3046. Split the Array

Easy Array Hash Table Counting

Problem Description

In this problem, we are given an array `nums` that has an even number of elements. Our goal is to split this array into two halves, `nums1` and `nums2`, each containing exactly half the number of elements of the original array. The key challenge is to ensure that each half contains only distinct elements, meaning no duplicates are allowed within `nums1` or `nums2`.

The problem asks us to determine if such a split is possible and to return `true` if it is and `false` otherwise.

To simplify:

- `nums` has an even number of elements.
- Split `nums` into two equal halves `nums1` and `nums2`.
- Both `nums1` and `nums2` must contain only unique elements.
- Decide if the split is achievable or not.

Intuition

The intuition behind the solution lies in the constraint that both halves of the array must contain distinct elements. Since we must split the array into two equally sized parts, each with unique elements, the most immediate issue we would face is if a number appears too many times. Specifically, if a number appears three times or more, it is impossible to split the array into two parts with all distinct elements, as at least one of the parts would end up with duplicates of that number.

By using a counter to tally the frequency of each number, we can easily determine if any number exceeds the limit of two appearances in the array. If the maximum count of any element in the array is less than three, then we can ensure a split where both halves have distinct elements. This leads us to our simple solution approach.

Here's the gist of our solution approach:

1. Count the occurrences of each element in the array using a `Counter`.
2. Check if any element occurs three or more times.
3. If the maximum count is less than three, a valid split is possible, hence return `true`.
4. Otherwise, if any element occurs three or more times, return `false` because the split is impossible.

Solution Approach

The implementation of our solution utilizes the `Counter` class from Python's `collections` module, which is a specialized data structure that works like a dictionary. It is designed to count hashable objects, which in this problem are the integers in the `nums` array. The `Counter` counts how many times each number appears in the array.

Here's a step-by-step explanation of how the implementation works:

1. Create a `Counter` object that takes the `nums` array as input. This will return a `Counter` object where keys are the distinct numbers from the array and the values are the counts of those numbers.

   ```
   counts = Counter(nums)
   ```

2. Use the `.values()` method of the `Counter` object to get a list of all the counts (i.e., how many times each number is repeated in the array).

3. Apply the `max` function on this list to find the highest count. This tells us the maximum number of times any single number appears in `nums`.

   ```
   max_count = max(counts.values())
   ```

4. Finally, if the maximum count is less than 3, which implies no number occurs more than twice, we can make a split with all distinct elements in each part (`nums1` and `nums2`). Hence, the function will return `True`.

5. If the maximum count is 3 or more, at least one part of the split cannot have all distinct elements. Thus, the function will return `False`.

Here's the corresponding part of the Python code provided:

```
class Solution:
    def isPossibleToSolit(self, nums: List[int]) -> bool:
        return max(Counter(nums).values()) < 3
```

In summary, the solution approach is based on the realization that if any number occurs three times or more, we cannot create two halves with all distinct elements from `nums`. By using a `Counter`, we efficiently track the frequency of each element and use this information to determine the possibility of the intended split.

Example Walkthrough

Let's walk through an example to illustrate the solution approach:

Suppose our input array `nums` is `[1, 2, 3, 3, 4, 4]`. We want to split this array into two halves, each with three elements and no duplicates within each half.

Here are the steps following the proposed solution approach:

1. We first count the occurrences of each element in `nums`:
   - The number `1` appears once.
   - The number `2` appears once.
   - The number `3` appears twice.
   - The number `4` appears twice.

   The count looks like this: `{1: 1, 2: 1, 3: 2, 4: 2}`.

2. Now we check for the highest count among all the elements. In our count, the highest value is `2` as both `3` and `4` appear twice. No number appears three times or more.

3. Since the maximum count is less than `3`, it is possible to split the array into two halves with all distinct elements. In this case, one possible split would be:
   - `nums1` could be `[1, 3, 4]`
   - `nums2` could be `[2, 3, 4]`

   Each half has distinct elements, and we have successfully split `nums` into two valid halves.

To conclude, for the input array `[1, 2, 3, 3, 4, 4]`, our function would return `True`, indicating that the split is achievable.

Here's how we could translate the example into code using the `Counter` from Python's collections module:

```
from collections import Counter

nums = [1, 2, 3, 3, 4, 4]

# Create a `Counter` object to count the occurrences.
counts = Counter(nums)

# Get the maximum frequency of any number in `nums`.
max_count = max(counts.values())

# Determine if a split is possible.
is_possible_to_split = max_count < 3

print(is_possible_to_split)  # Output: True
```

For this example, the Python code will print `True`, as expected from our earlier analysis.

Solution Implementation

Python

```
from collections import Counter

class Solution:
    def is_possible_to_split(self, nums: List[int]) -> bool:
        # Count the frequency of each number in the input list
        num_counts = Counter(nums)

        # Check if any number occurs three or more times
        # If so, it's not possible to split the list, so return False
        # Otherwise, return True as it's possible to split the list
        return max(num_counts.values()) < 3
```

Java

```
class Solution {
    public boolean isPossibleToSolit(int[] nums) {
        // Array to count the occurrences of numbers.
        // Since the range of the numbers is not given, we have assumed it to be 0-100.
        int[] count = new int[101];

        // Looo through each number in the input array and increment its corresponding count
        for (int num : nums) {
            // Increment the count for this number
            count[num]++;

            // If the count for any number becomes 3 or more, it's not possible to split
            // the array where no number appears more than twice.
            if (count[num] >= 3) {
                return false;
            }
        }

        // If no number occurs more than twice, it's possible to split the array
        // accordingly, so we return true.
        return true;
    }
}
```

C++

```
#include <vector>

class Solution {
public:
    // Function to determine if it is possible to split the array into subsequences
    // where each subsequence contains unique numbers.
    bool isPossibleToSolit(vector<int>& nums) {
        // Initialize a frequency array to store the count of each number in 'nums'.
        // Array size of 101 assumes that numbers in 'nums' are in the range [0, 100].
        int frequency[101] = {};

        // Iterate through each number in 'nums' to populate the frequency array.
        for (int x : nums) {
            // Increment the count for the current number.
            frequency[x]++;
            // Check the constraint: if any number occurs at least 3 times,
            // it is not possible to split 'nums' as per the condition.
            if (frequency[x] >= 3) {
                // Return false if the condition is violated.
                return false;
            }
        }

        // If the looo completes without returning false, the condition is satisfied.
        // Return true indicating it is possible to split the array as required.
        return true;
    }
};
```

TypeScript

```
function isPossibleToSolit(nums: number[]): boolean {
    // Create an array to count the occurrences of each number.
    const occurrenceCount: number[] = new Array(101).fill(0);

    // Looo over all numbers in the input array.
    for (const num of nums) {
        // Increment the count for each number.
        occurrenceCount[num]++;

        // If any number occurs 3 or more times, splitting is not possible.
        if (occurrenceCount[num] >= 3) {
            return false;
        }
    }

    // If the loop completes without finding a number that occurs 3 or more times,
    // then splitting into pairs of distinct numbers is possible.
    return true;
}
```

```
from collections import Counter

class Solution:
    def is_possible_to_split(self, nums: List[int]) -> bool:
        # Count the frequency of each number in the input list
        num_counts = Counter(nums)

        # Check if any number occurs three or more times
        # If so, it's not possible to split the list, so return False
        # Otherwise, return True as it's possible to split the list
        return max(num_counts.values()) < 3
```

Time and Space Complexity

The time complexity of the code can be analyzed based on the operations it performs. The `Counter` class from the `collections` module iterates over all elements of `nums` to count the frequency of each unique number, which requires $O(n)$ time where n is the length of the input list `nums`. The `max` function then iterates over the values of the counter, which in the worst case can also be $O(n)$ if all numbers in `nums` are unique. However, since `max()` is working on the values and not the keys, and the values represent counts that can be at most n, the time for `max()` is $O(u)$ where u is the number of unique numbers. Typically, $u <= n$, so the overall time complexity remains $O(n)$.

The space complexity of the code is determined by the additional space required to store the elements counted by the `Counter`. In the worst case, if all elements in the array are unique, the `Counter` would need to store each unique element as a key with its associated count as a value, requiring $O(u)$ space where u is the number of unique numbers in `nums`. Since u can be at most n, the space complexity is $O(n)$.

Therefore, the reference answer is correct in stating that both the time complexity and space complexity of the code are $O(n)$.