817. Linked List Components

Linked List Medium <u>Array</u> Hash Table

Problem Description

'connected components' are present in the array nums. A connected component here is defined as a sequence of numbers that appear consecutively in the linked list. Hence, if two numbers are adjacent in the linked list and both are present in nums, they form part of the same connected component. The question asks for the total count of such connected components. For example, consider the <u>linked list</u> 1->2->3->4 and the array nums = [2, 4]. Here, 2 and 4 each form their own connected

This problem presents a scenario where you're given two inputs: the head of a singly linked list, which contains unique integer

values, and an integer array nums, which contains a subset of the values found in the linked list. The task is to find out how many

component because they are not consecutive in the linked list. The answer would be 2. Intuition

The key to solving this problem lies in understanding the structure of a linked list and the definition of connected components in

occurrences of the values in nums. Here's a step-by-step intuition behind the solution: 1. We want to count the number of connected components, so we track that with an integer variable, which we can call ans.

the existence of each <u>linked list</u> node's value in nums.

indicating we've found a new connected component.

2. We create a set s from nums. Using a set is efficient for checking if an element exists within it. This is important because we'll need to check for

the context of this problem. We find connected components by traversing the linked list and checking for consecutive

- 3. We begin to traverse the linked list starting from the given head. For each node, we first check if we are currently looking at a value that is not in our set s (thus not in nums). If it is not, we simply move to the next node. 4. When we find a node with a value that is in the set s, this indicates the potential start of a connected component. We increment our ans count,
- 5. We continue traversing the linked list from this node onward, as long as the subsequent nodes' values are in the set s—which means we're still within the same connected component. 6. As soon as we find a node with a value not in s, or we reach the end of the list, we start looking for a new connected component (repeating the
- process from step 3). The result at the end, after we've traversed the entire <u>linked list</u>, is the count of connected components in nums.
- **Solution Approach**

The solution to this problem employs a simple linear traversal algorithm, which makes use of the singly linked list and set data

structures. The aim is to efficiently determine whether each node's value in the linked list is part of a connected component as defined by the array nums. Here is a breakdown of how the given Python solution achieves this:

An auxiliary data structure, a set (referred to as s in the reference code), is created from the list of integers nums. Sets provide an average time complexity of O(1) for look-up operations, which is crucial for determining whether a node's value is part of

nums.

A variable ans is initialized to 0. This variable keeps track of the number of connected components in nums. The solution then enters a while loop, which continues to iterate as long as there is a node (head) to process in the linked list.

Inside the top-level while loop, a nested while loop skips over any nodes whose values are not in the set s. This is because such nodes are not part of any connected component we're interested in.

The code increments ans by 1 if and only if it finds a node that contains a value in s, indicating the beginning of a potential

connected component. This increment happens before entering the next nested loop, ensuring that we count each connected component once. The condition head is not None ensures we don't count an extra component when we reach the end of the list.

Another nested while loop continues to traverse the linked list as long as consecutive nodes have values in the set s,

effectively walking through the nodes of a single connected component. Once the innermost while loop ends (because it encounters a node not in s or reaches the end of the list), control returns to the top-level while loop to seek the next connected component.

Through this method, the solution correctly identifies each connected component in the subset nums by traversing the linked list

once. This linear pass through the list results in a solution with O(n) time complexity, where n is the number of nodes in the linked

list. The space complexity of the solution is also O(n), which is due to the creation of the set from nums. **Example Walkthrough**

We're interested in finding how many connected components we can find in nums that correspond to consecutive elements in the linked list.

a. First node is 1, which is in s. This could be the start of a connected component, so we increment ans to 1. b. Move to the

a. The next node is 3, which is in s. This marks the start of another connected component. Increment ans to 2. b. Move to the

next node, which is 4 and also in s. However, since 4 is directly after 3 and both are in s, they belong to the same connected

Initialize the counter to zero: ans = 0.

Linked List: 1 -> 2 -> 3 -> 4

Array `nums`: [3, 4, 1]

next node, which is 2. It's not in s, so the potential connected component is complete.

component. No increment to ans. c. There are no more nodes to process.

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

Skip nodes until a node with a value in nums_set is found

Skip all subsequent nodes that are also in nums_set

Initialize count of connected components to 0

while head and head.val not in nums_set:

while head and head.val in nums_set:

// Definition for singly-linked list provided for context.

head = head->next;

++component_count;

head = head->next;

// Global definition for singly-linked list node.

// Move past the current component.

while (head && values_set.count(head->val)) {

return component_count; // Return the number of components found.

if (head) {

};

TypeScript

interface ListNode {

next: ListNode | null;

} else {

return componentCount;

Definition for singly-linked list.

isInComponent = false;

currentNode = currentNode.next;

// Return the total components count.

// Move to the next node in the list.

val: number;

class ListNode {

int val;

ListNode next;

Let's consider a linked list with the following values and the array nums = [3, 4, 1]:

Create a set s from the array nums, which gives us $s = \{1, 3, 4\}$.

Thus, we have traversed the linked list and counted a total of 2 connected components that are found in nums.

Definition for singly-linked list.

self.val = val

self.next = next

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

Traverse through the linked list

count += 1

head = head.next

class ListNode:

class Solution:

Start traversing the linked list:

Continue traversing:

Solution Implementation **Python**

count = 0 # Convert the nums list to a set for constant-time lookups nums_set = set(nums)

```
head = head.next
# If a node with a value in nums_set is found, increment count
if head:
```

while head:

```
# Return the total number of connected components
        return count
# Additional explanations about the code structure:
# 1. The first while loop progresses through nodes that are not part of any component until a start of a component is found.
# 2. Once a start of a component is detected, the connected component is counted with `count += 1`.
# 3. The second while loop continues to traverse through the linked list but only through the nodes
     that are part of the current component, until it reaches a node that is not part of nums_set.
# 4. The process continues until all nodes in the linked list are visited.
Java
class Solution {
    // Counts the number of connected components in the list that are present in the array 'nums'.
    public int numComponents(ListNode head, int[] nums) {
        int count = 0; // Counter for the number of components
        Set<Integer> set = new HashSet<>(); // HashSet to store elements of 'nums' for constant time access
       // Add all elements of the array 'nums' to the HashSet
        for (int value : nums) {
            set.add(value);
        // Traverse the linked list to find connected components
       while (head != null) {
            // Skip nodes until we find one that is contained in 'nums'
           while (head != null && !set.contains(head.val)) {
                head = head.next;
           // If a node is found in set, increment the component count once
            if (head != null) {
                count++;
                // Move past the current component
                while (head != null && set.contains(head.val)) {
                    head = head.next;
        return count; // Return the total number of components found
```

```
ListNode() {}
    ListNode(int val) { this.val = val; }
    ListNode(int val, ListNode next) { this.val = val; this.next = next; }
C++
#include <unordered_set>
#include <vector>
// 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:
    // Function to count the number of connected components in the linked list
    // that appear in 'nums' array as consecutive nodes.
    int numComponents(ListNode* head, std::vector<int>& nums) {
       // Convert the vector 'nums' into an unordered set for constant-time lookups.
       std::unordered_set<int> values_set(nums.begin(), nums.end());
       int component_count = 0; // Initialized the count of components to 0.
       while (head) {
           // Skip all nodes whose values do not appear in 'nums'.
           while (head && values_set.count(head->val) == 0) {
```

// If 'head' is not nullptr, we have encountered a component, so increment the count.

```
/**
* Counts the number of connected components in the linked list
* that are present in the given 'nums' array.
* @param head The head node of the linked list.
* @param nums An array of numbers representing the target values.
* @return The number of connected components that are subsets of 'nums'.
function numComponents(head: ListNode | null, nums: number[]): number {
   // Initialize a Set with the elements from 'nums' to facilitate constant-time checks.
   const numSet = new Set<number>(nums);
   // Variable to hold the count of connected components found.
   let componentCount = 0;
   // Iterator for traversing the linked list nodes.
   let currentNode = head;
   // Flag to keep track of whether we're inside a component.
    let isInComponent = false;
   // Traverse the linked list.
   while (currentNode !== null) {
       // If the current node's value is in 'numSet', we might be in a component.
       if (numSet.has(currentNode.val)) {
           // If 'isInComponent' is false, we've found the start of a new component.
            if (!isInComponent) {
```

// If the current node's value is not in 'numSet', we're not in a component.

isInComponent = true; // We're now inside a component.

componentCount++; // Increment our component count.

```
class ListNode:
   def __init__(self, val=0, next=None):
       self.val = val
       self.next = next
class Solution:
   def numComponents(self, head: Optional[ListNode], nums: List[int]) -> int:
       # Initialize count of connected components to 0
        count = 0
       # Convert the nums list to a set for constant-time lookups
       nums_set = set(nums)
       # Traverse through the linked list
       while head:
           # Skip nodes until a node with a value in nums_set is found
           while head and head.val not in nums_set:
                head = head.next
            # If a node with a value in nums_set is found, increment count
            if head:
               count += 1
           # Skip all subsequent nodes that are also in nums_set
           while head and head.val in nums_set:
                head = head.next
       # Return the total number of connected components
        return count
# Additional explanations about the code structure:
# 1. The first while loop progresses through nodes that are not part of any component until a start of a component is found.
# 2. Once a start of a component is detected, the connected component is counted with `count += 1`.
```

3. The second while loop continues to traverse through the linked list but only through the nodes

4. The process continues until all nodes in the linked list are visited.

that are part of the current component, until it reaches a node that is not part of nums_set.

Time Complexity: The time complexity of the code is analyzed by examining the operations performed at each step:

is reached.

Time and Space Complexity

1. There is a while loop that continues as long as there are nodes in the linked list. 2. Inside this loop, there are two nested while loops: • The first inner while loop iterates until a node's value that exists in s (set containing all values from nums) is found, or until the end of the list

move through that component. Therefore, if there are n nodes in the linked list, the algorithm will take O(n) time. **Space Complexity:**

• The second inner while loop continues as long as the nodes' values belong to s.

The space complexity is determined by the additional space used by the algorithm: A set s is created to store the values from nums. If there are m numbers in nums, the space complexity for the set will be 0(m).

No other data structures are used that grow with the size of the input. Thus, the overall space complexity is 0(m).

Each node of the linked list is visited at most twice: once to check for traversal to a component that is part of s, and once to

The provided Python code aims to determine the number of connected components in a linked list where every node's value

exists in the given nums list. To analyze the time complexity and space complexity of the code, let's break it down:

Hence, the time complexity is O(n) and the space complexity is O(m).