2130. Maximum Twin Sum of a Linked List

Medium Hints Linked List Two Pointers Leetcode Link

Problem Description

In this LeetCode problem, we are working with a linked list of even length `n`. The list has a special property: each node has a "twin," which is the node at the symmetric position from the other end of the list. For instance, in a list with `n` nodes, the `i`-th node is the twin of the `(n-1)`-th node, and so on. The concept of twins only applies to the first half of the nodes in the list.

The "twin sum" is the sum of the value of a node and its twin's value. Our goal is to calculate and return the maximum twin sum that can be found in the given linked list.

To clarify with an example, if we have a linked list `1 -> 2 -> 3 -> 4`, the twins are `(1, 4)` and `(2, 3)`, leading to twin sums of `5` and `5`. In this case, the maximum twin sum is `5`.

Intuition

The challenge in this problem is to efficiently find the twin of each node so that we can calculate the twin sums. Since the list is singly linked, we cannot directly access the corresponding twin for a node in the first half of the list without traversing it.

An intuitive approach involves reversing the second half of the linked list so that we can then pair nodes from the first half and the reversed second half to calculate their twin sums. By iterating through the two halves simultaneously, we can easily calculate each twin sum and determine the maximum twin sum.

Here is the step-by-step thought process behind the solution:

1. Find the middle of the linked list. Since we have a fast and a slow pointer, where the fast pointer moves at twice the speed of the slow pointer, when the fast pointer reaches the end of the list, the slow pointer will be at the middle.
2. Once the middle is found, reverse the second half of the list starting from the node right after the middle node.
3. Two pointers can now walk through both halves of the list at the same time. Since the second half is reversed, the corresponding twins will be the nodes that these two pointers point to.
4. As we iterate, we calculate the sum of the twins and keep track of the maximum sum that we find.
5. Once we've traversed both halves, the calculated maximum sum is returned as the result.

Solution Approach

The given solution in Python implements the approach we've discussed. The main components of the solution involve finding the midpoint of the linked list, reversing the second half of the list, and then iterating to find the maximum twin sum. Here is how each step is accomplished:

1. **Finding the Middle of the Linked List:** Use the classic two-pointer technique known as the "tortoise and hare" algorithm. One pointer (`slow`) moves one node at a time, while the other (`fast`) moves two nodes at a time. When `fast` reaches the end of the list, `slow` will be at the midpoint.

```
1 slow, fast = head, head.next
2 while fast and fast.next:
3     slow, fast = slow.next, fast.next.next
```

2. **Reversing the Second Half:** Once the midpoint is found, the solution defines a helper function `reverse(head)` to reverse the second half of the linked list starting from the node right after the middle (`slow.next`).

```
1  def reverse(head):
2      dummy = ListNode()
3      while head:
4          curr = head
5          next = curr.next
6          curr.next = dummy.next
7          dummy.next = curr
8          head = next
9      return dummy.next
```

   The reverse function makes use of a dummy node to easily reverse the linked list through pointer manipulation.

3. **Calculating Maximum Twin Sum:** After reversing the second half, we now have two pointers, `pa` that points at the head of the list and `pb` that points at the head of the reversed second half. We then iterate through the list using both pointers, which move in step with each other, calculating the sum of the values each points to and updating the `ans` variable if we find a larger sum.

```
1  pa = head
2  pb = slow.next
3  slow.next = None  # Break the list into two halves
4  pb = reverse(pb)
5
6  ans = 0
7  while pa and pb:
8      ans = max(ans, pa.val + pb.val)
9      pa = pa.next
10     pb = pb.next
```

4. **Returning the Result:** After the iteration, `ans` holds the maximum twin sum, which is then returned.

```
1 return ans
```

Overall, the solution iterates over the list twice: once to find the middle and once more to calculate the twin sum. This keeps the time complexity to O(n) where `n` is the number of nodes in the linked list. The space complexity is O(1) as it only uses a few extra pointers with no dependency on the size of the input list.

Example Walkthrough

To illustrate the solution approach, let's walk through a small example with the following linked list `[1 -> 4 -> 2 -> 3]`.

1. **Find the middle of the linked list:** We start with two pointers, `slow` and `fast`. `slow` moves one step at a time, while `fast` moves two steps. We initialize them as `slow = head` and `fast = head.next`. In our example:
   - `slow` starts at `1` and `fast` starts at `4`.
   - After the first step, `slow` is at `4`, and `fast` is at `2`.
   - After the second step, `slow` is at `2`, and `fast` has reached the end `None`. At this point, we stop the iteration and `slow` is at the midpoint of the list.
2. **Reverse the second half**. We reverse the list starting from the node right after the middle (`slow.next`), so we reverse the list from `3` onwards. The list is small, so after reversing, the second half would just be `(3 -> 2)`. We would have:

   Original list: `1 -> 4 -> 2 -> 3`
   After reversing: `1 -> 4 -> 3 -> 2`
3. **Calculating Maximum Twin Sum**: We initialize two pointers, `pa` at the head of the list (`1`) and `pb` at the head of the reversed second half (`3`). Iterating through both halves simultaneously:
   - On the first calculation, `pa.val` is `1`, and `pb.val` is `3`. The sum is `4`.
   - Move `pa` to the next node (`4`) and `pb` to its next node (`2`).
   - On the second calculation, `pa.val` is `4`, and `pb.val` is `2`. The sum is `6`.
   - We track the maximum sum found, which is `6` after this iteration.
4. **Returning the Result:** With the iterations complete and no more nodes to traverse, we found that the maximum twin sum in our example linked list `[1 -> 4 -> 2 -> 3]` is `6`. We return this value as the final result.

Python Solution

```
1  class ListNode:
2      def __init__(self, val=0, next=None):
3          self.val = val
4          self.next = next
5
6  class Solution:
7      def pairSum(self, head: Optional[ListNode]) -> int:
8          # define a function to reverse a linked list
9          def reverse_list(node):
10             prev = None
11             current_node = node
12             # iterate through the list and reverse the links
13             while current_node:
14                 next_node = current_node.next
15                 current_node.next = prev
16                 prev = current_node
17                 current_node = next_node
18             return prev
19
20         # Initialize slow and fast pointers to find the middle of the linked list
21         slow_pointer, fast_pointer = head, head.next
22         while fast_pointer and fast_pointer.next:
23             slow_pointer, fast_pointer = slow_pointer.next, fast_pointer.next.next
24
25         # Split the list into two halves
26         first_half = head
27         second_half_start = slow_pointer.next
28         slow_pointer.next = None  # Break the list to create two separate lists
29
30         # Reverse the second half of the list
31         second_half = reverse_list(second_half_start)
32
33         # Initialize the answer to zero
34         max_pair_sum = 0
35
36         # Traverse both halves simultaneously and find the max pair sum
37         while first_half_head and second_half_head:
38             current_pair_sum = first_half_head.val + second_half_head.val
39             max_pair_sum = max(max_pair_sum, current_pair_sum)
40             first_half_head = first_half_head.next
41             second_half_head = second_half_head.next
42
43         # Return the maximum pair sum found
44         return max_pair_sum
45
```

Java Solution

```
1  // Definition for singly-linked list.
2  class ListNode {
3      int val;
4      ListNode next;
5      ListNode() {}
6      ListNode(int val) { this.val = val; }
7      ListNode(int val, ListNode next) { this.val = val; this.next = next; }
8  }
9
10 class Solution {
11     // Function to find the maximum Twin Sum in a singly-linked list
12     public int pairSum(ListNode head) {
13         // Initialize two pointers, slow and fast
14         ListNode slow = head;
15         ListNode fast = head.next;
16
17         // Use fast and slow pointers to find the middle of the list
18         while (fast != null && fast.next != null) {
19             slow = slow.next; // Move slow pointer one step
20             fast = fast.next.next; // Move fast pointer two steps
21         }
22
23         // After the loop, slow will be at the middle of the list
24         // The next step is to reverse the second half of the linked list
25         ListNode firstHalf = head; // Start of the first half
26         ListNode secondHalfStart = slow.next; // Start of the second half
27         slow.next = null; // Split the list into two halves
28         ListNode reversedSecondHalf = reverseLinkedList(secondHalfStart); // Reverse the second half
29
30         // Initialize the maximum sum variable
31         int maxSum = 0;
32
33         // Traverse the two halves together
34         while (firstHalf != null) {
35             // Update maxSum to be the max of current maxSum and the sum of the current nodes from both halves
36             maxSum = Math.max(maxSum, firstHalf.val + reversedSecondHalf.val);
37             firstHalf = firstHalf.next; // Move to the next node in the first half
38             reversedSecondHalf = reversedSecondHalf.next; // Move to the next node in the reversed second half
39         }
40
41         // Return the maximum sum found
42         return maxSum;
43     }
44
45     // Helper function to reverse a singly-linked list
46     private ListNode reverseLinkedList(ListNode head) {
47         ListNode dummyHead = new ListNode(); // Create a new dummy node to help with the reversal
48         ListNode current = head; // Start from the head of the list
49
50         // Traverse the list and reverse pointers
51         while (current != null) {
52             ListNode next = current.next; // Keep the reference to the next node
53             current.next = dummyHead.next; // Reverse the pointer
54             dummyHead.next = current; // Move dummy next to the current node
55             current = next; // Move to the next node
56         }
57
58         // Return the reversed list, which starts at dummy.next
59         return dummyHead.next;
60     }
61 }
62
63
```

C++ Solution

```
1  /**
2   * Definition for singly-linked list.
3   * struct ListNode {
4   *     int val;
5   *     ListNode *next;
6   *     ListNode() : val(0), next(nullptr) {}
7   *     ListNode(int x) : val(x), next(nullptr) {}
8   *     ListNode(int x, ListNode *next) : val(x), next(next) {}
9   * };
10  */
11 class Solution {
12 public:
13     // Main function to find the twin sum of the linked list.
14     int pairSum(ListNode* head) {
15         // Initialize slow and fast to find the midpoint of the linked list.
16         ListNode* slow = head;
17         ListNode* fast = head->next;
18         while (fast && fast->next) {
19             slow = slow->next;
20             fast = fast->next->next;
21         }
22
23         // Split the linked list into two halves.
24         ListNode* firstHalf = head;
25         ListNode* secondHalfStart = slow->next;
26         slow->next = nullptr; // This null assignment splits the linked list into two.
27
28         // Reverse the second half of the linked list.
29         ListNode* reversedSecondHalf = reverse(secondHalfStart);
30
31         // Initialize the maximum pair sum as zero.
32         int maxPairSum = 0;
33
34         // Traverse the two halves in tandem to calculate the max pair sum.
35         while (firstHalf != nullptr && reversedSecondHalf != nullptr) {
36             maxPairSum = max(maxPairSum, firstHalf->val + reversedSecondHalf->val);
37             firstHalf = firstHalf->next;
38             reversedSecondHalf = reversedSecondHalf->next;
39         }
40
41         return maxPairSum;
42     }
43
44     // Helper function to reverse the linked list.
45     ListNode* reverse(ListNode* head) {
46         ListNode* prev = nullptr;
47         ListNode* current = head;
48
49         // Iterate over the linked list and reverse the pointers.
50         while (current) {
51             ListNode* next = current->next;
52             current->next = prev;
53             prev = current;
54             current = next;
55         }
56
57         // The 'prev' pointer now points to the new head of the reversed linked list.
58         return prev;
59     }
60 };
61
62
63
```

Typescript Solution

```
1  // Function to find the maximum twin sum of a linked list. The twin sum is defined as the sum of nodes that are equidistant from the
2  function pairSum(head: ListNode | null): number {
3      // Initialize two pointers, slow and fast. The fast pointer will move at twice the speed of the slow pointer.
4      let slow: ListNode | null = head;
5      let fast: ListNode | null = head.next;
6
7      // Move the fast pointer two nodes at a time and the slow pointer one node at a time. The loop ends when fast reaches the end of
8      while (fast && fast.next) {
9          slow = slow.next!; // Move slow one step
10         fast = fast.next.next; // Move fast two steps
11     }
12
13     // Reverse the second half of the linked list, starting from the slow pointer.
14     let prev: ListNode | null = null;
15     while (slow) {
16         const next: ListNode | null = slow.next;
17         slow.next = prev;
18         prev = slow;
19         slow = next;
20     }
21
22     // The 'prev' pointer now points to the head of the reversed second half of the list. 'head' points to the beginning of the first
23
24     // Initialize 'maxSum' to keep track of the maximum twin sum.
25     let maxSum: number = 0;
26
27     // Iterate through both halves of the list.
28     while (prev !== null && head !== null) {
29         let right: ListNode = prev!.val + head.val;
30
31         // Calculate the twin sum by adding values of 'left' and 'right' pointers and update 'maxSum' if a larger sum is found.
32         while (left.$$.right) {
33             maxSum = Math.maxSum, left.val + right.val);
34             left = left.next;
35             right = right.next!;
36         }
37
38         // Return the maximum twin sum after traversing the entire list.
39         return maxSum;
40     }
41 }
```

Time and Space Complexity

The given Python code defines a function to find the maximum twin sum in a singly-linked list, where the twin sum is defined as the sum of values from nodes that are equidistant from the start and end of the list.

Time Complexity

1. **Reversing the second half of the linked list:** The `reverse` function iterates over the nodes of the second half of the original linked list once, with a time complexity of $O(n/2)$, where `n` is the total number of nodes in the list, because it stops at the midpoint of the list due to the previous pointer manipulation.
2. **Finding the midpoint of the linked list:** This is achieved by the "tortoise and hare" technique, where we move a slow pointer by one step and a fast pointer by two steps. When the fast pointer reaches the end, the slow pointer will be at the midpoint, and this has a time complexity of $O(n/2)$ since the fast pointer traverses the list twice as fast as the slow pointer.
3. **Calculating the maximum twin sum:** This involves a single pass through the first half of the list while simultaneously traversing the reversed second half of the list, yielding a time complexity of $O(n/2)$.

Adding these together gives $2 * O(n/2) + O(n/2)$, simplifying to $O(n)$ as constants are dropped and $n/2$ is ultimately proportional to n.

Space Complexity

1. **Additional space for reversing:** The `reverse` function uses a constant amount of extra space, as it just rearranges the pointers without allocating any new list nodes.
2. **Space for pointers:** A few extra pointers are used for traversal and reversing the linked list (`dummy`, `curr`, `slow`, `fast`, `pa`, `pb`), which also contribute a constant amount of space.

Therefore, the space complexity of the provided code is $O(1)$ since it does not grow with the size of the input list and only a fixed amount of extra memory is utilized.