2807. Insert Greatest Common Divisors in Linked List

Medium Array Linked List Math

Leetcode Link

Problem Description

The problem requires us to modify a given singly linked list by inserting new nodes. Specifically, for every two adjacent nodes in the list, we need to insert a new node between them. The value of this new node is not just any number but the greatest common divisor (GCD) of the values of these two adjacent nodes. The GCD of two numbers is the largest number that divides both of them without leaving a remainder.

For example, if our linked list is `1 -> 4 -> 3 -> 8`, then after inserting new nodes, it should look like `1 -> 1 -> 4 -> 1 -> 3 -> 1 -> 8`, where the numbers `1`, `1`, and `1` are the GCDs of `1` and `4`, `4` and `3`, `3` and `8` respectively. Note that if the GCD is `1`, it means that the two numbers are coprime (they have no common divisor other than `1`).

Intuition

For solving this problem, we need to traverse the linked list and calculate the GCD of every two adjacent values. We do not need any extra space besides the new nodes that we are inserting, and we can do the insertion in a single pass through the list.

The steps involve:

- Start by pointing to the first node of the list.
- While we have not reached the end of the list (i.e., the current node's `next` is not `None`):
  - Calculate the GCD of the current node's value and the next node's value.
  - Create a new node with the calculated GCD value.
  - Insert this new node between the current node and the next node.
  - Move to the next pair of nodes to continue the process.

We use a utility function, `gcd`, which will calculate the greatest common divisor of two numbers. Python provides a built-in function for this in the `math` module, which could be employed to simplify our code. The process of inserting the new node involves manipulating the `next` pointers of the nodes to accommodate the new GCD node in between.

Overall, this approach works because linked lists allow for efficient insertion operations since, unlike arrays, we do not need to shift elements after the insertion point. We can simply rewire the `next` pointers appropriately to complete the insertion.

Solution Approach

The solution involves manipulation of linked list nodes and calculating the greatest common divisor (GCD) between adjacent nodes. The steps below detail the algorithm used in the provided code:

1. **Initialization**: We initialize two pointers, `pre` and `cur`. The `pre` holds the reference to the current node, where we will insert a new node after, and `cur` holds the reference to the next node, which will be used for calculating GCD. Initially, `pre` points to `head` and `cur` points to `head.next`.

2. **Traversal and Insertion**:
   - We enter a loop that continues to iterate as long as `cur` is not `None`, meaning there are still nodes to process.
   - Inside the loop, we calculate the GCD of `pre.val` and `cur.val` using a function `gcd`. This function is assumed to be predefined or imported from the `math` module in Python (`from math import gcd`).
   - We create a new `ListNode` with the calculated GCD, and its `next` points to `cur`. This effectively creates a new node between `pre` and `cur`.
   - Next, we rewire the `next` pointer of the `pre` node to point to the new node.
   - We update the `pre` pointer to now point to the `cur` node, since we've finished inserting the new node after what `pre` was pointing to before.
   - Lastly, we update `cur` to point to `cur.next`, moving forward in the linked list to the next pair of nodes.
3. **Algorithm Pattern**: The approach here doesn't use extra space for an auxiliary data structure but directly works on the list itself, modifying pointers as needed. It's an in-place algorithm with respect to linked list manipulation, although new nodes are created for storing the GCDs.

4. **GCD Function**: The GCD function utilized is a key part of this solution, which calculates the greatest common divisor of two given numbers. If Python's built-in `gcd` function is not being used, an implementation of the Euclidean algorithm would typically be employed.

This implementation's time complexity is O(n * log(max(A))), where `n` is the number of nodes in the list and `log(max(A))` represents the time complexity of computing the GCD for any two numbers in the list (assuming `A` is the set of integers in the nodes). The space complexity is O(1) ignoring the space taken by the output list; in a more practical sense, considering the new nodes, it would be O(n) due to the new nodes being inserted into the linked list.

Example Walkthrough

Let us consider a small linked list: `5 -> 10`. According to the problem, we need to find the GCD of every two adjacent nodes and insert a new node with that value in between them. In this case, we only need to find the GCD of 5 and 10. Using the Euclidean algorithm, we see that the GCD of 5 and 10 is 5.

Here is the step-by-step process based on the solution approach:

1. **Initialization**: We begin with two pointers, `pre` is pointing to the node with value 5 (head of the list), and `cur` is pointing to the node with value 10 (`head.next`).

2. **Traversal and Insertion**:
   - Since `cur` is not `None`, we enter the loop.
   - We compute the GCD of `pre.val` (5) and `cur.val` (10), which is 5.
   - A new node is created with the GCD value 5, and its `next` is set to point to `cur` (`ListNode(5, cur)`).
   - We then update `pre.next` to point to this new node. The linked list now looks like `5 -> 5 -> 10`.
   - We move `pre` to point to `cur`, which is the node with the value 10.
   - `cur` is updated to `cur.next`, which is `None` (end of the list).

By end of these steps, the traversal and insertion phase is complete, as there are no more nodes to process. The final linked list after the GCD insertion is `5 -> 5 -> 10`. It must be noted that if the linked list were longer, we would continue the process until we reach the end of the list.

3. **Algorithm Pattern**: In our example, we traversed the list and inserted the nodes without any extra data structures or modification of the original nodes' values. We only changed the `next` pointers to accommodate the new nodes.

4. **GCD Function**: The GCD value was calculated using the known algorithm, or we could have used Python's built-in `gcd` function for simplification (`from math import gcd`).

This example demonstrates the workings of our approach on a very small linked list. For linked lists with more nodes, the method would continue analogously until every pair of adjacent nodes has been processed.

Python Solution

```python
from math import gcd

# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next_node=None):
        self.val = val
        self.next = next_node

class Solution:
    def insertGreatestCommonDivisors(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # Initialize pointers: 'prev_node' pointing to the current node being processed,
        # and 'current_node' pointing to the next node in the list.
        prev_node, current_node = head, head.next

        # Iterate through the linked list until we reach the end.
        while current_node:
            # Compute Greatest Common Divisor (GCD) of the values in the 'prev_node' and 'current_node'.
            gcd_value = gcd(prev_node.val, current_node.val)

            # Insert a new node with the GCD value between the 'prev_node' and 'current_node'.
            prev_node.next = ListNode(gcd_value, current_node)

            # Update pointers: move 'prev_node' to the 'current_node' and
            # 'current_node' to the next node in the list.
            prev_node, current_node = current_node, current_node.next

        # Return the modified linked list head.
        return head
```

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; }
}

public class Solution {

    /**
     * Inserts a new node with the value of the greatest common divisor
     * between each pair of adjacent nodes in the linked list.
     *
     * @param head The head of the singly-linked list.
     * @return The head of the modified linked list, with new nodes added.
     */
    public ListNode insertGreatestCommonDivisors(ListNode head) {
        // Initialize two pointers for traversing the link list.
        ListNode prev = head;
        ListNode curr = head.next;

        // Iterate over the list until we reach the end.
        while (curr != null) {
            // Get the greatest common divisor of the values in the two nodes.
            int gcdValue = gcd(prev.val, curr.val);

            // Insert a new node with the GCD value between the two nodes.
            prev.next = new ListNode(gcdValue, curr);

            // Move the pointers forward to the next pair of nodes.
            prev = curr;
            curr = curr.next;
        }

        // Return the unchanged head of the modified list.
        return head;
    }

    /**
     * Helper function to calculate the greatest common divisor of two integers.
     *
     * @param a The first integer.
     * @param b The second integer.
     * @return The greatest common divisor of a and b.
     */
    private int gcd(int a, int b) {
        // If the second number, b, is 0, return the first number, a.
        if (b == 0) {
            return a;
        }
        // Keep calling the gcd function recursively, reducing the problem size.
        return gcd(b, a % b);
    }
}
```

C++ Solution

```cpp
#include <algorithm> // Include algorithm header to use std::gcd

// Definition for singly-linked list node.
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {} // Default constructor
    ListNode(int x) : val(x), next(nullptr) {} // Constructor initializing val
    ListNode(int x, ListNode *next) : val(x), next(next) {} // Constructor initializing both val and next
};

class Solution {
public:
    // Function to insert nodes containing the Greatest Common Divisor (GCD) of the
    // adjacent nodes' values, between those nodes in the list.
    ListNode* insertGreatestCommonDivisors(ListNode* head) {
        if (!head) return nullptr; // Return if the list is empty

        ListNode* current = head; // Current node is set to the head of the list
        ListNode* nextNode = head->next; // Next node is the one following current

        // Iterate over each pair of adjacent nodes until the end of the list
        while (nextNode) {
            // Calculate GCD of the values in the current and next nodes
            int gcdValue = std::gcd(current->val, nextNode->val);

            // Insert a new node with calculated GCD value between current and next nodes
            current->next = new ListNode(gcdValue, nextNode);

            // Move to the next pair of nodes, skipping over the newly inserted node
            current = nextNode;
            nextNode = nextNode->next;
        }

        return head; // Return the updated list with inserted GCD nodes
    }
};
```

Typescript Solution

```typescript
// Type definition for a node in a singly-linked list.
type ListNode = {
    val: number;
    next: ListNode | null;
};

/**
 * Inserts the greatest common divisor (GCD) of the values of each pair of adjacent nodes into the list.
 * @param head The head of the singly-linked list.
 * @returns the modified list with GCD nodes inserted.
 */
function insertGreatestCommonDivisors(head: ListNode | null): ListNode | null {
    // Pointers for the current and next node in the list
    let currentNode = head;
    let nextNode = head?.next;

    // Loop through each pair of adjacent nodes in the list
    while (nextNode) {
        // Calculate the GCD of the values of currentNode and nextNode
        const gcdValue = gcd(currentNode.val, nextNode.val);
        // Insert a new node with the GCD value between the currentNode and nextNode
        currentNode.next = { val: gcdValue, next: nextNode };
        // Move the currentNode pointer to the next node in the original list
        currentNode = nextNode;
        // Move the nextNode pointer to the node following the nextNode in the original list
        nextNode = nextNode.next;
    }

    return head;
}

/**
 * Calculates the greatest common divisor (GCD) of two non-negative integers.
 * @param a The first integer.
 * @param b The second integer.
 * @returns the GCD of a and b.
 */
function gcd(a: number, b: number): number {
    if (b === 0) {
        return a;
    }
    // Recursively call gcd with b and the remainder of a divided by b
    return gcd(b, a % b);
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given algorithm is determined by how many times it iterates through the linked list and how many times it computes the greatest common divisor (gcd) for adjacent nodes.

- The algorithm traverses each node of the linked list once to insert the computed gcds. If there are `n` nodes in the list, it makes `n-1` computations of gcd because we compute gcd for every pair of adjacent nodes.

- The time complexity of the `gcd` function depends on the values of the nodes. The worst-case scenario for Euclid's algorithm is when the two numbers are consecutive Fibonacci numbers, which leads to a time complexity of `O(log(min(a, b)))` where `a` and `b` are the values of the nodes.

Combining these two factors, the overall time complexity of the algorithm is `O(n * log(min(a, b)))`.

Space Complexity

- The space complexity is determined by the additional space needed by the algorithm which is not part of the input. In this case, we are inserting `n-1` new nodes to store the gcds, this requires `O(n)` additional space.

Therefore, the space complexity of the algorithm is `O(n)`.