Insertion sort

Ben Mezger

<2015-07-25 Sat>

1 Introduction

We need to sort numbers quite a lot when programming, for example, suppose you wrote a computer card game, each card will be shuffled at the beginning, and each user starts with 6 cards, but they need to be sorted, like the following.

Table 1: Shuffled card vs sorted cards.

Shuffled	6	3	1	4	2	5
Sorted	1	2	3	4	5	6

2 Solution

One solution for this problem, is to use an algorithm called *insertion sort*. Insertion sort is fairly simple, but it's a commonly used algorithm. Insertion sort works like we human sort cards. say we have 6 cards lying in the table, the cards are from 1-6, but they are not sorted. the cards on the table looks like the shuffled line from `Table 1`. Our algorithm, needs to take care of two elements, the first element (i) and the second element, (j). While j points to the second element, i points to j - *1*. We do so, because later, we can compare j with i and check if $i \geq j$, if so, we swap the element i with the j each time, until $i \leq j$.

A	6	3	1	4	2	5
current	i	j				

In C, this algorithm can be represented as the following; it receiving an array A and an total number of elements, so we can keep track where the list ends.

```c
void insertion_sort(int *A, size_t len){
    int j, i, key;

    for (j=1; j < (int) len; j++){
        key = A[j];
        // Insert A[j] into the sorted sequence A[1.. j-1]
        i = j - 1;
        while (i >= 0 && A[i] > key){
            A[i + 1] = A[i];
            i = i - 1;
        }
        A[i + 1] = key;
    }
```

Listing 1: Insertion sort algorithm in C

Line 4, we start j at the index 1 of array A up to or equal to len which is the total number of elements we have in array A. If *len* is 0, means our array is emtpy, so it will never enter line 4, and empty array is a sorted array. The same works if *len* is one, meaning one element in our array. Line 5, we assign *key* to the current value of A[j], we do this because otherwise we would lose the value of A[j] at in line 9. Line 7 is simple, we get the index before j. From line 8-10 is where the sorting begins. We check if i is ≥ 0 and if A[i] is $>$ key. If A[i] $>$ key is true, it means it's not sorted. So we will assign A[i + 1] = A[i] until i > 0, shuffling the elements to the left.

Table 2: Insertion sort operation

$\mathbf{A_1}$	6	3	1	4	2	5	current
$\mathbf{current_1}$	i	j					
$\mathbf{A_1}$	3	6	1	4	2	5	1 shift
$\mathbf{current_2}$		i	j				
$\mathbf{A_2}$	1	3	6	4	2	5	2 shifts
$\mathbf{current_3}$			i	j			
$\mathbf{A_3}$	1	3	4	6	2	5	3 shifts
$\mathbf{current_4}$				i	j		
$\mathbf{A_4}$	1	2	3	4	6	5	4 shifts
$\mathbf{current_5}$					i	j	
$\mathbf{A\{5\}}$	1	2	3	4	5	6	final

2.1 Analysis

In the chapter, I am assuming you are already familiar with Algorithm analysis.

Analyzing *insertion sort* algorithm can give us a good idea of when we should use this algorithm and when we should not. Analyzing can predict the resources the algorithm requires.

The time *insertion sort* requires depends on its input; if the input is 10 elements, it would take shorter than if the input was $6*10^3$. On the other hand, insertion sort also requires a different amount of time for two input sequences of the same size if one of the input sequences is halfway sorted.

2.1.1 Best case

The best case input is an array that is already sorted. If the array is already sorted, we have a linear running time: $\mathcal{O}(n)$.

2.1.2 Worst case

The simplest worst case for *insertion sort* algorithm is if the array is in reverse order. If the array is in reversed order, the inner loop (line 8-10) will scan and shift the **entire** "sorted" array before inserting the next element: $\mathcal{O}(n^2)$.

$\mathbf{A_1}$	6	5	4	3	2	1	current
$\mathbf{current_1}$	i	j					
$\mathbf{A_2}$	5	6	4	3	2	1	1 shift
$\mathbf{current_2}$		i	j				
$\mathbf{A_3}$	4	5	6	3	2	1	2 shifts
$\mathbf{current_3}$			i	j			
$\mathbf{A_4}$	3	4	5	6			3 shifts
$\mathbf{current_4}$				i	j		
$\mathbf{A_5}$	2	3	4	5	6	1	4 shifts
$\mathbf{current_5}$					i	j	
$\mathbf{A_6}$	1	2	3	4	5	6	5 shifts
$\mathbf{A_7}$	1	2	3	4	5	6	final

2.1.3 Average case

The average sort is **quadratic**, which means that insertion sort is impractical for sorting a large array: $\mathcal{O}(n^2)$ with a space complexity of $\mathcal{O}(1)$.