594. Longest Harmonious Subsequence

Easy Array Hash Table Sorting Leetcode Link

Problem Description

This problem asks for the length of the longest harmonious subsequence within an integer array `nums`. A harmonious array is defined as one where the difference between the maximum and the minimum values is exactly 1. It's important to note that a subsequence is different from a subset, as a subsequence maintains the original order of elements, and can be formed by removing zero or more elements from the array.

For example, given an array `nums` = [1,3,2,2,5,2,3,7], the longest harmonious subsequence is [3,2,2,2,3] with a length of 5.

Intuition

The solution to this problem leverages hashing to keep track of the frequency of each number in the given array. The essence of the approach is to check for each element `num`, whether there exists an element `num + 1` in the array. If it exists, then a harmonious subsequence can potentially be formed using the elements `num` and `num + 1`.

Since the elements need not be adjacent, we only care about the counts of the elements `num` and `num + 1`. We can find the total count of such pairs and keep updating the maximum found so far if the current count exceeds the previous maximum. Using a hash map or, in Python, a `Counter` object is ideal for this task as it allows us to efficiently store and access the frequencies of each element.

Here's the intuitive breakdown of the process:

1. Create a frequency counter (hash map) from the array to count occurrences of each element.
2. Iterate through each element `num` in the array.
3. Check if there's an element `num + 1` in the counter hashmap.
4. If so, calculate the sum of the count of `num` and `num + 1` since those two form a harmonious sequence.
5. Compare this sum with the current longest harmonious subsequence length, and update it if this one is longer.
6. Continue this process until you have checked all elements.
7. Return the longest length obtained.

This approach is efficient because it eliminates the need to consider actual subsequences. Instead, it relies on counts, which simplifies the process of verifying the harmonious condition.

Solution Approach

The implementation of the solution follows these steps:

1. **Counter Data Structure**: We use Python's `Counter` class from the `collections` module, which is a subclass of a dictionary. It is used to count objects and store them as dictionary keys and their counts as dictionary value. Here it is used to build a hash map of each number (`num`) in `nums` array to its frequency.

2. **Iterate Through Each Element**: We iterate through each element in the `nums` array. For each element `num`, we are going to check if `num + 1` is a key in the counter.

3. **Check and Calculate**: If `num + 1` exists in our counter, we then know we can form a harmonious subsequence with `num` and `num + 1`, since our Counter contains the counts of all numbers and we only want the difference between numbers to be exactly `1`. To calculate the length of this potential subsequence we add the count of `num` and the count of `num + 1`.

4. **Maximize Answer**: Now, having the sum of counts of `num` and `num + 1`, we compare it with our current answer (initially zero). If our sum is greater, we update the answer to this larger sum. Essentially, we are keeping track of the largest harmonious subsequence we've seen so far.

5. **Return the Result**: Finally, after iterating over all the elements in the `nums` array, we end up with the length of the longest harmonious subsequence in the `ans` variable.

The algorithm uses `O(N)` space due to the counter, where `N` is the number of elements in the `nums` array. The time complexity is also `O(N)`, since we iterate over the array once to build the counter and once again to find the `ans`.

Here's the key part of the code explained:

```python
counter = Counter(nums)  # Step 1: Build the counter hash map
ans = 0
for num in nums:  # Step 2: Iterate through each element
    if num + 1 in counter:  # Step 3: Check if `num + 1` is present
        ans = max(ans, counter[num] + counter[num + 1])  # Step 4: Maximize the answer
return ans  # Step 5: Return the result
```

This code snippet illustrates how we use the algorithms and data structures mentioned to find the length of the longest harmonious subsequence.

Example Walkthrough

Let's consider a small example to illustrate the solution approach with an array `nums = [1, 1, 2, 2, 3]`.

1. First, we create a counter from the array. Our counter will look like this: `{1: 2, 2: 2, 3: 1}`, where each key is the element from `nums`, and each value is how many times that element appears.

2. We now start to iterate through each element in `nums`. We take the first element `1` and check if `1 + 1` (which is `2`) is a key in the counter. It is, so we find the sum of the counts of `1` and `2`, which is `2 + 2 = 4`. We compare this sum `4` with our current answer (which is `0` since we just started) and update the answer to `4`.

3. We move to the next element, which is also `1`. Since we've already considered this number and the counter hasn't changed, the calculation would be the same, thus no change in the answer.

4. Next, we consider number `2`. We check if `2 + 1` (which is `3`) is a key in the counter. It is, so we find the sum of the counts of `2` and `3`, which is `2 + 1 = 3`. We compare this sum `3` with our current answer `4`, and since `3` is less than `4`, we don't update the answer.

5. Then, we take the next `2`, and just like the previous `2`, it yields the same calculation, so no change occurs.

6. Finally, we consider the last element `3` and check for `3 + 1` (which is `4`). This is not a key in the counter, so we don't have a harmonious subsequence involving the number `3`.

7. Having examined all the elements, we end up with the answer `4`, which is the length of the longest harmonious subsequence `[1, 1, 2, 2]`.

This example validates our solution approach: using a counter to efficiently compute the length of the longest harmonious subsequence by only considering the frequencies of `num` and `num + 1` for each number in the array. The final answer for this example is `4`.

Python Solution

```python
from collections import Counter

class Solution:
    def findLHS(self, nums: List[int]) -> int:
        # Count the frequency of each number in the list using Counter
        frequency_map = Counter(nums)

        # Initialize the variable to store the length of the longest harmonious subsequence
        longest_harmonious_subseq_length = 0

        # Iterate through each number in the nums list
        for num in nums:
            # Check if the number has a companion number that is one greater
            if num + 1 in frequency_map:
                # Harmonious subsequence found with num and num + 1
                # Calculate its length: count of num + count of num + 1
                current_length = frequency_map[num] + frequency_map[num + 1]
                # Update the answer with the maximum length found so far
                longest_harmonious_subseq_length = max(longest_harmonious_subseq_length, current_length)

        # Return the length of the longest harmonious subsequence found
        return longest_harmonious_subseq_length

```

Java Solution

```java
class Solution {
    public int findLHS(int[] nums) {
        // Create a HashMap to keep track of the frequency of each number
        Map<Integer, Integer> frequencyMap = new HashMap<>();

        // Count the occurrences of each number in the array.
        for (int num : nums) {
            frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
        }

        // Initialize variable to keep track of the longest harmonious subsequence
        int longestHarmoniousSubsequence = 0;

        // Iterate through the numbers in the array
        for (int num : nums) {
            // Check if the number that is one more than the current number exists in the map
            if (frequencyMap.containsKey(num + 1)) {
                // If it exists, calculate the sum of the frequencies of the current number
                // and the number that is one more than the current number
                int currentLength = frequencyMap.get(num) + frequencyMap.get(num + 1);

                // Update the longest harmonious subsequence if the current sum is greater
                longestHarmoniousSubsequence = Math.max(longestHarmoniousSubsequence, currentLength);
            }
        }

        // Return the length of the longest harmonious subsequence found
        return longestHarmoniousSubsequence;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
#include <algorithm>

class Solution {
public:
    int findLHS(vector<int>& nums) {
        std::unordered_map<int, int> frequencyMap; // Map to keep track of the frequency of each number in nums

        // Count the frequency of each number in the given nums array
        for (int num : nums) {
            ++frequencyMap[num]; // Increment the count for the number
        }

        int longestHarmoniousSequence = 0; // Variable to hold the length of the longest harmonious sequence

        // Iterate through the numbers in the array to find the longest harmonious sequence
        for (auto& [num, count] : frequencyMap) {
            // Check if there is a number in the map which is exactly one more than the current number
            if (frequencyMap.count(num + 1)) {
                // If found, update the longestHarmoniousSequence with the larger value between the previous
                // longest and the total count of the current number and the number that is one more.
                longestHarmoniousSequence = std::max(longestHarmoniousSequence, count + frequencyMap[num + 1]);
            }
        }

        // Return the length of the longest harmonious sequence found
        return longestHarmoniousSequence;
    }
};

```

Typescript Solution

```typescript
// Importing necessary types from 'collections' module
import { HashMap } from "collectable";

// Declare a HashMap to keep track of the frequency of each number in nums
let frequencyMap: HashMap<number, number> = HashMap.empty();

// Function to find the length of the longest harmonious subsequence in the nums array
function findLHS(nums: number[]): number {
    // Count the frequency of each number in the given nums array
    nums.forEach(num => {
        frequencyMap = HashMap.update<number, number>(n => (n || 0) + 1, num, frequencyMap);
    });

    let longestHarmoniousSequence: number = 0; // Variable to hold the length of the longest harmonious sequence

    // Iterate through the numbers in the map to find the longest harmonious sequence
    HashMap.forEach((count, num) => {
        // Check if there is a number in the map which is exactly one more than the current number
        if (HashMap.has(num + 1, frequencyMap)) {
            // If found, update the longestHarmoniousSequence with the larger value between the previous
            // longest and the total count of the current number and the number that is one more.
            const nextCount = HashMap.get(num + 1, frequencyMap) || 0;
            longestHarmoniousSequence = Math.max(longestHarmoniousSequence, count + nextCount);
        }
    }, frequencyMap);

    // Return the length of the longest harmonious sequence found
    return longestHarmoniousSequence;
}

```

Time and Space Complexity

Time Complexity

The function involves calculating the frequency of each number in the list, which can be done in `O(n)` time where `n` is the number of elements in `nums`. The `for` loop iterates through each element in `nums` once, and each lookup and update operation within the loop can be considered to have an average-case time complexity of `O(1)` due to the hash table (dictionary) operations in Python. Therefore, the total time complexity of this function is `O(n)`.

Space Complexity

The space complexity comes from the use of a counter (essentially a hash table), which stores each unique number and its corresponding frequency. In the worst case, if all elements in `nums` are unique, the space required would be `O(n)`. Thus, the overall space complexity of the function is `O(n)`.