2058. Find the Minimum and Maximum Number of Nodes Between Critical Points

Medium Linked List Leetcode Link

Problem Description

In this problem, we're dealing with a linked list where we need to find critical points - which are defined as either a local maxima or a local minima. A local maxima is considered as a node that has a value strictly greater than the nodes before and after it. Similarly, a local minima is a node that has a value strictly smaller than the nodes before and after it. However, a node can only be considered a critical point if it isn't the first or last node, meaning it has both a previous and a next node.

Your task is to calculate the minimum and maximum distance between any two critical points in the linked list. The distance is calculated based on the positions of the nodes, not their values. If there are less than two critical points, the answer will be `[-1, -1]`.

Intuition

To solve this problem, we need to traverse through the linked list to find the critical points and calculate the minimum and maximum distances between them.

1. We initiate a traversal starting with the second node in the list since the first node can't be a critical point by the problem's definition.
2. As we traverse, we compare the value of the current node with the values of its previous and next nodes to determine if it's a local maxima or minima.
3. When we identify a critical point, we calculate the distance from the last found critical point. If it's the first critical point found, we simply mark it as `first` and `last`; otherwise, we update the minimum distance every time we find a new critical point, and we set the maximum distance as the difference between the first critical point found and the current one, since this is likely to be the maximum distance as we traverse the list.
4. To keep track of the index of nodes, we use a counter that increments with each traversal step.
5. We utilize a pair of variables `[minDistance, maxDistance]` initialized to `[inf, -inf]` to keep track of the distances. These will be updated accordingly as we identify critical points.
6. At the end of traversal, we check whether we have found at least two critical points. If we have only found one or none, we return `[-1, -1]`. Otherwise, we return the distances found.

By following this approach, we ensure that we only pass through the list once (O(n) time complexity), and use O(1) auxiliary space, ensuring an efficient solution.

Solution Approach

The Solution Approach to finding the critical points in a linked list involves a single pass algorithm and judicious use of pointers and variables to keep track of the critical points and their distances.

1. **Algorithm**: We start by initializing our pointers, `prev` and `curr`, to the first and second nodes of the list respectively, because a critical point requires both a previous and next node; therefore, the first node cannot be one.
2. **Tracking First and Last Critical Point**: We declare variables `first` and `last` to remember the positions of the first and last critical points found. They are initially set to `None`.
3. **Initializing the Answer**: We prepare an array `ans` with two elements `[inf, -inf]` representing the minimum and maximum distances respectively. `inf` is a placeholder indicating that no minimum distance has been found yet, and `-inf` indicating that no maximum distance has been found yet.
4. **Traversing the List**: We iterate through the linked list with a `while` loop that continues as long as the current node has a next node.
5. **Identifying Critical Points**:
   - In each iteration, we check if `curr.val` (the value of the current node) is a local maximum or minimum compared to `prev.val` and `curr.next.val`.
   - This is done by checking if `curr.val` is strictly greater than both `prev.val` and `curr.next.val` for a local maximum or strictly smaller for a local minimum.
6. **Updating Distances**: If a critical point is found:
   - If it's the first critical point, we initialize both `first` and `last` to the current index `i`.
   - If it's not the first, we:
     - Calculate the distance from the last critical point by subtracting `last` from the current index `i` and update `ans[0]` with the minimum distance found so far.
     - Update `ans[1]` with the distance between the first critical point and the current one to maintain the maximum distance.
     - Then `last` is updated to the current index as the last seen critical point.
7. **Incrementing Index and Updating Pointers**: After checking for a critical point, we increment our index `i` and move our pointers ahead to the next elements (`prev` and `curr`) to continue our traversal.
8. **Returning the Result**: Once we finish traversing the list:
   - If `first` and `last` are the same, it means that either no critical points were found or there was only one critical point. In that case, we return `[-1, -1]`.
   - Otherwise, we return the distances stored in `ans`, which contain the minimum and maximum distances between critical points.

By applying this step-by-step approach, we efficiently calculate the required distances with straight forward logic and optimal time and space complexity.

Example Walkthrough

Consider the following linked list as an example:

```
List: 1 -> 3 -> 2 -> 5 -> 3 -> 4
```

Following our solution approach:

1. We initialize `prev` to node with value `1` and `curr` to node with value `3`, with an index `i` starting at `1`.
2. Our variables `first` and `last` are initialized to `None`. The array `ans` is set to `[inf, -inf]`.
3. As we traverse the list, we reach the second node with value `3`, it isn't a critical point because it isn't smaller or larger than both neighbours.
4. Move to the third node with value `2`, this is a local minima since `2` is less than both `3` and `5`. This is the first critical point, so we set both `first` and `last` to the current index `i = 2`. Since there's no previous critical point to compare with, we proceed without updating `ans`.
5. The next node, with value `5`, is a local maxima as it is greater than `2` and `3`. We find the distance to the last critical point (distance = 3 - 2 = 1), and update the `ans` array with the minimum distance thus far, `ans[0] = min(inf, 1) = 1`. We then set `last` to `3` since `5` is now the last critical point we've seen.
6. Moving forward, node `3` is not a critical point because it isn't a local minima or maxima.
7. Finally, the node with value `4` is a local maxima. We update the minimum distance if it's smaller than the current `ans[0]` value (it's not in this case: `1 < 2` is false), and then, we update the `ans[1]` with the maximum distance found which is from the first critical point to the current one (distance = 5 - 2 = 3). We also update `last` to the current index `i = 5`.
8. The traversal ends, and since `first != last`, we have found at least two critical points.
9. We conclude the minimum distance between critical points is `1` and the maximum is `3`, hence we return `[1, 3]`.

Here's the example summarized in the required markdown template:

```

Let's consider a linked list with the following sequence of nodes:

```

1 → 3 → 2 → 5 → 3 → 4

```

We'll walk through the solution approach step by step.

- We start at the second node (`3`) since the first node can't be a critical point.
- The second node (`3`) isn't a local maxima or minima, so we move to the next node.
- At the third node (`2`), we identify our first critical point, it's a local minima.
- Then we come across the fourth node (`5`) which is a local maxima. We find that the minimum distance to the previous critical point
- The fifth node (`3`) isn't a critical point, so we continue.
- Lastly, at the sixth node (`4`), we've got another local maxima. We now know the maximum distance is `3`, between the third node ar

After walking through the entire linked list, we determine the minimum distance between critical points is `1` and the maximum distar
```

Python Solution

```python
from typing import Optional, List
from math import inf

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

class Solution:
    def nodesBetweenCriticalPoints(self, head: Optional[ListNode]) -> List[int]:
        # Initialize markers for previous and current nodes
        prev_node, curr_node = head, head.next

        # Initialize the first and last positions of critical points to None
        first_critical, last_critical = None, None

        index = 1  # Start from index 1 since we are checking from the second node
        min_distance, max_distance = inf, -inf  # Set initial distances

        # Iterate through the linked list to find critical points
        while curr_node.next:
            # Check if the current node is a critical point (local min or local max)
            if curr_node.val < min(prev_node.val, curr_node.next.val) or \
               curr_node.val > max(prev_node.val, curr_node.next.val):
                # If it is the first critical point encountered, set both first and last
                if last_critical is None:
                    first_critical = last_critical = index
                else:
                    # Update the minimum distance if the current critical point
                    # is closer to the previous critical point than other pairs
                    min_distance = min(min_distance, index - last_critical)
                    # Update the maximum distance from the current to the first critical point
                    max_distance = index - first_critical
                    # Update the last critical point to the current index
                    last_critical = index
            # Move to the next index
            index += 1
            # Update prev_node and curr_node to step through the list
            prev_node, curr_node = curr_node, curr_node.next

        # If no critical points or only one critical point found, return [-1, -1]
        # else, return the found minimum and maximum distances
        return [min_distance, max_distance] if first_critical != last_critical else [-1, -1]

```

Java Solution

```java
class Solution {
    public int[] nodesBetweenCriticalPoints(ListNode head) {
        // Initialize pointers and set the index for iteration
        ListNode previousNode = head;
        ListNode currentNode = head.next;
        int firstCriticalIndex = 0, lastCriticalIndex = 0;
        int index = 1;

        // Initialize the answer array with maximum and minimum values respectively
        int[] answer = new int[] {Integer.MAX_VALUE, Integer.MIN_VALUE};

        // Iterate through the list until we reach the end
        while (currentNode.next != null) {
            // Check if the current node is a critical point (local minimum or maximum)
            if (currentNode.val < Math.min(previousNode.val, currentNode.next.val)
                    || currentNode.val > Math.max(previousNode.val, currentNode.next.val)) {
                // Update the first and last critical points found
                if (lastCriticalIndex == 0) {
                    // This means we found the first critical point
                    firstCriticalIndex = index;
                    lastCriticalIndex = index;
                } else {
                    // Update the minimum distance between critical points
                    answer[0] = Math.min(answer[0], index - lastCriticalIndex);
                    // Update the maximum distance from the first to the current critical point
                    answer[1] = index - firstCriticalIndex;
                    // Update the last critical point found
                    lastCriticalIndex = index;
                }
            }
            // Move to the next node in the list
            ++index;
            previousNode = currentNode;
            currentNode = currentNode.next;
        }

        // If only one critical point was found, then return [-1, -1]
        // Otherwise, return the array containing minimum and maximum distances
        return firstCriticalIndex == lastCriticalIndex ? new int[] {-1, -1} : answer;
    }
}

/**
 * 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;
    }
}

```

C++ Solution

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */

class Solution {
public:
    vector<int> nodesBetweenCriticalPoints(ListNode* head) {
        // Pointers to track the current and previous nodes in the list
        ListNode* previous = head;
        ListNode* current = head->next;

        // Variables to keep track of the positions of the first and last critical points
        int firstCriticalPosition = 0, lastCriticalPosition = 0;
        int index = 1; // Current index within the linked list

        // Initialize the answer with the maximum value for distance calculations
        vector<int> answer(2, INT_MAX);

        // Iterate through the linked list to find critical points
        while (current->next) {
            // Check for a critical point, which is either a local minimum or maximum
            bool isCritical = current->val < min(previous->val, current->next->val) ||
                              current->val > max(previous->val, current->next->val);

            if (isCritical) {
                if (lastCriticalPosition == 0) {
                    // This is the first critical point encountered
                    firstCriticalPosition = index;
                } else {
                    // Update the shortest distance between any two critical points
                    answer[0] = min(answer[0], index - lastCriticalPosition);
                    // Update the distance between the first and the current critical point
                    answer[1] = index - firstCriticalPosition;
                }
                // Update the position of the last critical point
                lastCriticalPosition = index;
            }
            ++index; // Move to the next index
            previous = current; // Move the previous pointer forward
            current = current->next; // Move the current pointer forward
        }
        // If no critical points or only one critical point was found, return {-1, -1}
        if (firstCriticalPosition == lastCriticalPosition) {
            return {-1, -1};
        }
        // Return the distances between critical points
        return answer;
    }
};

```

Typescript Solution

```typescript
/**
 * Definition for singly-linked list.
 */
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);
    }
}

/**
 * Finds the minimum and maximum distance between any two critical points in a linked list.
 * Critical points are defined as local minima or local maxima.
 * If fewer than two critical points are found, returns [-1, -1].
 * @param {ListNode | null} head - The head node of the linked list.
 * @return {number[]} - An array with two elements: the minimum and maximum distances between critical points.
 */
function nodesBetweenCriticalPoints(head: ListNode | null): number[] {
    let index = 1; // Stores the current node index.
    let previousValue = head.val; // Stores the value of the previous node.
    head = head.next; // Move head to the next node.
    let criticalPoints = []; // Stores the indices of the critical points.

    // Traverse the list to find all critical points.
    while (head && head.next !== null) {
        let currentValue = head.val;
        let postValue = head.next.val;
        // Check if the current node is a local maximum or minimum.
        if ((previousValue < currentValue && currentValue > postValue) ||
            (previousValue > currentValue && currentValue < postValue)) {
            criticalPoints.push(index);
        }
        previousValue = currentValue;
        index++;
        head = head.next;
    }

    // Determine the number of critical points.
    let numCriticalPoints = criticalPoints.length;
    // If there are less than two critical points, return [-1, -1].
    if (numCriticalPoints < 2) return [-1, -1];

    let minDistance = Infinity; // Initialize the minimum distance to infinity.
    // Calculate the minimum distance between consecutive critical points.
    for (let i = 1; i < numCriticalPoints; i++) {
        minDistance = Math.min(criticalPoints[i] - criticalPoints[i - 1], minDistance);
    }

    // Calculate the maximum distance, which is distance between the first and the last critical points.
    let maxDistance = criticalPoints[numCriticalPoints - 1] - criticalPoints[0];

    // Return the minimum and maximum distances as an array.
    return [minDistance, maxDistance];
}

```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the number of nodes in the linked list. The algorithm iterates through the list once with a single pass, checking each node to determine if it's a critical point—either a local minimum or a local maximum. The checks for minimum and maximum at each step are constant-time operations.

The space complexity of the code is `O(1)`. The space used by the algorithm does not scale with the input size as it only maintains a constant number of pointers and variables (`prev`, `curr`, `first`, `last`, `i`, and `ans`) regardless of the length of the list.