2058. Find the Minimum and Maximum Number of Nodes **Between Critical Points** Medium <u>Linked List</u>

Problem Description

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].

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

Leetcode Link

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.

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.

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

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,
- 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.

3. Initializing the Answer: We prepare an array ans with two elements [inf, -inf] representing the minimum and maximum

2. Tracking First and Last Critical Point: We declare variables first and last to remember the positions of the first and last critical

distances respectively. inf is a placeholder indicating that no minimum distance has been found yet, and -inf indicating that no

The Solution Approach to finding the critical points in a linked list involves a single pass algorithm and judicious use of pointers and

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.

points found. They are initially set to None.

smaller for a local minimum.

If it's not the first, we:

case, we return $\begin{bmatrix} -1, & -1 \end{bmatrix}$.

points.

and space complexity.

neighbours.

 $1 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 4$

Python Solution

class ListNode:

class Solution:

from math import inf

1 from typing import Optional, List

self.val = val

self.next = next_node

while curr_node.next:

else:

index += 1

10

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

40

42

49

50

51

52

54

55

56

57

58

59

61

60 }

41 }

*/

class ListNode {

int val;

C++ Solution

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

ListNode(int val, ListNode next) {

* Definition for singly-linked list.

Example Walkthrough

Following our solution approach:

since 5 is now the last critical point we've seen.

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

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

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

Definition for a singly-linked list node

def __init__(self, val=0, next_node=None):

6. **Updating Distances**: If a critical point is found:

the minimum distance found so far.

ensuring an efficient solution.

Solution Approach

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

If it's the first critical point, we initialize both first and last to the current index i.

Then last is updated to the current index as the last seen critical point.

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

o Otherwise, we return the distances stored in ans, which contain the minimum and maximum distances between critical

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

7. Incrementing Index and Updating Pointers: After checking for a critical point, we increment our index i and move our pointers

Calculate the distance from the last critical point by subtracting last from the current index i and update ans [0] with

Update ans [1] with the distance between the first critical point and the current one to maintain the maximum distance.

Consider the following linked list as an example: 1 List: 1 -> 3 -> 2 -> 5 -> 3 -> 4

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].

6. Moving forward, node 3 is not a critical point because it isn't a local minima or maxima.

8. The traversal ends, and since first != last, we have found at least two critical points.

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

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

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.

9. We conclude the minimum distance between critical points is 1 and the maximum is 3, hence we return [1, 3].

 We start at the second node (`3`) since the first node can't be a critical point. 5 - The second node (`3`) isn't a local maxima or minima, so we move to the next node. 6 - At the third node (`2`), we identify our first critical point, it's a local minima. 7 - Then we come across the fourth node (`5`) which is a local maxima. We find that the minimum distance to the previous critical point 8 - The fifth node (`3`) isn't a critical point, so we continue.

9 - 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

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

Initialize markers for previous and current nodes 12 13 prev_node, curr_node = head, head.next 14 # Initialize the first and last positions of critical points to None 15 16 first_critical, last_critical = None, None

if last_critical is None:

last_critical = index

Move to the next index

def nodesBetweenCriticalPoints(self, head: Optional[ListNode]) -> List[int]:

min_distance, max_distance = inf, -inf # Set initial distances

first_critical = last_critical = index

max_distance = index - first_critical

prev_node, curr_node = curr_node, curr_node.next

else, return the found minimum and maximum distances

Update prev_node and curr_node to step through the list

Iterate through the linked list to find critical points

index = 1 # Start from index 1 since we are checking from the second node

if curr_node.val < min(prev_node.val, curr_node.next.val) or \</pre>

curr_node.val > max(prev_node.val, curr_node.next.val):

Check if the current node is a critical point (local min or local max)

If it is the first critical point encountered, set both first and last

Update the maximum distance from the current to the first critical point

Update the minimum distance if the current critical point

is closer to the previous critical point than other pairs

return [min_distance, max_distance] if first_critical != last_critical else [-1, -1]

min_distance = min(min_distance, index - last_critical)

Update the last critical point to the current index

If no critical points or only one critical point found, return [-1, -1]

```
Java Solution
1 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};
10
11
12
           // Iterate through the list until we reach the end
           while (currentNode.next != null) {
13
               // Check if the current node is a critical point (local minimum or maximum)
14
               if (currentNode.val < Math.min(previousNode.val, currentNode.next.val)</pre>
15
                       || currentNode.val > Math.max(previousNode.val, currentNode.next.val)) {
16
                   // Update the first and last critical points found
17
                   if (lastCriticalIndex == 0) {
18
                       // This means we found the first critical point
19
20
                       firstCriticalIndex = index;
21
                       lastCriticalIndex = index;
                   } else {
23
                       // Update the minimum distance between critical points
24
                       answer[0] = Math.min(answer[0], index - lastCriticalIndex);
25
                       // Update the maximum distance from the first to the current critical point
26
                       answer[1] = index - firstCriticalIndex;
27
                       // Update the last critical point found
                       lastCriticalIndex = index;
28
29
30
31
               // Move to the next node in the list
32
               ++index;
33
               previousNode = currentNode;
               currentNode = currentNode.next;
34
35
36
37
           // If only one critical point was found, then return [-1, -1]
38
           // Otherwise, return the array containing minimum and maximum distances
39
           return firstCriticalIndex == lastCriticalIndex ? new int[] {-1, -1} : answer;
```

17 18 19 20

12

13

19

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

46

47

48

49

50

51

52

/**

*/

```
ListNode(int x, ListNode *next) : val(x), next(next) {}
    * };
10
    */
11
   class Solution {
   public:
       vector<int> nodesBetweenCriticalPoints(ListNode* head) {
14
           // 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
21
22
23
           // Initialize the answer with the maximum value for distance calculations
24
           vector<int> answer(2, INT_MAX);
25
26
           // Iterate through the linked list to find critical points
           while (current->next) {
27
28
               // Check for a critical point, which is either a local minimum or maximum
29
               bool isCritical = current->val < min(previous->val, current->next->val) ||
30
                                  current->val > max(previous->val, current->next->val);
31
32
               if (isCritical) {
33
                   if (lastCriticalPosition == 0) {
34
                       // This is the first critical point encountered
35
                       firstCriticalPosition = index;
36
                   } else {
37
                       // Update the shortest distance between any two critical points
                       answer[0] = min(answer[0], index - lastCriticalPosition);
38
                       // Update the distance between the first and the current critical point
39
                       answer[1] = index - firstCriticalPosition;
40
41
42
                   // Update the position of the last critical point
43
                   lastCriticalPosition = index;
44
               ++index; // Move to the next index
45
46
               previous = current; // Move the previous pointer forward
               current = current->next; // Move the current pointer forward
47
48
49
           // If no critical points or only one critical point was found, return \{-1, -1\}
           if (firstCriticalPosition == lastCriticalPosition) {
50
               return {-1, -1};
51
52
53
           // Return the distances between critical points
54
           return answer;
55
56 };
57
Typescript Solution
  1 /**
      * Definition for singly-linked list.
      */
     class ListNode {
         val: number;
         next: ListNode | null;
         constructor(val?: number, next?: ListNode | null) {
             this.val = (val === undefined ? 0 : val);
  8
             this.next = (next === undefined ? null : next);
  9
 10
 11 }
```

* Finds the minimum and maximum distance between any two critical points in a linked list.

* @return {number[]} - An array with two elements: the minimum and maximum distances between critical points.

* Critical points are defined as local minima or local maxima.

let index = 1; // Stores the current node index.

head = head.next; // Move head to the next node.

// Traverse the list to find all critical points.

while (head && head.next !== null) {

let currentValue = head.val;

let postValue = head.next.val;

previousValue = currentValue;

criticalPoints.push(index);

// Determine the number of critical points.

if (numCriticalPoints < 2) return [-1, -1];</pre>

let numCriticalPoints = criticalPoints.length;

for (let i = 1; i < numCriticalPoints; i++) {</pre>

* If fewer than two critical points are found, returns [-1, -1].

* @param {ListNode | null} head - The head node of the linked list.

function nodesBetweenCriticalPoints(head: ListNode | null): number[] {

let previousValue = head.val; // Stores the value of the previous node.

let criticalPoints = []; // Stores the indices of the critical points.

// Check if the current node is a local maximum or minimum.

// If there are less than two critical points, return [-1, -1].

if ((previousValue < currentValue && currentValue > postValue) ||

(previousValue > currentValue && currentValue < postValue)) {</pre>

let minDistance = Infinity; // Initialize the minimum distance to infinity.

let maxDistance = criticalPoints[numCriticalPoints - 1] - criticalPoints[0];

minDistance = Math.min(criticalPoints[i] - criticalPoints[i - 1], minDistance);

// Calculate the maximum distance, which is distance between the first and the last critical points.

// Calculate the minimum distance between consecutive critical points.

53 54 // Return the minimum and maximum distances as an array. 55 return [minDistance, maxDistance]; 56 } 57

Time and Space Complexity

index++;

head = head.next;

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 0(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.