1296. Divide Array in Sets of K Consecutive Numbers

Medium Greedy Array Hash Table Sorting Leetcode Link

Problem Description

Given an integer array `nums` and a positive integer `k`, the task is to determine whether the array can be divided into several sets of `k` consecutive numbers. To divide the array into sets of `k` consecutive numbers means to group the elements of the array such that each group contains `k` numbers and each number in a group follows the one before it with a difference of 1. The function should return `true` if such division is possible, otherwise, it should return `false`.

Intuition

To solve this problem, the intuition is to use a greedy approach. The steps followed are:

1. Count the occurrences of each number in the array using a hash map or counter. This step helps to quickly find how many times a number appears in `nums` without sorting the entire array.
2. Start from the smallest number in the array and try to build a consecutive sequence of length `k`. If the sequence cannot be formed due to a missing number, return `false`.
3. If a consecutive sequence of length `k` starting from a number is successfully formed, reduce the count of the numbers used in the sequence. If the count drops to zero, remove the number from the counter to avoid unnecessary checks in further iterations.
4. Repeat steps 2 and 3 until all numbers are used to form valid sequences or until you find a sequence that cannot be completed.

Using this approach, the solution checks in a sorted order if there are enough consecutive numbers following the current number to form a group of `k` elements. If at any point there aren't enough consecutive numbers to form a group of `k`, the function returns `false`. On the other hand, if all numbers can be grouped successfully, the function returns `true`.

Solution Approach

The provided solution implements a greedy algorithm to solve the problem by using the following steps:

1. **Counting Elements**: The solution uses Python's `Counter` from the `collections` module to count the frequency of each integer in the input `nums`. This `Counter` acts like a hash map, and it stores each unique number as a key and its frequency as the corresponding value.

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

2. **Sorting and Iterating**: After counting, the code sorts the unique numbers and iterates over them. The sorting ensures that we check for consecutive sequences starting with the smallest number.

   ```
   for v in sorted(nums):
   ```

3. **Forming Consecutive Groups**: Inside the loop, we check if the current number's count is non-zero, indicating that it hasn't been used up in forming a previous group. If the count is non-zero, the nested loop tries to form a group starting from this number `v` up to `v + k`.

   ```
   if cnt[v]:
       for x in range(v, v + k):
   ```

4. **Validating Consecutive Numbers**: For each number in the expected consecutive range `[v, v + k)`, check if the current number `x` is present in the counter (i.e., its count is not zero). If it is zero, this indicates that a consecutive sequence cannot be formed, and the function returns `False`.

   ```
   if cnt[x] == 0:
       return False
   ```

5. **Updating the Counter**: When a number `x` is found, its count is decremented since it's being used to form the current sequence. If the count reaches zero after decrementing, the number `x` is removed from the counter to prevent future unnecessary checks.

   ```
   cnt[x] -= 1
   if cnt[x] == 0:
       cnt.pop(x)
   ```

6. **Completing the Iteration**: This process continues until either a missing number is found (in which case `False` is returned), or all numbers are successfully grouped (in which case `True` is returned when the loop finishes).

Through these steps, the algorithm ensures that all possible consecutive sequences of length `k` are checked and formed, validating the possibility of dividing the array into sets of `k` consecutive numbers accurately.

Example Walkthrough

Let's illustrate the solution approach with an example:

Suppose we have an array `nums = [1, 2, 3, 3, 4, 4, 5, 6]` and `k = 4`. We want to find out if it's possible to divide this array into sets of `k` (4) consecutive numbers.

Following the solution approach:

1. **Counting Elements**: We count the frequency of each number using the `Counter`.

   ```
   from collections import Counter
   nums = [1, 2, 3, 3, 4, 4, 5, 6]
   cnt = Counter(nums)
   # Counter({3: 2, 4: 2, 1: 1, 2: 1, 5: 1, 6: 1})
   ```

2. **Sorting and Iterating**: We sort the `nums` and iterate over the distinct values. Here, the sorted unique values would be `[1, 2, 3, 4, 5, 6]`.
3. **Forming Consecutive Groups**: We start from the smallest number and try to form a group of `k` consecutive numbers. Starting with `1`, then `2`, `3`, and `4`.
4. **Validating Consecutive Numbers**: We check if each of these consecutive numbers is present in the counter with a non-zero count.
   - For `1`, `cnt[1] = 1`, thus we can use one `1`.
   - For `2`, `cnt[2] = 1`, we can use one `2`.
   - For `3`, `cnt[3] = 2`, we can use one `3`.
   - For `4`, `cnt[4] = 2`, we can use one `4`.
5. **Updating the Counter**: After decrementing, if any count becomes zero, we remove the number from the counter.
   - After using the numbers for the first group `[1, 2, 3, 4]`, our counter updates to:
   - Counter({3: 1, 4: 1, 5: 1, 6: 1})
6. **Completing the Iteration**: We repeat the process for the next smallest number with a non-zero count, which in this updated counter is `3`. We try to form the next group starting from `3`, and we would need a sequence `[3, 4, 5, 6]`.
   - For `3`, `cnt[3] = 1`, use one `3`.
   - For `4`, `cnt[4] = 1`, use one `4`.
   - For `5`, `cnt[5] = 1`, use one `5`.
   - For `6`, `cnt[6] = 1`, use one `6`.

   After this sequence, our counter is empty, which means we have successfully used all numbers to form groups of `k` consecutive numbers.

Since no step failed and we could form two groups `[1, 2, 3, 4]` and `[3, 4, 5, 6]` each with 4 consecutive numbers, the function would return `True`. Hence, it is possible to divide the given array `nums` into sets of `k` (4) consecutive numbers.

Python Solution

```python
from collections import Counter

class Solution:
    def isPossibleDivide(self, nums: List[int], k: int) -> bool:
        # Create a frequency count for all the numbers in nums
        num_count = Counter(nums)

        # Loop over each number after sorting nums
        for num in sorted(nums):
            # If this number is still in the count dictionary
            if num_count[num]:
                # Attempt to create a consecutive sequence starting at this number
                for x in range(num, num + k):
                    # If any number required for the sequence does not exist, return False
                    if num_count[x] == 0:
                        return False
                    # Decrease the count for this number since it's used in the sequence
                    num_count[x] -= 1
                    # If the count drops to zero, remove it from the dictionary
                    if num_count[x] == 0:
                        num_count.pop(x)

        # If the entire loop completes without returning False, it means all sequences can be formed
        return True

```

Java Solution

```java
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;

class Solution {

    /**
     * Checks if it is possible to divide the array into consecutive subsequences of length k.
     *
     * @param nums Input array of integers.
     * @param k Length of the consecutive subsequences.
     * @return true if division is possible, false otherwise.
     */
    public boolean isPossibleDivide(int[] nums, int k) {
        // Create a map to store the frequency of each number in the input array.
        Map<Integer, Integer> frequencyMap = new HashMap<>();
        for (int num : nums) {
            frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
        }

        // Sort the input array to ensure the numbers are in ascending order.
        Arrays.sort(nums);

        // Iterate over the sorted array to check if division into subsequences is possible.
        for (int num : nums) {
            // Only start a sequence if the current number is still in the frequencyMap.
            if (frequencyMap.containsKey(num)) {
                // Attempt to create a subsequence of length k starting with the current number.
                for (int i = num; i < num + k; ++i) {
                    // If the current number is not in the frequencyMap, division is not possible.
                    if (!frequencyMap.containsKey(i)) {
                        return false;
                    }
                    // Decrease the frequency of the current number.
                    frequencyMap.put(i, frequencyMap.get(i) - 1);
                    // If the frequency of a number becomes 0, remove it from the map.
                    if (frequencyMap.get(i) == 0) {
                        frequencyMap.remove(i);
                    }
                }
            }
        }

        // If the loop completes, then division into subsequences of length k is possible.
        return true;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to determine if it is possible to divide the vector of integers into groups of size 'k'
    // with consecutive numbers.
    bool isPossibleDivide(std::vector<int>& nums, int k) {
        // Creating a frequency map to count occurrences of each number
        std::unordered_map<int, int> frequencyMap;
        for (int num : nums) {
            ++frequencyMap[num];
        }

        // Sort the input vector to process numbers in ascending order
        std::sort(nums.begin(), nums.end());

        // Process each number in the sorted vector
        for (int num : nums) {
            // If there is still a count for this number, we need to form a group starting with this number
            if (frequencyMap.find(num) != frequencyMap.end()) {
                // Attempt to create a group of 'k' consecutive numbers
                for (int i = num; i < num + k; ++i) {
                    // If any number required to form the group is missing, return false
                    if (!frequencyMap.count(i)) {
                        return false;
                    }
                    // Decrement the count for the current number in the group
                    if (--frequencyMap[i] == 0) {
                        // If the count reaches zero, remove the number from the frequency map
                        frequencyMap.erase(i);
                    }
                }
            }
        }

        // If the function hasn't returned false, it's possible to divide the numbers as required
        return true;
    }
};

```

Typescript Solution

```typescript
function isPossibleDivide(nums: number[], k: number): boolean {
  // Create a frequency map to count occurrences of each number
  const frequencyMap: Record<number, number> = {};
  for (const num of nums) {
    frequencyMap[num] = (frequencyMap[num] || 0) + 1;
  }

  // Sort the input array to process numbers in ascending order
  nums.sort((a, b) => a - b);

  // Process each number in the sorted array
  for (const num of nums) {
    // If there is still a count for this number, we need to form a group starting with it
    if (num in frequencyMap) {
      // Attempt to create a group of 'k' consecutive numbers
      for (let i = num; i < num + k; i++) {
        // If any number required to form the group is missing, return false
        if (!(i in frequencyMap)) {
          return false;
        }
        // Decrement the count for the current number in the group
        if (--frequencyMap[i] === 0) {
          // If the count reaches zero, remove the number from the frequency map
          delete frequencyMap[i];
        }
      }
    }
  }

  // If the function hasn't returned false, it's possible to divide the numbers as required
  return true;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is determined by a few factors: the sorting of the input list `nums`, the construction of the counter `cnt`, and the nested loop where we check and decrement the count for each element over the range from `v` to `v + k`.

1. **Sorting `nums`**: The sort operation on the list `nums` has a time complexity of `O(N log N)`, where `N` is the number of elements in `nums`.
2. **Counter Construction**: Constructing the counter `cnt` is `O(N)` because we go through the list `nums` once.
3. **Nested Loop**: The nested loop involves iterating over each number in the sorted `nums` and then an inner loop that iterates up to `k` times for each unique number that has a non-zero count.

- It may seem like this gives us `O(Nk)`, but this is not entirely correct because each element is decremented once, and once it hits zero, it is popped from the counter and never considered again. Therefore, each element contributes at most `O(k)` before it's removed.
- The total number of decrements across all elements cannot exceed `N` (since each decrement corresponds to one element of `nums`), and since we have that outer loop that potentially could visit all `N` elements, we would multiply this by `k` giving us `O(Nk)`.

So, combining these together, the total time complexity is `O(N log N + N + Nk)` = `O(N log N + Nk)`. Since for large `N`, `N log N` dominates `N`, and `Nk` may either dominate or be dominated by `N log N` depending on the values of `N` and `k`, the final time complexity is often written with both terms.

Space Complexity

The space complexity is mostly determined by the counter `cnt` which stores a count of each unique number in `nums`.

- The counter can have at most `M` entries where `M` is the number of unique numbers in `nums`. In the worst case, if all numbers are unique, `M` is equal to `N`, giving us a space complexity of `O(N)`.

Therefore, the overall space complexity of the code is `O(N)`.