2610. Convert an Array Into a 2D Array With Conditions

Medium Array Hash Table

Problem Description

In this problem, we're given an integer array `nums`, and our goal is to transform this array into a two-dimensional (2D) array with specific conditions. The 2D array must use all the integers from `nums` and arrange them so that each row contains only unique integers, without any repetitions. Additionally, we need to minimize the number of rows in the 2D array. The final 2D array can have varying numbers of elements per row, and if there are several correct ways to create it, any valid answer is acceptable.

Intuition

The solution leverages the idea of distribution: we want to spread out the occurrence of each integer in `nums` across different rows of the resulting 2D array. Here's the thought process leading to the solution:

- First, we need to know how many times each number appears in `nums`. To do this efficiently, a `Counter` is used, which is essentially a hash table (dictionary) where each key is a unique number from `nums` and its corresponding value is the count of how many times it appears.
- Since each row must contain distinct integers, the number of times an integer appears dictates how many rows it will have to be spread across.
- The next step is to iterate through each unique number and its count, and add it to a new row in the 2D array until we've placed it the required number of times.
- To maintain the minimum number of rows, we only add a new row when it's necessary – that is, when the number of occurrences of the current number exceeds the current number of rows in the `ans` array.
- We'll append the number to increasingly indexed rows until we've placed it accordingly (based on its count).

Solution Approach

The implemented solution uses a hash table and array manipulation to satisfy the problem's requirements.

Here's a step-by-step breakdown of the implementation:

1. **Counting Elements:** We start by creating a `Counter` from the `nums` array. This `Counter` acts as a hash table that maps each unique integer in `nums` to the number of its occurrences.

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

2. **Preparing the Answer Array:** We initialize an empty list `ans`, which will become the 2D array we must return.

```
ans = []
```

3. **Populating the 2D Array:** We iterate over each unique element `x` and its occurrence count `v` in the `Counter`.

```
for x, v in cnt.items():
    for i in range(v):
```

4. **Row Management:** During the iteration for each unique element:
   - We check if the current row `i` exists in `ans`. If not, we create a new row (an empty list).

```
if len(ans) <= i:
    ans.append([])
```

   - We then append `x` to the proper row, filling the 2D array such that each row will have distinct integers. Since we iterate `v` times (the count of occurrences), `x` is added to each subsequent row.

```
ans[i].append(x)
```

By following this algorithm:

- We leverage the hash table for efficient lookups and counting.
- We use simple list manipulations to build the answer.
- We ensure minimal rows by checking if a row exists before adding to it, and only when necessary do we create a new row.

This approach ensures that each row will contain distinct elements from `nums`, and we only increase the total row count when the frequency of an integer necessitates an additional row, thus achieving the minimization of rows for our 2D array.

Finally, we return the `ans` list, which now represents the 2D array with the desired properties:

```
return ans
```

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose our input array `nums` is:

```
nums = [5, 5, 6, 6, 6, 7]
```

We will apply each step of the solution approach to this array.

1. Counting Elements

We create a Counter from the `nums` array that gives us the frequency of each unique number:

```
cnt = Counter(nums)  # Output: Counter({6: 3, 5: 2, 7: 1})
```

This tells us that the number 6 appears 3 times, 5 appears 2 times, and 7 appears 1 time.

2. Preparing the Answer Array

We initialize an empty list `ans` to represent our 2D array.

```
ans = []
```

3. Populating the 2D Array

We iterate over the elements and their counts in the `cnt` dictionary.

4. Row Management

For each unique element, we go through the count of its occurrences and manage rows accordingly.

For the number 6 (which appears 3 times):

```
# Since ans has 0 rows, we add a new row and insert 6.
ans = [[6]]
# Still needs to place 6 two more times, check next row which is empty, add a new row and insert 6.
ans = [[6], [6]]
# Still needs to place 6 one more time, check next row which is empty, add a new row and insert 6.
ans = [[6], [6], [6]]
```

Next, for the number 5 (which appears 2 times):

```
# There are already 3 rows, so we place 5 into the first row.
ans = [[6, 5], [6], [6]]
# Needs to place 5 one more time, we place it into the second row.
ans = [[6, 5], [6, 5], [6]]
```

Lastly, for the number 7 (which appears 1 time):

```
# There are already 3 rows, so we place 7 into the first row that does not have 7.
ans = [[6, 5, 7], [6, 5], [6]]
```

We have now placed each number from `nums` keeping the rows distinct and minimized the number of rows in the process.

Final 2D Array

The final `ans` array representing our 2D array is:

```
ans = [[6, 5, 7], [6, 5], [6]]
```

We return this array as our solution. It has the minimum number of rows, contains all integers from `nums`, and each row contains only unique integers.

Solution Implementation

Python

```
from collections import Counter

class Solution:
    def findMatrix(self, nums: List[int]) -> List[List[int]]:
        # Count the occurrences of each number in the input list
        num_counter = Counter(nums)

        # Initialize an empty list to store the resulting matrix
        matrix = []

        # Iterate through the counted numbers and their counts
        for number, count in num_counter.items():
            # Loop for the count number of times for each number
            for i in range(count):
                # If the matrix has fewer rows than needed, add a new row
                if len(matrix) <= i:
                    matrix.append([])

                # Append the current number to the i-th row of the matrix
                matrix[i].append(number)

        # Return the resulting matrix
        return matrix
```

Java

```
class Solution {
    public List<List<Integer>> findMatrix(int[] nums) {
        // Initialize a list to hold the final groups of numbers.
        List<List<Integer>> result = new ArrayList<>();
        // Find the length of the input array.
        int n = nums.length;
        // Create an array to keep track of the count of each number.
        int[] count = new int[n + 1];
        // Count the occurrences of each number in the input array.
        for (int num : nums) {
            ++count[num];
        }
        // Iterate over the possible numbers from 1 to n and organize them into the result list.
        for (int num = 1; num <= n; ++num) {
            int frequency = count[num];
            // For each occurrence of the number, place it into a sub-list.
            for (int i = 0; i < frequency; ++i) {
                // If the current sub-list doesn't exist, create it.
                if (result.size() <= i) {
                    result.add(new ArrayList<>());
                }
                // Add the current number to the appropriate sub-list.
                result.get(i).add(num);
            }
        }
        // Return the list of lists containing grouped numbers.
        return result;
    }
}
```

C++

```
#include <vector>

class Solution {
public:
    // Function that rearranges numbers into a sorted matrix based on their frequency
    std::vector<std::vector<int>> findMatrix(std::vector<int>& nums) {
        std::vector<std::vector<int>> matrix; // Will hold the final sorted matrix
        int n = nums.size();
        std::vector<int> count(n + 1, 0); // Initialize counting vector with zeros

        // Count how many times each number appears in the input vector
        for (int num : nums) {
            ++count[num];
        }

        // Iterate through each unique number in the input array
        for (int num = 1; num <= n; ++num) {
            int frequency = count[num]; // Get the frequency of the current number

            // Construct rows of the matrix based on the frequency of the current number
            for (int i = 0; i < frequency; ++i) {
                // If there are not enough rows in the matrix, add a new row
                if (matrix.size() <= i) {
                    matrix.push_back(std::vector<int>());
                }
                // Add the current number to the i-th row
                matrix[i].push_back(num);
            }
        }

        // Return the organized matrix
        return matrix;
    }
};
```

TypeScript

```
function findMatrix(nums: number[]): number[][] {
    // Initialize the answer matrix.
    const answerMatrix: number[][] = [];

    // Calculate the length of the input array.
    const length: number = nums.length;

    // Initialize a counter array of length `length + 1` to keep track of the frequency of each number.
    // Each index corresponds to a value from the input array.
    const frequencyCounter: number[] = new Array(length + 1).fill(0);

    // Count the frequency of each number in input array and update the frequencyCounter array.
    for (const num of nums) {
        ++frequencyCounter[num];
    }

    // Iterate through the possible numbers in the input array.
    for (let num = 1; num <= length; ++num) {
        // For each number, we create a row in the matrix for the number of times it appears.
        for (let i = 0; i < frequencyCounter[num]; ++i) {
            // If we don't have enough rows in the answerMatrix, add a new empty row.
            if (answerMatrix.length <= i) {
                answerMatrix.push([]);
            }
            // Append the number to the respective row in the matrix.
            answerMatrix[i].push(num);
        }
    }

    // Return the constructed matrix.
    return answerMatrix;
}
```

```
from collections import Counter

class Solution:
    def findMatrix(self, nums: List[int]) -> List[List[int]]:
        # Count the occurrences of each number in the input list
        num_counter = Counter(nums)

        # Initialize an empty list to store the resulting matrix
        matrix = []

        # Iterate through the counted numbers and their counts
        for number, count in num_counter.items():
            # Loop for the count number of times for each number
            for i in range(count):
                # If the matrix has fewer rows than needed, add a new row
                if len(matrix) <= i:
                    matrix.append([])

                # Append the current number to the i-th row of the matrix
                matrix[i].append(number)

        # Return the resulting matrix
        return matrix
```

Time and Space Complexity

The time complexity of the code is primarily determined by two factors: counting the elements in `nums` and then iterating over the count to build the `ans` list.

Counting the frequency of each element using `Counter(nums)` can be associated with a time complexity of $O(n)$, where n is the length of the array `nums`. This operation involves going through all elements once to determine their frequencies.

After counting, the code iterates over the count dictionary and, for each element `x`, it appends `x` to the lists in `ans` `v` times (where `v` is the frequency of `x`). Since the total number of append operations is equal to the length of the `nums` list (every number from the list is appended exactly once), this also has a time complexity of $O(n)$.

Therefore, the overall time complexity of the code is $O(n)$.

As for space complexity, we are using additional data structures: a dictionary for the counts and a list of lists for the `ans`. Since both the dictionary and the list of lists will at most store n entries (each corresponding to an element in the original `nums` list), the space complexity is also $O(n)$.