552. Student Attendance Record II

Hard | Dynamic Programming | Leetcode Link

Problem Description

The problem asks us to determine the number of possible attendance records of length `n` that allows a student to be eligible for an attendance award. A record is represented by a string consisting of the characters 'A' for absent, 'L' for late, and 'P' for present. A student qualifies for an award if they satisfy two conditions:

1. They are absent for fewer than 2 days in total.
2. They are not late for 3 or more consecutive days.

We must return the count of such attendance records modulo `10^9 + 7`, as this number could be very large.

Intuition

To arrive at the solution, we can use Dynamic Programming (DP). Since we need to keep track of absent and late counts, we can use a 3-dimensional DP array, where `dp[i][j][k]` represents the number of permutations of attendance records ending at day 'i', with 'j' representing whether the student has been absent 0 or 1 times, and 'k' representing the count of the latest consecutive late days (0, 1, or 2).

Thinking through the possible cases:

- The student can be present (P) on any given day, which doesn't affect their absences or increase the count of consecutive late days.
- They can be late (L), which resets the consecutive present day count but has to be carefully added so as not to surpass two consecutive late days.
- They can be absent (A), which increases the count of absences unless they have already been absent once.

The base case for the first day (i = 0) needs to be initialized to show that they can be present, late, or absent, but still be eligible for the award.

By iterating through the days and for each day computing the possible ending in 'P', 'L', or 'A', while adhering to the rules, we can incrementally construct our DP table up to day 'n'.

Finally, the sum of all possible records up to day 'n - 1' considering both absentee scenarios, and not exceeding two consecutive 'L's gives us the total count modulo `10^9 + 7`.

Solution Approach

The solution uses a dynamic programming approach to keep track of eligible attendance records by constructing a 3D DP array `dp` with dimensions `n`, `2`, and `3`; where `n` is the number of days, `2` represents being absent for either 0 or 1 day (`j` index), and `3` represents the number of consecutive late days (`k` index).

Here are the detailed steps of the algorithm:

1. **Initialization**: The index `[i][j][k]` in `dp` corresponds to the day `i`, absent status `j`, and consecutive late status `k`.
   - Day `i = 0`: Initialize `dp[0][0][0]`, `dp[0][0][1]`, and `dp[0][1][0]` to `1`. This covers the cases where the student is present, late, or absent for the first day.
2. **Filling the DP table**: Iterate over days `i` from `1` to `n - 1`:
   - **For a Present (P) day**: Update `dp[i][0][0]` and `dp[i][1][0]` to include all records from previous days where the student was present, late, or absent, ensuring that they are still eligible.
   - **For a Late (L) day**: The late status `k` is incremented by 1, but cannot exceed 2, which represents being late for 3 or more consecutive days. Set `dp[i][0][1]` to `dp[i - 1][0][0]`, `dp[i][0][2]` to `dp[i - 1][0][1]`, `dp[i][1][1]` to `dp[i - 1][1][0]`, and `dp[i][1][2]` to `dp[i - 1][1][1]`.
   - **For an Absent (A) day**: Only include records from the previous day where the student hasn't been absent before (`j=0`). Set `dp[i][1][0]` by summing up `dp[i - 1][0][0]`, `dp[i - 1][0][1]`, and `dp[i - 1][0][2]`.

   At each step, we take the result modulo `10^9 + 7` to avoid integer overflow due to large values.
3. **Computing the answer**: Sum up all the elements `dp[n - 1][j][k]`, where `j` can be `0` or `1`, and `k` can be `0`, `1`, or `2`, since we want to consider all possible eligible sequences. This sum gives the total number of attendance records of length `n` that allow a student to be eligible for an attendance award, modulo `10^9 + 7`.

Throughout the implementation, we're using the modulo operation due to the constraints that the answer may be very large and must be returned modulo `10^9 + 7` to fit within the integer range.

Example Walkthrough

Let's walk through an example using the provided solution approach to calculate the number of valid attendance records for `n = 3`.

1. **Initialization**:
   - We start with day `i = 0` and initialize our `dp` array such that `dp[0][0][0] = 1` (student was present), `dp[0][0][1] = 1` (student was late), and `dp[0][1][0] = 1` (student was absent).
2. **Filling the DP table**:
   - For day `i = 1`, we have:
     - Present (P) case: Since the student can be present after any attendance status of the previous day without restrictions, we set `dp[1][0][0] = sum(dp[0][0]) = 2` (from being P or L on the first day), and `dp[1][1][0] = sum(dp[0][1]) + sum(dp[0][0]) = 3` (from being P on the first day and A on the second, L on the first and P on the second, or A on the first and P on the second).
     - Late (L) case: The student can only be late for a maximum of two consecutive days, so we update `dp[1][0][1]` (was P now L) and `dp[1][1][1]` (was A now L), `dp[1][0][1] = dp[0][0][0] = 1` and `dp[1][1][1] = dp[0][1][0] = 1`.
     - Absent (A) case: The student can be absent for at most once, so `dp[1][1][0]` (was P now A) is already covered in Present case, no update needed for this case.
   - For day `i = 2`, we repeat the process:
     - Present (P) case: `dp[2][0][0] = sum(dp[1][0]) = 3` (from all P and L scenarios of the previous day where the student wasn't absent), and `dp[2][1][0] = sum(dp[1][1]) + sum(dp[1][0]) = 4` (from all P and L scenarios of the previous day and adding A scenarios where the student wasn't absent on previous days).
     - Late (L) case: Update the `dp` array considering the consecutive L scenarios, `dp[2][0][1] = dp[1][0][0] = 3` and `dp[2][0][2] = dp[1][0][1] = 1`; similarly for `j=1`, `dp[2][1][1] = dp[1][1][0] = 3` and `dp[2][1][2] = dp[1][1][1] = 1`.
     - Absent (A) case: `dp[2][1][0] = sum(dp[1][0]) = 3` (from all non-absent previous day records).
3. **Computing the answer**:
   - After filling in the `dp` table, we sum up all the possibilities on the last day `i = 2` considering both `j` (absent count) and `k` (consecutive late count), resulting in the final count: `dp[2][0][0] + dp[2][0][1] + dp[2][0][2] + dp[2][1][0] + dp[2][1][1] + dp[2][1][2] = 3 + 3 + 1 + 3 + 3 + 1 = 14`.

So, for `n = 3`, there are 14 valid attendance records that meet the criteria for an awards eligibility, and the final answer modulo `10^9 + 7` would be 14 (since 14 is already less than `10^9 + 7`).

Python Solution

```python
class Solution:
    def checkRecord(self, n: int) -> int:
        MOD = 10**9 + 7  # Define the modulus for the problem, to prevent overflow.
        # Initialize a 3D DP array where:
        # 1st dimention is the day,
        # 2nd dimention is the absence count (0 or 1, because more than 1 is not allowed),
        # 3rd dimention is the late count (0, 1, or 2, because more than 2 in a row is not allowed).
        dp = [[[0, 0, 0], [0, 0, 0]] for _ in range(n)]

        # Base cases: there is one way to have a sequence end in either 'P', 'L', or 'A' on the first day
        dp[0][0][0] = 1  # Present
        dp[0][0][1] = 1  # Late
        dp[0][1][0] = 1  # Absent

        # Iterate over each day starting from the second one.
        for i in range(1, n):
            # If the day ends in 'A' (Absent), the sequence must not have any 'A's before.
            # We can have 0, 1, or 2 'L's (Late) before this 'A'.
            dp[i][1][0] = sum(dp[i - 1][0][l] for l in range(3)) % MOD

            # If the day ends in 'L' (Late), the previous day can have 0 or 1 'L' or be 'P' (Present).
            dp[i][0][1] = dp[i - 1][0][0]  # One possible 'L' before current day
            dp[i][0][2] = dp[i - 1][0][1]  # Two possible 'L's before current day
            dp[i][1][1] = dp[i - 1][1][0]  # One 'L', with an 'A' at some point before
            dp[i][1][2] = dp[i - 1][1][1]  # Two 'L's, with an 'A' before

            # If the day ends in 'P' (Present), there are no constraints for this particular day.
            for j in range(2):  # j=0: no 'A' yet, j=1: there has been an 'A'
                dp[i][j][0] = sum(dp[i - 1][j][l] for l in range(3)) % MOD

        # Calculate the total number of valid attendance record combinations
        # by summing over all possibilities on the last day.
        total = 0
        for j in range(2):
            for k in range(3):
                total = (total + dp[n - 1][j][k]) % MOD
        return total  # Return the total number of valid combinations.

```

Java Solution

```java
class Solution {
    private static final int MOD = 1000000007;

    public int checkRecord(int n) {
        // dp[i][j][k]: number of valid sequences of length i, with j 'A's and a trailing 'L's of length k.
        long[][][] dp = new long[n][2][3];

        // Base cases
        dp[0][0][0] = 1; // P
        dp[0][0][1] = 1; // L
        dp[0][1][0] = 1; // A

        // Building the DP table for subsequences of length i
        for (int i = 1; i < n; i++) {
            // Adding 'A' to the sequence ending without 'A's and less than 2 'L's
            dp[i][1][0] = (dp[i - 1][0][0] + dp[i - 1][0][1] + dp[i - 1][0][2]) % MOD;

            // Adding 'L' to the sequence, considering previous 'L's and 'A's
            dp[i][0][1] = dp[i - 1][0][0];                 // Previous has no trailing 'L'
            dp[i][0][2] = dp[i - 1][0][1];                 // Previous has 1 trailing 'L'
            dp[i][1][1] = dp[i - 1][1][0];                 // Previous has an 'A' and no trailing 'L'
            dp[i][1][2] = dp[i - 1][1][1];                 // Previous has an 'A' and 1 trailing 'L'

            // Adding 'P' to the sequence, considering previous 'A's and 'L's
            dp[i][0][0] = (dp[i - 1][0][0] + dp[i - 1][0][1] + dp[i - 1][0][2]) % MOD;
            dp[i][1][0] = (dp[i][1][0] + dp[i - 1][1][0] + dp[i - 1][1][1] + dp[i - 1][1][2]) % MOD;
        }

        // Sum up all valid sequences
        long ans = 0;
        for (int j = 0; j < 2; j++) {     // 0 or 1 'A's
            for (int k = 0; k < 3; k++) { // 0 to 2 trailing 'L's
                ans = (ans + dp[n - 1][j][k]) % MOD; // Aggregate counts
            }
        }
        return (int) ans; // Final answer
    }
}

```

C++ Solution

```cpp
constexpr int MOD = 1e9 + 7;

class Solution {
public:
    int checkRecord(int n) {
        // Define 'll' as shorthand for 'long long' type
        using ll = long long;
        // Create a 3D vector to hold the state information
        // dp[i][j][k] represents the number of valid sequences of length i
        // where j tracks the absence count (0 for no A, 1 for one A)
        // and k tracks the late count (0, 1, or 2 consecutive L's)
        vector<vector<vector<ll>>> dp(n, vector<vector<ll>>(2, vector<ll>(3)));

        // base cases for first day
        dp[0][0][0] = 1; // P
        dp[0][0][1] = 1; // L
        dp[0][1][0] = 1; // A

        for (int i = 1; i < n; ++i) {
            // For A: append A after sequences that do not contain A ('j' == 0)
            dp[i][1][0] = (dp[i - 1][0][0] + dp[i - 1][0][1] + dp[i - 1][0][2]) % MOD;

            // For L: append L after sequences ending in no L or 1 L
            dp[i][0][1] = dp[i - 1][0][0]; // no L followed by L
            dp[i][0][2] = dp[i - 1][0][1]; // 1 L followed by another L
            dp[i][1][1] = dp[i - 1][1][0]; // no L followed by L, already contains A
            dp[i][1][2] = dp[i - 1][1][1]; // 1 L followed by another L, already contains A

            // For P: append P after any sequence
            dp[i][0][0] = (dp[i - 1][0][0] + dp[i - 1][0][1] + dp[i - 1][0][2]) % MOD;
            // for sequences that already contain A, append P
            dp[i][1][0] = (dp[i][1][0] + dp[i - 1][1][0] + dp[i - 1][1][1] + dp[i - 1][1][2]) % MOD;
        }

        // Calculate the final result by summing up all possible sequences of length 'n'
        ll result = 0;
        for (int absence = 0; absence < 2; ++absence) {
            for (int late = 0; late < 3; ++late) {
                result = (result + dp[n - 1][absence][late]) % MOD;
            }
        }
        return result;
    }
};

```

Typescript Solution

```typescript
// MOD constant for modulo operation to prevent overflow
const MOD: number = 1e9 + 7;

// A function to check the number of valid sequences of attendance records of length n
function checkRecord(n: number): number {
    // Define 'll' as alias for 'number' type since TypeScript doesn't have 'long long' type
    type ll = number;

    // Create a 3D array to hold the state information
    // dp[i][j][k] represents the number of valid sequences of length i
    // where j tracks the absence count (0 for no A, 1 for one A)
    // and k tracks the late count (0, 1, or 2 consecutive L's)
    let dp: ll[][][] = Array.from({ length: n }, () =>
        Array.from({ length: 2 }, () => Array(3).fill(0))
    );

    // Base cases for the first day
    dp[0][0][0] = 1; // P present
    dp[0][0][1] = 1; // L late
    dp[0][1][0] = 1; // A absent

    for (let i = 1; i < n; ++i) {
        // For A: append A after sequences that do not contain A ('j' == 0)
        dp[i][1][0] = (dp[i - 1][0][0] + dp[i - 1][0][1] + dp[i - 1][0][2]) % MOD;

        // For L: append L after sequences ending in no L or 1 L
        dp[i][0][1] = dp[i - 1][0][0]; // No L followed by L
        dp[i][0][2] = dp[i - 1][0][1]; // 1 L followed by another L
        dp[i][1][1] = dp[i - 1][1][0]; // No L followed by L, already contains A
        dp[i][1][2] = dp[i - 1][1][1]; // 1 L followed by another L, already contains A

        // For P: append P after any sequence
        dp[i][0][0] = (dp[i - 1][0][0] + dp[i - 1][0][1] + dp[i - 1][0][2]) % MOD;
        // For sequences that already contain A, append P
        dp[i][1][0] = (dp[i][1][0] + dp[i - 1][1][0] + dp[i - 1][1][1] + dp[i - 1][1][2]) % MOD;
    }

    // Calculate the final result by summing up all possible sequences of length 'n'
    let result: ll = 0;
    for (let absence = 0; absence < 2; ++absence) {
        for (let late = 0; late < 3; ++late) {
            result = (result + dp[n - 1][absence][late]) % MOD;
        }
    }
    return result;
}

```

Time and Space Complexity

Time Complexity

The algorithm uses a dynamic programming approach to calculate the number of ways a student can attend classes over `n` days without being absent for consecutive 3 days and without being absent for 2 days in total. The time complexity is determined by the triple-nested loops:

- The outer loop runs `n` times, which represents each day.
- For each day, there are 2 states of absence (either the student has been absent once or not at all), and 3 states for tardiness (the student has not been late, has been late once, or has been late twice).

Hence, the time complexity of the algorithm is `O(2 * 3 * n)`, which simplifies to `O(n)` because the constants can be removed in Big O notation.

Space Complexity

The space complexity is determined by the space required to store the dynamic programming states. The `dp` array is a two-dimensional array where the first dimension is `n`, and the second dimension is a 2×3 matrix to store all different states for absences and lates.

Therefore, the space complexity is `O(2 * 3 * n)`, which simplifies to `O(n)` as the constants can be disregarded.