1121. Divide Array Into Increasing Sequences

Hard Greedy Array

Leetcode Link

Problem Description

The LeetCode problem in question requires us to determine if a sorted integer array `nums` can be divided into one or more disjoint increasing subsequences where each subsequence is of at least length `k`. A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements. Disjoint subsequences are such that they do not share any common elements. The `nums` array is given to be sorted in non-decreasing order (i.e., it can have duplicates, but the sequence is not decreasing). The function should return `true` if such a division is possible or `false` otherwise.

Intuition

To arrive at the solution, we must understand that if we can form disjoint increasing subsequences of at least length `k`, then the most frequent number in `nums` limits the number of these subsequences we can create.

This is because each subsequence can have at most one occurrence of any number, and hence the number of subsequences cannot exceed the number of occurrences of the most frequent number.

To find the solution, we follow these steps:

1. We calculate the frequency of the most frequent number, `mx`, by using the `groupby` function from the `itertools` module in Python, which groups consecutive identical elements. We convert each group to a list and get its length to find out how many times that particular number appears in the array.
2. We then multiply this frequency `mx` by `k` to find the minimum array length needed to have `mx` disjoint subsequences of length `k`. This is because each subsequence needs at least one occurrence of the most frequent number and each must be of length `k` or more.
3. We compare this minimum required length `mx * k` to the actual length of `nums`. If `mx * k` is less than or equal to the length of `nums`, then we can divide `nums` into the required disjoint increasing subsequences and we return `true`. Otherwise, we return `false` because there would not be enough numbers to form subsequences of length `k` with the most frequent number appearing in all of them.

By following this approach, the solution effectively ensures whether there are enough elements in the array to distribute among subsequences of the required length without any overlap.

Solution Approach

The implementation of the solution uses:

- **The `groupby` method:** This function is used for grouping elements in an iterable. If you pass a sorted iterable to it, it groups all consecutive duplicate elements together. In Python, it is available in the `itertools` module.
- **The `max` function:** After grouping the elements with `groupby`, the code calculates the maximum group size, which represents the frequency of the most common element in the sorted array `nums`. This is done by mapping each grouped sequence to its length and taking the maximum of these lengths.
- **List comprehension:** This is used for compactly applying operations to sequences. In the reference solution, list comprehension is utilized within the `max` function to create a list of lengths of the groups returned by `groupby`.
- **Comparison:** Finally, the decision to return `true` or `false` is decided by comparing the product of the maximum group size and `k` with the length of the `nums` array. If this product is less than or equal to the length of `nums`, then `true` is returned. Otherwise, `false` is returned.

The code snippet works as follows:

- `mx = max(len(list(x)) for _, x in groupby(nums))`: This line creates grouped sequences of identical consecutive elements in the sorted array `nums` using `groupby`. For each group, it takes the length by converting the group iterator to a list. Among these lengths, the maximum length is taken, which is stored in `mx`.
- `return mx * k <= len(nums)`: This checks whether the array has enough elements to form the required increasing subsequences. If the most frequent element's count (`mx`) times the minimum subsequence length (`k`) is less than or equal to the total number of elements in `nums`, then the condition is satisfied and `true` is returned. If not, `false` is returned.

In this implementation, the use of `groupby` is key because it allows us to easily find the frequency of the most common element in the sorted array, which is crucial for determining whether the subsequences can be formed.

Example Walkthrough

Let's illustrate the solution approach using a small example.

Consider the sorted integer array `nums = [1, 2, 3, 3, 4, 4, 5, 5, 5]` and let `k = 3`. We want to determine if this array can be divided into disjoint increasing subsequences each of at least length 3.

Following the described approach:

1. We use the `groupby` method to group identical consecutive elements. Thus, we get the groups `[(1), (2), (3, 3), (4, 4), (5, 5, 5)]`.
2. We calculate the maximum frequency `mx`. The groups' lengths are `[1, 1, 2, 2, 3]`, so `mx` is `3` because 5 occurs three times.
3. We then calculate `mx * k`. Here, `mx` is 3 and `k` is 3, so `mx * k` is `9`.
4. We compare `mx * k` with the length of `nums`. The length of `nums` is `9`, which is equal to `mx * k` (both are 9).
5. Since `mx * k` <= the length of `nums`, the function would return `true`. This means we can divide `nums` into one or more disjoint increasing subsequences of length at least 3. One such division could be `[[1, 3, 5], [2, 4, 5], [3, 4, 5]]`.

This example illustrates how by using the `groupby` function and checking the maximum frequency of a number times the minimum subsequence length `k`, we can determine if a sorted array can be divided into the required subsequences.

Python Solution

```python
from itertools import groupby
from typing import List

class Solution:
    def canDivideIntoSubsequences(self, nums: List[int], k: int) -> bool:
        # Calculate the maximum frequency of any number in the list
        max_frequency = max(len(list(group)) for _, group in groupby(nums))

        # The number of subsequences of size k we can create is equal to the maximum frequency.
        # If the total length of nums is at least as large as this number, then we can
        # divide nums into subsequences of size k, where each subsequence is strictly increasing.
        return max_frequency * k <= len(nums)

```

Java Solution

```java
class Solution {
    // Method to check if an array can be divided into subsequences each of length k
    public boolean canDivideIntoSubsequences(int[] nums, int k) {
        int currentCount = 0; // to hold count of current element
        int lastValue = 0; // to store the value of the last element processed

        // Iterate over each element in the array
        for (int currentValue : nums) {
            // If the current element is the same as the last, increment the count
            // Otherwise, reset the count for a new value
            currentCount = (lastValue == currentValue) ? currentCount + 1 : 1;

            // If the number of times an element appears multiplied by k exceeds the array length,
            // it's not possible to divide the array into subsequences of length k
            if (currentCount * k > nums.length) {
                return false;
            }

            // Update the last processed value to the current value
            lastValue = currentValue;
        }

        // If we didn't return false during the loop, it's possible to divide the array
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    bool canDivideIntoSubsequences(vector<int>& nums, int k) {
        int currentStreak = 0; // Count of how many times the current number has appeared consecutively
        int previousValue = 0; // The value of the previous element in the array

        // Iterate through each number in the given 'nums' vector
        for (int currentValue : nums) {

            // If the current value is the same as the previous value, increment the streak
            // Otherwise, reset the streak count to 1 for the new number
            currentStreak = (previousValue == currentValue) ? currentStreak + 1 : 1;

            // If the number of times a particular element needs to be repeated ('currentStreak' times 'k')
            // exceeds the total length of the 'nums' vector, it is not possible to divide into subsequences.
            if (currentStreak * k > nums.size()) {
                return false;
            }

            // Update the previous value to the current value for the next iteration
            previousValue = currentValue;
        }

        // If the loop completes without returning false, it means the 'nums' vector can be
        // divided into subsequences of length 'k' without violating the rules
        return true;
    }
};

```

Typescript Solution

```typescript
function canDivideIntoSubsequences(nums: number[], k: number): boolean {
    let currentStreak = 0; // Count of consecutive appearances of the current number
    let previousValue = 0; // The value of the previous element in the array

    // Iterate through each number in the 'nums' array
    for (let currentValue of nums) {
        // If the current value is the same as the previous value,
        // increment the streak, otherwise reset streak to 1
        if (previousValue === currentValue) {
            currentStreak++;
        } else {
            currentStreak = 1;
        }

        // If the current streak times k exceeds the array size,
        // it's not possible to divide into subsequences
        if (currentStreak * k > nums.length) {
            return false; // Return early as the condition is violated
        }

        // Update the previousValue to the currentValue for the next iteration
        previousValue = currentValue;
    }

    // If we can iterate through the array without returning false,
    // it means the array can be divided into subsequences of length 'k'
    return true;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is primarily determined by the `groupby` operation from the `itertools` library and the calculation of the maximum group size (`mx`).

The `groupby` operation is `O(n)`, where `n` is the length of the list `nums`. This is because `groupby` iterates through the list once in order to group adjacent elements together.

The calculation of the maximum group size involves iterating over each group generated by `groupby` and finding the length of the longest group created. This step is also `O(n)` because in the worst case, it will iterate through all elements of `nums` once, if all elements are the same and thus belong to a single group.

Thus, the time complexity of the function is `O(n)` overall, as both the `groupby` operation and the calculation of the maximum group size are linear in terms of the input size.

Space Complexity

The space complexity of this code involves the extra space required to store the groups formed by the `groupby` function.

For each unique element in `nums`, a new group is formed, and a generator object is created for each group. However, since these groups are not stored in memory all at once but instead one at a time as they are iterated over, the space complexity does not grow with the number of unique elements.

The variable `mx` only requires constant space, and since `groupby` does not create a list of all groups but just an iterable, the space complexity for the storage of groups is also constant.

Thus, the space complexity is `O(1)`, because the space required does not grow with the size of the input list `nums`.