

## Assignment - 01:

You are given an array of integers, and you are required to sort this array using one of the following sorting algorithms: Bubble Sort, Selection Sort, or Insertion Sort. Your task is to implement the chosen sorting algorithm and analyze its time complexity.

1. Implement one of the sorting algorithms mentioned above (Bubble Sort, Selection Sort, or Insertion Sort) in Python.

Implementation of Insertion Sort algorithm in Python:

```
def insertion_sort(arr):  
    for i in range(1, len(arr)):  
        key = arr[i]  
        j = i - 1  
        while j >= 0 and key < arr[j]:  
            arr[j + 1] = arr[j]  
            j -= 1  
        arr[j + 1] = key  
    return arr
```

This function will sort an input list arr using the Insertion Sort algorithm.

Now that we've implemented the sorting algorithm, please let me know if you'd like to proceed with the next steps of your assignment or if you have any specific questions about the implementation.

2. Apply your sorting algorithm to the given array of integers ([5,2,9,1,5,6]) .

Certainly! Let's apply the Insertion Sort algorithm to the given array [5, 2, 9, 1, 5, 6] step by step:

\*Step 1:\* The initial array is [5, 2, 9, 1, 5, 6].

**\*Step 2:\*** Start with the second element (index 1) which is 2. We'll compare it to the elements to its left and insert it into the correct position within the sorted part of the array.

- Compare 2 with 5. Since 2 is smaller, we shift 5 to the right.

- Array after this step: [2, 5, 9, 1, 5, 6]

**\*Step 3:\*** Move to the next element, which is 9. Compare it to the elements to its left and insert it into the correct position.

- Compare 9 with 5. 9 is larger, so we don't shift.

- Compare 9 with 2. 9 is larger, so we don't shift.

- Insert 9 into the correct position.

- Array after this step: [2, 5, 9, 1, 5, 6]

**\*Step 4:\*** Continue this process for the remaining elements.

- For 1, we compare it to 9, 5, and 2, and insert it into the correct position.

- Array after this step: [1, 2, 5, 9, 5, 6]

- For the first 5, we compare it to 9 and insert it into the correct position.

- Array after this step: [1, 2, 5, 5, 9, 6]

- Finally, for 6, we compare it to 9 and insert it into the correct position.

- Array after this step: [1, 2, 5, 5, 6, 9]

**\*Step 5:\*** The entire array is now sorted in ascending order.

So, the sorted array using the Insertion Sort algorithm is [1, 2, 5, 5, 6, 9].

3. Provide the sorted array as the output.

The sorted array using the Insertion Sort algorithm for the input [5, 2, 9, 1, 5, 6] is:

[1, 2, 5, 5, 6, 9]

This is the sorted array in ascending order.

4. Analyze the time complexity of the sorting algorithm you implemented. Explain whether it is a stable sort and how it performs on different types of input data (e.g., already sorted, reverse sorted, random data).

Let's analyze the time complexity, stability, and performance of the Insertion Sort algorithm in detail.

#### **\*Time Complexity:\***

The time complexity of the Insertion Sort algorithm is as follows:

- **\*Best Case:\***  $O(n)$  comparisons and  $O(1)$  swaps. This occurs when the input array is already sorted. In this case, the algorithm simply checks each element and doesn't need to move them around.

- **\*Worst Case:\***  $O(n^2)$  comparisons and  $O(n^2)$  swaps. This happens when the input array is sorted in reverse order, and the algorithm needs to perform the maximum number of comparisons and swaps.

- **\*Average Case:\***  $O(n^2)$  comparisons and  $O(n^2)$  swaps. On average, Insertion Sort has quadratic time complexity, making it less efficient for large datasets compared to more advanced sorting algorithms like Quick Sort and Merge Sort.

#### **\*Stability:\***

Insertion Sort is a stable sorting algorithm. This means that it maintains the relative order of equal elements. If you have elements with the same value in the input, their order in the sorted array will be the same as in the original array.

#### **\*Performance on Different Input Data:\***

1. **\*Already Sorted Input:\*** Insertion Sort performs well when the input is already sorted (best-case scenario) because it requires only  $O(n)$  comparisons and no swaps. This makes it a good choice for data that is partially sorted or nearly sorted.

2. **\*Reverse Sorted Input:** Insertion Sort performs poorly on reverse sorted data (worst-case scenario) because it needs to perform the maximum number of comparisons and swaps. In this case, it becomes one of the least efficient sorting algorithms.

3. **\*Random Input:** For randomly ordered data, Insertion Sort has an average-case time complexity of  $O(n^2)$ , which makes it less efficient compared to more advanced sorting algorithms like Quick Sort or Merge Sort. It's suitable for small to moderately sized datasets but becomes inefficient for larger datasets.

In a holistic view, Insertion Sort is a stable sorting algorithm that performs well on nearly sorted data but is not efficient for large or reverse sorted datasets. It is generally used when simplicity and stability are more important than sorting speed. For large datasets or when performance is critical, other sorting algorithms like Quick Sort or Merge Sort are preferred.

5. Compare the time complexity of your chosen sorting algorithm with at least one other sorting algorithm (e.g., Quick Sort, Merge Sort, or Python's built-in sorted function). Explain the differences and scenarios where one algorithm might be preferred over the other.

Let's compare the time complexity of Insertion Sort with Quick Sort, which is a more efficient sorting algorithm, and Python's built-in sorted function. We'll also discuss scenarios where one algorithm might be preferred over the other.

1. **\*Insertion Sort vs. Quick Sort:**

- **\*Insertion Sort:** As previously discussed, Insertion Sort has an average-case time complexity of  $O(n^2)$ . It performs well on small datasets and nearly sorted data but becomes inefficient for larger datasets.

- **\*Quick Sort:** Quick Sort has an average-case time complexity of  $O(n \log n)$ . It is significantly faster than Insertion Sort for larger datasets. Quick Sort is often the preferred choice for sorting large datasets efficiently. However, it's not a stable sort, which means it may change the relative order of equal elements.

**\*Scenarios:**

- Use Insertion Sort when simplicity and stability are more important than sorting speed, and the dataset is small or partially sorted.

- Use Quick Sort when sorting speed is critical, and the dataset is large or not partially sorted. Quick Sort is one of the fastest sorting algorithms for general use.

## 2. **\*\*Insertion Sort vs. Python's sorted Function:\*\***

Insertion Sort: As discussed, Insertion Sort has an average-case time complexity of  $O(n^2)$  and is typically used for small datasets or partially sorted data. You have control over its implementation.

Python's sorted Function: Python's built-in sorted function uses an adaptive sorting algorithm called Timsort, which combines Merge Sort and Insertion Sort. It has an average-case time complexity of  $O(n \log n)$  and is highly optimized for various scenarios. It is a stable sort, preserving the relative order of equal elements.

Scenarios:

Use Insertion Sort when you need to implement your sorting logic, and simplicity or customizability is essential.

-Use Python's sorted function for most general sorting tasks in Python, as it is highly optimized and efficient. It's also a stable sort, which is advantageous when maintaining the relative order of equal elements is crucial.

In summary, the choice of sorting algorithm depends on the specific requirements of your task:

- For simplicity, small datasets, or nearly sorted data, Insertion Sort can be suitable.
- For general-purpose sorting in Python with good performance and stability, the built-in sorted function is often the best choice.
- For large datasets where sorting speed is critical, Quick Sort is a more efficient option. However, Quick Sort is not stable, so consider this factor when deciding between Quick Sort and sorted.