**Insertion Sort: Explanation, Algorithm, and Complexity**

Let’s go over how **insertion sort** works, step-by-step, and then dive into its **time complexity** and **space complexity**.

**1. What is Insertion Sort?**

Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. It divides the list into two parts: the **sorted part** and the **unsorted part**. Initially, the sorted part contains just the first element, and the algorithm repeatedly takes an element from the unsorted part and inserts it into the correct position in the sorted part.

**2. How Insertion Sort Works (Algorithm):**

* **Step 1:** Start by considering the first element as the already sorted part of the list.
* **Step 2:** Take the next element from the unsorted part (the second element) and store it in a temporary variable.
* **Step 3:** Compare this element to the elements in the sorted part, moving the larger elements in the sorted part one position to the right to make space for this element.
* **Step 4:** Insert the stored element into the correct position in the sorted part.
* **Step 5:** Repeat this process for all remaining elements in the unsorted part, growing the sorted part of the list until the entire list is sorted.

**3. Insertion Sort Python Code:**

python

Copy code

def insertion\_sort(my\_list):

for i in range(1, len(my\_list)): # Start with the second element (index 1)

curr = my\_list[i] # Store the current element in a temporary variable

j = i

# Compare with the sorted part of the list

while j > 0 and my\_list[j - 1] > curr: # Shift elements to the right to make space for curr

my\_list[j] = my\_list[j - 1]

j -= 1

# Insert the current element into the correct position

my\_list[j] = curr

**Code Explanation:**

* for i in range(1, len(my\_list)): Start from the second element (index 1) because we assume the first element is already sorted.
* curr = my\_list[i]: Store the element from the unsorted part in a variable (curr).
* The while loop: Shifts elements in the sorted part to the right until the correct position for curr is found.
* my\_list[j] = curr: Insert the stored element into the correct position.

**4. Time Complexity of Insertion Sort:**

**Best Case Time Complexity:**

* In the **best case**, the list is already sorted. The while loop never needs to shift elements because each new element is already in its correct position.
* For each element, the algorithm just compares it once, so the best-case time complexity is **O(n)**.

**Worst Case Time Complexity:**

* In the **worst case**, the list is sorted in reverse order. This means that every new element needs to be compared with all the elements in the sorted part of the list, and all elements need to be shifted.
* The worst-case time complexity is **O(n²)** because each element has to be compared with every other element in the worst case.

**Average Case Time Complexity:**

* On average, the elements are partially sorted, and the algorithm still needs to shift elements for insertion, but not as much as in the worst case.
* The average-case time complexity is also **O(n²)**, though it’s typically faster than selection sort and bubble sort in practice because insertion sort only makes necessary swaps.

**Time Complexity Summary:**

* **Best Case**: **O(n)** (the list is already sorted)
* **Worst Case**: **O(n²)** (the list is sorted in reverse order)
* **Average Case**: **O(n²)** (typical case)

**5. Space Complexity of Insertion Sort:**

Insertion sort is an **in-place algorithm**, meaning it does not need extra space for another list, and it only uses a few extra variables (curr, i, j).

* **Space complexity** is **O(1)**, which means it uses a constant amount of additional memory, regardless of the input size.

**6. Insertion Sort Example:**

Let’s walk through an example of how **insertion sort** works step-by-step on a small list:

**Input List:**

[5, 2, 9, 1, 6]

1. **First Element (5)**: Already sorted (since it’s the first element).
2. **Second Element (2)**: Compare with 5. 2 is smaller than 5, so we swap them.
   * List after step: [2, 5, 9, 1, 6]
3. **Third Element (9)**: Compare with 5. 9 is larger, so no swaps are needed.
   * List after step: [2, 5, 9, 1, 6]
4. **Fourth Element (1)**: Compare with 9, 5, and 2. Since 1 is the smallest, we shift all of them and place 1 in the first position.
   * List after step: [1, 2, 5, 9, 6]
5. **Fifth Element (6)**: Compare with 9. 6 is smaller than 9, so we swap 6 and 9.
   * Final sorted list: [1, 2, 5, 6, 9]

**7. How to Explain Insertion Sort’s Complexity in a Quiz:**

If you're asked to explain the **complexity** of insertion sort in a quiz, you can structure your answer like this:

**Question:** Explain the time and space complexity of insertion sort.

**Answer:**

Insertion sort is a simple, comparison-based sorting algorithm that divides the list into two parts: the sorted part and the unsorted part. It repeatedly takes an element from the unsorted part and inserts it into the correct position in the sorted part.

1. **Time Complexity**:
   * **Best Case**: **O(n)** – When the list is already sorted, insertion sort only compares each element once with the previous element.
   * **Worst Case**: **O(n²)** – In the worst case, when the list is sorted in reverse order, each new element has to be compared with all previous elements, requiring many shifts and swaps.
   * **Average Case**: **O(n²)** – On average, insertion sort still requires shifts and comparisons, but fewer than in the worst case.

Overall, insertion sort has **O(n²)** time complexity in the average and worst cases, but it is more efficient than other quadratic sorting algorithms when the list is almost sorted.

1. **Space Complexity**:
   * Insertion sort is an **in-place** algorithm, meaning it only uses a constant amount of extra space (O(1)), regardless of the input size.

**Conclusion**: Insertion sort is efficient for small or nearly sorted lists, but it is not suitable for large datasets due to its **O(n²)** time complexity. It is, however, faster than selection and bubble sort in practice for small or semi-sorted inputs.