2046. Sort Linked List Already Sorted Using Absolute Values

Medium | Linked List | Two Pointers | Sorting | Leetcode Link

Problem Description

The given problem asks us to take the `head` of a singly-linked list, where the list is sorted in non-decreasing order based on the **absolute values** of its nodes. The goal is to re-sort this list, but this time according to the **actual values** of its nodes, still maintaining a non-decreasing order. This challenge essentially involves sorting negative numbers (where applicable) before the non-negative ones while adhering to the non-decreasing sequence.

Intuition

The intuition behind the solution is to leverage the fact that the list is already sorted by absolute values. This characteristic implies that all negative values, if any, will be at the beginning of the list - but they will be in decreasing order because their absolute values are sorted in non-decreasing order. Conversely, the non-negative values will follow in non-decreasing order.

Therefore, the solution approach involves iterating through the list and moving any encountered negative values to the front. Since we know that the list is effectively partitioned into a non-increasing sequence of negative values followed by a non-decreasing sequence of non-negative values, moving each negative value we encounter to the front of the list will naturally sort it.

By iterating just once through the linked list, whenever a negative value is found, it's plucked out of its position and inserted before the current `head`. This insert-at-head operation ensures that the ultimately larger (less negative) values encountered later are placed closer to the head, thereby achieving a non-decreasing order of actual values.

The key points of the approach are:

- No need for a complex sorting algorithm since the list is already sorted by absolute values.
- Negative values must precede non-negative ones for actual value sorting.
- Iterating just once throughout the list is sufficient.
- A simple insert-at-head operation for negative values during iteration results in the desired sorted order.

The solution preserves the initial list's order in terms of absolute values while reordering only where necessary to get the actual values sorted, hence achieving the re-sorting in an efficient manner.

Solution Approach

The provided solution uses a simple while loop and direct manipulation of the linked list nodes to achieve the task without additional data structures.

The algorithm follows these steps:

1. Initialize two pointers `prev` and `curr`. `prev` starts at the `head` of the list, and `curr` starts at the second element (`head.next`), since there is no need to move the head itself.
2. Iterate through the linked list with a `while` loop, which continues until `curr` is `None`, meaning we've reached the end of the list.
3. For each node, we check if `curr.val` is negative.
   - If it is negative, we conduct the following re-linking process: a. Detach the `curr` node by setting `prev.next` to `curr.next`. This removes the current node from its position in the list. b. Move the `curr` node to the front by setting `curr.next` to `head`. This places the current node at the beginning of the list. c. Update `head` to be `curr` since `curr` is now the new head of the list. d. Move the `curr` pointer to the next node in the list by setting it to `t` (the node following the moved node).
   - If `curr.val` is not negative, we simply move forward in the list by setting `prev` to `curr` and `curr` to `curr.next`, as no changes are needed for non-negative or zero values.
4. Repeat this process until the `curr` pointer reaches the end of the list.

The algorithm uses the two-pointer technique to traverse and manipulate the linked list in place without requiring extra space for sorting. The key insight here is that the insertion of negative nodes at the head can be done in constant time, which makes the solution very efficient. There are no complicated data structures or algorithms needed; the solution makes direct alterations to the list's nodes, which takes advantage of the properties of linked lists for efficient insertion operations.

As for complexity:

- The time complexity is O(n), where `n` is the number of nodes in the list, since every node is visited once.
- The space complexity is O(1), as no additional space proportional to the input size is used.

This approach is efficient and takes advantage of the already partially sorted nature of the list based on absolute values to achieve the fully sorted list by actual values with minimal operations.

Example Walkthrough

Let's consider the following singly-linked list sorted by absolute values:

`2 -> -3 -> -4 -> 5 -> 6`

According to the problem statement, our goal is to reorder this list based on the actual values, preserving non-decreasing order.

Here's a step-by-step walkthrough using the solution approach provided:

1. We initialize `prev` as the `head` (node with value 2) and `curr` as the second node (node with value -3).
2. We start the `while` loop and iterate over the list. In our example, the first `curr` is negative.
3. Since the value of `curr` (-3) is negative, we insert it before the head: a. Detach `curr`: we remove -3 from the list. The list becomes: `2 -> -4 -> 5 -> 6` b. Move `curr` to the front: we insert -3 before the current head. The list becomes: `-3 -> 2 -> -4 -> 5 -> 6` c. Update `head` to `curr`: -3 is the new head of the list. d. Move `curr` to next node: `curr` now points to -4.
4. Repeat the process for the next `curr`, which again has a negative value (-4): a. Detach `curr`: we remove -4 from the list. The list becomes: `-3 -> 2 -> 5 -> 6` b. Move `curr` to the front: we insert -4 before the head. The list becomes: `-4 -> -3 -> 2 -> 5 -> 6` c. Update `head` to `curr`: -4 is the new head of the list. d. Move `curr` to next node: `curr` now points to 5.
5. Now `curr` points to a positive value (5), so we just move the `prev` to `curr` and `curr` to its next node, which is 6.
6. As `curr` points to another positive value (6), we again just advance `prev` and `curr` accordingly.
7. We've reached the end of the list. The list is now correctly sorted by actual values in non-decreasing order: `-4 -> -3 -> 2 -> 5 -> 6`

While iterating through the list, we only moved the negative values, which were already sorted in non-increasing order, to the front. We didn't need to touch the positive values because they were already in non-decreasing order. This process respects the previously sorted absolute values while ordering the elements by their actual values.

Python Solution

```python
# Definition for a singly-linked list node.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val  # Value of the node
        self.next = next  # Reference to the next node in the list

class Solution:
    def sortLinkedList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # Initialize pointers.
        # `previous` will track the node before `current`.
        previous = None
        current = head

        # Traverse the linked list starting from the head until there are no more nodes.
        while current:
            # Check if the current node's value is negative.
            if current.val < 0:
                # If this is the first node (head of the list), no need to move it.
                if previous is None:
                    current = current.next
                    continue

                # If current node is negative, shift it to the beginning of the list.
                previous.next = current.next  # Link the previous node to the next node, effectively removing `current` from its curr
                current.next = head  # Make `current` node point to the current head
                head = current  # Update `head` to be the `current` node

                # Move on to the next node
                current = previous.next
            else:
                # If current node is not negative, just move `previous` and `current` pointers one step forward.
                previous = current
                current = current.next

        return head  # Return the modified list with all negative values moved to the beginning
```

Java Solution

```java
/**
 * Definition for singly-linked list.
 */
class ListNode {
  int val;
  ListNode next;

  ListNode() {}

  ListNode(int val) { this.val = val; }

  ListNode(int val, ListNode next) { this.val = val; this.next = next; }
}

class Solution {
  /**
   * This method sorts a linked list such that all nodes with negative values
   * are moved to the front.
   *
   * @param head The head of the singly-linked list.
   * @return The head of the sorted singly-linked list.
   */
  public ListNode sortLinkedList(ListNode head) {
    // Previous and current pointers for traversal
    ListNode previous = head;
    ListNode current = head.next;

    // Iterate through the list
    while (current != null) {
      // When a negative value is encountered
      if (current.val < 0) {
        // Detach the current node with the negative value
        ListNode temp = current.next;
        previous.next = temp;

        // Move the current node to the front of the list
        current.next = head;
        head = current;

        // Continue traversal from the next node
        current = temp;
      } else {
        // If the current value is not negative, move both pointers forward
        previous = current;
        current = current.next;
      }
    }

    // Return the updated list with all negative values at the front
    return head;
  }
}
```

C++ Solution

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* sortLinkedList(ListNode* head) {
        // Initialize pointers for traversal
        ListNode* previous = head;          // Pointer to track the node just before the current one
        ListNode* current = head->next;     // Pointer to track the current node during traversal

        // Iterate through the list
        while (current != nullptr) {
            // When the current node has a negative value, reposition it to the start of the list
            if (current->val < 0) {
                ListNode* temp = current->next; // Hold the next node to continue traversal later
                previous->next = temp;          // Remove the current node from its original position
                current->next = head;           // Insert the current node at the beginning of the list
                head = current;                 // Update the head to the new first node
                current = temp;                 // Continue traversal from the next node
            } else {
                // If the current node is non-negative, move both pointers forward
                previous = current;
                current = current->next;
            }
        }

        return head; // Return the sorted list
    }
};
```

Typescript Solution

```typescript
/**
 * Definition for singly-linked list.
 */
interface ListNode {
  val: number;
  next: ListNode | null;
}

/**
 * Sorts a singly linked list such that all negative values are moved to the front.
 * @param {ListNode | null} head - The head of the linked list to be sorted.
 * @returns {ListNode | null} The head of the sorted linked list.
 */
function sortLinkedList(head: ListNode | null): ListNode | null {
  // There is nothing to sort if the list is empty or has only one node.
  if (!head || !head.next) {
    return head;
  }

  // Initialize pointers for traversal.
  let previous: ListNode | null = head; // Pointer to track the node just before the current one.
  let current: ListNode | null = head.next; // Pointer to track the current node during traversal.

  // Iterate through the list.
  while (current !== null) {
    // When the current node has a negative value, reposition it to the start of the list.
    if (current.val < 0) {
      let temp: ListNode | null = current.next; // Hold the next node to continue traversal later.
      previous.next = temp; // Remove the current node from its original position.
      current.next = head; // Insert the current node at the beginning of the list.
      head = current; // Update the head to the new first node.
      current = temp; // Continue traversal from the next node.
    } else {
      // If the current node is non-negative, move both pointers forward.
      previous = current;
      current = current.next;
    }
  }

  // Return the head of the sorted list.
  return head;
}
```

Time and Space Complexity

Time Complexity

The given code is intended to sort a singly-linked list by moving all negative value nodes to the beginning of the list in one pass. It iterates through the list only once, with a fixed set of actions for each node. The main operations within the loop involve pointer adjustments which take constant time. Hence, no matter the input size of the singly-linked list, the algorithm guarantees that each element is checked exactly once.

The time complexity of the code is `O(n)`, where `n` is the number of nodes in the linked list.

Space Complexity

The space complexity of the code is related to the extra space used by the algorithm, excluding the space for the input.

The provided code uses a constant amount of space: variables `prev`, `curr`, and `t`. No additional data structures are used that grow with the input size. All changes are made in-place by adjusting the existing nodes' `next` pointers.

Thus, the space complexity of the code is `O(1)` since it uses constant extra space.