Longest Consecutive Sequence

Given an array of integers, find the length of the longest consecutive sequence. You can rearrange the integers in any order. For example, in the array $[1, 2, 3, 4, 100, 101, 102]$, the longest consecutive sequence is $[1, 2, 3, 4]$, with a length of 4.

Brute Force Solution

The brute force approach involves picking an element x and then searching for $x + 1$, $x + 2$, and so on, to determine the longest sequence starting from x.

1. Initialize the `longest` sequence length to 1.
2. Iterate through the array. For each element:
    - Set the `counter` to 1 and the current element to x.
    - While searching linearly in the array finds $x + 1$:
        - Increment x to $x + 1$.
        - Increment the `counter`.
    - Update `longest` with the maximum of `longest` and the `counter`.

```
longest = 1
for i = 0 to n:
    x = array[i]
    counter = 1
    while linear_search(array, x + 1) is found:
        x = x + 1
        counter = counter + 1
    longest = max(longest, counter)
```

- Time Complexity: $O(n^2)$ due to the nested loop (outer loop for each element and inner loop for linear search).
- Space Complexity: $O(1)$ as no extra space is used.

Better Solution

To optimize, we can sort the array first. Sorting brings consecutive elements together, making it easier to identify sequences.

1. Sort the array.
2. Initialize `longest` to 1, `last_smaller` to the minimum possible integer value, and `count_current` to 0.
3. Iterate through the sorted array.
    - If the current element requires a preceding number to be part of the sequence, check the `last_smaller`. If `last_smaller` indicates that a number can't be part of the sequence, it becomes a new sequence with a length of 1.
    - If the current element is the same as `last_smaller`, it's a duplicate, and we don't include it in the sequence.
    - If the current element is consecutive (i.e., it's one greater than `last_smaller`), it can be part of the sequence. Increment `count_current`, update `longest` if `count_current` is greater, and update `last_smaller` to the current element.
    - If the current element is not consecutive, it can't be part of the sequence, so it starts a new sequence with a length of 1.

Imagine the last smaller represents that consecutive sequences will have 1 number larger than itself.

Longest Consecutive Sequence Algorithm

Let's explore an algorithm to find the longest consecutive sequence in an array.

Initial Approach

1. Sort the array.
2. Initialize `longest = 1`, `count = 0`, and `lastSmaller = Integer.MIN_VALUE`.
3. Iterate through the array from `i = 0` to `n`.
    - If `arr[i]` is part of the sequence:
        - If `arr[i] == lastSmaller + 1`, increment `count` and update `lastSmaller = arr[i]`.
    - If `arr[i] == lastSmaller`, do nothing.
    - Otherwise, start a new sequence with `lastSmaller = arr[i]`.
    - Update `longest = max(longest, count)`.

The time complexity for this approach is $n \log n$ due to sorting, plus $O(n)$ for iteration. However, this approach distorts the original array, which might not be desirable.

Optimal Solution Using Sets

To avoid distorting the array and potentially improve time complexity, we can use a set data structure.

1. Insert all elements into an unordered set.
2. Iterate through the set.
3. For each element, check if its predecessor exists in the set.
    - If the predecessor does not exist, it's potentially the start of a sequence.
    - Look for consecutive elements (i.e., $x + 1$, $x + 2$, etc.) in the set to determine the sequence length.
    - Update the longest sequence found so far.

Here's a breakdown:

Time Complexity: The time complexity hinges on the nature of set operations and the iterations. Inserting elements into the set takes $O(n)$. The outer loop iterates through each element of the set, which in the worst case is $O(n)$. Inside the loop, we check for consecutive elements using `set.find()`, which takes $O(1)$ on average for unordered sets, but could degrade to $O(n)$ in the worst-case scenario of hash collisions.

Optimization Hack

To avoid redundant checks, only start looking for a sequence from a number if that number doesn't have a predecessor in the set. For instance, in the sequence `100, 101, 102`, start looking from `100` because `99` is not present. This ensures we only start from the beginning of a sequence.

Code Example (C++)

```cpp
int longestConsecutive(vector<int>& nums) {
    if (nums.size() == 0) return 0;
    int longest = 1;
    unordered_set<int> s;
    for (int num : nums) {
        s.insert(num);
    }
    for (int x : s) {
        if (s.find(x - 1) == s.end()) {
            int current = 1;
            int next = x + 1;
            while (s.find(next) != s.end()) {
                current++;
                next++;
            }
            longest = max(longest, current);
        }
    }
    return longest;
}
```

Explanation of the C++ Code Snippet

- If the input vector is empty, the function immediately returns 0, as there can be no consecutive sequence.
- It initializes `longest` to 1 because if the vector is not empty, the minimum possible length of a consecutive sequence is 1.
- The code inserts all elements from the input vector `nums` into an unordered set `s`. This ensures uniqueness and allows for $O(1)$ average time complexity for element lookup.
- The code iterates through each element `x` in the set `s`.
- It checks if `x - 1` exists in the set. If `x - 1` is not in the set, `x` is considered the potential start of a consecutive sequence.
- If `x` is the potential start, the code enters a loop to find how many consecutive elements exist in the set starting from `x`.
- The variable `current` keeps track of the length of the current consecutive sequence.
- The variable `next` is used to look for the next consecutive element in the set.
- The `while` loop continues as long as `next` exists in the set.
- Inside the `while` loop, `current` is incremented, and `next` is incremented to look for the next consecutive element.
- After the `while` loop finishes (when there are no more consecutive elements), the code updates `longest` with the maximum value between `longest` and `current`.
- After iterating through all elements in the set, the function returns the final value of `longest`, which represents the length of the longest consecutive sequence found in the input vector.

Key Data Structures

Data Structure	Purpose
Unordered Set	Stores unique elements and allows for efficient lookups (average case of $O(1)$).
Vector (Initial)	Stores the initial array of numbers.

Summary

By using an unordered set and checking for the absence of predecessors, we can efficiently find the longest consecutive sequence in an array without modifying the original array. This approach balances time complexity and space usage, making it an optimal solution.

Time Complexity Analysis

Let's analyze the time complexity of the given algorithm, considering the use of an **unordered set**.

Average Case Scenario

If we assume the unordered set takes $O(1)$ (big O of 1) on average for insertion and lookup:

1. Iterating and putting all elements into the set: $O(n)$
2. Iterating through the set: $O(n)$
3. The `while` loop might seem like another n, but it's not.

Understanding the While Loop's Iterations

Consider the example sequence `100, 1, 200, 4, 3, 2`.

- Initial iterations through the set: 7 iterations.
- From 100, we check 101, then 102 (3 iterations).
- From 1, we check 2, 3, 4 (4 iterations).

Total iterations: 7 + 3 + 4 = 14, which is approximately $2n$. This is because we only start checking from the "first" number in a sequence and don't revisit intermediate numbers.

Therefore, the overall time complexity is $O(n + 2n) = O(3n)$, which simplifies to $O(n)$.

Worst Case Scenario

If collisions occur in the unordered set, it might take $O(log(n))$ (big O of log n) for set operations in the worst case. If that's the case, then brute force would be better.

Space Complexity

The space complexity is $O(n)$ due to storing elements in the unordered set. This assumes all elements are unique.

Summary of Complexities

Here's a summary table:

Operation	Time Complexity (Average)	Space Complexity
Building the set	$O(n)$	$O(n)$
Finding Longest Sequence	$O(n)$	
Total	$O(n)$	$O(n)$