Insertion Sort Algorithm

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst-case complexity are of $O(n^2)$, where **n** is the number of items.

How Insertion Works

a. If it is the first element, it is already sorted. return 1;

b. Pick next element

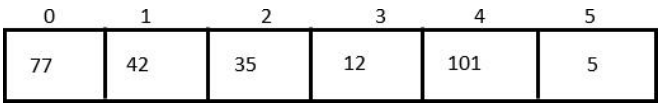
c. Compare with all elements in the sorted sub-list

d. Shift all the elements in the sorted sub-list that is greater than the value to be sorted

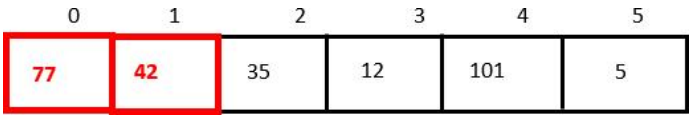
e. Insert the value

f. Repeat until list is sorted

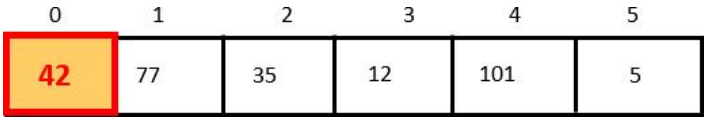
We take an unsorted array for our example.



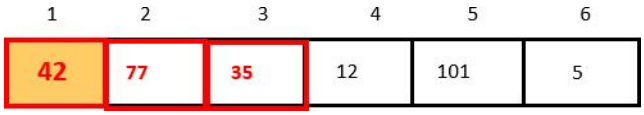
Insertion sort compares the first two elements



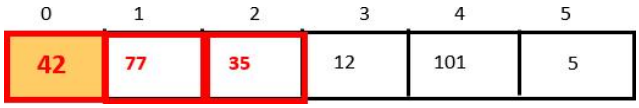
It finds that 77 is not in the correct position. We will swaps 77 with 42.



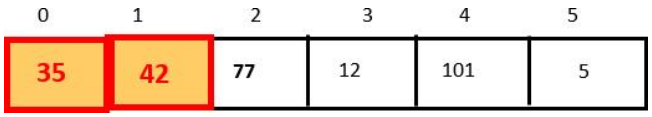
By now we have 42 in the sorted sub-list. Next, it compares 77 with 35.



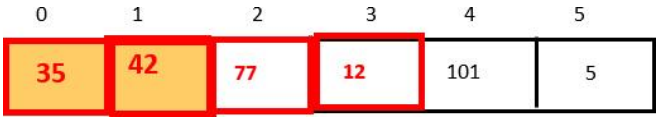
These values are not in a sorted order. So we swap them.



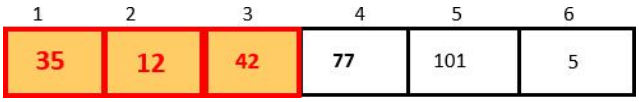
It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 42, and 35 is less than 42. So, we swap them two.



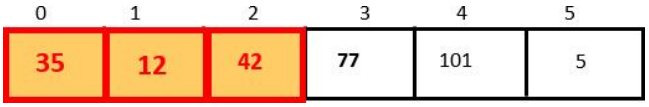
By now we have 35 and 42 in the sorted sub-list.



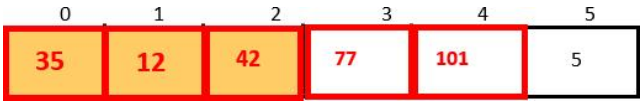
Next, it compares 77 with 12. These values are not in a sorted order. So we swap them.



However, swapping makes 12 and 42 unsorted. Hence, we swap them too

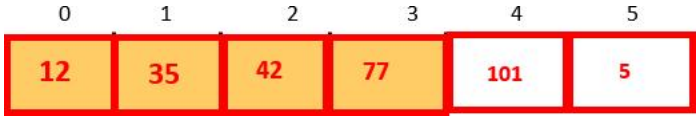


Again we find 12 and 35 in an unsorted order. Swap them again



By the end of third iteration, we have a sorted sub-list of 3 items.

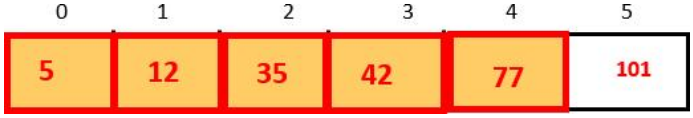
Insertion sort moves ahead and compares 77 with 101.



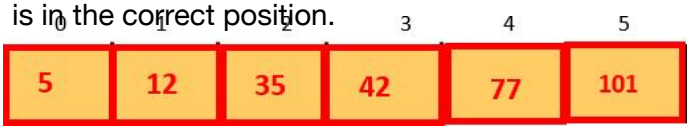
Now, we have a sorted sub-list of 4 items. Again, we will compare the next two elements 101 and 5It finds that both 77 and 101 are already in ascending order.

Again we find 101 and 5 in an unsorted order. Swap it.

It swaps 101 with 5. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list element 12,15,42 and 77 is greater than 5. Hence, the sorted sub-list remains sorted after swapping.



Now we have 5 elements in the sorted sub-list. Next, it compare the last two elements. And finds that 101 is in the correct position.



Now, all elements are in the right position.

Implementing the Insertion sort algorithm in C#

```
using System;

namespace InsertionSort
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] myarray = { 4, 1, 9, -13, 90, 56, 81, 34, -2, -15, 60, 88 };
            Console.WriteLine("Array before sorting...");
            foreach (int x in myarray)
                Console.Write(x + "  ");
            Sort.InsertionSort(myarray);
            Console.WriteLine("\nArray after sorting...");
            foreach (int x in myarray)
                Console.Write(x + "  ");
            Console.WriteLine();
            Console.ReadKey();
        }
    }
    class Sort
    {
        public static void InsertionSort(int[] a1)
        {
            int j, x;
            for(int i=1;i<a1.Length;i++)
            {
                x = a1[i];
                j = i - 1;
                while((j>=0) && (a1[j]>x))
                {
                    a1[j + 1] = a1[j];
                    j--;
                }
                a1[j + 1] = x;
            }
        }
    }
}
```

```
Unsorted Array
4  1  9  -13  90  56  81  34  -2  -15  60  88
Sorted Array
-15  -13  -2  1  4  9  34  56  60  81  88  90
```