**4. Test Cases:**

* Test your optimized function with the following lists:
  1. **Input:** [64, 25, 12, 22, 11]
     + **Expected Output:** [11, 12, 22, 25, 64]
  2. **Input:** [29, 10, 14, 37, 13]
     + **Expected Output:** [10, 13, 14, 29, 37]
  3. **Input:** [3, 5, 2, 1, 4]
     + **Expected Output:** [1, 2, 3, 4, 5]
  4. **Input:** [1, 2, 3, 4, 5] (Already sorted list)
     + **Expected Output:** [1, 2, 3, 4, 5]
  5. **Input:** [5, 4, 3, 2, 1] (Reverse sorted list)
     + **Expected Output:** [1, 2, 3, 4, 5]

Write code for Insertion Sort that manages arrays with duplicate elements during the sorting process. Ensure the algorithm's behavior when encountering duplicate values, including whether it preserves the relative order of duplicates and how it affects the overall sorting outcome.

**Examples:**

**1. Array with Duplicates**:

* + **Input**: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
  + **Output**: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]

1. **All Identical Elements**:
   * **Input**: [5, 5, 5, 5, 5]
   * **Output**: [5, 5, 5, 5, 5]
2. **Mixed Duplicates**:
   * **Input**: [2, 3, 1, 3, 2, 1, 1, 3]
   * **Output**: [1, 1, 1, 2, 2, 3, 3, 3]

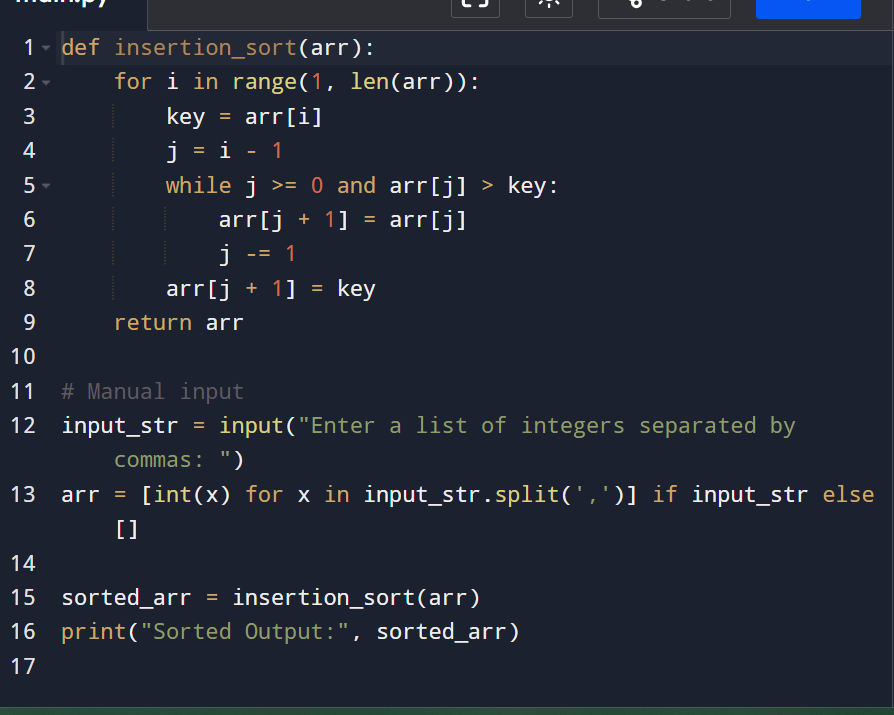
**Aim:**

To implement Insertion Sort that correctly sorts arrays containing duplicate elements, preserves their relative order (stable sort), and works correctly across various input patterns.

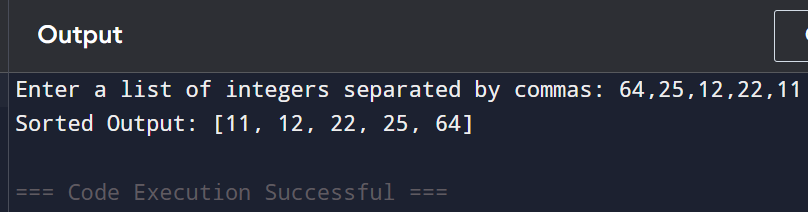
**Algorithm:**

1. Start from index 1 (second element) of the array.
2. Store the current element (key).
3. Compare it with previous elements.
4. Shift elements greater than key to the right.
5. Insert key at its correct position.
6. Repeat until the array is sorted.

**Code:**

****

**Input and output:**

****

**Result: the program is executed successfully and output is verified**

**Performance analysis:**

**Time Complexity:**

| **Case** | **Time Complexity** |
| --- | --- |
| **Best (sorted)** | **O(n)** |
| **Average** | **O(n²)** |
| **Worst (reverse)** | **O(n²)** |

**Space Complexity:**

* **O(1) — In-place sorting**