

CAB302 - Assignment 1

Luke Josh

April 22, 2016

Contents

1	Introduction	2
1.1	Background	2
1.2	Semantics of the Algorithm	2
1.3	An Example	3
2	Theoretical Analysis	3
2.1	Basic Operation	3
2.2	Best Case	3
2.3	Worst Case	3
2.4	Average Case	4
3	Methodology, Tools and Techniques	4
4	Applied Analysis	5
4.1	Tests	5
4.2	Number of Operations	5
5	Source Code	5
6	Bibliography	7
	References	7

1 Introduction

1.1 Background

Insertion sort is an algorithm that takes an input of an unsorted (or sorted) list of numbers, and rearranges them into ascending order. It belongs to a group of algorithms known as sorting algorithms, but is by far from the best. There are a number of more complex algorithms that can work much faster (especially as the size of the array increases), however, it is extremely simple, easy to implement, and a great tool to teach computer science students about algorithms and complexity.

The algorithm can be thought of as separating the array into two subarrays, one that is sorted, and one that isn't. Traditionally, the algorithm begins at the leftmost element - splitting the array into the far left element, and every else to the right. Initially, the first element of the array is sorted - as there is nothing to compare it to. The remaining unsorted elements are then iteratively compared to the sorted elements, and are placed into the index in which they belong. The process is often described as how a human would normally sort a hand of cards. An implementation of the algorithm is included below:

1.2 Semantics of the Algorithm

```
function INSERTIONSORT( $A[0..n-1]$ )  
  for  $i \leftarrow 1$  to  $n-1$  do  
     $v \leftarrow A[i]$   
     $j \leftarrow i-1$   
    while  $j \geq 0$  and  $A[j] > v$  do  
       $A[j+1] \leftarrow A[j]$   
       $j \leftarrow j-1$   
    end while  
     $A[j+1] \leftarrow v$   
  end for  
end function
```

In words, the algorithm can be expressed as follows:

1. Take the n th element of the array, starting from the 2nd element, call this the subject element
2. Compare the n th element with the $n-1$ th element (the element to the left)
3. If the element to the left is larger than the subject element, swap the two elements
4. Else, if there is no item to the left, or the item to the left is less than the subject element, do nothing, and proceed along the array
5. Repeat for all elements of the array

1.3 An Example

This can be observed in a simple example, take the array $A = [1, 3, 4, 2, 0]$ - the algorithm would observed the following steps:

1. Take the Second element, $v = 1$, the element to the left is lower, continue
2. Take the third element, $v = 3$, the element to the left is greater, so call the third element, 4, the subject element
3. Perform $A[j+1] \leftarrow A[j]$, which becomes $A[3] \leftarrow A[2]$, thus $A = [1, 3, 4, 4, 0]$
4. Repeat this operation along the list, until the item to the left is no longer less than the subject element: $A = [1, \mathbf{3}, 3, 4, 0]$
5. Now, as the previous element is not less than the subject element, instead of making the element equal to the one before it ($A[j+1] \leftarrow A[j]$), we set the element to be the subject element ($A[j+1] \leftarrow v$), thus, $A = [1, 2, 3, 4, 0]$
6. Repeat the process for the last element, which yields $A = [0, 1, 2, 3, 4]$

2 Theoretical Analysis

2.1 Basic Operation

To analyse the efficiency of insertion sort, we must define the *basic operation* of the algorithm. The basic operation is said to be the operation which has the most influence on the running time of the algorithm, which can usually be said to be the operation which is performed with the highest frequency. The operation, in this case, is the check $A[j] > v$. This check is performed more than any other in the algorithm.

2.2 Best Case

The best case efficiency for insertion sort is when the array is already sorted. For a sorted array, $A = [a_1, a_2, \dots, a_n]$, each element $a_i \geq a_{i-1} \forall i \in [2, 3, \dots, n]$, and thus, each element that is to be sorted will only check against the element to it's left. Therefore, for a sorted array A with n elements, the number of comparisons that must be performed is $c = n - 1$, giving the algorithm a running time of $\Theta(n)$.

2.3 Worst Case

The worse case efficiency for insertion sort is for an array that is sorted in descending order. An array of this fashion will require a check for each value to the left of each value in the array. For example, if we have the array $A = [5, 4, 3, 2, 1]$, for the first value to be checked, $v = 4$, we must check that the value to the left is not greater, so, 1 comparison. For the second value to be checked, $v = 3$, we must check the two values to the left, so 2 comparisons. It continues in this fashion, and we have the number of comparisons, $c = 1 + 2 + 3 + 4$.

Extending this to an array $B = [b_1, b_2, b_3, \dots, b_n]$, the number of comparisons, c becomes

$$c = 1 + 2 + 3 + \dots + (n - 1) \quad (1)$$

$$c = \sum_{j=1}^{n-1} j \quad (2)$$

$$c = \left(\sum_{j=1}^n j \right) - n \quad (3)$$

$$c = \frac{n(n+1)}{2} - n \quad (4)$$

$$\therefore c = \frac{n(n-1)}{2} \quad (5)$$

2.4 Average Case

Assuming an array $A = [a_1, a_2, \dots, a_n]$ whose values could take any possible permutation of the natural numbers, the average case number of basic operations can be calculated. In this case, we can assume that since all elements are randomly selected from the set of natural numbers, that each element is equally likely to be greater than or less than each other element, and from that, we can state that for each element, half of the elements to the left will have to be checked before one is found that is larger than it, on average. Thus, we can state that for each element a_i in A , approximately $\frac{i}{2}$ comparisons must be made:

$$c = \sum_{j=1}^{n-1} \frac{j}{2} \quad (6)$$

$$c = \left(\sum_{j=1}^n \frac{j}{2} \right) - \frac{n}{2} \quad (7)$$

$$c = \frac{\left(\sum_{j=1}^n j \right)}{2} - \frac{n}{2} \quad (8)$$

$$c = \frac{n(n+1)}{4} - \frac{n}{2} \quad (9)$$

The average case efficiency is much more accurately calculated and quoted by McQuain (McQuain, 2000), as $\frac{n^2}{4}$. Both of these values for the average case efficiency give the algorithm a running time of $\Theta(n^2)$.

3 Methodology, Tools and Techniques

The following sections regarding the tests that were performed to test the algorithm were run on relatively powerful personal computer, with the following

specifications:

- Intel Core i7 @ 2.40GHz
- 8GB DDR3 Memory @ 1600MHz
- Running Ubuntu 14.(64 bit)

All tests were written in C++, and were compiled using the CLion IDE. Average case tests are performed by generating pseudo-random arrays using the c++ rand functions. Graphs that are shown were produced by reading outputs into a Python script using the matplotlib library (Hunter, 2007).

This report was typeset and formatted using the L^AT_EX package.

4 Applied Analysis

4.1 Tests

A number of tests can be applied to the operation on this algorithm to test it's efficiency, and how well it conforms to the theoretical analysis above. This report will detail a count of the number of basic operations performed to sort an array, along with the time taken to sort an array.

4.2 Number of Operations

4.2.1 Best Case

5 Source Code

```
#include <iostream>
#include <fstream>
#include <cstdio>
#include <ctime>
#include <python2.7/Python.h>

using namespace std;

int insertion_sort_op_count(int array_to_sort[], int length){
    int v;
    int j;
    int op_count = 0;

    for(int i = 1; i < length; i++){
        v = array_to_sort[i];
        j = i - 1;
        op_count++;

        while(j >= 0 && array_to_sort[j] > v){
            if (j >= 0){
                op_count++;
            }
        }
    }
}
```

```

        array_to_sort[j + 1] = array_to_sort[j];
        j--;
    }
    array_to_sort[j + 1] = v;
}
return op_count;
}

double insertion_sort_time(int array_to_sort[], int length){
    int v;
    int j;
    clock_t start;
    double duration;

    for(int i = 0; i < length; i++){
        v = array_to_sort[i];
        j = i - 1;

        while(j >= 0 && array_to_sort[j] > v){
            if (j >= 0){
            }
            array_to_sort[j + 1] = array_to_sort[j];
            j--;
        }
        array_to_sort[j + 1] = v;
    }
    return (clock() - start) / (double) CLOCKS_PER_SEC;
}

void print_array(int array[], int length){
    cout << "[";
    for (int i = 0; i < length; i++){
        cout << array[i] << "_";
    }
    cout << "]" << endl;
}

void populate_data(int empty[], int length, int lower, int upper){
    for (int i = 0; i < length; i++){
        empty[i] = rand() % (upper - lower + 1) + lower;
    }
}

void test_range_op_count(int lower, int upper, int iterations){
    ofstream data_file("data.txt");
    int op_count;

    for (int length = lower; length < upper; length++){
        cout << length << "_/_ " << upper << endl;
        op_count = 0;
        for(int repeat = 0; repeat < iterations; repeat++) {
            int gen_array[length];
            populate_data(gen_array, length, 0, length * 2);
            op_count += insertion_sort_op_count(gen_array, length);
        }

        int data_point[2] = {length, op_count / iterations};
    }
}

```

```

        data_file << data_point[0] << "_" << data_point[1] << endl;
    }
    data_file.close();
}

void test_range_time(int lower, int upper, int iterations){
    ofstream data_file("data.txt");
    double time;

    for (int length = lower; length < upper; length++){
        cout << length << "_/" << upper << endl;
        time = 0;
        for(int repeat = 0; repeat < iterations; repeat++) {
            int gen_array[length];
            populate_data(gen_array, length, 0, length * 2);
            time += insertion_sort_time(gen_array, length);
        }

        double data_point[2] = {length, time / (double) iterations};
        data_file << data_point[0] << "_" << data_point[1] << endl;
    }
    data_file.close();
}

void make_plot(){
    Py_Initialize();
    PyObject *sys_path = PySys_GetObject("path");
    PyList_Append(sys_path, PyString_FromString("."));
    PyRun_SimpleString("import _plotter\nplotter.main()\n");
    Py_Finalize();
}

int mainlol(){
    time_t t;
    srand(time(&t));
    test_range_time(1, 500, 5);
    make_plot();
    return 0;
}

int main(){
    int length = 10;
    int arr[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
    int op_count = insertion_sort_op_count(arr, 10);
    cout << "OP_COUNT:_" << op_count << endl;
    print_array(arr, 10);
}

```

6 Bibliography

References

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing In Science and Engineering*, 9(3), 90–95.

McQuain. (2000). *Data structures and algorithms*.
<http://courses.cs.vt.edu/cs3114/Fall09/wmcquain/Notes/T14.SortingAnalysis.pdf>.
Computer Science at Virginia Tech. (Online; accessed 15-April-2016)