920. Number of Music Playlists

Hard | Math | Dynamic Programming | Combinatorics | Leetcode Link

Problem Description

In this problem, we're faced with the task of creating a music playlist with a certain number of constraints. You have a collection of `n` unique songs and your goal is to listen to `goal` songs during your trip. However, to keep the playlist engaging and avoid boredom, you need to follow two rules: firstly, you must play every song at least once, and secondly, you cannot replay a song unless `k` other different songs have been played in the interim.

The challenge is to calculate the total number of different playlists you can create that meet these conditions. Since the number of playlists can be incredibly large, the result must be reported modulo `10^9 + 7`, which is a common technique in computational problems to keep numbers within a manageable range and prevent overflow.

Intuition

To solve this puzzle, we make use of Dynamic Programming (DP), which is a method for solving complex problems by breaking them down into simpler subproblems. The fundamental concept is to create a two-dimensional array `f` with `goal + 1` rows and `n + 1` columns. Each element `f[i][j]` represents the number of playlists of length `i` that contain exactly `j` different songs.

We start by initializing our DP table with `f[0][0] = 1`, assuming that there is one way to create a playlist of length zero with zero different songs (the base case).

Next, we fill in the DP table row by row. For each position `f[i][j]`, we consider two scenarios:

1. We add a new song to the playlist, which we haven't listened to before. This can be done in `(n - j + 1)` ways because we have `(n - j + 1)` songs that haven't been played yet. This means the current number of playlists can be derived from `f[i - 1][j - 1]`.

2. We replay a song that has already been played, but only if `k` other songs have been played since its last occurrence. This can be done in `(j - k)` ways if `j > k` because we have `(j - k)` songs eligible for replay. We derive this possibility from `f[i - 1][j]`.

The two possibilities are added together to form the solution for `f[i][j]`, and we ensure to take the modulo `10^9 + 7` at each step to handle the large numbers.

After filling in the DP table, the answer that we're looking for will be in `f[goal][n]` which represents the number of different playlists of length `goal` that include all `n` different songs.

The key to understanding this approach is recognizing that we can make independent choices for each position in the playlist while respecting the constraints. Dynamic Programming is perfectly suited for this as it enables us to build the solution incrementally while reusing previously computed states.

Solution Approach

Following the intuition behind using Dynamic Programming to solve this problem, we can explain how the provided Python code implements the solution.

The algorithm makes use of a two-dimensional list `f` with dimensions `(goal + 1) x (n + 1)` to represent our DP table, where `f[i][j]` is the number of playlists of length `i` with `j` distinct songs.

Let's walk through the implementation steps:

1. **Initialization**: The DP table `f` is initialized to zero, and the base case `f[0][0]` is set to 1 (as there is one way to have a playlist of zero length that contains zero songs).

```python
f = [[0] * (n + 1) for _ in range(goal + 1)]
f[0][0] = 1
```

2. **Filling DP table**: We iterate over each possible playlist length `i` from 1 to `goal` and for each length, we consider the number of distinct songs `j` from 1 to `n`.

```python
for i in range(1, goal + 1):
    for j in range(1, n + 1):
```

3. **Calculating possibilities**: For each cell `f[i][j]`:

   - Add a new song: We look at the previous number of playlists with one fewer song `f[i - 1][j - 1]` and multiply by the number of new songs available to be played `(n - j + 1)`.

   - Replay a song: If there are more than `k` songs already played (`j > k`), we add the number of playlists from the previous song count `f[i - 1][j]` and multiply it by the number of songs that can be replayed `(j - k)`.

   This is aligned with the transition equation from the solution approach reference:

```
f[i][j] = f[i - 1][j - 1] * (n - j + 1) + f[i - 1][j] * (j - k), where i >= 1, j >= 1
```

   In the Python code, this is implemented concisely as:

```python
f[i][j] = f[i - 1][j - 1] * (n - j + 1)
if j > k:
    f[i][j] += f[i - 1][j] * (j - k)
f[i][j] %= mod
```

   The modulo operation ensures that we keep the numbers within the specified bounds to avoid overflow.

4. **Returning the result**: After filling in the entire table, we return the value of `f[goal][n]`, which gives us the total number of possible playlists we can create, fitting within the given constraints.

The time complexity of this solution is `O(goal * n)`, as we have a double for-loop iterating over `goal` and `n`, and the space complexity is also `O(goal * n)` due to the size of the DP table.

Overall, the use of dynamic programming enables us to solve this problem efficiently by building up the solution through state transitions, taking advantage of computed sub-solutions, and carefully considering the constraints of the problem within each step.

Example Walkthrough

Let's clarify the solution approach with a small example where `n = 3` unique songs, `goal = 3` total songs to listen to, and `k = 1`, which is the minimum number of different songs to play before a song can be repeated.

1. **Initializing the DP Table**: We create our table `f` with dimensions `4 x 4` (since we are including 0 in our indexing) and set `f[0][0] = 1`.

```
f = [
      [1, 0, 0, 0],
      [0, 0, 0, 0],
      [0, 0, 0, 0],
      [0, 0, 0, 0],
    ]
```

2. **Filling in the DP Table**: We loop through each song length `i` and each distinct song count `j`.

   For `i=1` and `j=1`: We add a new song to the playlist. The number of ways to choose one new song from `n` songs is `n - j + 1 = 3 - 1 + 1 = 3`. Therefore, `f[1][1] = f[0][0] * 3 = 3`.

   For `i=1` and `j=2` or `j=3`, we cannot create a playlist with one song that contains two or three unique songs, so these combinations are infeasible and remain `0`.

   Continuing the process, the table updates as follows:

```
f = [
      [1, 0, 0, 0],
      [0, 3, 0, 0],
      [0, 0, 6, 0],
      [0, 0, 0, 6],
    ]
```

   For `i=2, j=1`: This is not possible because we must have at least `i` different songs in playlist of length `i`.

   For `i=2, j=2`: There are two slots in the playlist and two unique songs that have not been played yet, so `f[2][2] = 3 * 2 = 6`.

   Since `j > k`, we can replay a song. Thus, for `i=2` and `j=2`, we can also add a song that was played once. There are `j - k` options, so `f[2][2]` also includes `f[1][2] * (2 - 1) = 0 * 1 = 0`. The total for `f[2][2]` remains `6`.

   The table becomes:

```
f = [
      [1, 0, 0, 0],
      [0, 3, 0, 0],
      [0, 6, 6, 0],
      [0, 0, 0, 6],
    ]
```

   And similarly, for `i=3, j=3`: We can pick a new unique song, and we can also use a previously used song, so `f[3][3]` is `f[2][2] * (3 - 2 + 1) + f[2][3] * (3 - 1) = 6 * 2 + 0 * 2 = 12`.

   Table finally looks like:

```
f = [
      [1, 0, 0, 0],
      [0, 3, 0, 0],
      [0, 6, 6, 0],
      [0, 0, 12, 12],
    ]
```

3. **Returning the result**: The number of different playlists we can create with `n = 3` unique songs in a goal of `3` songs is `f[3][3]`, which is `12`.

Here, the use of dynamic programming allows us to break the problem down into manageable chunks and cleverly count the number of different playlists by ensuring diversity in the songs played and respecting the constraints imposed by `k`.

Python Solution

```python
class Solution:
    def numMusicPlaylists(self, total_songs: int, playlist_length: int, min_diff_songs: int) -> int:
        # Define the modulus for taking the result modulo 10^9 + 7 as required.
        MOD = 10**9 + 7

        # Initialize a 2D list (dp table), where dp[i][j] represents the number of playlists of length i with exactly j unique songs
        dp = [[0] * (total_songs + 1) for _ in range(playlist_length + 1)]

        # There is one way to have a playlist of length 0 with 0 songs
        dp[0][0] = 1

        # Fill the dp table
        for i in range(1, playlist_length + 1):
            for j in range(1, total_songs + 1):
                # Case 1: Add a new song which hasn't been played before - multiply by the number of new songs available
                dp[i][j] = dp[i - 1][j - 1] * (total_songs - j + 1)

                # Case 2: Add a song which has been played before, but not in the last k songs, if j is large enough
                if j > min_diff_songs:
                    dp[i][j] += dp[i - 1][j] * (j - min_diff_songs)

                # Take modulo to avoid integer overflow
                dp[i][j] %= MOD

        # The result will be the number of playlists for the given length with the total number of songs
        return dp[playlist_length][total_songs]

```

Java Solution

```java
class Solution {

    // This function calculates the number of possible playlists that can be created
    // with 'n' different songs such that each playlist is 'goal' songs long, and
    // each song must not be repeated until 'k' other songs have played.
    public int numMusicPlaylists(int totalSongs, int playlistLength, int minDistance) {
        final int MOD = (int) 1e9 + 7; // The modulo value to keep numbers in a manageable range

        // 'dpTable' is a dynamic programming table where
        // dpTable[i][j] represents the number of playlist of length 'i' with 'j' unique songs.
        long[][] dpTable = new long[playlistLength + 1][totalSongs + 1];

        // Base case: 0 playlists of length 0
        dpTable[0][0] = 1;

        // Fill the dpTable, row by row, for all playlist lengths and song counts
        for (int i = 1; i <= playlistLength; ++i) {
            for (int j = 1; j <= totalSongs; ++j) {
                // If we are to add a new song to the playlist, multiply with the number of new songs left
                dpTable[i][j] = dpTable[i - 1][j - 1] * (totalSongs - j + 1);
                dpTable[i][j] %= MOD;

                // If we can reuse a song again, we add the case where the last song in the playlist
                // is a song we have already used, which is j - k permutations when j is greater than k
                if (j > minDistance) {
                    dpTable[i][j] += dpTable[i - 1][j] * (j - minDistance);
                    dpTable[i][j] %= MOD; // Apply modulo to keep it within the integer range
                }
            }
        }

        // Result is the number of playlists of length 'goal' using exactly 'n' unique songs
        return (int) dpTable[playlistLength][totalSongs];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the number of playlists that can be created
    int numMusicPlaylists(int numSongs, int playlistLength, int repeatK) {
        const int MOD = 1e9 + 7; // Defining the modulus value for large numbers
        vector<vector<long long>> dp(playlistLength + 1, vector<long long>(numSongs + 1, 0));
        // Initialize Dynamic Programming table with dp[length][songs]

        dp[0][0] = 1; // Base case: 0 songs for a 0 length playlist

        // Iterate through all playlist lengths from 1 to playlistLength
        for (int i = 1; i <= playlistLength; ++i) {
            // Iterate through all possible number of distinct songs from 1 to numSongs
            for (int j = 1; j <= numSongs; ++j) {
                // Case when a new song is added to the playlist
                dp[i][j] = dp[i - 1][j - 1] * (numSongs - j + 1) % MOD;
                // Case when we reuse a song that is not in the last k songs
                if (j > repeatK) {
                    dp[i][j] = (dp[i][j] + dp[i - 1][j] * (j - repeatK)) % MOD;
                }
            }
        }

        // Return the number of playlist of length playlistLength with numSongs unique songs
        return dp[playlistLength][numSongs];
    }
};

```

Typescript Solution

```typescript
function numMusicPlaylists(N: number, goal: number, K: number): number {
    const MOD = 1e9 + 7;
    const dp = new Array(goal + 1).fill(0).map(() => new Array(N + 1).fill(0));
    dp[0][0] = 1;

    // dp[i][j] will be the number of playlists of length i that have exactly j unique songs
    for (let i = 1; i <= goal; ++i) {
        for (let j = 1; j <= N; ++j) {
            // The last song of the playlist is a new song (not played in the last K songs)
            // Multiply by the number of new songs that can be placed here, which is (N - j + 1)
            dp[i][j] = dp[i - 1][j - 1] * (N - j + 1) % MOD;

            // If we have more than K unique songs to choose from, we can play a song that's not
            // played in the last K songs from the existing j songs.
            if (j > K) {
                // Multiply by the number of choices to pick from existing songs (j - K)
                dp[i][j] = (dp[i][j] + dp[i - 1][j] * (j - K)) % MOD;
            }
        }
    }
    // The result is the number of playlists of length 'goal' that have exactly 'N' unique songs
    return dp[goal][N];
}

```

Time and Space Complexity

Time Complexity

The time complexity of the algorithm is determined by two nested loops. The outer loop runs `goal` times, where `goal` is the total number of slots in the playlist. The inner loop runs `n` times, where `n` is the total number of unique songs. Thus, the total operations can be represented as `goal * n`.

For each pair `(i, j)`, the algorithm computes `f[i][j]` with at most two operations, one for each of the possible cases. Since each operation is computed in constant time, the overall time complexity is `O(goal * n)`.

Space Complexity

The original space complexity is due to the 2D list `f`, which has dimensions `[goal + 1]` by `[n + 1]`. Therefore, the space complexity is `O(goal * n)`.

According to the reference answer, we can optimize the space complexity by using a rolling array. A rolling array means we only maintain two rows at any time - the current row being calculated and the previous row. This optimization reduces the space complexity from `O(goal * n)` to `O(n)`, as we only need to store two rows each of `n + 1` elements, and at each step, we overwrite the previous row with the new one.