128. Longest Consecutive Sequence

Medium Union Find Array Hash Table

Problem Description

The problem asks us to find the length of the longest sequence of consecutive numbers in an unsorted array called `nums`. A sequence of numbers is considered consecutive if every number follows the previous one without any gaps. For example, `[1, 2, 3]` is a consecutive sequence, but `[1, 3, 4]` is not. Our task is to find the longest such sequence in the given array.

The tricky part of this problem is that we need to come up with a solution that has a time complexity of `O(n)`. This means we cannot afford the luxury of sorting the array as it would typically require `O(n * log n)` time. Thus, we must find a way to keep track of sequences efficiently, despite the order of elements being arbitrary.

Intuition

To solve this problem in `O(n)` time, we need to think of a data structure that allows us to quickly check if an element exists in the set and if we can extend a consecutive sequence. A hash table, or in Python a set, is an ideal candidate because it allows us to query the existence of an element in constant `O(1)` time.

Here's the intuition for the solution approach:

1. Convert the `nums` array into a set to eliminate duplicates and allow for `O(1)` existence checks. It takes `O(n)` time to convert the list to a set.
2. We iterate through each number `x` in the original array. For each `x`, we have two conditions:
   - If `x - 1` is not in the set, `x` could be the start of a new sequence.
   - If `x - 1` is in the set, `x` is part of a sequence that started before `x` and we don't need to check it as it will already be covered when we check the beginning of its sequence.
3. When we find a number `x` that is the start of a new sequence (because `x - 1` is not in the set), we then proceed to check how long this sequence is by continuously incrementing `y` (initialized as `x + 1`) as long as `y` is present in the set.
4. Each time we extend the sequence, we update the length of the current sequence and update the answer `ans` if the current sequence is longer than the previously recorded longest sequence.

This approach guarantees we only make a constant number of passes through the array and that we only consider each sequence from its beginning, ensuring our algorithm runs in `O(n)` time.

Solution Approach

The solution approach can be decomposed into key steps that align with the algorithmic design and utilize data structures such as hash tables effectively.

Step 1: Building a Set Firstly, we convert the given list `nums` into a set `s`. This process removes any duplicate elements and facilitates constant time checks for the presence of integers. This is critical as it allows for the linear time algorithm we're aiming for.

```
s = set(nums)
```

Step 2: Iteration and Sequence Detection We iterate through each number `x` in the list `nums`. For each number, we check if its predecessor (`x - 1`) is in the set.

```
for x in nums:
    if x - 1 not in s:
```

If `x - 1` is not in the set, it implies that `x` could potentially be the start of a new consecutive sequence.

Step 3: Extension of the Sequence When we find that `x` could be the start of a sequence, we try to find out where the sequence ends. We initialize a variable `y` as `x + 1` and while `y` is in the set, we keep incrementing `y` by one to extend the sequence.

```
y = x + 1
while y in s:
    y += 1
```

Step 4: Update Longest Sequence Length After we find a sequence starting with `x` and ending before `y`, the length of this sequence is `y - x`. If this length is greater than any previously found sequences, we update our answer `ans`.

```
ans = max(ans, y - x)
```

Step 5: Return the Result Once we've considered each number in the array, we return `ans` as the answer to the problem, which represents the length of the longest consecutive elements sequence found.

This approach takes advantage of the hash table pattern via the set `s`, which provides us with the constant time lookups needed to achieve an overall `O(n)` time complexity. Thus, we harness the capability of hash tables to manage our computations efficiently and satisfy the problem's constraints.

Example Walkthrough

Let's illustrate the solution approach using a small example. Consider the unsorted array `nums = [4, 1, 3, 2, 6]`. Our goal is to find the longest sequence of consecutive numbers in this array.

Step 1: Building a Set

We transform the `nums` array into a set:

```
s = set([4, 1, 3, 2, 6])  # s = {1, 2, 3, 4, 6}
```

Duplications are removed and we can check for existence in constant time.

Step 2: Iteration and Sequence Detection

We iterate through `nums`. Assume our iteration order is the same as the array's order.

Iteration 1: `x = 4`

- We check if `3` (`x - 1`) is in the set:

```
if 3 not in s:  # False, hence we skip
```

Since `3` is present, `4` is not the start of a new sequence.

Iteration 2: `x = 1`

- We check if `0` (`x - 1`) is in the set:

```
if 0 not in s:  # True, thus 1 might be a sequence start
```

Since `0` is not present, `1` is a start of a new sequence.

Step 3: Extension of the Sequence

We extend the sequence from `1` onwards to find its length:

```
y = 2
while y in s:
    y += 1  # y becomes 3, then 4, stops at 5
```

The sequence we found is `1, 2, 3, 4`.

Step 4: Update Longest Sequence Length

We calculate the length of the current sequence `4 - 1` which is `3`.

```
ans = max(ans, 4 - 1)  # if ans was 0, it becomes 3
```

We update `ans` to the length of this sequence if it is the longest found so far.

Iterations continue with `2`, `3`, and `6` but no other new sequence is found with a length greater than `3`.

Step 5: Return the Result

After iterating through all numbers, the longest sequence found is from `1` to `4`, which has a length of `4`. Thus, `ans = 4` and is returned as the solution.

Solution Implementation

Python

```
class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        # Create a set from the list for O(1) lookups
        num_set = set(nums)
        longest_streak = 0

        # Iterate over each number in the list
        for num in nums:
            # Check if it's the start of a sequence
            if num - 1 not in num_set:
                # Initialize the current number as the possible start of a sequence
                current_num = num
                current_streak = 1

                # Increment the current num to find the length of the streak
                while current_num + 1 in num_set:
                    current_num += 1
                    current_streak += 1

                # Update the longest streak with the maximum streak found
                longest_streak = max(longest_streak, current_streak)

        # Return the length of the longest consecutive sequence
        return longest_streak
```

Java

```
class Solution {
    public int longestConsecutive(int[] nums) {
        // Create a hash set to store the unique elements of the array.
        Set<Integer> numSet = new HashSet<>();
        for (int num : nums) {
            numSet.add(num);
        }

        // Initialize the variable for the longest consecutive sequence.
        int longestStreak = 0;

        // Go through each element in the array.
        for (int num : nums) {
            // Check if current number is the beginning of a sequence.
            if (!numSet.contains(num - 1)) {
                // Initialize the current number as the potential start of the sequence.
                int currentNum = num;
                // Initialize the current streak length.
                int currentStreak = 1;

                // Expand the current streak if consecutive numbers are found.
                while (numSet.contains(currentNum + 1)) {
                    currentNum += 1;
                    currentStreak += 1;
                }

                // Update the longest streak found so far.
                longestStreak = Math.max(longestStreak, currentStreak);
            }
        }

        // Return the longest streak length.
        return longestStreak;
    }
}
```

C++

```
#include <vector>
#include <unordered_set>
#include <algorithm>

class Solution {
public:
    int longestConsecutive(std::vector<int>& nums) {
        // Create an unordered set to hold unique elements for constant-time lookups.
        std::unordered_set<int> numbersSet(nums.begin(), nums.end());
        int longestStreak = 0; // Variable to store the length of the longest consecutive sequence found.

        // Iterate over each element in the vector.
        for (int num : nums) {
            // Check if the current number is the beginning of a sequence by looking for num - 1.
            if (!numbersSet.count(num - 1)) {
                // If num is the start of a sequence, look for all consecutive numbers starting with num + 1.
                int currentNum = num + 1;

                // Continue checking for the next consecutive number in the sequence.
                while (numbersSet.count(currentNum)) {
                    currentNum++;
                }

                // Update the longest streak with the length of the current sequence.
                longestStreak = max(longestStreak, currentNum - num);
            }
        }

        // Return the longest length of consecutive sequence found.
        return longestStreak;
    }
};
```

TypeScript

```
// Function to find the length of the longest consecutive elements sequence.
function longestConsecutive(nums: number[]): number {
    // Initializing a set to store unique numbers from the input.
    const numSet: Set<number> = new Set(nums);
    let longestStreak = 0; // Stores the length of the longest consecutive sequence.

    // Iterate over each number in the set.
    for (const num of numSet) {
        // Check if current number is the beginning of a sequence.
        if (!numSet.has(num - 1)) {
            let currentNum = num;       // Starting number of the current sequence.
            let currentStreak = 1;      // Initializing current streak length.

            // Incrementally check consecutive numbers.
            while (numSet.has(currentNum + 1)) {
                currentNum++;
                currentStreak++;
            }

            // Update the longest streak if current one is longer.
            longestStreak = Math.max(longestStreak, currentStreak);
        }
    }
    // Return the length of the longest consecutive sequence.
    return longestStreak;
}
```

```
class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        # Create a set from the list for O(1) lookups
        num_set = set(nums)
        longest_streak = 0

        # Iterate over each number in the list
        for num in nums:
            # Check if it's the start of a sequence
            if number - 1 not in num_set:
                # Initialize the current number as the possible start of a sequence
                current_num = number
                current_streak = 1

                # Increment the current num to find the length of the streak
                while current_num + 1 in num_set:
                    current_num += 1
                    current_streak += 1

                # Update the longest streak with the maximum streak found
                longest_streak = max(longest_streak, current_streak)

        # Return the length of the longest consecutive sequence
        return longest_streak
```

Time and Space Complexity

The given code is designed to find the length of the longest consecutive elements sequence in an unsorted array. It utilizes a set to achieve an average time complexity of `O(n)`.

Time Complexity:

The algorithm has two main parts:

1. Creating a set from the list of numbers, which takes `O(n)` time.
2. Looping through each number in the array and extending the consecutive sequence if the current number is the start of a sequence. This part is also `O(n)` on average because each number is visited only once during the sequence extension process.

Combining these two parts still results in a total of `O(n)` time complexity since other operations inside the loop are constant time on average, such as checking for membership in the set and updating the `ans` variable.

Space Complexity:

The space complexity is `O(n)` because a set is created to store the elements of the array, and no other data structures that depend on the size of the input are used.