879. Profitable Schemes

Hard · Array · Dynamic Programming · Leetcode Link

Problem Description

In this LeetCode problem, we are given `n`, which represents the number of members in a group. We also have a list of crimes, where each crime can generate a certain amount of profit (`profit[i]`) and requires a specific number of group members (`group[i]`) to participate. The constraint is that a member cannot participate in more than one crime.

Our goal is to find the number of different "profitable schemes" we can form. A profitable scheme is defined as a subset of these crimes that yields at least a certain `minProfit` while utilizing `n` or fewer members in total.

We need to calculate the total number of such subsets of crimes that meet the criteria and return this number modulo `10^9 + 7` to avoid large output values.

Intuition

To tackle this problem, we can consider it as a variation of the classic 0/1 knapsack problem. The two constraints we are dealing with are the total number of members we have (`n`) and the minimum profit we aim to achieve (`minProfit`). The twist here is that, unlike the classic knapsack problem where we typically have only one constraint (weight), we have two constraints (group size and profit).

We can approach this problem with either recursion with memoization or dynamic programming. The basic intuition for both methods is to consider each crime and decide whether we include it in our scheme or not. We make this decision based on whether including the crime will keep the total number of participants within `n` and help us reach at least `minProfit`.

Recursion with Memoization:

We recursively try to build our solution by defining a function, say `dfs(i, j, k)`, which tells us the number of schemes possible starting from the `i`-th job while having chosen `j` members and already accumulated a profit of `k`. We explore two possibilities at each step: including the current crime or excluding it. To prevent recalculations and improve efficiency, we use memoization to store intermediate results in a three-dimensional array.

Dynamic Programming:

Dynamic Programming (DP) offers a more systematic approach to this problem. Instead of recursively computing the answers, we iteratively build up a problem table `f` with dimensions representing the number of jobs considered so far, the number of employees used, and the current profit. The recurrence relation updates the table by considering the inclusion or exclusion of each job, much like the recursive approach, but done iteratively.

For the DP solution, we initialize our table with the understanding that, for zero jobs, we can only achieve zero profit with one scheme (doing nothing). We then iterate over each job and update the schemes count for various combinations of members used and profit gained, considering both options of including or excluding the crime.

Finally, whether we choose recursion with memoization or dynamic programming, the common intuition is to try each subset of crimes, enforce our constraints (not exceeding member limit and achieving at least `minProfit`), and count the number of valid schemes that arise from these choices.

Solution Approach

The problem has been approached with two algorithms: recursion with memoization and dynamic programming. Both methods aim to compute the number of ways in which crimes can be combined such that the total number of members used does not exceed `n` and the profit is at least `minProfit`. Here's how each approach works:

Recursion with Memoization:

The recursion with memoization approach uses a depth-first search (DFS) function named `dfs(i, j, k)` that represents number of schemes that can be formed starting from the `i`-th crime while having engaged `j` members and amassed a profit of `k`.

The base case of the recursion is when all crimes have been considered (`i == n`). If the accumulated profit `k` is greater than or equal to `minProfit` at this point, then we have found one valid scheme. If not, the scheme is invalid.

During the recursion, for each crime, we have the choice of either including it or not:

- If we choose not to include the crime, we simply move onto the next with `dfs(i + 1, j, k)`.
- If we do include it and have enough members left (`j + group[i] <= n`), the profit increases by `profit[i]` (but not exceeding `minProfit`) and the number of members increases by `group[i]`. This gives us `dfs(i + 1, j + group[i], min(k + profit[i], minProfit))`.

Memoization is used to store these results in a three-dimensional array `f[i][j][k]` to ensure that each unique state is only computed once. This greatly reduces the number of redundant calculations, thus optimizing the function.

Dynamic Programming:

The dynamic programming approach involves iteratively filling out a three-dimensional array `f[i][j][k]`, which holds similar meanings to the parameter of our `dfs` function. The dimensions `i`, `j`, and `k` respectively represent the number of jobs considered up to this point, the number of members involved, and the current accumulated profit.

Initialization is straightforward: `f[0][j][0] = 1`, signifying that having zero jobs only allows for a profit of `0`, which has one trivial scheme (doing nothing).

From there, for each job `i`, we enumerate through all possible employee counts `j` and profit values `k` to update the DP table based on our choices:

- If job `i` is skipped, `f[i][j][k]` remains the same as `f[i - 1][j][k]`.
- If job `i` is included, the count `f[i][j][k]` is increased by the count `f[i - 1][j - group[i - 1]][max(0, k - profit[i - 1])]`. This addition represents the schemes where job `i` contributes to the number of members and profit considering the constraints.

The answer to the problem would be the value of `f[m][n][minProfit]` after populating the table, as it represents the total number of schemes for `m` jobs with `n` members achieving at least `minProfit`.

Both the recursion with memoization and dynamic programming approaches provide a way to systematically traverse the search space of all possible crime scheme combinations, while ensuring the constraints are met, and efficiently count the number of valid schemes.

Example Walkthrough

Imagine a scenario where there are `n = 2` members in a group and the given list of crimes with their respective profits and group requirements is as follows: `profit = [1, 2]` and `group = [1, 2]`. The minimum profit we want to achieve is `minProfit = 2`.

Let's walk through the dynamic programming approach to see how this problem can be solved.

Initially, we start by creating a 3D array `f` with dimensions `[3][3][3]` representing the number of jobs (including a 'zero' job), the number of members (0 through 2), and the profit values (0 through 2), respectively.

We know that no profit can be made without any jobs, so we initialize `f[0][j][0] = 1` for `0 <= j <= n`. This reflects the fact that there is one way to achieve zero profit with any number of available members by not committing any crimes.

Now, using the DP approach, we iteratively update the array `f` as follows:

1. Consider job 1 (crime 1 with profit 1 requiring 1 member): We look at combinations of employee counts and profits, and update the table by considering if we include job 1 or skip it.

   Including job 1 when we have 0 members already used and 0 profit so far would update `f[1][1][1]`, indicating one scheme to achieve at least 1 profit with 1 member. Skipping job 1 would keep values same as previous, that is `f[0][0][0] = 1`.

2. Consider job 2 (crime 2 with profit 2 requiring 2 members): Now we proceed to update our array with the second job. This job requires both members to participate and yields the profit we want.

   If we include job 2 with 0 members and 0 profit used so far, we cannot do so because we don't have enough members (as it requires 2 and we only have 0). So, we skip to the entries where we have enough members.

   The entry `f[2][2][2]` is updated (now has value 1) representing one way to achieve a profit of 2 using both members by including the second job. Entries like `f[2][1][1]` remain as they were when derived from `job 1`, representing the available schemes to achieve profit 1 with 1 member, which cannot contribute to achieving profit 2 and so are not altered.

After populating our DP table, we look for `f[NUM_JOBS][NUM_MEMBERS][minProfit]` to find our answer. Here, `f[2][2][2]` represents the total number of schemes to achieve a profit of at least 2 while using up to 2 members, which is 1 (only by committing the second crime).

Therefore, the total number of profitable schemes for `n=2`, `profit=[1,2]`, `group=[1,2]`, and `minProfit=2` is 1, indicating there is only one way to achieve the minimum profit without exceeding the number of group members. This solution counts all distinct arrangements that meet our criteria modulo `10^9 + 7`.

Python Solution

```python
class Solution:
    def profitableSchemes(self, n: int, minProfit: int, group: List[int], profit: List[int]) -> int:
        # Define the modulus value as constant
        MOD = 10**9 + 7
        # Get the length of the group list which indicates the number of crimes
        num_crimes = len(group)
        # Create a 3D DP array with dimensions (num_crimes+1) x (n+1) x (minProfit+1)
        dp = [[[0] * (minProfit + 1) for _ in range(n + 1)] for _ in range(num_crimes + 1)]

        # Initialize the base case where for 0 crimes there's 1 way to make $0 with any number of people
        for j in range(n + 1):
            dp[0][j][0] = 1

        # Iterate over each crime
        for i, (members, gain) in enumerate(zip(group, profit), 1):
            # Loop over the number of people available from 0 to n
            for j in range(n + 1):
                # Loop over the range of profits from 0 to minProfit
                for k in range(minProfit + 1):
                    # Copy the value from the previous crime plan
                    dp[i][j][k] = dp[i - 1][j][k]
                    # Check if the number of members needed for the current crime is less than or equal to the number of people ava
                    if j >= members:
                        # Update the current state considering taking the current crime
                        dp[i][j][k] += dp[i - 1][j - members][max(0, k - gain)]
                        # Apply modulus to keep the value within the integer range
                        dp[i][j][k] %= MOD

        # Return the total number of ways to achieve at least minProfit
        return dp[num_crimes][n][minProfit]

```

Java Solution

```java
class Solution {
    public int profitableSchemes(int n, int minProfit, int[] group, int[] profit) {
        final int MOD = (int) 1e9 + 7; // Modulo for the final result to prevent overflow
        int m = group.length; // total number of crimes
        int[][][] dp = new int[m + 1][n + 1][minProfit + 1]; // dp array to store the results

        // Initialization: with 0 crimes, there is 1 way to get 0 profit with any number of members
        for (int j = 0; j <= n; ++j) {
            dp[0][j][0] = 1;
        }

        // Fill the dp table
        for (int i = 1; i <= m; ++i) { // for each crime
            for (int j = 0; j <= n; ++j) { // for each possible number of gang members
                for (int k = 0; k <= minProfit; ++k) { // for each profit from 0 to minProfit
                    // Counting the number of profitable schemes without the current crime
                    dp[i][j][k] = dp[i - 1][j][k];
                    // Counting profitable schemes including the current crime, if possible
                    if (j >= group[i - 1]) {
                        dp[i][j][k] = (dp[i][j][k] + dp[i - 1][j - group[i - 1]][Math.max(0, k - profit[i - 1])]) % MOD;
                    }
                }
            }
        }

        // The answer is the number of profitable schemes with ‘m’ crimes, using up to ‘n’ members
        // and achieving at least 'minProfit' profit.
        return dp[m][n][minProfit];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int profitableSchemes(int G, int P, vector<int>& group, vector<int>& profit) {
        // m is the number of crimes
        int m = group.size();
        // Initializing the 3D dynamic programming array with dimensions
        // m + 1 (for number of crimes), G + 1 (for gang members), and P + 1 (for minimum profit)
        int dp[m + 1][G + 1][P + 1];
        memset(dp, 0, sizeof(dp)); // Zero-initialize the dp array

        // Base case: for zero crimes, we have one way to achieve zero profit, irrespective of the number of members
        for (int j = 0; j <= G; ++j) {
            dp[0][j][0] = 1;
        }
        // Modulo for large numbers to avoid integer overflow
        const int MOD = 1e9 + 7;

        // Dynamic Programming to fill the dp array
        for (int i = 1; i <= m; ++i) {
            for (int j = 0; j <= G; ++j) {
                for (int k = 0; k <= P; ++k) {
                    // Case where we don't commit the i-th crime
                    dp[i][j][k] = dp[i - 1][j][k];
                    // Case where we commit the i-th crime, if there are enough gang members
                    if (j >= group[i - 1]) {
                        // We use max(0, k - profit[i - 1]) to ensure non-negative index when profit is higher than k
                        dp[i][j][k] = (dp[i][j][k] + dp[i - 1][j - group[i - 1]][max(0, k - profit[i - 1])]) % MOD;
                    }
                }
            }
        }
        // Returning the total number of profitable schemes with at most G members and at least P profit
        return dp[m][G][P];
    }
};

```

Typescript Solution

```typescript
// Define a constant for modulo as per the problem statement
const MOD = 1e9 + 7;

function profitableSchemes(G: number, P: number, group: number[], profit: number[]): number {

    // m represents the number of possible crimes
    const m = group.length;

    // Initialize a 3D dynamic programming array
    const dp: number[][][] = [...Array(m + 1)].map(() =>
        [...Array(G + 1)].map(() => Array(P + 1).fill(0))
    );

    // Base case: for zero crimes, there is one way to achieve zero profit
    for (let j = 0; j <= G; ++j) {
        dp[0][j][0] = 1;
    }

    // Main Dynamic Programming loop to populate dp array
    for (let i = 1; i <= m; ++i) {
        for (let j = 0; j <= G; ++j) {
            for (let k = 0; k <= P; ++k) {
                // Case where the i-th crime is not committed
                dp[i][j][k] = dp[i - 1][j][k];
                // Case where the i-th crime is committed, if enough gang members are available
                if (j >= group[i - 1]) {
                    // Calculating the new profit index, but ensuring non-negative index
                    const newProfit = Math.max(0, k - profit[i - 1]);
                    // Adding this scheme's count to the total
                    dp[i][j][k] = (dp[i][j][k] + dp[i - 1][j - group[i - 1]][newProfit]) % MOD;
                }
            }
        }
    }

    // Return the total count of schemes that can achieve at least P profit with at most G members
    return dp[m][G][P];
}

// Example usage of the function
// const G = 10; // Number of gang members
// const P = 5;  // Minimum profit required
// const group = [2, 3, 5]; // Group sizes required for each crime
// const profit = [6, 7, 8]; // Profit of each crime
// console.log(profitableSchemes(G, P, group, profit)); // Outputs the result

```

Time and Space Complexity

The time complexity of the provided code is `O(m * n * minProfit)`, where `m` represents the number of jobs, `n` represents the number of workers, and `minProfit` is the target minimum profit. This stems from the triple nested loops where the outermost loop runs for `m` jobs, the middle loop for `n + 1` workers (from `0` to `n`), and the innermost loop for `minProfit + 1` different profit targets (from `0` to `minProfit`).

The space complexity of the code is also `O(m * n * minProfit)`. This is due to the 3-dimensional array `f` that is being created to store results for subproblems. The dimensions of this array are `(m + 1) * (n + 1) * (minProfit + 1)`, corresponding to the number of jobs, workers plus one (to include the case of 0 workers), and minimum profit targets plus one (to include the case of 0 profit), respectively.