2638. Count the Number of K-Free Subsets

Medium Array Dynamic Programming Sorting Leetcode Link

Problem Description

In this problem, you're given an integer array `nums`, with distinct elements, and an integer `k`. Your task is to count the number of subsets of `nums` that are "k-Free." A subset is "k-Free" if no two elements in the subset have an absolute difference equal to `k`. It's important to remember that a subset can be any collection of elements from the array, including no elements at all, which means the empty set is always a "k-Free" subset.

Intuition

To solve this problem, we should start by understanding that we're looking for subsets where no two elements differ by `k`. Since the elements of `nums` are distinct, we can sort `nums` and group them by their remainder when divided by `k`. Each of these groups will contain elements that differ by, at a minimum, `k` from elements of other groups, meaning the absolute difference of `k` can only occur within the same group.

After the grouping, we can separate the problem into smaller subproblems: counting "k-Free" subsets within each group, independent of the others. We tackle this with dynamic programming, where `f[i]` will represent the number of "k-Free" subsets we can form using the first `i` elements of a group. The base cases are `f[0] = 1`, since there's only one empty subset of zero elements, and `f[1] = 2`, which represents the two subsets of a single-element group: the empty set and the set containing that element.

When we have at least two elements (i.e., `i >= 2`), we check if the absolute difference between the last two elements being considered is `k`. If it is, we can't include both of them in a "k-Free" subset. Hence, the number of "k-Free" subsets involving these two elements is the same as `f[i - 1] + f[i - 2]`: either we include the last element (and count subsets not including the previous one), or we don't (counting subsets up to the previous one). If the difference isn't `k`, then including the last element doesn't restrict us in any way, so we have `f[i] = f[i - 1] * 2`, as each existing subset can either include or exclude the last element, effectively doubling the count.

Once we have the count for each group, we multiply the counts together to get the total number of "k-Free" subsets, which gives us our answer.

Solution Approach

The solution to this problem employs both sorting and dynamic programming, alongside additional data structures, to manage and compute the required counts of "k-Free" subsets.

1. **Sorting**: We begin by sorting the array `nums` to organize elements in ascending order. This allows us to easily group elements by their remainders when divided by `k`.

2. **Grouping Elements**: After sorting, the array elements are grouped based on their remainder modulo `k`. This is done using a dictionary where the key is the remainder and the value is a list of elements with that remainder. In Python, this is realized using a `defaultdict` from the `collections` module. The expression `g[x % k].append(x)` populates our groups.

3. **Dynamic Programming**: For each group of elements (with the same remainder), we use a dynamic programming approach to count the subsets. We initialize an array `f` with a length of `m + 1`, where `m` is the number of elements in the current group. `f[i]` will store the number of "k-Free" subsets possible with the first `i` elements of the group.

4. **Base Cases**: `f[0]` is `1`, representing the empty set, and `f[1]` is `2`. The latter represents the subsets possible with one element: one subset is the empty set, and another is the set containing just that one element.

5. **Recurrence Relation**: For `i >= 2`, we have a decision based on the difference between the `i-1`th and the `i-2`th elements. If `arr[i - 1] - arr[i - 2]` is equal to `k`, we cannot have both elements in a subset, so `f[i]` is the sum of `f[i - 1]` (excluding `arr[i - 1]`) and `f[i - 2]` (including `arr[i - 1]` and excluding `arr[i - 2]`). Otherwise, `f[i]` is twice `f[i - 1]`, as we can freely choose to include or exclude the `i-1`th element in each subset counted by `f[i - 1]`.

6. **Final Computation**: After computing the number of subsets for each group, the variable `ans` is used to hold the cumulative product of subset counts from each group. The final result is the value of `ans` after all groups have been processed.

Using this approach, we efficiently compute the count of "k-Free" subsets without having to examine each subset of `nums` individually. The dynamic programming aspect drastically reduces the complexity by breaking the problem down into manageable states for which the solution can be constructed iteratively.

Example Walkthrough

Let's take a small example to illustrate the solution approach. Consider the integer array `nums = [1, 2, 4, 5, 7]`, and let `k = 3`. We want to count the number of "k-Free" subsets, where no two elements have an absolute difference of `3`.

1. **Sorting**: First, we sort the array `nums`, but since our array is already sorted, we can move to the next step.

2. **Grouping Elements**: Next, we group elements based on their remainder when divided by `k`. The remainders and their groups will look like this:

   - Remainder 0: [4, 7] (since 4%3=1 and 7%3=1)
   - Remainder 1: [1]
   - Remainder 2: [2, 5]

   Each of these groups can be used to form "k-Free" subsets independently.

3. **Dynamic Programming**:

   - For the group with remainder 0 (`[4, 7]`), we start with our dynamic programming array `f` with the base case `f[0]=1`. We have `f[1]=2`, representing the number of subsets when we consider up to the first element of this group. Since `7-4 != k`, we use the relation `f[i]=f[i-1]*2`, giving us `f[2] = f[1]*2 = 4`. Thus, there are 4 subsets possible with elements `[4, 7]`.

   - For the group with remainder 1 (`[1]`), there are only two possible subsets (`f[1] = 2`): the empty set and the set `[1]`.

   - For the group with remainder 2 (`[2, 5]`), we start similarly with `f[0] = 1`, `f[1] = 2`. For `f[2]`, since `5-2 != k`, we can double the previous count, resulting in `f[2] = f[1]*2 = 4`.

4. **Final Computation**: We multiply the counts of each group's "k-Free" subsets to get the overall count. Multiplying the counts from each group together, we have `4 * 2 * 4 = 32`. There are 32 "k-Free" subsets in the array `nums`.

By following these steps, we've demonstrated the method to find the number of "k-Free" subsets efficiently using dynamic programming without checking each possible subset explicitly.

Python Solution

```python
from collections import defaultdict

class Solution:
    def count_the_num_of_k_free_subsets(self, nums: List[int], k: int) -> int:
        # Sort the numbers in ascending order
        nums.sort()

        # Group numbers by their remainder when divided by k
        remainder_groups = defaultdict(list)
        for num in nums:
            remainder_groups[num % k].append(num)

        # Initialize answer as 1
        ans = 1

        # Loop through each group of numbers with the same remainder
        for group in remainder_groups.values():
            group_size = len(group)
            # The dp array holds the count of k-free subsets up to the current index
            dp = [0] * (group_size + 1)
            dp[0] = 1  # One way to create a subset including no elements
            dp[1] = 2  # Two ways to create a subset including the first element

            # Calculate the number of k-free subsets for the current group
            for i in range(2, group_size + 1):
                # If the difference between current and previous is k, we may merge the last two sets
                if group[i - 1] - group[i - 2] == k:
                    dp[i] = dp[i - 1] + dp[i - 2]
                else:
                    # Otherwise, we can either include or exclude the current number
                    dp[i] = dp[i - 1] * 2

            # Multiply the result by the number of k-free subsets for the current group
            ans *= dp[group_size]

        # Return the total number of k-free subsets
        return ans

```

Java Solution

```java
class Solution {
    public long countTheNumOfKFreeSubsets(int[] nums, int k) {
        // Sort the input array.
        Arrays.sort(nums);
        // Group numbers in the array by their modulus k.
        Map<Integer, List<Integer>> modulusGroups = new HashMap<>();
        for (int num : nums) {
            modulusGroups.computeIfAbsent(num % k, x -> new ArrayList<>()).add(num);
        }
        // Initialize answer to 1 since we will multiply the counts.
        long answer = 1;
        // Iterate through each group formed by the modulus operation.
        for (List<Integer> group : modulusGroups.values()) {
            int size = group.size();
            // Dynamic programming array to hold the count of k-free subsets
            long[] dp = new long[size + 1];
            dp[0] = 1; // Base case: One way to form an empty subset.
            dp[1] = 2; // Base case: Either include the first element or not.
            // Fill up the dp array with the count of k-free subsets for each size.
            for (int i = 2; i <= size; ++i) {
                // Check if current and previous elements are k apart
                if (group.get(i - 1) - group.get(i - 2) == k) {
                    // If yes, we can form new subsets by adding the current element to subsets ending at i - 2
                    dp[i] = dp[i - 1] + dp[i - 2];
                } else {
                    // If not, subsets can either include or not include the current element
                    dp[i] = dp[i - 1] * 2;
                }
            }
            // Multiply the total answer with the count of k-free subsets of current size.
            answer *= dp[size];
        }
        // Return the final count of k-free subsets as the answer.
        return answer;
    }
}

```

C++ Solution

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

class Solution {
public:
    long long countTheNumOfKFreeSubsets(vector<int>& nums, int k) {
        // Sort the input array.
        sort(nums.begin(), nums.end());

        // Create a map to categorize numbers by their modulo with k.
        unordered_map<int, vector<int>> groupsByModulo;
        for (int num : nums) {
            groupsByModulo[num % k].push_back(num);
        }

        long long answer = 1; // Initialize the answer with 1 for multiplication.

        // Iterate through each group categorized by modulo with k.
        for (auto& [modulo, group] : groupsByModulo) {
            int groupSize = group.size();

            // Create a dynamic programming array to store intermediate results.
            vector<long long> dp(groupSize + 1);
            dp[0] = 1; // Base case: There's 1 way to make a subset with 0 elements.
            dp[1] = 2; // If there's 1 element, we can either include or exclude it.

            // Fill in the dynamic programming array.
            for (int i = 2; i <= groupSize; ++i) {
                // If the difference between two consecutive numbers is k, we have additional case to consider.
                if (group[i - 1] - group[i - 2] == k) {
                    // We can either add the current element in a new subset or add it to a subset without the previous element.
                    dp[i] = dp[i - 1] + dp[i - 2];
                } else {
                    // Otherwise, we can choose to include or exclude the current element for each subset.
                    dp[i] = dp[i - 1] * 2;
                }
            }

            // Multiply the answer by the number of ways to make subsets for this group.
            answer *= dp[groupSize];
        }

        // Return the final answer.
        return answer;
    }
};

```

Typescript Solution

```typescript
function countTheNumOfKFreeSubsets(nums: number[], k: number): number {
    // Sort the array in non-decreasing order
    nums.sort((a, b) => a - b);

    // Create a map to group numbers by their modulo k value
    const moduloGroups: Map<number, number[]> = new Map();

    // Populate the moduloGroups map
    for (const num of nums) {
        const modulo = num % k;
        if (!moduloGroups.has(modulo)) {
            moduloGroups.set(modulo, []);
        }
        moduloGroups.get(modulo)!.push(num);
    }

    // Initialize answer to 1 as a starting multiplication identity
    let answer: number = 1;

    // Iterate over each group in the map
    for (const groupArray of moduloGroups.values()) {
        const groupSize = groupArray.length;

        // Initialize dynamic programming array with base cases
        const subsetCounts: number[] = new Array(groupSize + 1).fill(1);
        subsetCounts[1] = 2; // 2 ways for a single number: include or exclude

        // Calculate the number of K-free subsets for the group
        for (let i = 2; i <= groupSize; ++i) {
            // If the difference between consecutive elements is exactly k
            if (groupArray[i - 1] - groupArray[i - 2] === k) {
                subsetCounts[i] = subsetCounts[i - 1] + subsetCounts[i - 2];
            } else {
                // Else, the number of subsets is double of the previous
                subsetCounts[i] = subsetCounts[i - 1] * 2;
            }
        }

        // Update answer by multiplying by the number of subsets for this group
        answer *= subsetCounts[groupSize];
    }

    // Return the total number of K-free subsets
    return answer;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is as follows:

1. First, sorting the `nums` array which takes `O(n * log n)` time where `n` is the total number of elements in `nums`.
2. The for loop iterates over each number in sorted `nums` array and performs operations of constant time complexity to fill the dictionary `g`. This step runs in `O(n)` time.
3. For each mod group gathered in `g`, we calculate a dynamic programming solution which has a time complexity of `O(m)` where `m` is the length of that group in `g`.
4. Assuming that the elements are distributed uniformly across the groups, in the worst case, the size of each group could be close to `n`. Hence each dp calculation for a group in worst case takes `O(n)` time. This step is repeated for each unique modulus (up to `k` groups), leading to a complexity of `O(k * n)` in the worst case.
5. Multiplying the results of dynamic programming solutions for each group is constant for each iteration, adding only a negligible amount of time to the overall complexity.

Given the typical constraint where `k` is much smaller than `n`, the `O(n * log n)` term from sorting dominates the overall time complexity. Therefore, the reference answer's claim of `O(n * log n)` time complexity holds.

Space Complexity

The space complexity of the provided code is as follows:

1. The sorted array `nums` requires `O(n)` space.
2. The dictionary `g` which contains the elements of `nums` grouped by their modulus `k` can also take up to `O(n)` space in total.
3. The array `f` in the dynamic programming section is recreated for each group and is of size `m + 1` where `m` is the length of the current group being considered; this space would be reused for each group. The peak space taken by `f` is the size of the largest group, which is at most `n`.

Combining these two we see the space complexity is `O(n)` to account for the storage of sorted `nums` and the dictionary `g` with all its lists, matching the reference answer's claim of `O(n)` space complexity.