2147. Number of Ways to Divide a Long Corridor

Hard Math String Dynamic Programming Leetcode Link

Problem Description

The corridor in the library has a number of seats and decorative plants arranged in a line. This arrangement is represented as a string `corridor` where the character `'S'` represents a seat and `'P'` represents a plant. There's already one room divider at each end of the corridor. Our goal is to add more dividers in between elements of the corridor so we can divide the corridor into sections. Each section must contain exactly two seats. There is no constraint on the number of plants a section can have. We have to find the total number of different ways we can add dividers to achieve this. Two ways of adding dividers are considered different if the position of any divider is different between the two. The final answer needs to be returned as a modulo of `10^9 + 7`. If it is not possible to divide the corridor according to the given rules, we must return `0`.

Intuition

To solve this problem, we adopt a recursive approach with memoization, commonly known as dynamic programming. The intuition here is that if we encounter a seat ('S'), we track it. If by adding this seat, we form a pair of seats (meaning we have encountered two seats so far), we have two choices: either put a divider right after this pair of seats or move to the next symbol in the corridor. If we decide to put a divider, we reset our seat count to start forming new sections. If we come across a plant ('P'), it doesn't affect the count of the seats but we still move forward making further recursive calls.

We use the `dfs` function which is defined as `dfs(i, cnt)`. Here `i` is the current position in the corridor string and `cnt` is the current count of seats we've encountered so far without hitting a pair. If `cnt` exceeds two we return `0` since any section having more than two seats is invalid. When `i` reaches the end of the corridor and we have exactly two seats in the current section (`cnt == 2`), this is one valid way so we return `1`. Otherwise, we return `0` since it's an incomplete or invalid state.

We continue this recursive approach until we've explored all possibilities. To speed up the process, we use memoization (`@cache` decorator) to remember results of subproblems that we have already solved to avoid redundant calculations.

Each time we have a pair of seats, we can add the current number of ways to the final answer since it represents a branching point where we can make a choice to add or not add a divider. The final answer could be very large, hence we use modulo operation with `10^9 + 7` to keep our final count within integer limits.

Solution Approach

The solution to the problem employs recursive depth-first search (DFS) with memoization, a common dynamic programming technique, to explore all the possible ways to divide the corridor while ensuring the business rule of exactly two seats per section is met. The code defines a helper function `dfs(i, cnt)` which uses the following approach:

1. **Base Case:** When the recursion reaches the end of the string (i.e., `i == n`), it checks if the count of seats `cnt` is exactly `2`. If it is, it returns `1`, indicating one valid division is found. Otherwise, it returns `0`.

2. **Seat Counting:** It increments the count `cnt` if the current character is a seat (`'S'`). If this count goes above `2`, it returns `0` immediately, as more than two seats would violate the corridor division rule.

3. **Recursion:** There are two recursive calls within the `dfs` function:
   - The first call, `dfs(i + 1, cnt)`, is made to explore the possibility of not placing a divider after the current index, regardless of how many seats have been seen so far (as long as it's not more than two).
   - The second call, `dfs(i + 1, 0)`, is made only when `cnt` is exactly `2`, indicating that a section with two seats has been completed, and a divider may be installed here. This recursive call starts the count over from zero for the next section.

4. **Memoization:** The `@cache` decorator is used above the `dfs` function to memoize the results. This ensures that once a particular state (`i`, `cnt`) has been computed, it will not be computed again, which significantly reduces the number of calculations.

5. **Modulo Operation:** The result of each recursive call (if a divider is placed) is taken modulo `10^9 + 7`. This is done because the final answer may be very large, and we want to avoid integer overflow by keeping the number within the specified limits.

6. **Combination Summation:** The function `dfs` keeps a running total of valid ways to divide the corridor as it recursively explores each possibility. Every time a section completes with two seats, it adds to the possible combinations and continues to the next recursive case.

The main function initializes the required parameters (such as the length of the corridor `n` and the modulo `mod`) and starts the recursion by calling `dfs(0, 0)`, which indicates starting from the first index with a seat count of zero.

After exploring all options, the `dfs.cache_clear()` is called to clear the cache, which is a good practice to release the memory used by the cache when it is no longer needed, and the total number of ways (`ans`) is returned.

Example Walkthrough

Let's illustrate the solution approach using a small example corridor string `corridor = "SSPPSPS"`. We aim to find the total number of ways we can divide this corridor into sections so that each section contains exactly two seats.

Here's a step-by-step walkthrough of the solution approach:

1. We initialize our recursion with `dfs(0, 0)` which means we start at index `0` with `0` seats counted so far.

2. We encounter the first seat at index `0`. Now our call is `dfs(1, 1)` since we've found `1` seat.

3. The next character at index `1` is also a seat. We're now calling `dfs(2, 2)` and have a branching point. We can:
   - Place a divider (`dfs(3, 0)`) because we completed a section with exactly two seats.
   - Not place a divider and just continue to the next character.

4. Assuming we place a divider, our new state is `dfs(3, 0)`. At index `3`, we find a plant and the call becomes `dfs(4, 0)`.

5. We reach another 'S' at index `4`, making our state `dfs(5, 1)`.

6. At index `5`, we find a plant, and again, continue to `dfs(6, 1)`.

7. At index `6`, we reach the last seat, and our state is `dfs(7, 2)`. Since we have two seats, we have two options:
   - End the recursion (returning `1`) because we are at the end (`i == n`) and we have a valid section.
   - Call `dfs(7, 0)` to start looking beyond the last index for more seats, which is not possible since we're at the end.

8. We can now backtrack to previous choices where we didn't place a divider and look at potential divisions there, applying the same decision process.

9. Throughout the exploration, the `@cache` decorator ensures we don't recompute states we've already processed.

10. Once all possible paths are walked through, the total number of ways is summed up, taking into account the modulo `10^9 + 7` to keep the numbers within bounds.

For each index we are considering, we have two main choices leading to two recursive calls: advance with the same count of seats or reset the seat count if we placed a divider. This process will explore all valid combinations of placing dividers.

In this specific example, we have two places where we can potentially place additional dividers (after the first and the last seat). So the example `corridor = "SSPPSPS"` can be divided in the following ways:

- "SS" | "PPSPS"
- "SSPPSPS"

Therefore, there are `2` ways to divide the corridor into sections with exactly two seats each.

Python Solution

```python
from functools import lru_cache  # Importing cache mechanism for memoization

class Solution:
    def numberOfWays(self, corridor: str) -> int:
        # This function uses depth-first search to count the number of ways
        # to partition the corridor keeping the constraints in mind.
        @lru_cache(None)  # Using least recently used cache for memoization
        def dfs(index, seat_count):
            # If we've reached the end of the corridor, return 1 if we have exactly 2 seats,
            # otherwise return 0 since the current pathway is invalid.
            if index == corridor_length:
                return int(seat_count == 2)

            # Increment seat_count if the current position has a seat 'S'
            seat_count += corridor[index] == 'S'

            # If the count exceeds 2, we cannot make any more partitions, thus return 0
            if seat_count > 2:
                return 0

            # Continue searching along the corridor with the current seat count
            ways = dfs(index + 1, seat_count)

            # If we have exactly 2 seats so far, we have an option to partition here
            if seat_count == 2:
                ways += dfs(index + 1, 0)  # Reset seat_count after partitioning
                ways %= mod  # Take mod to prevent overflow

            return ways

        corridor_length = len(corridor)  # Store the length of the corridor
        mod = 10**9 + 7  # Modulus to avoid integer overflow issues
        result = dfs(0, 0)  # Start the DFS from index 0 with seat count 0
        dfs.cache_clear()  # Clear the cache after computation
        return result  # Return the total number of ways to partition the corridor

# This code can now be used as follows:
# sol = Solution()
# print(sol.numberOfWays("SSPPSPS"))

```

Java Solution

```java
import java.util.Arrays; // Import Arrays to use Arrays.fill()

class Solution {
    // Variables used across methods
    private String corridor;
    private int corridorLength;
    private int[][] memoization;
    private static final int MODULO = (int) 1e9 + 7; // Using a more descriptive constant name

    public int numberOfWays(String corridor) {
        this.corridor = corridor;
        corridorLength = corridor.length();
        memoization = new int[corridorLength][3]; // Store the computed values for dynamic programming

        // Initialize memoization array with -1, which signifies that the value has not been computed
        for (var element : memoization) {
            Arrays.fill(element, -1);
        }

        // Start the search from position 0 with 0 'S' plants encountered
        return depthFirstSearch(0, 0);
    }

    // Helper method for depth-first search
    // 'index' represents the current index in the corridor string
    // 'seatCount' counts the number of 'S' plants encountered
    private int depthFirstSearch(int index, int seatCount) {
        // If we reach the end of the string
        if (index == corridorLength) {
            // Check if we have exactly 2 'S' plants, which is required to form a valid section
            return seatCount == 2 ? 1 : 0;
        }

        // If adding a seat exceeds the allowed number for a valid section, return 0
        seatCount += corridor.charAt(index) == 'S' ? 1 : 0;
        if (seatCount > 2) {
            return 0;
        }

        // Check for memoized results to avoid recomputation
        if (memoization[index][seatCount] != -1) {
            return memoization[index][seatCount];
        }

        // Continue to the next seat
        int possibleWays = depthFirstSearch(index + 1, seatCount);

```

```
        // If there are 2 'S' plants, we have an option to start a new section
        if (seatCount == 2) {
            possibleWays += depthFirstSearch(index + 1, 0);
            // Ensure the result is within the modulo range
            possibleWays %= MODULO;
        }

        // Memoize the calculated ways for current state
        memoization[index][seatCount] = possibleWays;
        return possibleWays;
    }
}

```

C++ Solution

```
class Solution {
public:
    const int MOD = 1e9 + 7; // Constant for the modulus operation

    // Function to compute the number of ways to sit people in the corridor
    int numberOfWays(string corridor) {
        int corridorLength = corridor.size(); // Get the length of the corridor
        vector<vector<int>> dp(corridorLength, vector<int>(3, -1)); // Dynamic programming table initialized with -1, representing

        // Declare the depth-first search function to find the number of ways
        // cnt represents the current number of people sitting consecutively. It must not exceed 2.
        function<int(int, int)> dfs = [&](int index, int count) {
            // If we reach the end of the string, return 1 if count is 2, else 0
            if (index == corridorLength) return count == 2 ? 1 : 0;

            // Increment the count if we find a seat (represented by 'S')
            count += corridor[index] == 'S';

            // If count exceeds 2, return 0 as it's an invalid placement
            if (count > 2) return 0;

            // Return the precomputed value if it's already calculated
            if (dp[index][count] != -1) return dp[index][count];

            // Recursive case: continue to the next index with the current count
            int ways = dfs(index + 1, count);

            // If we have exactly 2 people sitting, we can reset the count and add the ways from this point
            if (count == 2) {
                ways += dfs(index + 1, 0);
                ways %= MOD; // Ensure the result stays within the bounds of the modulus
            }

            // Save the computed value in the dynamic programming table before returning
            dp[index][count] = ways;
            return ways;
        };

        return dfs(0, 0); // Start the DFS from index 0 with 0 people sitting consecutively
    }
};

```

Typescript Solution

```
// Constant for the modulus operation
const MOD: number = 1e9 + 7;

// Function to compute the number of ways to sit people in the corridor
function numberOfWays(corridor: string): number {
    let corridorLength: number = corridor.length; // Get the length of the corridor
    let dp: number[][] = Array.from({ length: corridorLength }, () => Array(3).fill(-1)); // Dynamic programming table initialized

    // The depth-first search function finds the number of ways
    // `count` represents the current number of people sitting consecutively. It must not exceed 2.
    const dfs = (index: number, count: number): number => {
        // If we reach the end of the string, return 1 if count is 2, else 0
        if (index === corridorLength) return count === 2 ? 1 : 0;

        // Increment the count if we find a seat (represented by 'S')
        count += corridor[index] === 'S' ? 1 : 0;

        // If count exceeds 2, return 0 as it's an invalid placement
        if (count > 2) return 0;

        // Return the precomputed value if it's already calculated
        if (dp[index][count] !== -1) return dp[index][count];

        // Recursive case: continue to the next index with the current count
        let ways = dfs(index + 1, count);

        // If we have exactly 2 people sitting, we can reset the count and add the ways from this point
        if (count === 2) {
            ways += dfs(index + 1, 0);
            ways %= MOD; // Ensure the result stays within the bounds of the modulus
        }

        // Save the computed value in the dynamic programming table before returning
        dp[index][count] = ways;
        return ways;
    };

    return dfs(0, 0); // Start the DFS from index 0 with 0 people sitting consecutively
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code mainly comes from the recursive function `dfs`. The function has two parameters: the index `i` and the count of 'S' seen so far `cnt`. The recursion has three cases:

1. Base case when `i == n`, which takes O(1) time.
2. The recursive case without changing `cnt`, which in the worst case, can be called for each character in the `corridor` string—resulting in O(n).
3. The recursive case where `cnt == 2` can double the number of calls, as it leads to another call with `cnt` reset to `0`.

With memoization, we prevent recalculating the results for the same state `(i, cnt)`. There are `n` different values for `i` and 3 values for `cnt` (0, 1, and 2), giving us at most `O(n)` states.

Each state requires constant time to process, except for the recursive calls. Since memoization ensures we compute each state at most once, and the recursion fan-out is capped (only two recursive calls per invocation), the overall time complexity is O(n).

Space Complexity

The space complexity includes the space for the recursive stack and the memoization cache.

1. Recursive stack: In the worst case, there could be O(n) calls in the call stack since `dfs` gets called with each increasing index `i` up to `n`.
2. Memoization cache: We store a result for each possible state `(i, cnt)`. As established, there are `n` positions for `i`, and `cnt` can have 3 values leading to `O(n)` space complexity for the cache.

Therefore, the overall space complexity of the code is also O(n), with the primary contributions being the recursion stack and the memoization cache.