1207. Unique Number of Occurrences

Easy | Array | Hash Table

Leetcode Link

Problem Description

In this problem, we are given an array of integers, `arr`. Our task is to determine whether the array has the property that no two different numbers appear the same number of times. In other words, each integer's frequency (how often it occurs in the array) should be unique. If this property holds, we return `true`, otherwise, we return `false`.

For example, suppose the input array is `[1, 2, 2, 1, 1, 3]`. Here, the number 1 occurs three times, number 2 occurs twice, and number 3 occurs once. Since all these frequencies (3, 2, 1) are unique, our function would return `true`.

However, if we have an array like `[1, 2, 2, 3, 3, 3]`, where 1 occurs once, 2 occurs twice, and 3 also occurs three times, we see that the frequencies are not unique. In this case, our function would return `false`.

Intuition

To solve this problem, we can follow these steps:

1. Count the occurrences of each value in the array.
2. Check if the counts are all unique.

The intuition behind the solution lies in the frequency counting mechanic. We can use a data structure that allows us to count the occurrence of each element efficiently. The Python `Counter` class from the `collections` module is perfect for this job as it creates a dictionary with array elements as keys and their counts as values.

Once we have the counts, we need to check if they are unique. We can convert the counts into a set, which only contains unique elements. If the length of the set of counts (unique frequencies) is equal to the length of the original dictionary of counts, then all frequencies were unique, and we can return `true`. Otherwise, we return `false`.

Let's illustrate this with our [1, 2, 2, 1, 1, 3] example:

- Step 1: `Counter(arr)` would give us `{1: 3, 2: 2, 3: 1}` indicating that 1 appears thrice, 2 appears twice, and 3 appears once.
- Step 2: We check with `len(set(cnt.values())) == len(cnt)` which compares `len({3, 2, 1})` to `len({1: 3, 2: 2, 3: 1})`. Since both lengths are 3, they are equal, and the function returns `true`.

This simple yet efficient approach helps us to determine the uniqueness of each number's occurrences in the given array.

Solution Approach

The solution implementation harnesses the Python `collections.Counter` class to count the frequency of each element in the input array `arr`. The `Counter` class essentially implements a hash table (dictionary) under the hood, which allows efficient frequency counting of hashable objects. Here's a breakdown of the implementation steps according to the provided code:

Step by Step Implementation:

1. The `Counter(arr)` creates a dictionary-like object, where keys are the elements from `arr`, and the values are their respective counts. This step uses a hash table internally to store the counts, offering O(n) time complexity for counting all elements in the array, where `n` is the length of the array.

   For example, if `arr = [1, 2, 2, 3]`, `Counter(arr)` would produce `Counter({2: 2, 1: 1, 3: 1})`.

2. The expression `set(cnt.values())` takes all the values from the dictionary returned by `Counter(arr)`—which are the frequencies of elements—and converts them into a set. Since a set can only contain unique elements, this casting effectively filters out any duplicate counts.

   Continuing the example above, `set(cnt.values())` converts `[2, 1, 1]` to `{1, 2}`.

3. The comparison `len(set(cnt.values())) == len(cnt)` is checking whether the number of unique frequencies (length of the set) is equal to the number of distinct elements in `arr` (length of the Counter dictionary). An equality indicates that all frequencies are unique.

   In our continuing example, `len({1, 2}) == len(Counter({2: 2, 1: 1, 3: 1}))` reduces to `2 == 3`, which is `False`, reflecting that not all occurrences are unique since 1 and 3 both occur once.

This solution is both elegant and efficient, utilizing the properties of sets and hash tables to check for the uniqueness of the elements' frequencies in the array. There's no need for additional loops or explicit checks for duplicates; the data structures do the heavy lifting. The overall time complexity is O(n), dominated by the counting process. The space complexity is also O(n), as it's necessary to store the counts of elements which, in the worst case, can be as many as the number of elements in the array if all are unique.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have the input array `arr = [4, 5, 4, 6, 6, 6]`. We want to determine if no two different numbers in the array have the same frequency.

Following the steps outlined in the solution approach:

Step 1: Utilize the `Counter` class to count the occurrences of each element in `arr`.

We execute `cnt = Counter(arr)` and get `Counter({4: 2, 6: 3, 5: 1})`. This tells us that the number 4 appears twice, 6 appears three times, and 5 appears once.

Step 2: Convert the counts to a set to check uniqueness.

We convert the values, which are the frequencies, to a set using `set(cnt.values())` and obtain `{1, 2, 3}`. This set represents the unique frequencies of the numbers in our array.

Step 3: Compare the length of the set of counts to the length of the dictionary of counts to determine if the frequencies are all unique.

We perform the comparison `len(set(cnt.values())) == len(cnt)`. The length of our set `{1, 2, 3}` is `3`, and the length of our Counter dictionary `{4: 2, 6: 3, 5: 1}` is also `3`.

Since `3 == 3`, we can affirm that all counts are unique and therefore return `true` for our input array.

The array `[4, 5, 4, 6, 6, 6]` confirms our solution's criteria, meaning that no two different numbers in our array appear the same number of times. Hence, our illustrated function returns `true`.

Python Solution

```python
from collections import Counter
from typing import List

class Solution:
    def uniqueOccurrences(self, arr: List[int]) -> bool:
        # Create a Counter object that counts occurrences of each element in the array.
        element_count = Counter(arr)

        # Convert the values of the Counter (which represent the occurrences of each unique element) to a set.
        # This will remove any duplicate counts.
        unique_occurrences = set(element_count.values())

        # Check if the number of unique occurrences is equal to the number of unique elements.
        # If they are equal, it means that no two elements have the same number of occurrences.
        return len(unique_occurrences) == len(element_count)

```

Java Solution

```java
class Solution {
    // This method checks if all elements in the array have unique occurrence counts.
    public boolean uniqueOccurrences(int[] arr) {
        // Create a hashmap to store the counts of each number.
        Map<Integer, Integer> countMap = new HashMap<>();

        // Iterate over the array and populate the countMap.
        for (int number : arr) {
            // If the number is already in the map, increment its count, otherwise insert it with count 1.
            countMap.merge(number, 1, Integer::sum);
        }

        // Create a hashset containing all the values (occurrence counts) from the countMap.
        Set<Integer> occurrenceSet = new HashSet<>(countMap.values());

        // If the size of the set (unique occurrences) is the same as the size of the map (unique numbers),
        // it means that all occurrence counts are unique and we return true. Otherwise, return false.
        return occurrenceSet.size() == countMap.size();
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
#include <unordered_set>
using namespace std;

class Solution {
public:
    // Function to check if the array has a unique number of occurrences for each element
    bool uniqueOccurrences(vector<int>& arr) {
        // Map to store the frequency of each element
        unordered_map<int, int> frequencyMap;

        // Increment the frequency count for each element in arr
        for (int element : arr) {
            ++frequencyMap[element];
        }

        // Set to store unique occurrence counts
        unordered_set<int> occurrencesSet;

        // Iterate through the frequency map
        for (auto& keyValue : frequencyMap) {
            int occurrence = keyValue.second; // Get the occurrence/frequency count of the element

            // Check if the occurrence count is already in the occurrences set
            if (occurrencesSet.count(occurrence)) {
                // If already present, it's not unique and returns false
                return false;
            }
            // If not present, add the occurrence count to the set
            occurrencesSet.insert(occurrence);
        }

        // All occurrence counts were unique, return true
        return true;
    }
};

```

Typescript Solution

```typescript
// This function checks if all the elements in the array `arr` have unique occurrences
function uniqueOccurrences(arr: number[]): boolean {
    // Create a Map to count occurrences of each element
    const occurrenceCount: Map<number, number> = new Map();

    // Iterate over each element in the array
    for (const num of arr) {
        // If the element is already in the Map, increment its occurrence count
        // Otherwise, add the element with an occurrence count of 1
        occurrenceCount.set(num, (occurrenceCount.get(num) || 0) + 1);
    }

    // Create a Set from the values of the Map
    // This will automatically remove any duplicate occurrence counts
    const uniqueCounts: Set<number> = new Set(occurrenceCount.values());

    // Compare the size of the Set (unique counts) with the size of the Map (all counts)
    // If they are equal, all occurrence counts are unique
    return uniqueCounts.size === occurrenceCount.size;
}

```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the array `arr`. This is because counting the occurrences of each element in the array with `Counter(arr)` requires a single pass over all elements in `arr`, and then converting the counts into a set and comparing the sizes involves operations that are also `O(n)` in the worst case.

The space complexity of the code is also `O(n)`. The counter object `cnt` will store as many entries as there are unique elements in `arr`. In the worst case, where all elements are unique, the space required for the counter would be `O(n)`. Additionally, when the counts are converted to a set to ensure uniqueness, it occupies another space that could at most be `O(n)`, if all counts are unique.