Sortieren durch direktes Einfügen (Insertion Sort)

Introduction

If you're majoring in Computer Science, *Insertion Sort* is most likely one of the first sorting algorithms you have heard of. It is intuitive and easy to implement, but it's very slow on large arrays and is almost never used to sort them.

Insertion sort is often illustrated by comparing it to sorting a hand of cards while playing rummy. For those of you unfamiliar with the game, most players want the cards in their hand sorted in ascending order so they can quickly see which combinations they have at their disposal.

The dealer deals out 14 cards to you, you pick them up one by one and put them in your hand in the right order. During this entire process your hand holds the sorted "subarray" of your cards, while the remaining face down cards on the table are unsorted - from which you take cards one by one, and put them in your hand.

Insertion Sort is very simple and intuitive to implement, which is one of the reasons it's generally taught at an early stage in programming. It's a *stable*, *in-place* algorithm, that works really well for nearly-sorted or small arrays.

Let's elaborate these terms:

- **in-place**: Requires a small, constant additional space (no matter the input size of the collection), but rewrites the original memory locations of the elements in the collection.
- **stable**: The algorithm maintains the relative order of equal objects from the initial array. In other words, say your company's employee database returns *"Dave Watson"* and *"Dave Brown"* as two employees, in that specific order. If you were to sort them by their (shared) first name, a *stable* algorithm would guarantee that this order remains unchanged.

Another thing to note: Insertion Sort doesn't need to know the entire array in advance before sorting. The algorithm can receive one element at a time. Which is great if we want to add more elements to be sorted - the algorithm only *inserts* that element in it's proper place without "re-doing" the whole sort.

Insertion Sort is used rather often in practice, because of how efficient it is for small (~10 item) data sets. We will talk more about that later.

How Insertion Sort Works

An array is partitioned into a "sorted" subarray and an "unsorted" subarray. At the beginning, the sorted subarray contains only the first element of our original array.

The first element in the unsorted array is evaluated so that we can *insert* it into its proper place in the sorted subarray.

The insertion is done by moving all elements larger than the new element one position to the right.

Continue doing this until our entire array is sorted.

Keep in mind however that when we say an element is larger or smaller than another element - it doesn't necessarily mean larger or smaller integers.

We can define the words "larger" and "smaller" however we like when using custom objects. For example, point *A* can be "larger" than point *B* if it's further away from the center of the coordinate system.

We will mark the sorted subarray with bolded numbers, and use the following array to illustrate the algorithm:

8, 5, 4, 10, 9

The first step would be to "add" *8* to our sorted subarray.

8, 5, 4, 10, 9

Now we take a look at the first unsorted element - *5*. We keep that value in a separate variable, for example `current`, for safe-keeping. *5* is less than *8*. We move *8* one place to the right, effectively overwriting the *5* that was previously stored there (hence the separate variable for safe keeping):

8, *8*, 4, 10, 9 (`current` = 5)

5 is lesser than all the elements in our sorted subarray, so we insert it in to the first position:

5, **8**, 4, 10, 9

Next we look at number *4*. We save that value in `current`. *4* is less than *8* so we move *8* to the right, and do the same with *5*.

5, **5**, **8**, 10, 9 (`current` = 4)

Again we've encountered an element lesser than our entire sorted subarray, so we put it in the first position:

4, **5**, **8**, 10, 9

10 is greater than our rightmost element in the sorted subarray and is therefore larger than any of the elements to the left of *8*. So we simply move on to the next element:

4, **5**, **8**, **10**, 9

9 is less than *10*, so we move *10* to the right:

4, **5**, **8**, **10**, *10* (`current` = 9)

However, *9* is greater than *8*, so we simply insert *9* right after *8*.

4, **5**, **8**, **9**, **10**

Insertion Sort in Practice

Insertion sort may seem like a slow algorithm, and indeed in most cases it is too slow for any practical use with its $O(n^2)$ time complexity. However, as we've mentioned, it's very efficient on small arrays and on nearly sorted arrays.

This makes *Insertion Sort* very relevant for use in combination with algorithms that work well on large data sets.

For example, Java used a *Dual Pivot Quick Sort* as the primary sorting algorithm, but used Insertion Sort whenever the array (or subarray created by Quick Sort) had less than 7 elements.

Another efficient combination is simply ignoring all small subarrays created by Quick Sort, and then passing the final, nearly-sorted array, through Insertion Sort.

Another place where Insertion Sort left its mark is with a very popular algorithm called *Shell Sort*. Shell Sort works by calling Insertion Sort to sort pairs of elements far apart from each other, then incrementally reducing the gap between elements to should be compared.

Essentially making a lot of Insertion Sort calls to first small, and then nearly-sorted larger arrays, harnessing all the advantages it can.

Conclusion

Insertion Sort is a very simple, generally inefficient algorithm that nonetheless has several specific advantages that make it relevant even after many other, generally more efficient algorithms have been developed.

It remains a great algorithm for introducing future software developers into the world of sorting algorithms, and is still used in practice for specific scenarios in which it shines.