2585. Number of Ways to Earn Points

Hard Array Dynamic Programming

Leetcode Link

Problem Description

In the given LeetCode problem, there is an exam composed of `n` types of questions. The objective is to find the number of different ways to achieve a specific score, referred to as `target`. The questions are presented as a 2D array, `types`, where each element `types[i]` contains two values: `count_i` and `marks_i`. `count_i` tells us the number of available questions of the `i`th type, while `marks_i` tells us the points awarded for each question of that type.

The challenge lies in combining these questions in such a way that the sum of the points equals the `target` score. A key detail to consider is that all questions of the same type are identical in terms of points and cannot be distinguished by sequence. The result should be calculated modulo (10^9 + 7) to handle large numbers.

For example, if you have three types of questions, each type offering 1, 2, and 3 points and the `target` is 5, you must figure out all possible combinations of those questions that add up to exactly 5 points.

Intuition

The solution uses a dynamic programming (DP) approach to tackle this problem methodically. DP is a strategy often used to solve counting problems like this, particularly when the problem involves making choices at several steps, and the objective is to count the total number of ways to achieve a certain outcome.

We define a DP matrix `f` where each entry `f[i][j]` represents the number of ways to score `j` points using only the first `i` types of questions. The dynamic programming table is filled in a bottom-up manner using the recurrence relation described in the solution approach.

To calculate the number of ways `f[i][j]`, we iterate through each possible number of questions we could answer of the `i`th type (from 0 to `count[i]`) and add the ways from the previous iteration (`i-1`) that would lead to the current score (`j`). This accumulation process ultimately gives us the number of ways to reach every potential score up to the `target`, using the available questions.

By carefully iterating over the candidates and updating our dynamic programming table, we can progressively build up the solution to include all types of questions. In the end, `f[n][target]` will give us the exact number of ways one could achieve the `target` score, where `n` is the total number of question types available in the exam.

Solution Approach

To implement the solution, we use a two-dimensional list (or array in some languages) `f` of size `(n + 1) * (target + 1)` as our dynamic programming table, where `n` is the number of question types.

This DP table `f` allows us to store intermediate results: `f[i][j]` will hold the number of ways to score exactly `j` points with the first `i` types of questions. The process begins by initializing `f[0][0]` to 1, which represents the fact that there is one way to score 0 points without any questions. All other entries in `f` start at 0, as initially, there are no known ways to reach scores higher than 0.

The solution iterates over each type of question `i` using a nested for loop. The outer loop runs through the question types, and the inner loops go through the target scores (from 0 to `target`) and the number of questions of the current type (from 0 to `count[i]`). For each question type, it updates the DP table entries for all possible scores `j` up to `target`.

Here is the key part of the algorithm:

```
for i in range(1, n + 1):
    count, marks = types[i - 1]
    for j in range(target + 1):
        for k in range(count + 1):
            if j - k * marks >= 0:
                f[i][j] = (f[i][j] + f[i - 1][j - k * marks]) % mod
```

In the above code block:

- `count` and `marks` are extracted from the `types` array for the `i`th question type.
- We need to check if the current target score `j` can be reached by answering `k` questions of type `i`, which is possible if `j` is greater than or equal to `k * marks`.
- If this condition is satisfied, we retrieve the count of ways to achieve the remaining score (`j - k * marks`), which is found in `f[i-1][j - k * marks]`. We add this count to `f[i][j]` to include the additional ways facilitated by the current question type.
- As the number of combinations can be very large, we use the modulo operation with `mod = 10**9 + 7` to avoid integer overflow and to keep the numbers within the specified limit.

After filling out the DP table, the final answer, which is the number of ways to reach exactly `target` points, is found at `f[n][target]`.

This dynamic programming approach provides an efficient way to solve the counting problem by breaking it down into smaller sub-problems and building up the solution incrementally.

Example Walkthrough

Let's consider an example where we have `n = 2` types of questions and the `target` score is 5. The available questions are given as `types = [[2, 3], [1, 5]]`, meaning we have two type 1 questions with 3 points each, and one type 2 question with 5 points.

Our dynamic programming table `f` will be initialized to zero with dimensions `(n + 1) * (target + 1)`, resulting in a 3×6 matrix since `n = 2` and `target = 5`. The first row `f[0]` and the first column `f[x][0]` for all x are initialized to signify the number of ways to achieve a score of 0 with different available types. In other words, `f[0][0]` will be set to 1 because there is only one way to achieve a score of 0 (by not selecting any questions), and all other cells in the first column are set to 1 as they signify the number of ways to reach a score of 0 with x available types.

Our table looks like this after initialization:

```
1 f[ ][0][1][2][3][4][5]
2 f[0][1][0][0][0][0][0]
3 f[1][ ][ ][ ][ ][ ][ ]
4 f[2][ ][ ][ ][ ][ ][ ]
```

Now, we fill the table using the approach described earlier:

1. For `i = 1`, we have `count = 2` and `marks = 3`. We iterate over `j` from `0` to `5` and `k` from `0` to `2`. This gives us:
   - For `j = 0`: `k` can be `0`, so `f[1][0] = 1`.
   - For `j = 1, 2, 4`: `k` can only be `0` since `j` is less than `marks * k` for `k > 0`.
   - For `j = 3`: `k` can be `0` or `1` (since 3 points can be achieved by one question of the first type), so `f[1][3] = f[0][3] + f[0][0]` (since `f[0][3]` is `0`, `f[1][3]` becomes `1`).
   - For `j = 5`: `k` can be `0` (impossible) or `1` (but we only have 3 points questions, so it's not possible either).

   The table now looks like this:

```
1 f[ ][0][1][2][3][4][5]
2 f[0][1][0][0][0][0][0]
3 f[1][1][0][0][1][0][0]
4 f[2][ ][ ][ ][ ][ ][ ]
```

2. For `i = 2`, `count = 1` and `marks = 5`. We follow the same process for `j` from `0` to `5` and `k` from `0` to `1`.
   - For `j = 5`: `k` can be `0` or `1` (since we can achieve 5 points with one question of the second type), so `f[2][5] = f[1][5] + f[1][0]` (since `f[1][5]` is `0`, `f[2][5]` becomes `1`).

   After filling the table, we have:

```
1 f[ ][0][1][2][3][4][5]
2 f[0][1][0][0][0][0][0]
3 f[1][1][0][0][1][0][0]
4 f[2][1][0][0][1][0][1]
```

In this example, the number of ways to achieve exactly `target = 5` points is found in `f[2][5]`, which is `1`. Therefore, there is one way to achieve the target score using the available questions.

Python Solution

```
from typing import List

class Solution:
    def waysToReachTarget(self, target: int, types: List[List[int]]) -> int:
        num_types = len(types)  # Total number of different types
        mod = 10**9 + 7  # Modulo value for the result to prevent integer overflow

        # Initialize 2D array dp to store the number of ways to accumulate each sum leading up to the target
        dp = [[0] * (target + 1) for _ in range(num_types + 1)]
        dp[0][0] = 1  # There is one way to reach target 0 using 0 types

        # Loop through each type
        for i in range(1, num_types + 1):
            count, value = types[i - 1]  # Extract the count and value from the current type tuple

            # Loop through all possible sums from 0 to the target
            for current_target in range(target + 1):
                # Loop through how many times to use the current type, from 0 to its count
                for times_used in range(count + 1):
                    # Only continue if the current sum minus the current total value of the type is not negative
                    if current_target >= times_used * value:
                        # Update the number of ways to reach the current sum modified by modulo
                        dp[i][current_target] = (
                            dp[i][current_target] + dp[i - 1][current_target - times_used * value]
                        ) % mod

        # Return the number of ways to reach the target using all types
        return dp[num_types][target]
```

Java Solution

```
class Solution {
    // Method to calculate the number of ways to reach a specific target using defined types
    public int waysToReachTarget(int target, int[][] types) {
        int numTypes = types.length; // Total number of types provided
        final int MOD = (int) 1e9 + 7; // Modulo value for the result to prevent overflow

        // 2D array for dynamic programming, where f[i][j] represents the number of ways to reach j using first i types
        int[][] dp = new int[numTypes + 1][target + 1];

        // Base case initialization: there's one way to achieve a target of 0 (not using any types)
        dp[0][0] = 1;

        // Iterate over all types
        for (int i = 1; i <= numTypes; ++i) {
            int count = types[i - 1][0]; // Max number available for current type
            int marks = types[i - 1][1]; // Marks that each type contributes to the target

            // Iterate over all sub-targets up to the main target
            for (int j = 0; j <= target; ++j) {
                // Try using 0 to count of the current type
                for (int k = 0; k <= count; ++k) {
                    // Ensure that the current composition does not exceed the sub-target j
                    if (j >= k * marks) {
                        // Update dp with the number of ways by adding the value from the previous type (i-1)
                        // remaining sub-target (j - k * marks), modulo MOD to manage large numbers
                        dp[i][j] = (dp[i][j] + dp[i - 1][j - k * marks]) % MOD;
                    }
                }
            }
        }

        // Return the result from the dp table that corresponds to using all types to reach exact target
        return dp[numTypes][target];
    }
}
```

C++ Solution

```
#include <vector>
#include <cstring>

class Solution {
public:
    // Function to calculate the number of ways to reach a target score using given types of scores
    int waysToReachTarget(int target, std::vector<std::vector<int>>& types) {
        int numTypes = types.size(); // Number of different score types
        const int MOD = 1e9 + 7; // Modulo value for avoiding integer overflow
        int ways[numTypes + 1][target + 1]; // 2D array for dynamic programming
        std::memset(ways, 0, sizeof(ways)); // Initialize all elements of ways to 0
        ways[0][0] = 1; // Base case: there's one way to reach a score of 0

        // Populating the 'ways' table using dynamic programming
        for (int i = 1; i <= numTypes; ++i) {
            int count = types[i - 1][0]; // Number of available scores of current type
            int marks = types[i - 1][1]; // The score value of the current type
            for (int j = 0; j <= target; ++j) {
                for (int k = 0; k <= count; ++k) {
                    if (j >= k * marks) {
                        // Adding the number of ways of reaching (j - k * marks) to ways[i][j]
                        ways[i][j] = (ways[i][j] + ways[i - 1][j - k * marks]) % MOD;
                    }
                }
            }
        }

        // Return the number of ways to reach the target score using all types
        return ways[numTypes][target];
    }
};
```

Typescript Solution

```
function waysToReachTarget(target: number, scoreTypes: number[][]): number {
    const numTypes = scoreTypes.length;  // Number of different score types
    const modulus = 10 ** 9 + 7;  // Modulus for the result to prevent overflow
    // Initialize a 2D array to store ways to reach each target for each type
    const ways: number[][] = Array.from({ length: numTypes + 1 }, () => Array(target + 1).fill(0));
    ways[0][0] = 1;  // Base case: one way to reach a target of 0 with 0 types

    // Iterate over each type of score
    for (let i = 1; i <= numTypes; ++i) {
        const [maxCount, scoreValue] = scoreTypes[i - 1];
        // Iterate over all possible targets from 0 to the desired target
        for (let currentTarget = 0; currentTarget <= target; ++currentTarget) {
            // Try using each count of the current score type up to its maximum
            for (let count = 0; count <= maxCount; ++count) {
                if (currentTarget >= count * scoreValue) {
                    // Update the number of ways to reach the current target
                    ways[i][currentTarget] =
                        (ways[i][currentTarget] + ways[i - 1][currentTarget - count * scoreValue]) % modulus;
                }
            }
        }
    }

    // Return the number of ways to achieve the target score using all score types
    return ways[numTypes][target];
}
```

Time and Space Complexity

The algorithm consists of three nested loops: the outermost loop runs through each type of question (`n` types in total), the middle loop iterates through all possible target scores up to `target`, and the innermost loop iterates up to `count + 1` times, where `count` represents the maximum number of questions of a given type.

The time complexity of this nested loop structure is `O(n * target * count)`. This is because for each of the `n` types, we perform `target + 1` iterations in the middle loop, and up to `count + 1` iterations in the inner loop for each iteration of the middle loop.

The space complexity arises from the 2D array `f`, which has dimensions `(n + 1) * (target + 1)`. Hence, the space complexity is `O(n * target)` since we need to store a computed value for every combination of question types and scores up to the target. The additional constants (`+1`) do not affect the asymptotic complexities and are therefore omitted in Big O notation.