Insertion Sort

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is $O(n^2)$, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as -

- o Simple implementation
- o Efficient for small data sets
- o Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Now, let's see the algorithm of insertion sort.

Algorithm

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element, and store it separately in a **key.**

Step3 - Now, compare the **key** with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

Working of Insertion sort Algorithm

Now, let's see the working of the insertion sort Algorithm.

To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.

Let the elements of array are -

| 12 | 31 | 25 | 8 | 32 | 17 |

Initially, the first two elements are compared in insertion sort.

| 12 | 31 | 25 | 8 | 32 | 17 |

Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.

| 12 | 31 | 25 | 8 | 32 | 17 |

Now, move to the next two elements and compare them.

| 12 | 31 | 25 | 8 | 32 | 17 |

| 12 | 31 | 25 | 8 | 32 | 17 |

Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

| 12 | 25 | 31 | 8 | 32 | 17 |

Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

| 12 | 25 | 31 | 8 | 32 | 17 |

| 12 | 25 | 31 | 8 | 32 | 17 |

Both 31 and 8 are not sorted. So, swap them.

| 12 | 25 | 8 | 31 | 32 | 17 |

After swapping, elements 25 and 8 are unsorted.

| 12 | 25 | 8 | 31 | 32 | 17 |

So, swap them.

| 12 | 8 | 25 | 31 | 32 | 17 |

Now, elements 12 and 8 are unsorted.

| 12 | 8 | 25 | 31 | 32 | 17 |

So, swap them too.

| 8 | 12 | 25 | 31 | 32 | 17 |

Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.

| 8 | 12 | 25 | 31 | 32 | 17 |

Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

| 8 | 12 | 25 | 31 | 32 | 17 |

Move to the next elements that are 32 and 17.

| 8 | 12 | 25 | 31 | 32 | 17 |

17 is smaller than 32. So, swap them.

8	12	25	31	17	32

8	12	25	31	17	32

Swapping makes 31 and 17 unsorted. So, swap them too.

8	12	25	17	31	32

8	12	25	17	31	32

Now, swapping makes 25 and 17 unsorted. So, perform swapping again.

8	12	17	25	31	32

Now, the array is completely sorted.