1269. Number of Ways to Stay in the Same Place After Some Steps

Hard · Dynamic Programming · Leetcode Link

Problem Description

In this problem, we are given an array of size `arrLen`, and we start with a pointer at index `0`. We can take a total of `steps` number of moves, and at each move, we have three choices:

- Move the pointer to the left (if the pointer is not at the leftmost position),
- Move the pointer to the right (if not at the rightmost position),
- Or don't move the pointer at all.

The goal is to calculate the number of unique ways we can make these moves so that after exactly `steps` steps, our pointer is returned to the starting position at index `0`. Because the answer can potentially be very large, we will return the result modulo `10^9 + 7`, which is a common technique to prevent integer overflow in programming contests.

Intuition

The problem is a perfect candidate for dynamic programming because it deals with counting ways and optimizing over a range of choices at each step.

Intuitively, we can approach this using a depth-first search (DFS) strategy that explores all possible movements from a given position at a step. However, a naive DFS solution would be too slow because it involves a lot of repeated calculations. Therefore, we can apply memoization to cache the results of subproblems and avoid redundant computations.

The DFS function `dfs(i, j)` keeps track of the current index `i` and the remaining steps `j`. Here are the key points:

- If the pointer is outside the array bounds (`i < 0` or `i >= arrLen`) or if the number of steps remaining is not sufficient to return to the start (`i > j`), there are no valid ways, so we return `0`.
- If the pointer is at the starting position and no more steps remain (`i == 0` and `j == 0`), we've found a valid way, so we return `1`.
- Otherwise, we recursively call `dfs` for all potential moves: stay in place, move left, or move right. We do this by decreasing the number of remaining steps `j` by one and either keeping the index `i` the same or adjusting it by `-1` or `1` depending on the move.

We sum the results from the three recursive calls, apply the modulo operation to keep the number within the bounds, and then return the sum as the answer from the current state. The `@cache` decorator in Python ensures that once a subproblem has been solved, its result is stored and can be reused in subsequent calls without recalculating.

Our main function `numWays` initializes the modulo value and calls the helper function `dfs` starting from the first position with all steps available, and returns the total count of unique ways.

Solution Approach

The solution makes use of recursive depth-first search (DFS) combined with memoization, which is a form of dynamic programming where we remember the results of the subproblems we have already solved.

DFS and Recursion

The `dfs(i, j)` function is the heart of our solution. It recursively searches through all potential moves starting from a given index `i` with `j` steps remaining. The recursion allows the function to branch out into every possible move scenario. When we reach the base cases — either stepping out of the array bounds or reaching index 0 with no steps left — the function stops and returns either 0 (not a valid scenario) or 1 (a valid scenario), respectively.

Memoization with Caching

To optimize the recursive calls, we use the `@cache` decorator from the Python `functools` module. This automatically stores any results returned by `dfs(i, j)` so that if the function is called again with the same arguments `i` and `j`, it does not compute everything from scratch but instead returns the cached result. This greatly reduces the number of calculations and thus speeds up the execution.

Modulo Operation

The `mod` variable is set to `10**9 + 7`, a large prime number, and it's used to perform modulo operations to keep the numbers we work with within the boundaries of integer limits. This is a common approach in problems involving counting, where the numbers can grow exponentially. In our code, after updating the `ans` (the number of ways to return to index `0`), we take `ans % mod` to ensure that we only keep the remainder and avoid integer overflow.

Execution

The helper function `dfs` is called from the `numWays` method with parameters `0` and `steps`, which signify that we start at index `0` with all `steps` remaining. By exploring all possibilities and pruning unnecessary ones using memoization, we are able to calculate the exact number of ways to return to the start after the stipulated number of steps.

```python
class Solution:
    def numWays(self, steps: int, arrLen: int) -> int:
        @cache
        def dfs(i, j):
            if i > j or i >= arrLen or i < 0 or j < 0:
                return 0
            if i == 0 and j == 0:
                return 1
            ans = 0
            for k in range(-1, 2):
                ans += dfs(i + k, j - 1)
                ans %= mod
            return ans

        mod = 10**9 + 7
        return dfs(0, steps)
```

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have `arrLen = 4` and `steps = 3`. This means we have an array `[0, 1, 2, 3]` (indices of the array) and we can make three moves. We want to find out in how many unique ways we can return to index `0`.

1. We start at `(index = 0, steps = 3)` and call `dfs(0, 3)`.
2. From here, we can move to index `1`, stay at index `0`, or try to move to the left, which isn't possible since we're at the leftmost position.
3. When we call `dfs(1, 2)` (move to the right):
   - We can move back to index `0`, go further to index `2` (which isn't wise because we don't have enough steps to come back), or stay at index `1`.
   - The recursive call `dfs(0, 1)` from this position will lead us back to the starting index with one step remaining. Here, we can only stay in place for the last step.
   - Hence, from the move to index `1` and back, we get one unique way to return to the starting point.
4. When we call `dfs(0, 2)` (stay in place):
   - We are again presented with the choice to stay, move right to index `1`, or move left (not possible).
   - If we stay and call `dfs(0, 1)`, our next step can only be to stay again at index `0`, which gives us another unique way.
   - If we move to index `1` and call `dfs(1, 1)`, the only option is to move back to index `0`, which we can do.
   - Thus, staying in place on the first move gives us two more unique ways.
5. We wouldn't consider moving left from the starting position since it is not possible.
6. Each call to `dfs` saves its result, so any overlapping subproblems don't require re-computation.
7. After exploring all possibilities, we find that there are `1 + 2 = 3` unique ways to return to index `0` in `3` steps for an `arrLen` of `4`.

Putting this into the context of our algorithm, the recursive calls and the memoization ensure that we efficiently count all unique ways with the DFS approach, considering the implications of each possible move at every step. The modulo operation is applied to each update to the number of ways, which keeps our numbers within the bounds of typical integer values.

Python Solution

```python
class Solution:
    def numWays(self, steps: int, arrLen: int) -> int:
        # The function dfs computes the number of ways to end at index i after j steps
        from functools import lru_cache
        @lru_cache(maxsize=None)
        def dfs(current_index: int, remaining_steps: int) -> int:
            # If out of bounds or more steps than the array length, return 0
            if current_index > remaining_steps or current_index >= arrLen or current_index < 0 or remaining_steps < 0:
                return 0
            # Base case: If at the start and no steps left, there's 1 way
            if current_index == 0 and remaining_steps == 0:
                return 1
            # Initialize the number of ways to 0
            number_of_ways = 0
            # Iterate over the three directions: left, stay, right
            for move in range(-1, 2):
                number_of_ways += dfs(current_index + move, remaining_steps - 1)
                number_of_ways %= mod # Modulo operation for each addition to avoid overflow
            return number_of_ways

        # Define the modulo constant
        mod = 10**9 + 7
        # Call the dfs function starting from index 0 and with given steps
        return dfs(0, steps)
```

Java Solution

```java
public class Solution {
    private Integer[][] memoizationCache;
    private int arrayLength;

    public int numWays(int steps, int arrLen) {
        // Initializing the memoization cache with the number of steps and steps + 1
        // because the farthest we can go is being equal to the number of steps if we move only in one direction.
        memoizationCache = new Integer[steps][steps + 1];
        arrayLength = arrLen;
        // This is a DFS starting at index 0 with the number of steps available.
        return dfs(0, steps);
    }

    private int dfs(int currentPosition, int remainingSteps) {
        // Base case conditions:
        // If the position is beyond the number of remaining steps or array length,
        // or if the position is negative or there are no steps left, we return 0 as no way can be formed.
        if (currentPosition > remainingSteps || currentPosition >= arrayLength || currentPosition < 0 || remainingSteps < 0) {
            return 0;
        }

        // If we are at the starting position and there are no steps remaining,
        // this is a valid way to finish within the array bounds.
        if (currentPosition == 0 && remainingSteps == 0) {
            return 1;
        }

        // If we have already computed the number of ways from this position with the remaining steps,
        // we can return the cached result to save computation time.
        if (memoizationCache[currentPosition][remainingSteps] != null) {
            return memoizationCache[currentPosition][remainingSteps];
        }

        // This variable will accumulate the number of ways we can end at the starting position.
        int totalWays = 0;
        // The modulus value given in the problem statement to prevent integer overflow.
        final int mod = (int) 1e9 + 7;

        // We iterate through three possibilities - moving left, staying in place, or moving right.
        for (int stepDirection = -1; stepDirection <= 1; ++stepDirection) {
            // The totalWays is the sum of ways from the new position after taking the step
            // We use modulo operation to keep the result within integer limits.
            totalWays = (totalWays + dfs(currentPosition + stepDirection, remainingSteps - 1)) % mod;
        }

        // The result is stored in the memoization cache before returning it.
        memoizationCache[currentPosition][remainingSteps] = totalWays;

        return totalWays;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <cstring>
#include <functional>

class Solution {
public:
    int numWays(int steps, int arrLen) {
        // Using std::vector to handle dynamic 2D array for memoization
        std::vector<std::vector<int>> memo(steps, std::vector<int>(steps + 1, -1));
        const int MOD = 1e9 + 7;  // Define the modulo constant

        // Define the depth-first search function with memoization
        std::function<int(int, int)> dfs = [&](int position, int remainingSteps) -> int {
            // Base case: If out of bounds or steps remaining are less than distance to the start
            if (position >= arrLen || position < 0 || remainingSteps < position) {
                return 0;
            }
            // If at the start position with no remaining steps, return 1 way
            if (position == 0 && remainingSteps == 0) {
                return 1;
            }
            // Check if we have already computed the number of ways for this state
            if (memo[position][remainingSteps] != -1) {
                return memo[position][remainingSteps];
            }
            // Recursive case: Explore staying in the same place, moving left, or moving right
            int countWays = 0;
            for (int step = -1; step <= 1; ++step) {
                countWays = (countWays + dfs(position + step, remainingSteps - 1)) % MOD;
            }
            // Memoize and return the computed number of ways for current state
            return memo[position][remainingSteps] = countWays;
        };

        // Call the helper function starting at position 0 with 'steps' remaining
        return dfs(0, steps);
    }
};
```

Typescript Solution

```typescript
// Define the modulo constant to prevent overflow.
const MODULO: number = 10 ** 9 + 7;

// Create a memoization table to store computed values.
const memo: number[][] = Array.from({ length: steps }, () => Array(steps + 1).fill(-1));

// The recursive DFS helper function to compute the number of ways.
function dfs(position: number, remainingSteps: number): number {
    // Out of bounds or more steps to return than remaining ones are invalid scenarios.
    if (position > remainingSteps || position >= arrLen || position < 0) {
        return 0;
    }

    // Base case: when at the start and no more steps left, there's one way.
    if (position === 0 && remainingSteps === 0) {
        return 1;
    }

    // Return the cached value if it's computed already.
    if (memo[position][remainingSteps] !== -1) {
        return memo[position][remainingSteps];
    }

    // Initialize the number of ways to 0.
    let numberOfWays: number = 0;

    // Iterate over the possible steps one can make: -1, 0, +1.
    for (let step = -1; step <= 1; step++) {
        // Explore the next state in the DFS and update the number of ways, keeping the result within the modulo.
        numberOfWays = (numberOfWays + dfs(position + step, remainingSteps - 1)) % MODULO;
    }

    // Memorize the computed value before returning.
    return memo[position][remainingSteps] = numberOfWays;
}

// The main function to return the number of ways to stay in the array after taking a certain number of steps.
function numWays(steps: number, arrLen: number): number {
    return dfs(0, steps);
}
```

Time and Space Complexity

The given code defines a dynamic programming function `dfs(i, j)` where `i` is the current position and `j` is the remaining number of steps. The function is memoized using Python's `cache` decorator, meaning previously computed results for certain `(i, j)` pairs will be stored and reused.

Time Complexity

The time complexity depends on the number of unique recursive calls to `dfs(i, j)`. Let `min(steps, arrLen)` be `m` which represents the maximum possible unique positions we can be on the path.

Given that from each position `i`, we can move to `i-1`, `i`, or `i+1` for the next step, the recursion creates a ternary tree of calls. However, the use of memoization cuts down the number of unique calls to the actual number of different states the problem can be in.

There are at most `m` different positions and at each position, we can have at most `steps` different `j` values (number of steps remaining). Therefore, the number of unique states is `O(m * steps)`.

For each state `(i, j)`, we consider 3 possible next states by iterating over `k = -1, 0, 1`, each constant time operations. Hence, the total time complexity is:

`O(m * steps * 3)` which simplifies to `O(m * steps)`.

Since `m = min(steps, arrLen)`, the time complexity becomes `O(min(steps, arrLen) * steps)`.

Space Complexity

The space complexity consists of the space used by the call stack during recursion and the space used to store the memoized results.

1. **Call Stack**: In the worst-case scenario, the maximum depth of the recursive call stack is equal to the number of steps `steps`, because we can only move `steps` times before we run out of moves.
2. **Memoization Dictionary**: There are `O(m * steps)` unique states being stored in the dictionary, where `m` is `min(steps, arrLen)` as discussed earlier.

Therefore, the overall space complexity, accounting for both the call stack and the memoization storage, is `O(m * steps + steps)` which simplifies to `O(m * steps)` where `m = min(steps, arrLen)`.

Putting it all together, the space complexity is also `O(min(steps, arrLen) * steps)`.