Documentation for the problem of Sort List

Problem Description

Given the head of a linked list, the task is to return the list after sorting it in ascending order.

Example 1:

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

Example 2:

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

Example 3:

- **Input:** head = []
- **Output:** []

Constraints

- The number of nodes in the list is in the range [0, 50,000].
- Node values are in the range [-100,000, 100,000].

Follow-up

Can you sort the linked list in O(n log n) time and O(1) memory (i.e., constant space)?

Solution Approach

The provided solution implements a sorting algorithm specifically designed for linked lists, known as Merge Sort. This algorithm efficiently sorts the linked list with a time complexity of O(n log n) and uses a constant amount of additional memory. Here's a step-by-step breakdown of how the solution works:

Steps

1. **Base Case Handling:**
   - If the list is empty or contains a single node, it is already sorted. Therefore, return the head of the list.

2. **Finding the Midpoint:**
   - Use the slow and fast pointer technique to find the midpoint of the list. The slow pointer advances one node at a time, while the fast pointer advances two nodes at a time. When the fast pointer reaches the end, the slow pointer will be at the midpoint.

3. **Splitting the List:**
   - Once the midpoint is found, split the list into two halves. The first half starts from the head of the list and ends at the node before the midpoint. The second half starts from the midpoint and continues to the end of the list.

4. **Recursive Sorting:**
   - Recursively apply the sorting algorithm to both halves of the list. This will eventually break the list down into sublists of size 1 or 0, which are inherently sorted.

5. **Merging Sorted Halves:**
   - After sorting both halves, merge them into a single sorted list. Use a dummy node to facilitate the merging process. Compare nodes from both halves and attach the smaller node to the sorted list. Continue until all nodes from both halves are processed.

6. **Returning the Result:**
   - Return the sorted list starting from the node next to the dummy node, which represents the head of the newly merged sorted list.

Key Points

- **Time Complexity:** O(n log n), where n is the number of nodes in the list. This is because the list is divided into halves recursively (log n levels) and each level involves merging all nodes (O(n) operations).

- **Space Complexity:** O(1) auxiliary space, as the sorting is done in-place without using additional data structures beyond a few pointers.

Advantages

- **Efficient Sorting:** Merge Sort is particularly well-suited for linked lists due to its ability to sort in O(n log n) time.

- **No Extra Space:** The in-place merging ensures that the algorithm uses constant space, aside from the recursive stack.

Use Cases

Useful for scenarios where sorting linked lists is required, such as in real-time data processing where linked lists are used for their efficient insertion and deletion capabilities.

This documentation covers the problem description, constraints, and the approach to solving it, including the key steps and considerations.