CAB302 - Assignment 1

Luke Josh

April 22, 2016

Contents

1 Introduction

1.1 Background

Insertion sort is an algorithm that takes an input of an unsorted (or sorted) list of numbers, and rearrangesthem into asscending 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 seperating the array into two subarrays, one that is sorted, and one that isn't. Traditionaalgorithm begins at theleftmost element - splitting the array into the far left element, and every else to the right. Initially, the first element of tis sorted -as there is nothing to compare it to. The remaining unsorted elements are then itteratively comparedto the sorted elements, and are placed into the index in which they belong. The process is often described as how a human would 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
```

1.3 Algorithm in words

In words, the algorithm can be expressed as follows:

1. Take the nth element of the array, starting from the 2nd element, call this the subject element
2. Compare the nth element with the n-1th 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

1.4 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, $A = [a_1, a_2, ..., a_n]$, each element $a_i \geq a_{i-1} \forall i \in [2, 3, ...n]$, 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 $B = [b_1, b_2, b_3, ..., b_n]$, the number of comparisons, c becomes

$$c = 1 + 2 + 3 + ... + (n - 1) \tag{1}$$

$$c = \sum_{j=1}^{n-1} j \tag{2}$$

$$c = (\sum_{j=1}^{n} j) - n \tag{3}$$

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

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

2.4 Average Case

Assuming an array $A = [a_1, a_2, ..., a_n]$ who's values could take any possible permutation of the natural numbers, the áverage caseńumber 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:

$$c = \sum_{j=1}^{n-1} \frac{j}{2} \tag{6}$$

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

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

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

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 14.04 (64 bit)

All tests were written in C++, and were compiled using the CLion IDE. Average case tests are performed by generating 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 LaTeX package.

4 Experimental 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. All referenced figures are included in Appendix A.

4.2 Number of Operations

The number of basic operations for the Insertion Sort algorithm is a well defined concept, and can be quantitatively measured and compared to a literature value. The following sections outline results from several tests performed using the included code. Referring to the pseudo code

4.2.1 Best Case

Recall from above that the best case running time occurs when an array $A = [a_1, a_2, ..., a_n]$ is already sorted such that $a_n \leq a_{n+1} \forall\ n \in [1, 2, ..., n-1]$, and that it is expected to run with n - 1 operations. To test this, we can generate arrays that are sorted, and then measure the number of operations required to complete the algorithm. To ensure that the relation is indeed correct, the expected and actual number of basic operations will be calculated for arrays of size 2 through 1000. Two graphs have been produced for this test - one that displays the data that was measured (Figure 1), and another that shows the expected output (Figure 2), these have been included separately to highlight the fact that the two data are exactly the same.

4.2.2 Worst Case

The worst case for Insertion Sort was said to be an array $A = [a_1, a_2, ..., a_n]$, sorted in reverse order such that $a_n \geq a_{n+1} \forall\ n \in\ [1, 2, ..., n-1]$. The expected number of operations here was said to be $\frac{n(n-1)}{2}$. To test this, data is again generated, but this time in descending order. To ensure correctness, it has also been tested on arrays of size 2 through 1000. For this test case, only one image was shown, displaying the expected output and the actual output on the same graph. Barely any distinction is visible between these two lines, indicating that our expected number of operations is indeed correct. The graphs have been included as Figure 3.

4.2.3 Average Case

The average case for Insertion Sort was said to be an array of numbers, each number being equally likely to be greater than or less than each other number, the expected number of operations was quoted to be $\frac{n^2}{4}$. To test the expected value of basic operations, arrays of varying sizes were generated, with each element being a random number between zero and twice the number of elements. Twice the number of elements was chosen just so that there are less occurrences of duplicate values, and that the array has a greater chance of entropy, and thus, a more accurate "average" result. We generate a number of these arrays of each size, calculate the number of operations, and take the average - this is to ensure functional correctness, and to get a better reading of the average case. Arrays of size 2 through 1000 were tested, taking an average of 10 generations each time.

A graph has been included as Figure 4 showing the expected and theoretical data for this test. Unlike the previous trials on best and worst case data, these trials will not produce the exact expected value each time, but when averaged, will tend towards the expected value. This can be verified visually, the blue dots representing a measured number of operations for a particularly sized array does not form a seemingly continuous line like the other tests.

4.3 Processing Time

The time taken to

5 Appendices

5.1 Appendix A - Graphs

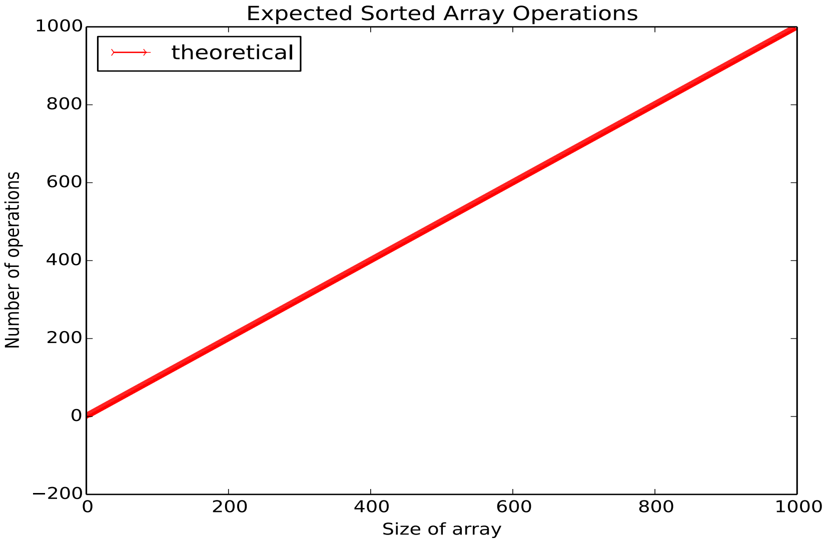

Figure 1: Expected number of operations for sorted arrays of size 2 through 1000

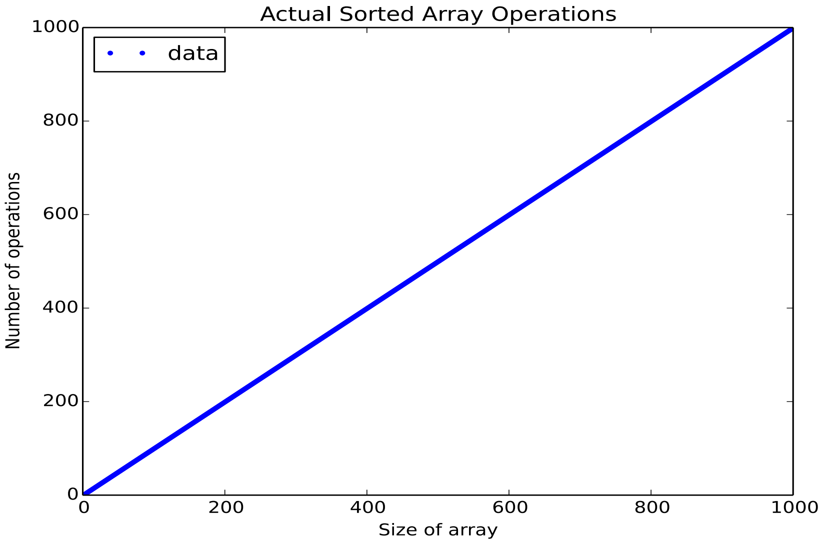

Figure 2: Recorded number of operations for sorted arrays of size 2 through 1000

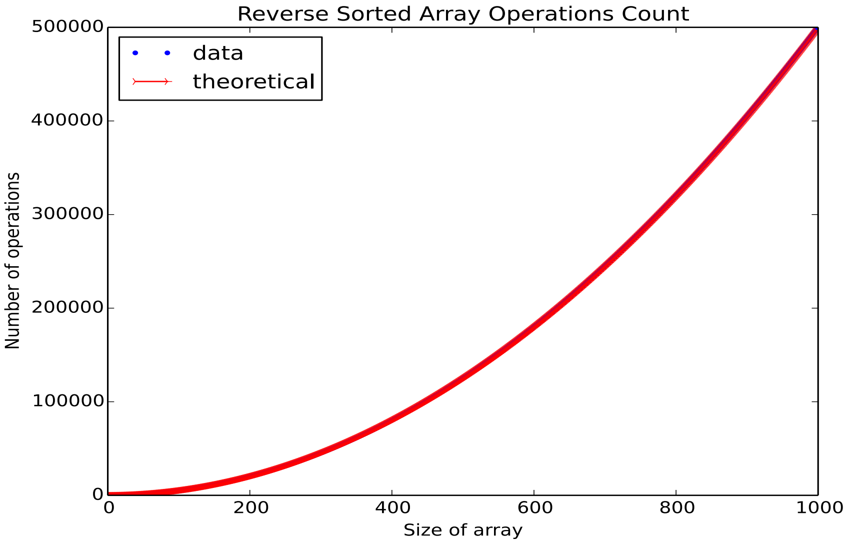

Figure 3: Number of operations on reversed sorted arrays of size 2 through 1000

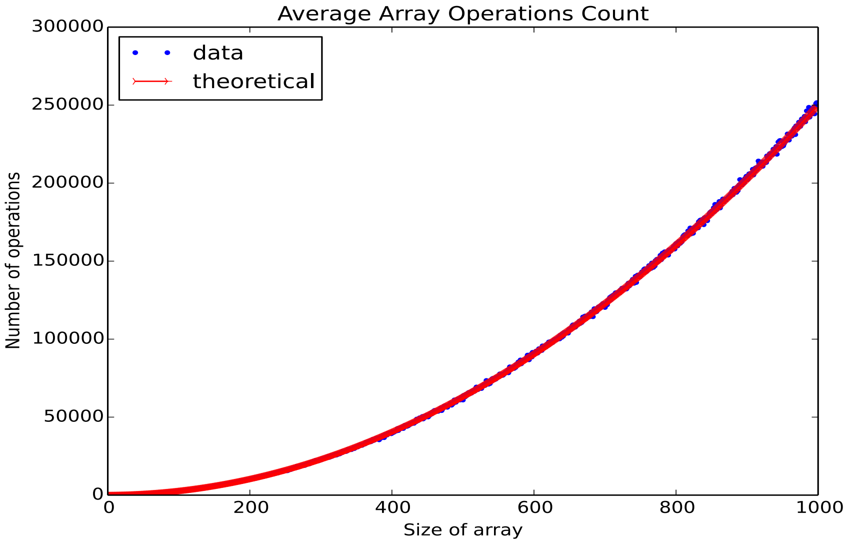

Figure 4: Number of operations on average arrays, for arrays of size 2 through 1000, averaged over 10 trials

5.2 Source Code

```
#include <iostream>
#include <fstream>
#include <cstdio>
#include <ctime>

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){
            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;
    start = clock();

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

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

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

void write_data_points_to_file(string filename, double data_points[][2], int num_points){
    ofstream file(filename);

    for(int i = 0; i < num_points; i++){
        file << data_points[i][0] << ”␣” << data_points[i][1] << endl;
    }

    file.close();

}

void test_range_operations(string filename, int lower, int upper, int iterations, void (*genFu
    double operations;
    double data_points[upper - lower][2];
    for (int length = lower; length <= upper; length++){
        cout << length << ”␣/␣” << upper << endl;
        operations = 0;
        for(int repeat = 0; repeat < iterations; repeat++) {
            int gen_array[length];
            genFunction(gen_array, length, 0, upper * 2);
            operations += insertion_sort_op_count(gen_array, length);
        }
        data_points[length - lower][0] = length;
        data_points[length - lower][1] = operations / (double) iterations;
    }
    write_data_points_to_file(filename, data_points, upper - lower);
}

void test_range_time(string filename, int lower, int upper, int iterations, void (*genFunction
    double time;
    double data_points[upper - lower][2];
    for (int length = lower; length <= upper; length++){
        cout << length << ”␣/␣” << upper << endl;
        time = 0;
        for(int repeat = 0; repeat < iterations; repeat++) {
            int gen_array[length];
            genFunction(gen_array, length, 0, upper * 2);
            time += insertion_sort_time(gen_array, length);
        }
        data_points[length - lower][0] = length;
        data_points[length - lower][1] = time / (double) iterations;
    }
    write_data_points_to_file(filename, data_points, upper - lower);
}

int main(){
    //Test number of operations for sorted arrays
    test_range_operations(”sorted_array_op_count.txt”, 2, 1000, 10, &populate_sorted_data);

    //Test number of operations for sorted arrays
    test_range_operations(”reverse_sorted_array_op_count.txt”, 2, 1000, 10, &populate_reverse_s

    //Test number of operations for random, ”average” arrays
    test_range_operations(”random_array_op_count.txt”, 2, 1000, 10, &populate_random_data);
}
```

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)