148 Sort List

Problem Statement

The problem presents us with a singly linked list and requires us to sort it in ascending order. A singly linked list is a data structure where each node contains a value and a reference (or a pointer) to the next node. The head of the linked list is the first node, and this is the only reference to the linked list that is provided. The challenge is to rearrange the nodes such that their values are in ascending order from the head to the end of the list.

Intuition

The given solution uses a divide and conquer strategy. Specifically, it uses the merge sort algorithm adapted for linked lists. Here's the intuition behind this approach

1.Divide phase: First, we split the linked list into two halves. This is done by using a fast and slow pointer approach to find the middle of the linked list. The slow pointer moves one step at a time, while the fast pointer moves two steps. When the fast pointer reaches the end of the list, the slow pointer will be at the middle. We then break the list into two parts from the middle.

2.Conquer phase: We recursively call the sortList function on these two halves. Each recursive call will further split the lists into smaller sublists until we are dealing with sublists that either have a single node or are empty.

3.Merge phase: After the sublists are sorted, we need to merge them back together. We use a helper pointer to attach nodes in the correct order (the smaller value first). This is similar to the merge operation in the traditional merge sort algorithm on arrays.

4.Base case: The recursion stops when we reach a sublist with either no node or just one node, as a list with a single node is already sorted.

By following these steps, the sortList function continues to split and merge until the entire list is sorted. The dummy node (dummy in the code) is used to simplify the merge phase, so we don't have to handle special cases when attaching the first node to the sorted part of the list. At the end, dummy.next will point to the head of our sorted list, which is what we return.

Time and Space Complexity

The given Python code is an implementation of the merge sort algorithm for sorting a linked list. Let's analyze the time complexity and space complexity of the code.

Time Complexity

The merge sort algorithm divides the linked list into halves recursively until each sublist has a single element. Then, it merges these sublists while sorting them. This divide-and-conquer approach leads to a time complexity of O(n log n), where n is the number of nodes in the linked list. This is because:

->The list is split into halves repeatedly, contributing to the log n factor (each divide step cuts the linked list size in half).

->In each level of the recursion, all n elements have to be looked at to merge the sublists, contributing to the n factor.

Therefore, combining both, we get a total time complexity of O(n log n).

Space Complexity

The space complexity of the code depends on the implementation of the merge sort algorithm. In this particular version, merge sort is not done in-place; new nodes are not created, but pointers are moved around.

However, due to the use of recursion, there is still a space complexity concern due to the recursive call stack. The maximum depth of the recursion will be O(log n), as each level of recursion splits the list in half. Hence, the space complexity is O(log n), which is derived from the depth of the recursion stack.

In summary:

-> Time Complexity: O(n log n)

->Space Complexity: O(log n)