Lesson:

Insertion Sort

PW SKILLS

Pre-Requisites

- C++ arrays and loops

List of Concepts Involved

- Insertion Sort Visualization
- Insertion Sort Code
- Insertion Sort Time and Space Complexity
- Is Insertion Sort Stable ?
- Applications of Insertion sort

As of now we are familiar with the concept of Bubble sort and selection sort from the previous lessons, we will now expand our knowledge further to learn Insertion sort.

Topic 1: Insertion Sort Visualization

Algorithm Explanation:
Let's suppose you have an array of integer values and you want to sort these values in non-decreasing order. Then the logic of Insertion sort states that you need to insert the element into the correct position by looping over the array, thereby forming a sorted array on the left.
Then increase the pointer and similarly for the next element find its correct position in the left sorted array and insert it. Do this till the pointer reaches the end of the array.

Insertion sort is a way of sorting where we pick one element , find its correct position and then insert it there or vice versa where we pick a position and find the correct element which should be there in the sorted array and then put it there. If we pick the positions from left to right then it boils down to finding the smallest element because the leftmost position should have the smallest element and put it there and then repeat the same process for all the indices.

- Traverse the array from left to right
- Compare the current element to its previous element and keep swapping it until the previous element is smaller than the current element.
- This way each element reaches it correct position

Example (with all the steps):

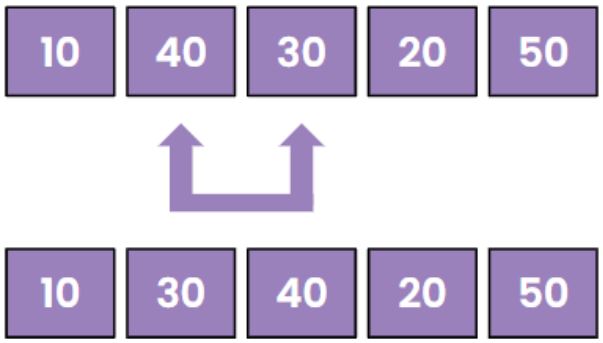
Array = [10, 40, 30, 20, 50]

In the above example, as we iterate through the array from left to right, we keep inserting elements into their corresponding correct positions thereby forming a sorted array on the left.

First Pass: 10 is to be compared with its predecessor, here there is no previous element so simply move to the next element.
The sorted part of the array on the left is currently 10.
[**10**, 40, 30, 20, 50]

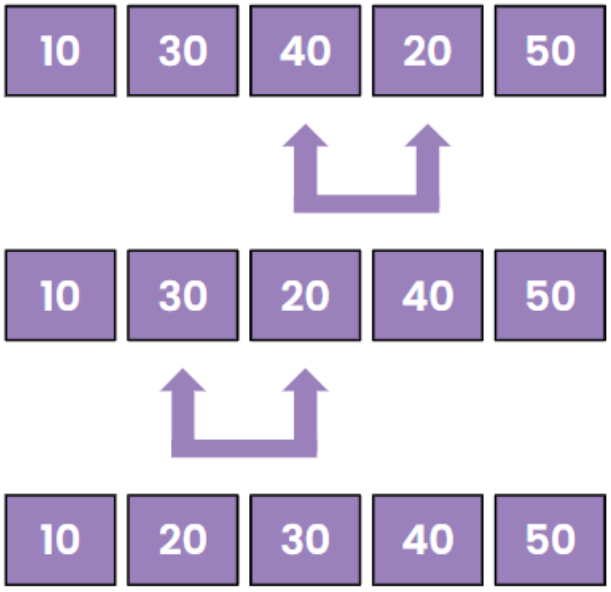
Second Pass: 40 is compared with its predecessor and the previous element is already smaller than the current element (10 < 40) so no need to swap anything and move to the next element.
The sorted part of the array on the left is currently [10,40].
[10, **40**, 30, 20, 50]

Third Pass: 30 is now compared with its predecessor and here it is 40. As 40 < 30, we need to keep swapping the current element with its predecessors until the previous element is smaller than the current element. This way 30 will reach its correct position.
[10, 40, **30**, 20, 50] (arrow between 40 and 30)
[10, **30**, 40, 20, 50]



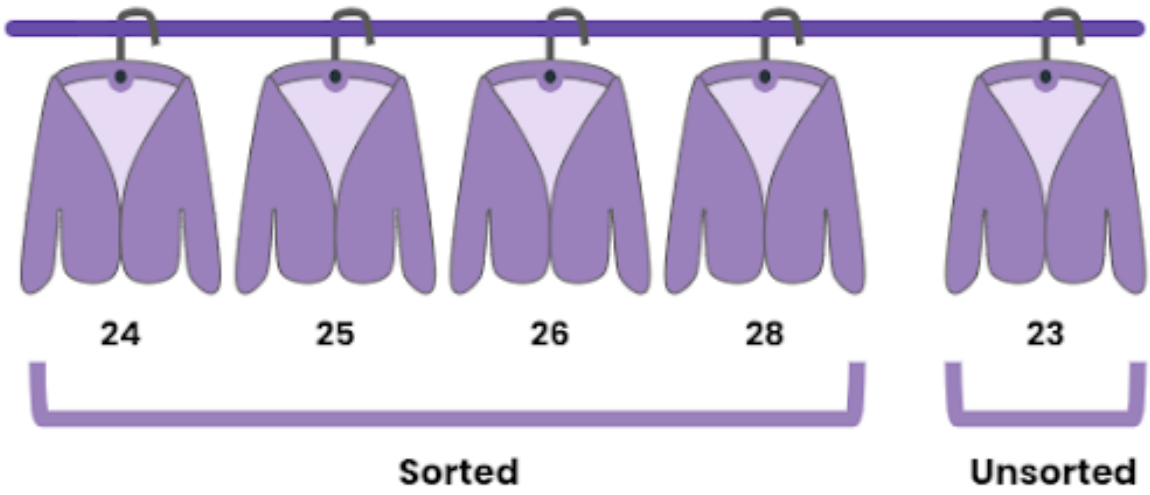
The sorted part of the array on the left is currently [10,30,40].

Fourth Pass: 20 is now compared with its predecessor that is 40 so as it is smaller than 40. We keep swapping it, it now reaches and array looks after swapping will be [10,30,20,40]. Again 20 is checked with its predecessor and as it is smaller than 30 also, it will be swapped again and the array will look like [10,20,30,40]. Now 20 has reached its correct position and no more swaps are needed.
[10, 30, 40, **20**, 50] (arrow between 20 and 40)
[10, 30, **20**, 40, 50] (arrow between 30 and 20)
[10, **20**, 30, 40, 50]



Fifth Pass: Here the current element is 50 and is compared with its predecessor and no swaps are needed here. So the final array looks like this - [10,20,30,40,50]
[10,20,30,40,**50**]

Real life example 1 :
Have you taken note, how tailors organize customers' shirts in their closet, according to measure. So they embed a new shirt at the right position, for that, they move existing shirts until they discover the correct place. If you consider the wardrobe as an array and shirts as a component, you may discover that we ought to move existing components to discover the correct place for the unused element.

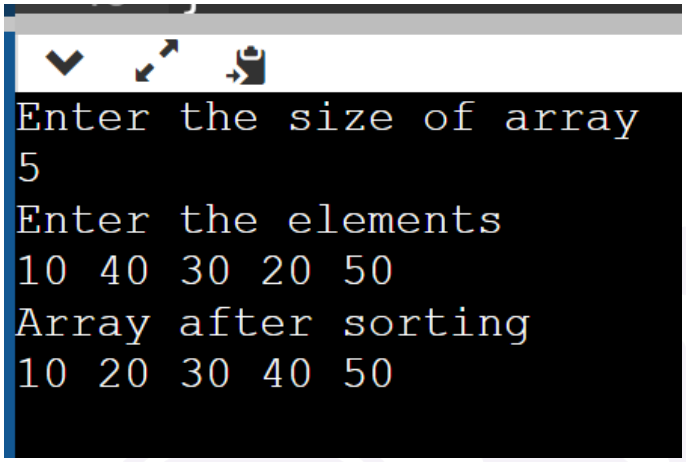


Real life example 2:
Imagine that you are playing a game of cards. You're holding the cards in sorted order. Let's assume that the merchant hands you a new card. You may discover the position of the new card and embed it within the correct position. The existing cards more prominent than the unused card would be moved right by one place.

The same logic is used in insertion sort. You loop over the Array starting from the first index. Each new position is analogous to a new card, & you need to insert it in the correct place in the sorted subarray to your left.

Topic : Insertion Sort Code:

https://pastebin.com/c3xnKxXD



Topic : Insertion Sort Time Complexity:

The time complexity of the following algorithm will be **O(n²)** as for every particular element we find its correct position and insert it at that particular position. This requires O(n) time complexity for one element. We are doing this for every element so **O(n²)** time complexity in total.

Topic : Insertion Sort Space Complexity:

It does not require any extra space so space complexity is O(1).

Topic : Is insertion sort stable?

Here we just pick an element and place it in its correct place and in the logic we are only swapping the elements if the element is larger than the key, i.e. we are not swapping the element with the key when it holds equality condition,so insertion sort is stable sort.

Topic : Insertion Sort Applications

The insertion sort is used when:

- the array is has a small number of elements
- there are only a few elements left to be sorted

Upcoming Class Teasers:

- Problems Based on sorting algorithms.