2784. Check if Array is Good

Easy Array Hash Table Sorting [Leetcode Link]

Problem Description

The problem presents an integer array `nums` and defines a specific type of array called `base[n]`. The `base[n]` is an array of length `n + 1` that contains each integer from `1` to `n - 1` exactly once and includes the integer `n` exactly twice. For example, `base[3]` would be `[1, 2, 3, 3]`. The primary task is to determine if the given array `nums` is a permutation of any `base[n]` array.

A permutation in this context means any rearrangement of the elements. So, if `nums` is a rearrangement of all the numbers from `1` to `n - 1` and the number `n` appears exactly twice, the array is considered "good," and the function should return `true`, otherwise `false`. The problem simplifies to checking whether the array contains the correct count of each number to match the `base` definition.

Intuition

The intuition behind the solution involves counting the occurrences of each number in the array `nums`.

- Identify `n` as the length of the input array `nums` minus one, since a `base[n]` array has a length of `n + 1`. This identification is based on the definition mentioned in the problem description.
- Utilize a Counter (a collection type that conveniently counts occurrences of elements) to tally how often each number appears in `nums`.
- Subtract 2 from the count of `n` because `n` should appear exactly twice in a good `base` array.
- Then subtract 1 from the counts of all other numbers from `1` to `n - 1` because each of these numbers should appear exactly once in a good `base` array.
- Finally, check if all counts are zero using `all(v == 0 for v in cnt.values())`. If they are, it means that the input array has the exact count for each number as required for it to be a permutation of a `base[n]` array, and the function should return `true`. If even one count is not zero, it indicates that there's a discrepancy in the required number frequencies, and the function should return `false`.

Solution Approach

The implementation of the solution utilizes a standard Python library called `collections.Counter`, which is a subclass of dictionary specifically designed to count hashable objects.

Here's the step-by-step breakdown:

1. Compute `n` as the length of the array `nums` minus one. This is because we expect the array to be a permutation of a `base[n]`, which has length `n + 1`.

   ```
   n = len(nums) - 1
   ```

2. Initialize a Counter with `nums` which automatically counts the frequency of each element in the array.

   ```
   cnt = Counter(nums)
   ```

3. Adjust the counted frequencies to match the expectations of a `base[n]` array. According to `base[n]`, the number `n` should appear twice, and all the numbers from `1` to `n - 1` should appear once. To reflect this in our counter, we subtract 2 from the count of `n` and subtract 1 from the counts of all other numbers within the range `1` to `n`.

   ```
   cnt[n] -= 2
   for i in range(1, n):
       cnt[i] -= 1
   ```

4. After the adjustments, a "good" array would leave all counts in the Counter at zero. Verify that this is true by applying the `all` function on a generator expression that checks if all `values` in the Counter are zero.

   ```
   return all(v == 0 for v in cnt.values())
   ```

5. If any value in the Counter is not zero, then the array cannot be a permutation of "base" because it does not contain the correct frequency of numbers. In such a case, the function will return `false`.

This solution approach utilizes the Counter data structure to perform frequency counting efficiently. The adjustment steps ensure that the counts match the unique `base[n]` array's requirements. The final `all` check succinctly determines the validity of `nums` being a good array, keeping the implementation both effective and elegant.

Example Walkthrough

Let's use an example to illustrate the solution approach.

Example Input

Consider the array `nums = [3, 1, 2, 3]`.

Steps

1. First, calculate `n` by taking the length of `nums` and subtracting one to account for the fact that there should be `n + 1` elements in a good `base` array. In this case, `len(nums) - 1` equals `4 - 1` which is `3`. Hence, `n = 3`.

   ```
   n = len(nums) - 1  # n = 3
   ```

2. Initialize a Counter with the array `nums`. This will count how many times each number appears in `nums`.

   ```
   cnt = Counter(nums)  # cnt = {3: 2, 1: 1, 2: 1}
   ```

3. Alter the counted frequencies to mimic a `base[n]` array. The number `n` should appear twice, so we subtract 2 from its count. All other numbers from `1` to `n - 1` should appear once, so we subtract 1 from their counts.

   ```
   cnt[n] -= 2  # cnt[3] -= 2 gives cnt = {3: 0, 1: 1, 2: 1}
   for i in range(1, n):
       cnt[i] -= 1  # iterating and subtracting 1, cnt = {3: 0, 1: 0, 2: 0}
   ```

4. All values in the Counter should now be zero for a good `base` array. Using the `all` function we check each value:

   ```
   return all(v == 0 for v in cnt.values())  # This evaluates to `True`
   ```

5. Since each number from `1` to `n - 1` is included exactly once and `n` is included exactly twice, and all adjusted counts are zero, the function will return `true`. This means `nums` is indeed a permutation of a `base[3]` array.

Following this approach, the given array `nums = [3, 1, 2, 3]` is confirmed to be a good array and thus the expected output for this example would be `True` as it fits the criteria of a `base[n]` array permutation.

Python Solution

```python
from collections import Counter # Import the Counter class from the collections module

class Solution:
    def isGood(self, nums: List[int]) -> bool:
        # Compute the length of the input array minus one
        length_minus_one = len(nums) - 1

        # Create a counter to record the frequency of each number in the input array
        num_counter = Counter(nums)

        # Decrease the count of the number 'length_minus_one' in the counter by 2
        num_counter[length_minus_one] -= 2

        # Iterate through the range from 1 to 'length_minus_one'
        for i in range(1, length_minus_one):
            # Decrease the count of 'i' in the counter by 1
            num_counter[i] -= 1

        # Return True if all counts in the counter are zero, else return False
        return all(count == 0 for count in num_counter.values())

```

Java Solution

```java
class Solution {

    // Method to check if the array 'nums' meets a certain condition
    public boolean isGood(int[] nums) {
        // Assuming nums.length - 1 is the maximum number that can be in 'nums'
        int n = nums.length - 1;

        // Create a counter array with size enough to hold numbers up to 'n'
        int[] count = new int[201]; // Assumes the maximum value in nums is less than or equal to 200

        // Count the occurrences of each number in nums and store in 'count'
        for (int number : nums) {
            ++count[number];
        }

        // Decrement the count of the last number 'n' by 2 as per the assumed constraint
        count[n] -= 2;

        // Decrement the count of numbers from 1 to n-1 (inclusive) by 1
        for (int i = 1; i < n; ++i) {
            count[i] -= 1;
        }

        // Check for any non-zero values in 'count', which would indicate 'nums' did not meet the condition
        for (int c : count) {
            if (c != 0) {
                return false;
            }
        }

        // If all counts are zero, it means 'nums' meets the condition
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to determine if the given vector 'nums' is "good" by certain criteria.
    bool isGood(vector<int>& nums) {

        // Calculate the size of 'nums' and store it in 'lastIndex'.
        int lastIndex = nums.size() - 1;

        // Initialize a counter array 'count' to hold frequencies of numbers in the range [0, 200].
        vector<int> count(201, 0); // Extended size to 201 to cover numbers from 0 to 200.

        // Populate 'count' vector with the frequency of each number in 'nums'.
        for (int num : nums) {
            ++count[num];
        }

        // The problem description might mention that the last element is counted twice,
        // so this line compensates for that by decrementing twice.
        count[lastIndex] -= 2;

        // The problem may specify that we should decrement the frequency
        // count of all numbers from 1 to 'lastIndex - 1'.
        for (int i = 1; i < lastIndex; ++i) {
            --count[i];
        }

        // Check the 'count' vector. If any element is not zero, return false.
        // An element not being zero would indicate the 'nums' vector is not "good".
        for (int counts : count) {
            if (counts != 0) {
                return false;
            }
        }

        // If all elements in 'count' are zero, the vector 'nums' is "good".
        return true;
    }
};

```

Typescript Solution

```typescript
function isGood(nums: number[]): boolean {
    // Get the size of the input array.
    const size = nums.length - 1;

    // Initialize a counter array with all elements set to 0.
    const counter: number[] = new Array(201).fill(0);

    // Count the occurrence of each number in the input array.
    for (const num of nums) {
        counter[num]++;
    }

    // Decrement the count at the index equal to the size of the input array by 2.
    counter[size] -= 2;

    // Decrement the count for each index from 1 up to size-1.
    for (let i = 1; i < size; ++i) {
        counter[i]--;
    }

    // Check if all counts are non-negative.
    return counter.every(count => count >= 0);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided function is determined by a few major steps:

1. `n = len(nums) - 1`: This is a constant time operation, `O(1)`.
2. `cnt = Counter(nums)`: Building the counter object from the `nums` list is `O(N)`, where `N` is the number of elements in `nums`.
3. `cnt[n] -= 2`: Another constant time operation, `O(1)`.
4. The loop `for i in range(1, n): cnt[i] -= 1`: This will execute `N-1` times (since `n = len(nums) - 1`), and each operation inside the loop is constant time, resulting in `O(N)` complexity.
5. The `all` function combined with the generator expression `all(v == 0 for v in cnt.values())`: Since counting the values in a `Counter` object and then iterating through them is `O(N)`, this step is `O(N)` as well.

Adding up all the parts, the overall time complexity is `O(N)` + `O(N)` + `O(N)` which simplifies to `O(N)`, because in Big O notation we keep the highest order term and drop the constants.

Space Complexity

The space complexity is also determined by a few factors:

1. `cnt = Counter(nums)`: Storing the count of each number in `nums` requires additional space which is proportional to the number of unique elements in `nums`. In the worst case, if all elements are unique, this will be `O(N)`.
2. The `for` loop and the `all` function does not use extra space that scales with the size of the input, as they only modify the existing `Counter` object.

Therefore, the space complexity is `O(N)`, where `N` is the number of elements in `nums` and assuming all elements are unique.