Courses Practice Roadmap Pro

Algorithms and Data Structures for Beginners

11 / 35

4 Stacks

Linked Lists

10 - Insertion Sort

Mark Lesson Complete View Code Prev Next

Sorting

Suggested Problems

Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Sort An Array	Medium		

Insertion Sort

Insertion sort is one sorting algorithm used to sort data in different data structures. It is one of the simplest sorting algorithms that works best when the data size is small (we will discuss why this is the case soon).

Concept

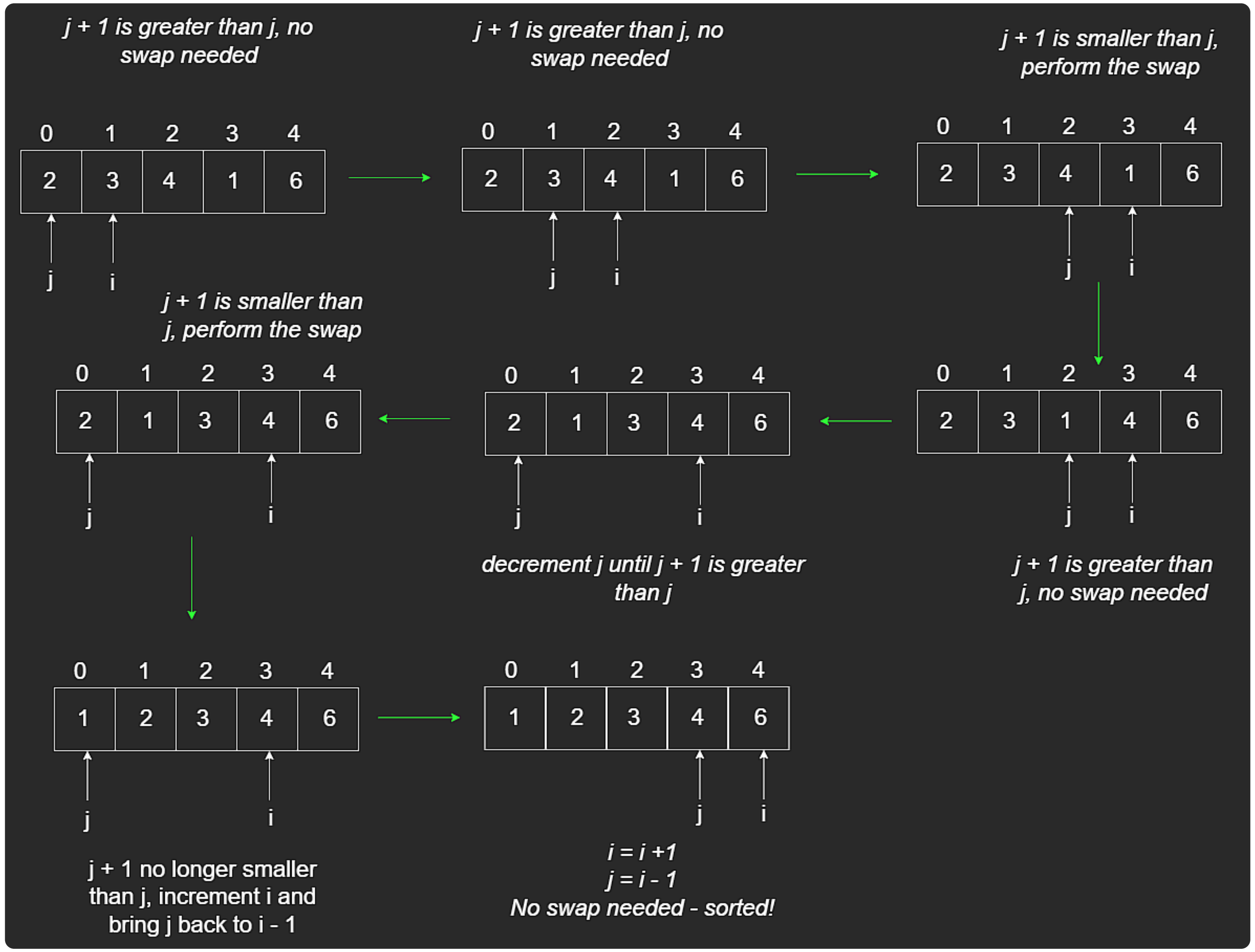
Say that we have an array of size 5, populated with values: `[2,3,4,1,6]` . Our goal is to sort the array so we end up with `[1,2,3,4,6]` . Insertion sort says to break the arrays into sub-arrays and sort them individually, which results in a sorted array. If we had an array of size 1, that would already be sorted because there is no other element to compare it to. As such, we assume that the first element is sorted because we treat it as its own subarray.

The next subarray will be of size 2, which is `[2,3]`. To perform the sort, the two elements will need to be compared. For an array of size 2, this is trivial. However, when we get to the full array of size 5, there is no way to keep track of where each element is without using pointers. So, let's take two pointers, `i` and `j`.

- `j` will always be behind `i` such that it will never cross `i`.
- The `i` pointer points of the index $n - 1$ where n is the size of the current sub-array.
- The `j` pointer starts off with being one index behind `i` and as long as `j` does not go out of bounds, that is, it is not at a negative index, and the `j+1` element is smaller than the `jth` element, we keep decrementing `j`. This will ensure that we have sorted all the elements before the `ith` index before moving to the next sub-array (iteration). This is demonstrated in the pseudocode below.

```
fn insertionSort(arr):
    for i = 1 until end of the array:
        j = i - 1
        // j is not out of bounds and j + 1 is smaller than j
        while j >= 0 and arr[j + 1] < arr[j]:
            // arr[j] and arr[j + 1] are out of order so swap them
            tmp = arr[j + 1]
            arr[j + 1] = arr[j]
            arr[j] = tmp
            j -= 1
    return arr
```

Applying the code to the example above, the steps look like the following.


j + 1 is greater than j, no swap needed
j + 1 is greater than j, no swap needed
j + 1 is smaller than j, perform the swap
j + 1 is smaller than j, perform the swap
decrement j until j + 1 is greater than j
j + 1 is greater than j, no swap needed
j + 1 no longer smaller than j, increment i and bring j back to i - 1
i = i +1
j = i - 1
No swap needed - sorted!

> *You may be thinking, will insertion sort work on different data types apart from the integers? The answer is yes. As long as there is a way to compare two values, this algorithm will work on any data type.*

> *The terms above such as subarray and subproblem may remind you of the concept of recursion, which we just learned. So, why not just use recursion to perform the sort? We could, but implementing it recursively will not have any advantage over the iterative solution. In fact, the recursive solution usually takes more time and memory. And the iterative implementation is more intuitive.*

Stability

Stability in a sorting algorithm refers to the relative order of the elements after the sorting is done. Take `[7,3,7]` for example. There are two 7s, one at the `0th` index and the other at the `2nd` index. We know that the relative order of these two 7s will stay the same since 3 will swap with the 7 at the `0th` index and then the while loop will never be implemented.

This is called a **stable** sorting algorithm. Insertion sort is stable, meaning that it is guarenteed that the relative order will remain the same. In an unstable sorting algorithm, this is not guarenteed and we shall see why in a few chapters.

Time Space Complexity

For performing insertion sort on any data set of size n, in the best case, the array is already sorted which would cost $O(n)$. This is because we will still have to go through every element in the array but our while loop would never execute. Can you think of the worst case scenario? In the worst case, all of the elements are sorted in reverse order, which means the while loop will execute n times inside the for loop. This leads to an $O(n^2)$ time complexity.

A Deeper Dive - why $O(n^2)$?

In the worst case, insertion sort performs n^2 operations, where n is the size of the array. In the first for loop iteration, we sort the first two elements, then 3, 4, and finally 5. Well, if all of elements are in a reverse order, not only do we have to go through the entire array - the for loop, but also perform swap at every single step, which in total will be n. So in total, we can conclude that this is n^2. For sure, there is a very neat mathematical proof, but for the purpose of this course, this explanation is enough.

Closing Notes

Insertion sort is a great algorithm when the input size is small, but falls short when n is very large due to its $O(n^2)$ complexity. Still it is good to know the pros and cons.