Longest Consecutive Sequence

Created	@2025年4月13日 上午2:43
Question Type	Array
Difficulty	Medium
LeetCode Question Link	https://leetcode.com/problems/longest-consecutive-sequence/description/

1. Question Self-Understanding:

1.1 Description

The goal is to find the longest sequence of consecutive elements in an array. The order of the elements in the input does not matter; for example, even if 3 appears before 2, the sequence [2, 3] is still considered consecutive. The description indicates that the key task in the algorithm is to identify the next consecutive number relative to the current element.

1.2 Input

The input is a list of integers.

1.3 Input Assumptions

N/A

1.4 Output

The output is an integer representing the length of the longest consecutive sequence.

1.5 Example:

```
Input: nums = [1,0,1,2]
Output: 3
```

1.6 Other Q&A

- **Question:** What is the definition of "consecutive"? Can numbers be repeated?
    - **Answer:** Based on the example, consecutive numbers must be strictly increasing. Duplicates are not considered consecutive (for instance, the sequence [0, 1, 1, 2] is not valid, and the expected result is 3).

2. Attempt 1

2.1 Thought

- To find the consecutive sequence, we first sort the array in ascending order. Then, we iterate through the sorted array to determine how far each index can extend a consecutive sequence. Finally, we return the length of the longest sequence encountered.

2.2 Pseudo-Code: (Ignore this part. This is a draft of code for thinking purposes.)

```
class Solution:
    def longestConsecutive(self, nums: List[int]) → int:
        nums ← order(nums)
        final_result = 0
        for-loop the number to see if the next number is not consecutive of the curr
                if it is yes:
                    update the length to be the max(temp + 1, final_result)
                else:
                    reset the longest length and continuous
                    contiunous
```

2.3 Implementation through python:

```
class Solution:
    def longestConsecutive(self, nums: List[int]) → int:
        if not nums:
            return 0
        # in-place sorting to avoid extra space usage
        nums.sort()

        longest_streak = 0
        current_streak = 1

        curret_index = 0
        while curret_index < len(nums) - 1:
            if nums[curret_index] == nums[curret_index + 1]:
                curret_index += 1
                continue

            if nums[curret_index] + 1 == nums[curret_index + 1]:
                current_streak += 1
            else:
                longest_streak = max(longest_streak, current_streak)
                current_streak = 1
            curret_index += 1
        longest_streak = max(longest_streak, current_streak)
        return longest_streak
```

2.4 Time and Space Complexity

2.4.1 Time Complexity

- Sorting the array takes O(n log n) time (using Timsort in Python), and iterating through the array takes O(n) time. Therefore, the overall time complexity is O(n log n).

2.4.2 Space Complexity

- Sorting in-place minimizes extra space usage, resulting in an O(1) space complexity. However, if modifying the input is not allowed and a copy of the array must be created before sorting, the space complexity becomes O(n).

3. Attempt 2

3.1 Thought

- From the previous attempt, we see that the algorithm mainly wastes time on sorting. Is it possible to avoid sorting entirely? We realize we need to check whether the next number in the list is consecutive. The key action is determining if the next number exists. Since we don’t care about the order, we can leverage a set to optimize our approach.
- Let’s look at an example:

  `[1, 2, 3, 5, 6, 100]`

  This is the result after sorting.

If we start at `1` in the unsorted array, we just need to check how far the consecutive sequence goes, which in this example would be up to `3`. Then, when moving to the next value, if there is a previous number, it might already belong to a sequence (like `2`), so we wouldn't need to start a new sequence from there. However, this approach could still result in an \(O(n^2)\) worst-case scenario.

- Another idea is: if we find the end of a sequence, we can remember that number and start from it. This concept is part of the third approach. Let's proceed step by step and focus on Approach 2 first.

3.2 Pseudo-Code: (Ignore this part. It's a draft for brainstorming.)

```
class Solution:
    def longestConsecutive(self, nums: List[int]) → int:
        # turn the nums into a set
        # for-loop through nums to see if the current value has no previous numb
er in theset
            # if yes:
                # find how far it can go
            # if no:
                # continue
```

3.3 Implementation through python:

```
class Solution:
    def longestConsecutive(self, nums: List[int]) → int:
        if not nums:
            return 0

        nums_set = set(nums)
        longest_streak = 0
```

```
for num in nums:
    # check if num is the start of a sequence
    if num - 1 not in nums_set:
        current_num = num
        current_streak = 1

        # continue while the next consecutive number exists
        while current_num + 1 in nums_set:
            current_num += 1
            current_streak += 1

        longest_streak = max(longest_streak, current_streak)

return longest_streak
```

3.4 Time and Space Complexity

3.4.1 Time Complexity

In the worst-case scenario, this approach can run in $O(n^2)$. However, if the proportion of the longest consecutive sequence is small relative to the total length of `nums`, it can approach $O(n)$. Also, having many duplicate numbers may reduce the effectiveness of this method.

⚠️ Actually, the previous reasoning needs some clarification. In reality:

- The **outer loop** iterates through n elements.
- The **inner loop** is only triggered when a sequence's starting element is found. In the worst-case scenario—when all numbers form one long consecutive sequence—the inner loop will run for O(n) time for that starting element. For each of the remaining n-1 elements, only an O(1) check is performed to verify if they have a predecessor.

Thus, the overall time complexity is:

$$O(n) + (n-1) \times O(1) = O(2n)$$

Since constant factors are ignored in Big-O notation, this simplifies to:

$$O(n)$$

3.4.2 Space Complexity

We create a set that has the same size as `nums`, so the extra space used is $O(n)$.

4. Attempt 3

4.1 Thought

Recall what we discussed in Attempt 2:

- There is another idea: if we find the terminator of the sequence, why don't we remember that number and start with it? This will be the third approach. Let's work through it step by step, starting with Approach 2.

- However, we realize that this approach won’t work because you don’t know when the next number will appear unless the array is sorted. Nonetheless, we can address the issue of duplicate numbers potentially reducing the number of checks required with a small modification. Furthermore, what happens if the longest streak is already greater than half the length of the array? In that case, we could ignore the rest.

4.2 Pseudo-Code: (Ignore this part. This is a draft of code for thinking purposes.)

```
class Solution:
    def longestConsecutive(self, nums: List[int]) → int:
        # Convert nums into an array
        # Iterate over the set:
            # If the current number does not have a previous number:
                # Determine how far the sequence can go and store the result
                # Check if the streak is already greater than the remaining portion
```

4.3 Implementation through python:

```
class Solution:
    def longestConsecutive(self, nums: List[int]) → int:

        if not nums:
            return 0
        # Convert nums into a set
        nums_set = set(nums)

        longest_streak = 0

        # Iterate over nums_set
        for num in nums_set:
```

```
        # Check if the current number is the start of a sequence
        if num - 1 not in nums_set:
            current_num = num
            current_streak = 1

            # Use a while loop to find the longest streak
            while current_num + 1 in nums_set:
                current_num += 1
                current_streak += 1

            longest_streak = max(longest_streak, current_streak)

    return longest_streak
```

4.4 Time and Space Complexity

4.4.1 Time Complexity

- ~~In the worst-case scenario, this approach can run in $O(n^2)$. However, if the proportion of the longest consecutive sequence is small relative to the total length of `nums`, it can approach $O(n)$. Also, having many duplicate numbers may reduce the effectiveness of this method.~~
- This approach will be faster by ignoring duplicates and by early returning when possible. However, the worst-case scenario remains.

4.4.2 Space Complexity

- We create a set that has the same size as `nums`, so the extra space used is $O(n)$.