LeetCode 147. Insertion Sort List

1. Problem Title & Link

- **147. Insertion Sort List**
- https://leetcode.com/problems/insertion-sort-list/

2. Problem Statement (Short Summary)

We are given the **head of a singly linked list**. We need to sort the list in ascending order using the **Insertion Sort algorithm** and return the sorted list.

3. Examples (Input → Output)

Input: head = [4,2,1,3]
Output: [1,2,3,4]

Input: head = [-1,5,3,4,0]
Output: [-1,0,3,4,5]

4. Constraints

- The number of nodes in the list is in the range [1, 5000].
- -5000 <= Node.val <= 5000

5. Thought Process (Step by Step)

- Insertion Sort works well for linked lists because **insertion can be done in O(1)** if we have the right spot.
- For each node in the list:
    - Remove it from current position.
    - Insert it into the correct position in the already-sorted part.
- Maintain a **dummy head** to simplify insertion logic.

6. Pseudocode (Language-Independent)

```
function insertionSortList(head):
    dummy = new Node(-∞)   # helps with insertion
    curr = head

    while curr is not null:
        prev = dummy
```



```
        # find insertion spot
        while prev.next != null and prev.next.val < curr.val:
            prev = prev.next

        next_node = curr.next
        # insert curr between prev and prev.next
        curr.next = prev.next
        prev.next = curr
        curr = next_node

    return dummy.next
```

7. Code Implementation

✅ **Python**

```
class Solution:
     def insertionSortList(self, head: Optional[ListNode]) ->
Optional[ListNode]:
        dummy = ListNode(0)   # dummy node
        curr = head

        while curr:
            prev = dummy
            # find position to insert
            while prev.next and prev.next.val < curr.val:
                prev = prev.next

            next_node = curr.next
            curr.next = prev.next
            prev.next = curr
            curr = next_node

        return dummy.next
```

✅ **Java**

```
class Solution {
    public ListNode insertionSortList(ListNode head) {
        ListNode dummy = new ListNode(0);
        ListNode curr = head;

        while (curr != null) {
            ListNode prev = dummy;
            // find correct spot
                 while (prev.next != null && prev.next.val <
curr.val) {
                prev = prev.next;
            }

            ListNode nextNode = curr.next;
            curr.next = prev.next;
            prev.next = curr;
            curr = nextNode;
        }
        return dummy.next;
    }
}
```

8. Time & Space Complexity Analysis

- **Time Complexity:** $O(n^2)$ in worst case (for every node we may scan the sorted part).
- **Space Complexity:** O(1) (in-place, only uses dummy node).

9. Common Mistakes / Edge Cases

- Forgetting to update curr with next_node.
- Infinite loop if links are not updated properly.
- Assuming list is not empty → must handle single-node case.

10. Variations / Follow-Ups

- Implement **Selection Sort** on a linked list.
- Convert the list into array → sort → rebuild list (not efficient).
- Sort **doubly linked list** with insertion sort (simpler because you can go backwards).

11. Dry Run (Step by Step Execution)

👉 Input: [4, 2, 1, 3]

- Initialize dummy → ∅

Step 1: curr = 4

- Sorted list: [4]

Step 2: curr = 2

- Compare with 4 → insert before
- Sorted list: [2, 4]

Step 3: curr = 1

- Compare with 2 → insert before
- Sorted list: [1, 2, 4]

Step 4: curr = 3

- Compare with 1, 2, 4 → insert between 2 and 4
- Sorted list: [1, 2, 3, 4]

✅ Final Output: [1, 2, 3, 4]