2597. The Number of Beautiful Subsets

Medium Array Dynamic Programming Backtracking

Problem Description

This problem involves analyzing a collection of positive integers, provided within an array `nums`, along with a given positive integer `k`. The objective is to determine the count of "beautiful" subsets that you can derive from the original array. A subset is considered beautiful if none of its element pairs have an absolute difference equal to `k`.

In more simple terms, you are tasked with finding every possible combination of numbers from the given array such that no two numbers in any combination are `k` units apart from each other. Remember that a subset means any selection of elements from `nums`, including possibly none - meaning, the full array `nums` itself can be a subset as long as no two of its elements have an absolute difference of `k`. Each subset is unique if it's derived from a unique set of indices chosen for deletion from `nums`.

Importantly, because subsets can vary in size from just one element up to the full length of the original array, there are potentially many such subsets, and you're required to report their count.

Intuition

The solution approach leverages recursive backtracking, which is a well-suited technique for problems involving combinatorial computations such as this one. The primary idea is to iterate through the array and make a decision at each step whether to include or exclude the current number from a subset we're building. We adapt this common pattern slightly by keeping track of the numbers we've included so far and their counts in a counter structure, which is essential for applying our beautiful subset rule regarding the absolute difference.

When we include a number in a subset, we check the counter to ensure that including it would not violate the condition of having an absolute difference of `k` with any other included numbers. If including it is valid, we add it to our counter, recurse to consider further numbers, and upon returning from the recursion, we backtrack by reverting the change to our counter to ensure it's ready for the next iteration.

In tackling the problem with backtracking, we systematically explore all subsets, validating and counting all the beautiful ones. The technique is exhaustive and ensures that we consider each possible subset.

The process begins with an answer counter set to `-1` to exclude the empty set from the solution, as specified by the problem statement, before starting our Depth-First Search (DFS) traversal. Through this traversal, we either pass over a number or include it, depending on whether it meets the beautiful subset condition. By the end, we'll have traversed all possibilities and accumulated the total count of valid subsets in our answer variable.

Solution Approach

The solution to this problem utilizes recursive backtracking alongside a counting mechanism, which together form a powerful approach to generate and validate the subsets of `nums`.

The centerpiece of this implementation is the `dfs` function, which stands for Depth-First Search. This recursive function is what allows us to consider every subset of `nums` for its beauty according to the provided condition related to `k`. The base case of this recursion occurs when all elements have been considered, at which point one valid subset path has been completed and the `ans` is incremented by one.

The `ans` variable is used to accumulate the number of beautiful subsets found. It is initialized to `-1` because the empty set is not counted, according to the problem statement.

The data structure `cnt` is a `Counter` from the Python collections library and it serves to maintain the count of the numbers included in the current subset being evaluated. The `Counter` provides us a fast way to check the existence and count of elements.

In each recursive call, there are two choices:

1. **Exclude the current number `i` from the subset.**
   - This doesn't impact the `cnt`, hence we simply call `dfs(i + 1)` and move on to the next number without changing the state.
2. **Include the current number `i` in the subset.**
   - Before we can make this choice, a validation check decides if the number can be included by ensuring that neither `nums[i] + k` nor `nums[i] - k` is currently in the subset (dictated by seeing if they are not in the `Counter`).
   - If the number passes validation, it's count in `cnt` is incremented by 1 and `dfs(i + 1)` is called to consider the next number.
   - After the recursive call returns, we backtrack by decrementing the count of `nums[i]` in `cnt` undo the last choice. This step is crucial and ensures that the state is maintained correctly for further exploration of subsets.

It is worth noticing that the decision to increment `ans` unconditionally with every completed and returned recursion path subtly captures all non-empty subsets' combinations.

Finally, the function `beautifulSubsets` returns the accumulated count in `ans`.

It must be mentioned that this recursive and backtracking algorithm is exhaustive, which means that it operates well for smaller inputs but may suffer from performance issues with large inputs due to the exponential number of possible subsets. Nonetheless, this brute-force approach cleverly uses counting to avoid illegal subsets and ensures a comprehensive search through the subset space.

Example Walkthrough

Let us consider a small example to illustrate the solution approach. Suppose we have an array `nums = [1, 3, 5]` and `k = 2`.

We want to count all the beautiful subsets of `nums` where no two numbers have an absolute difference equal to `k`. Here, we will be looking for subsets where no two numbers are exactly 2 units apart.

Initially, our `ans` is set to `-1` to exclude the empty set. We now perform DFS to explore all subsets.

1. Start with an empty subset, and empty counter `cnt`.
2. For the first element `1`, we have two choices:
   - **Exclude `1`:** Recursively continue with the next element of `nums`.
   - **Include `1`:** Update `cnt` by incrementing the count of `1`. Now, `cnt = Counter({1: 1})`. Recursively continue with the next element, ensuring not to include `3` in any subset because `3 - 1 = k`.
3. For the next element, `3`, following the same choices:
   - **Exclude `3`:** No change in `cnt`. Continue to the next element.
   - As we have `1` in our `cnt`, **including `3` is not valid** because `abs(3 - 1) = k`. We cannot proceed with this option.
4. Next, for the element `5`, we again have two choices:
   - **Exclude `5`:** If we previously excluded `3`, a subset `[1]` is counted, `ans = ans + 1`. We will proceed to backtrack and consider the other options.
   - **Include `5`:** This is valid if `1` was excluded. Now we have a subset `[1, 5]`, and `cnt` is updated to `Counter({1: 1, 5: 1})`, `ans = ans + 1`. We backtrack after this since there are no more elements to consider.

By the end, we will have considered all possible subsets, and our `ans` variable will contain the count of all non-empty beautiful subsets. For this example, the subsets we count are `[1]`, `[3]`, `[5]`, `[1, 5]`. Therefore, the returned `ans` is `3` (since we started with `-1`).

Now imagine we had not started with `ans = -1`, we would have also implicitly counted the empty subset, resulting in `ans` being `4`.

This walkthrough demonstrates how we utilize recursive backtracking to manage state and systematically explore all possible subsets, validating against the given condition and counting those that are beautiful.

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def beautifulSubsets(self, nums: List[int], k: int) -> int:
        # Helper depth-first search function to generate subsets and count the beautiful ones.
        def depth_first_search(index: int) -> None:
            nonlocal beautiful_count # This allows us to modify a variable outside of the nested function's scope.
            # Base case: check if we have considered all numbers.
            if index == len(nums):
                beautiful_count += 1 # Found a beautiful subset.
                return

            # Recursive case 1: Skip the current number and move to the next.
            depth_first_search(index + 1)

            # Recursive case 2: Include the current number if it can form a beautiful subset.
            if count[nums[index] - k] == 0 and count[nums[index] + k] == 0:
                count[nums[index]] += 1 # Include the current number in the count.
                depth_first_search(index + 1) # Move to the next number.
                count[nums[index]] -= 1 # Backtrack by removing the current number from the count.

        # Initialize the answer to -1 to account for the empty subset which is not considered beautiful.
        beautiful_count = -1
        count = Counter() # Counter to track the occurrences of numbers in the current subset.

        # Begin the depth-first search with the first number in the list.
        depth_first_search(0)

        return beautiful_count # Return the final count of beautiful subsets.

# Example usage:
# solution = Solution()
# result = solution.beautifulSubsets([1,2,3,4], 1)
# print(result) # Output will depend on the input list and the value of k
```

Java

```java
class Solution {
    private int[] nums; // Array of given numbers
    private int[] count = new int[1010]; // Frequency array to keep track of elements in the subset
    private int totalBeautifulSubsets = 0; // Total count of beautiful subsets found
    private int k; // The difference value used to determine beauty of a subset

    // Method to find the count of beautiful subsets from a given array with respect to 'k'
    public int beautifulSubsets(int[] nums, int k) {
        this.k = k;
        this.nums = nums;
        findBeautifulSubsets(0); // start the DFS with the first element in the array
        return totalBeautifulSubsets;
    }

    // Recursive method to perform DFS and find all subsets that are considered beautiful
    private void findBeautifulSubsets(int index) {
        // Base case: if we've considered all elements in 'nums'
        if (index == nums.length) {
            totalBeautifulSubsets++; // we've formed a subset, increment the beautiful subsets count
            return;
        }
        // Option 1: Exclude the current element and explore further subsets
        findBeautifulSubsets(index + 1);

        // Check if including nums[index] in the subset still keeps it beautiful
        boolean isBeautifulWithNext = nums[index] + k == count.length || count[nums[index] + k] == 0;
        boolean isBeautifulWithPrevious = nums[index] - k < 0 || count[nums[index] - k] == 0;

        // If both conditions are true, the number can be included in the subset.
        if (isBeautifulWithNext && isBeautifulWithPrevious) {
            count[nums[index]]++; // include the current element by incrementing its frequency
            findBeautifulSubsets(index + 1); // explore further subsets including this element
            count[nums[index]]--; // backtrack and exclude the current element
        }
    }
}
```

C++

```cpp
class Solution {
public:
    int beautifulSubsets(vector<int>& nums, int k) {
        int countBeautifulSubsets = -1; // Initialize the counter for beautiful subsets to -1.
        int countNums[1010]{}; // Array to keep track of the frequency of each number.
        int size = nums.size(); // Store the size of the input vector 'nums'.

        // Define a recursive lambda function to perform depth-first search for beautiful subsets.
        function<void(int)> dfs = [&](int index) {
            if (index >= size) { // Base case: If the end of the array is reached,
                ++countBeautifulSubsets; // increment the count of beautiful subsets.
                return; // Exit the current recursive call.
            }
            dfs(index + 1); // Recursive call to continue without including the current number.

            // Check if the number is considered beautiful in the context of the subset being formed.
            bool isBeautifulIncrement = nums[index] + k == 0 || countNums[nums[index] + k] == 0;
            bool isBeautifulDecrement = nums[index] - k < 0 || countNums[nums[index] - k] == 0;

            // If both conditions are true, the number can be included in the subset.
            if (isBeautifulIncrement && isBeautifulDecrement) {
                ++countNums[nums[index]]; // Increment the count for the current number.
                dfs(index + 1); // Recursive call to continue with the current number included.
                --countNums[nums[index]]; // Backtrack: Decrement the count after exploring.
            }
        };

        dfs(0); // Start the depth-first search from the first index.
        return countBeautifulSubsets; // Return the final count of beautiful subsets.
    }
};
```

TypeScript

```typescript
function beautifulSubsets(nums: number[], k: number): number {
    let answer: number = -1; // Initialize answer to -1 to exclude the base case.
    const count: number[] = new Array(1010).fill(0); // Count array with size 1010 to track occurrences.
    const len: number = nums.length; // Cache the length of nums.

    // Helper function to perform depth-first search.
    function dfs(index: number) {
        if (index >= len) {
            // If the end of the array is reached, increment the answer.
            answer++;
            return;
        }

        // Recursive case: proceed to next element without including current.
        dfs(index + 1);

        // Check if adding the current element is valid.
        const isAddableAbove: boolean = nums[index] + k >= 1010 || count[nums[index] + k] === 0;
        const isAddableBelow: boolean = nums[index] - k < 0 || count[nums[index] - k] === 0;

        // If the current element can be added without breaking the 'beautiful' condition,
        if (isAddableAbove && isAddableBelow) {
            count[nums[index]]++; // Increment the count for this number.
            dfs(index + 1); // Continue to next element with current included.
            count[nums[index]]--; // Decrement the count after backtracking.
        }
    }

    // Start depth-first search from index 0.
    dfs(0);
    return answer; // Return the total count of 'beautiful' subsets.
}
// Usage.
const nums = [1, 2, 3]; // Example array.
const k = 1; // Example difference.
console.log(beautifulSubsets(nums, k)); // Call the function with example inputs and log the result.
```

```python
from collections import Counter

class Solution:
    def beautifulSubsets(self, nums: List[int], k: int) -> int:
        # Helper depth-first search function to generate subsets and count the beautiful ones.
        def depth_first_search(index = 0):
            nonlocal beautiful_count # This allows us to modify a variable outside of the nested function's scope.
            # Base case: check if we have considered all numbers.
            if index >= len(nums):
                beautiful_count += 1 # Found a beautiful subset.
                return

            # Recursive case 1: Skip the current number and move to the next.
            depth_first_search(index + 1)

            # Recursive case 2: Include the current number if it can form a beautiful subset.
            if count[nums[index] - k] == 0 and count[nums[index] + k] == 0:
                count[nums[index]] += 1 # Include the current number in the count.
                depth_first_search(index + 1) # Move to the next number.
                count[nums[index]] -= 1 # Backtrack by removing the current number from the count.

        # Initialize the answer to -1 to account for the empty subset which is not considered beautiful.
        beautiful_count = -1
        count = Counter() # Counter to track the occurrences of numbers in the current subset.

        # Begin the depth-first search with the first number in the list.
        depth_first_search(0)

        return beautiful_count # Return the final count of beautiful subsets.

# Example usage:
# solution = Solution()
# result = solution.beautifulSubsets([1,2,3,4], 1)
# print(result) # Output will depend on the input list and the value of k
```

Time and Space Complexity

Time Complexity

The time complexity of the code is $O(2^n)$ where n is the length of the array `nums`. This is because in the worst case, for every element in the array, the `dfs` function is called twice: once when the element is included in the subset and once when the element is excluded. This results in a binary tree with 2^n leaves, representing all the possible subsets.

Space Complexity

The space complexity of the code is $O(n)$ due to the recursion depth and the additional space used by the `cnt` counter. In the worst case, the depth of the recursion call stack will be n when we keep including elements until we reach the end of the array. Moreover, `cnt` is a counter that keeps track of how many times an element appears in the current subset which takes at most $O(n)$ space if all elements are distinct.