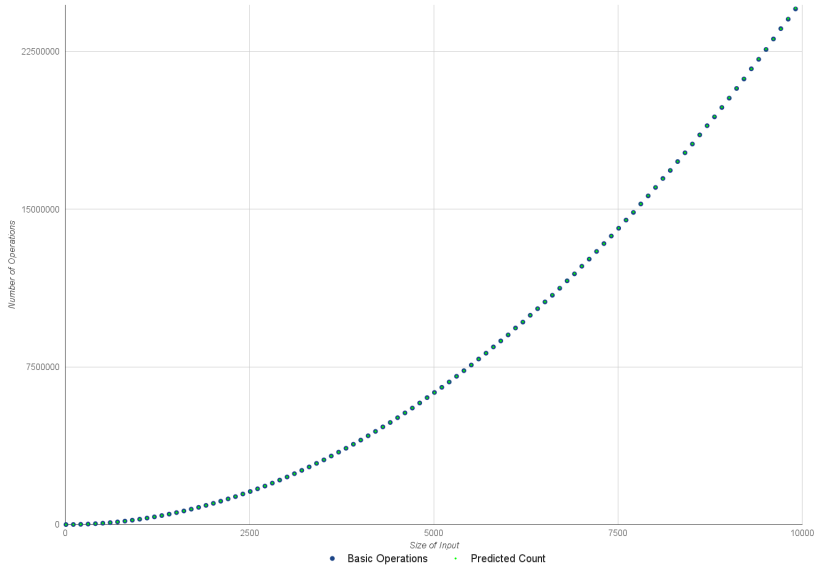

Figure 13: Each blue circle represents the average amount of basic operations it took over 100 iterations to sort an array, where the size of said array can be seen on the horizontal axis. The green circles which are inside of the blue circles represent the count that represents the hypothesised number of operations given the size of the input ($\frac{n^2}{4}$).

M Total Basic Operations vs Size of Input

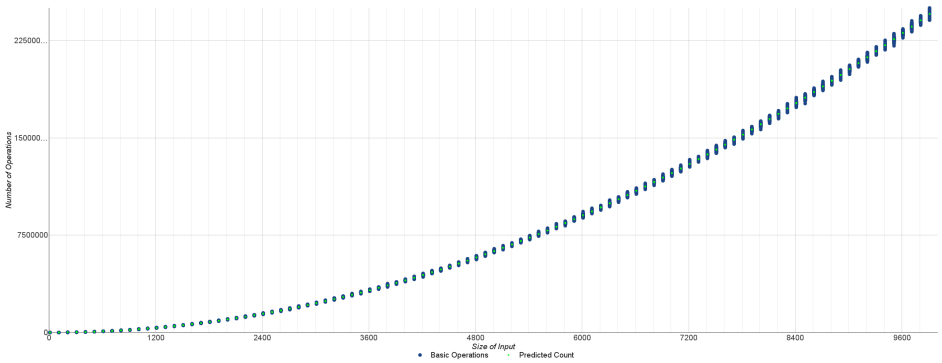

Figure 14: The stark green circles are the count that represents the predicted number of operations given the size of the input which is a the graphical form of $\frac{n^2}{4}$. The blue circles, almost indistinguishable from being a blue line, are points making up every amount of basic operations that was completed at a given size of input. On this graph there are 100 points per Size of input so what is truly being represented here is the range of values for the experiment results.