Insertion Sort

- Insertion sort is a simple sorting algorithm that builds the final sorted array one element at a time.
- It works by taking elements from the input one at a time and inserting them into their correct position in the already sorted part of the array.
- The algorithm maintains a sorted and an unsorted portion of the array. Initially, the first element is considered sorted.
- It iterates through the unsorted portion, picking one element at a time, and inserts it into the correct position in the sorted portion.
- The key idea behind insertion sort is to shift larger elements to the right to create space for inserting the current element in its correct position.
- The algorithm repeats this process until all elements are placed in their correct positions, resulting in a fully sorted array.

Steps for Insertion Sort:

1. Start with the second element (index 1) and consider it as the "key."
2. Compare the key with the elements before it in the sorted portion.
3. If the key is smaller than the compared element, shift the compared element to the right.
4. Repeat the comparison and shifting until you find the correct position for the key.
5. Insert the key into its correct position in the sorted portion.
6. Move on to the next element (index 2) and repeat the process until the last element.

Time Complexity:

- **Best Case: O(n)**
- **Average Case & Worst Case: O(n^2)**
- **It is not suitable for large datasets, as its time complexity makes it inefficient compared to more advanced sorting algorithms like merge sort or quicksort.**

Space Complexity:

- **Space Complexity: In insertion sort, the sorting is done in-place, meaning that the algorithm does not require any additional space proportional to the input size. It operates directly on the given array, rearranging the elements within the array itself. Thus the Space Complexity is O(1)**

Advantages:

- Simple implementation and easy to understand.
- Efficient for small datasets or partially sorted arrays.

Disadvantages:

- Inefficient for large datasets due to its quadratic time complexity.
- Not stable when it comes to sorting elements with equal values (i.e., the relative order of equal elements may not be preserved).

Note:

- Insertion sort works like arranging a deck of cards, where you pick cards one by one and insert them in their correct order into your hand.
- It's best suited for small datasets or nearly sorted data.
- Be cautious when using it for large datasets, as other sorting algorithms may be more efficient in such cases.

Here's a step-by-step visualization of the insertion sort algorithm using the example array [5, 2, 8, 12, 3]:

Step 1: Array: [5, 2, 8, 12, 3]

Step 2: Consider the second element (2) as the key. Compare it with the element before it (5). Since 2 is smaller than 5, we shift 5 to the right and insert 2 into its correct position.

Array: [2, 5, 8, 12, 3]

Step 3: Consider the third element (8) as the key. Compare it with the elements before it (5 and 2). Since 8 is greater than both, it remains in its position.

Array: [2, 5, 8, 12, 3]

Step 4: Consider the fourth element (12) as the key. Compare it with the elements before it (8, 5, and 2). Since 12 is greater than all, it remains in its position.

Array: [2, 5, 8, 12, 3]

Step 5: Consider the fifth element (3) as the key. Compare it with the elements before it (12, 8, 5, and 2). Since 3 is smaller than all, we shift all larger elements to the right and insert 3 into its correct position.

Array: [2, 3, 5, 8, 12]

After completing all the steps, the array is sorted in ascending order using the insertion sort algorithm: [2, 3, 5, 8, 12].

Approach 1: Function to perform iterative insertion sort

The **insertionSort** function implements the iterative approach of the insertion sort algorithm.

1. It starts with a **for** loop that iterates through the array **arr**, starting from the second element (index 1) since the first element is considered already sorted.
2. Inside the loop, it assigns the current element (**arr[i]**) to a temporary variable **temp**.
3. It initializes a variable **j** with the value of **i - 1**, which represents the index of the element just before the current element.
4. Then, it enters a **while** loop that checks two conditions: if **j** is greater than or equal to 0 (to ensure we don't go out of bounds) and if the element at **arr[j]** is greater than the **temp** value.
5. If both conditions are true, it shifts the element at **arr[j]** one position to the right (**arr[j+1] = arr[j]**) and decreases **j** by 1.
6. The **while** loop continues until either one of the conditions becomes false.
7. After exiting the **while** loop, it inserts the **temp** value into its correct position by assigning it to **arr[j+1]**.
8. The **for** loop continues to the next element, repeating the process until all elements are sorted.
9. Finally, the sorted array is printed using the **print** function.

Approach 2: Function to perform recursive insertion sort

The **insertionSortRecursive** function implements the recursive approach of the insertion sort algorithm.

1. The function takes an additional parameter **index** that represents the current index of the element being processed.
2. It starts with a base case: if **index** is equal to the size of the array **arr**, it means all elements have been processed, so the function simply returns.
3. Inside the function, it performs the same steps as the iterative approach to insert the current element (**arr[index]**) into its correct position.
4. It initializes a variable **temp** with the value of **arr[index]** and a variable **j** with the value of **index - 1**.
5. Then, it enters a **while** loop similar to the iterative approach, shifting elements and finding the correct position for the **temp** value.

6. After exiting the **while** loop, it inserts the **temp** value into its correct position.
7. The function then makes a recursive call to itself, passing **arr** and **index+1** as arguments to process the next element.
8. The recursive calls continue until the base case is reached.
9. Finally, the sorted array is printed using the **print** function.