Divide and Conquer Merge Sort Medium **Linked List** Two Pointers Sorting **Leetcode Link**

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

Problem Description

148. Sort List

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. Sorting linked lists is a bit tricky since, unlike arrays, we don't have direct access to the nodes based on index and we cannot use traditional indexing methods. Intuition

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

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

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

- 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.
- 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

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

- 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.
- **Solution Approach**

The implementation follows the intuition outlined previously and can be broken down as follows: 1. Base Case Handling: The function first checks whether the linked list is empty or contains only a single node by looking for head is None or head.next is None. If either condition holds true, it returns head as it stands for an already sorted list or no list at all.

2. Dividing the List: The list is split into two halves by employing two pointers, slow and fast. These pointers start at head with

fast one step ahead (pointing to head.next). They then loop through the list, with slow moving one node at a time (slow.next)

and fast moving two nodes at a time (fast.next.next). This continues until fast reaches the end of the list. At this point, slow

• The split is performed by setting t to slow.next (which is the start of the second half of the list) and then severing the list by

sortList.

1 4 -> 2 -> 1 -> 3

Example Walkthrough

Let's illustrate the solution approach using a small example. Imagine we have a linked list:

We want to sort this list in ascending order using the merge sort algorithm designed for linked lists.

list, slow ends up on node 2 and fast ends up past node 3 (signifying the end of the list).

points to the node just before the midpoint of the list.

4. Merging: Once the base cases return, the merge phase begins.

- setting slow.next to None. This leaves us with two separate lists: one starting from head and ending at slow, and the other starting from t. 3. Recursive Sorting: The function is recursively called on the two halves of the list, self.sortList(head) for the first half and self.sortList(t) for the second half. The recursive calls continue to split the sublists until they can no longer be split (i.e., the base case).
- A cur pointer references it and is used to keep track of the last node in the sorted list as we merge nodes from 11 and 12. o In a loop, we compare the values of the nodes at the heads of 11 and 12. We attach the smaller node to cur. next and advance the corresponding pointer (11 or 12) as well as cur. • This loop continues until either 11 or 12 is exhausted. Once that happens, the remaining nodes of the non-exhausted list are

appended to the end of the merged list because they are already sorted. This is done by the line cur.next = 11 or 12.

5. Returning the Sorted List: The head of the dummy node (dummy) does not contain any value and was just a placeholder to ease

the merge process. Therefore, dummy.next refers to the first node of the merged, sorted list, which is the output of the function

A dummy node is created with dummy = ListNode(), which serves as the starting node of the sorted list.

- In summary, this solution utilizes the merge sort algorithm adapted for linked lists and employs a recursive divide and conquer approach to sorting. It's efficient and effective for sorting linked lists as it doesn't rely on random access memory and works well with the sequential nature of a linked list.
- end. 3. Recursive Sorting: We now call sortList(head) which sorts the sublist 4 -> 2, and sortList(t) which sorts the sublist 1 -> 3. In the sublist 4 -> 2, we would further divide it into 4 and 2, and since these are single nodes, they serve as our base cases and

1. Base Case Handling: Check if the list is empty or has one node. Our list has multiple nodes, so we move to dividing the list.

2. Dividing the List: We create two pointers, slow and fast. They start off with fast being one node ahead. As we step through the

After the while loop, slow is on 2, so we split the list into head (pointing at 4) to slow (pointing at 2) and t (pointing at 1) to the

4. Merging: We have our sublists sorted: 4, 2, 1, and 3. We now need to merge them. This is done using the dummy node approach and comparing one by one.

Python Solution

class Solution:

16

17

18

19

20

21

22

23

24

25

26

28

29

30

31

32

33

34

35

36

37

48

49

50

51

52

53

54

56

57

58

59

60

61

62

64

65

66

68

74

75

76

77

78

79

80

81

82

83

84

85

87

86 }

67 }

class ListNode {

int val;

C++ Solution

* };

public:

11 class Solution {

*/

10

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

52

53

54

55

56

58

57 };

* struct ListNode {

int val;

ListNode *next;

ListNode next;

ListNode() {}

ListNode(int val) {

this.val = val;

this.val = val;

this.next = next;

* Definition for singly-linked list.

ListNode(): val(0), next(nullptr) {}

ListNode* sortList(ListNode* head) {

ListNode* fast = head->next;

while (fast && fast->next) {

fast = fast->next->next;

// Split the list into two halves

ListNode* midNext = slow->next;

// Recursively sort both halves

// Merge the two sorted halves

ListNode* current = dummyHead;

while (leftHalf && rightHalf) {

ListNode* leftHalf = sortList(head);

ListNode* dummyHead = new ListNode();

ListNode* rightHalf = sortList(midNext);

// Choose the smaller value from either half

// Move to the next node in the merged list

// The merged sorted list is pointed to by the dummy head's next node

if (leftHalf->val <= rightHalf->val) {

current->next = leftHalf;

leftHalf = leftHalf->next;

current->next = rightHalf;

ListNode* sortedHead = dummyHead->next;

delete dummyHead; // Clean up the dummy node

rightHalf = rightHalf->next;

slow = slow->next;

ListNode* slow = head;

slow->next = nullptr;

} else {

return sortedHead;

Typescript Solution

ListNode(int x) : val(x), next(nullptr) {}

if (!head || !head->next) return head;

ListNode(int x, ListNode *next) : val(x), next(next) {}

// Base case: if the list is empty or has only one element, it is already sorted.

// Use the fast and slow pointer approach to find the middle of the list

ListNode(int val, ListNode next) {

First, 2 and 4 are merged into 2 -> 4.

○ Then 1 and 3 are merged into 1 → 3.

def sortList(self, head: ListNode) -> ListNode:

Split the list into two halves.

Merge the two sorted halves.

while left_half and right_half:

current = current.next_node

if left_half.value <= right_half.value:</pre>

left_half = left_half.next_node

current.next_node = right_half

* Sorts a linked list using the merge sort algorithm.

* @param head The head node of the linked list.

while (left != null && right != null) {

if (left.val <= right.val) {</pre>

current.next = left;

current.next = right;

current.next = (left == null) ? right : left;

right = right.next;

current = current.next;

left = left.next;

} else {

return dummyHead.next;

* Definition for singly-linked list.

// Choose the node with the smaller value from either left or right,

// If any nodes remain in either list, append them to the end of the merged list.

// Return the head of the merged sorted list, which is the next node of the dummy node.

// and append it to the current node of the merged list.

* @return The sorted linked list.

right_half = right_half.next_node

Attach the remaining elements, if any, from either half.

current.next_node = left_half

temp = slow.next_node

slow.next_node = None

dummy_node = ListNode()

current = dummy_node

else:

are already sorted.

 Finally, we merge 2 -> 4 and 1 -> 3 into one sorted list. Our pointers would compare 2 and 1 and choose 1, moving one step. Compare 2 and 3 and choose 2, and so forth until the list is fully merged into 1 -> 2 -> 3 -> 4.

Similarly, the sublist 1 -> 3 is divided into 1 and 3. Again, these are single nodes and are sorted by definition.

- 5. Returning the Sorted List: The dummy. next will point to 1 which is the start of our sorted linked list, and that's what we return. Thus, we have successfully sorted the original linked list using the merge sort algorithm: 1 -> 2 -> 3 -> 4.
- 1 # Definition for singly-linked list. 2 class ListNode: def __init__(self, value=0, next_node=None): self.value = value self.next_node = next_node
- if head is None or head.next_node is None: 10 return head 11 12 # Find the middle of the list to split the list into two halves. 13 slow, fast = head, head.next_node 14 15 while fast and fast.next_node:

Compare the current elements of both halves and attach the smaller one to the result.

Base case: if the list is empty or has only one element it is already sorted.

slow, fast = slow.next_node, fast.next_node.next_node

left_half, right_half = self.sortList(head), self.sortList(temp)

current.next_node = left_half or right_half 38 39 # Return the head of the sorted list. return dummy_node.next_node 40 41

Java Solution

class Solution {

/**

```
9
       public ListNode sortList(ListNode head) {
           // Base cases: if the list is empty or has just one element, it is already sorted.
10
           if (head == null || head.next == null) {
11
12
                return head;
           // Find the midpoint of the list using the slow and fast pointer approach.
16
           ListNode slow = head;
17
           ListNode fast = head.next;
18
           while (fast != null && fast.next != null) {
                slow = slow.next; // moves one step at a time
19
20
                fast = fast.next.next; // moves two steps at a time
21
22
23
           // Split the list into two halves.
24
           ListNode mid = slow.next;
25
           slow.next = null;
26
27
           // Recursively sort each half.
28
           ListNode leftHalf = sortList(head);
29
           ListNode rightHalf = sortList(mid);
30
31
           // Merge the two halves and return the merged sorted list.
32
           return merge(leftHalf, rightHalf);
33
34
35
       /**
36
        * Merges two sorted linked lists into one sorted linked list.
37
38
        * @param left The head node of the first sorted linked list.
39
        * @param right The head node of the second sorted linked list.
        * @return The head node of the merged sorted linked list.
40
41
42
       private ListNode merge(ListNode left, ListNode right) {
43
           // Create a dummy node to serve as the starting point for the merged list.
           ListNode dummyHead = new ListNode();
44
45
           // Use a pointer to build the new sorted linked list.
46
           ListNode current = dummyHead;
47
```

current = current->next; 46 47 48 // If there are remaining nodes in either half, append them to the merged list 49 current->next = leftHalf ? leftHalf : rightHalf; 50 51

```
// TypeScript definition for a singly-linked list node
   class ListNode {
       val: number;
       next: ListNode | null;
       constructor(val?: number, next?: ListNode | null) {
            this.val = val === undefined ? 0 : val;
            this.next = next === undefined ? null : next;
 8
 9
10
11
   /**
    * Sorts a singly-linked list using merge sort algorithm.
    * @param {ListNode | null} head - The head of the singly-linked list to be sorted.
    * @returns {ListNode | null} - The head of the sorted singly-linked list.
16
    */
   function sortList(head: ListNode | null): ListNode | null {
       // Base case: if the list is empty or has only one node, it's already sorted
       if (head == null || head.next == null) {
20
            return head;
21
22
23
       // Use fast and slow pointers to find the middle of the linked list
       let slow: ListNode = head;
24
25
       let fast: ListNode = head.next;
26
       while (fast != null && fast.next != null) {
27
            slow = slow.next;
28
           fast = fast.next.next;
29
30
       // Use recursion to sort both halves of the list
31
32
       let mid: ListNode = slow.next;
33
       slow.next = null;
34
       let sortedList1: ListNode = sortList(head);
35
       let sortedList2: ListNode = sortList(mid);
36
37
       // Merge the two sorted halves
38
       let dummy: ListNode = new ListNode(); // Temporary dummy node to simplify merge process
39
        let current: ListNode = dummy; // Current node for merge process
40
       // Merge the lists by selecting the smallest of the two nodes at each step
41
       while (sortedList1 != null && sortedList2 != null) {
42
            if (sortedList1.val <= sortedList2.val) {</pre>
43
44
               current.next = sortedList1;
45
               sortedList1 = sortedList1.next;
46
           } else {
47
               current.next = sortedList2;
               sortedList2 = sortedList2.next;
48
49
50
           current = current.next;
51
52
53
       // Attach remaining nodes (if any) from the non-empty list
        current.next = sortedList1 == null ? sortedList2 : sortedList1;
54
55
56
        return dummy.next; // The head of the sorted list is next to the dummy node
57 }
58
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

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

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.

Space Complexity

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

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

Time Complexity: 0(n log n)

derived from the depth of the recursion stack.

Space Complexity: O(log n)

In summary:

and space complexity of the code.

nodes in the linked list. This is because:

Time Complexity