790. Domino and Tromino Tiling

Medium Dynamic Programming Leetcode Link

Problem Description

You are given a problem related to tiling a `2 x n` board with two types of shapes: a `2 x 1` domino and an `L` shaped tromino. The shapes can be rotated as needed to fit into the tiling of the board. The task is to find out how many different ways you can completely cover the `2 x n` board using these tiles. It's important to note that two tilings are considered different only if there's at least one pair of adjacent cells in the board that is covered in one tiling and not covered in the other. Since the number of ways can be very large, the answer should be returned modulo `10^9 + 7`.

Intuition

To solve this problem, we can use dynamic programming because the problem has an optimal substructure and overlapping subproblems. Each state in our dynamic programming represents the number of ways to tile a board up to positions `(i, j)`, where `i` is the position on the upper row and `j` is the position on the lower row.

Starting from a fully untiled board `(0, 0)`, we consider all possible moves we can make with either of our two types of tiles—one that takes us to `(i + 1, j + 1)` which represents placing a domino tile vertically, to `(i + 2, j + 2)` representing placing two domino tiles horizontally next to each other, to `(i + 1, j + 2)` or `(i + 2, j + 1)` representing placing a tromino tile. Every move is a transition to a new state, and the number of ways to tile the board for that state is the sum of ways of reaching it from all previous states.

The recursion ends when we consider the entire board tiled (`i == n` and `j == n`), which is our base case and we return 1 as there is exactly one way to have a whole tiled board from an entirely tiled board—do nothing.

Since there are overlapping subproblems (we could reach a state from different previous moves), memoization via the `@cache` decorator is used to avoid redundant calculations. Lastly, every addition of ways is done modulo `10^9 + 7` to prevent integer overflow and adhere to the problem constraints of returning the count of ways modulo `10^9 + 7`.

Solution Approach

The solution to this problem makes use of a recursive depth-first search (DFS) approach along with memoization to count the number of ways to tile the `2 x n` board. The DFS function `dfs(i, j)` is calculating the number of tilings for the board up to the point `(i, j)`, bearing in mind that `i` and `j` represent the ending positions on the upper and lower rows, respectively.

The solution uses the `@cache` decorator to memorize intermediate results. This is because the DFS function will be called many times with the same arguments, and we want to avoid recalculating those results each time. The memoization optimizes our approach from an exponential time complexity to a polynomial time complexity.

Here is the step-by-step approach:

1. If either `i` or `j` exceeds `n`, we've gone beyond the board, which means this is not a valid tiling, so return 0.
2. If `i == n` and `j == n`, we succeeded in tiling the entire board and thus return 1.
3. If `i == j`, both rows are equally tiled thus far. We can place either two dominoes side by side (horizontally), one domino vertically, or a tromino (which will cover `2 x 1` area and one additional square either above or below). We accordingly call the `dfs` function for the following possible next steps: `dfs(i + 2, j + 2)`, `dfs(i + 1, j + 1)`, `dfs(i + 2, j + 1)`, and `dfs(i + 1, j + 2)`.
4. If `i > j`, the upper row is ahead, so we can only place a domino horizontally on the lower row or a tromino that covers two cells on the lower row and one on the upper. Thus, the new states to go to are `dfs(i, j + 2)` and `dfs(i + 1, j + 2)`.
5. If `i < j`, the lower row is ahead, so we can place a domino vertically on the upper row to reach `dfs(i + 2, j)` or place a tromino in a way that covers two cells on the upper row and one on the lower to reach `dfs(i + 2, j + 1)`.

In each of these cases, we add up the counts returned by all possible step(s) taken from the current state, and this sum is then returned modulo `10^9 + 7` as required by the problem statement. This result is then used for further calculations as we backtrack through the recursion.

When the initial call to `dfs(0, 0)` completes, we have computed the total number of valid tilings for the entire board, and this result is returned as the final answer.

Example Walkthrough

Let's walk through a small example using a `2 x 3` board to illustrate the solution approach with the method `dfs(i, j)`.

1. Start by calling `dfs(0, 0)` since no tiles have been placed yet.

2. Since `i == j`, there are three moves we can consider:

   - Place two horizontal dominoes to cover the whole first column, `dfs(2, 2)`.
   - Place one vertical domino to cover the first cells of both rows, `dfs(1, 1)`.
   - Place one tromino (L-shaped), which will leave one cell in the second column on one of the rows uncovered, so we call both `dfs(1, 2)` and `dfs(2, 1)`.
3. For each recursive call `dfs(2, 2)`, `dfs(1, 1)`, `dfs(1, 2)`, and `dfs(2, 1)`, repeat steps similar to step 2.

   When calling `dfs(2, 2)` from `dfs(0, 0)`:

   - `i == j` and they are equal to `n`, so it's a complete tiling and returns 1.

   When calling `dfs(1, 1)` from `dfs(0, 0)`:

   - `i == j`, do the similar three steps:
     - `dfs(3, 3)`: Goes beyond the board, thus returns 0.
     - `dfs(2, 2)`: Complete tiling, returns 1.
     - `dfs(2, 3)` and `dfs(3, 2)`: Goes beyond the board, thus both return 0.

   When calling `dfs(1, 2)` from `dfs(0, 0)`:

   - `i < j`, only two moves are possible:
     - `dfs(3, 2)`: Goes beyond the board, thus returns 0.
     - `dfs(3, 3)`: Goes beyond the board, thus returns 0.

   When calling `dfs(2, 1)` from `dfs(0, 0)`:

   - `i > j`, only two moves are possible:
     - `dfs(2, 3)`: Goes beyond the board, thus returns 0.
     - `dfs(3, 3)`: Goes beyond the board, thus returns 0.
4. To get the result for `dfs(0, 0)` add up all the ways from the recursive calls:

   - From `dfs(2, 2)` we got 1 way.
   - From `dfs(1, 1)` we consider the result from `dfs(2, 2)`, which is 1 way.
   - Calls `dfs(1, 2)` and `dfs(2, 1)` do not add more ways as they result in 0 valid tilings.
5. So, `dfs(0, 0)` would return 1 (from `dfs(2, 2)`) + 1 (from `dfs(1, 1)`, which includes the result from `dfs(2, 2)`) modulo `10^9 + 7`, giving us a total of 2 ways to tile the `2 x 3` board.

The result from `dfs(0, 0)` is the final answer for the problem, which is 2 different ways to completely cover the `2 x 3` board using the given domino and tromino shapes.

Python Solution

```python
from functools import lru_cache  # Import lru_cache for memoization

class Solution:
    def numTilings(self, n: int) -> int:
        # A helper function with memoization to count the number of ways to tile the board.
        # i represents the tiles placed in the first row, and j represents the tiles in the second row.
        @lru_cache(None)  # Use lru_cache for memoization to improve performance
        def count_ways(first_row: int, second_row: int) -> int:
            # Base case: If we exceed the board size, there's no way to tile.
            if first_row > n or second_row > n:
                return 0
            # Base case: If both rows are completely tiled, we've found one valid tiling.
            if first_row == n and second_row == n:
                return 1

            # Initialization of possible ways to tile.
            ways = 0
            # When both rows have the same number of points covered by tiles.
            if first_row == second_row:
                ways = (
                    count_ways(first_row + 2, second_row + 2) +  # Place a 2x2 square.
                    count_ways(first_row + 1, second_row + 1) +  # Place two 2x1 tiles, one in each row.
                    count_ways(first_row + 2, second_row + 1) +  # Place a 'L' shaped tromino.
                    count_ways(first_row + 1, second_row + 2)    # Place an inverted 'L' shaped tromino.
                )
            elif first_row > second_row:
                # If the first row has more tiles than the second row.
                ways = count_ways(first_row, second_row + 2) + count_ways(first_row + 1, second_row + 2)
            else:
                # If the second row has more tiles than the first row.
                ways = count_ways(first_row + 2, second_row) + count_ways(first_row + 2, second_row + 1)

            # Return the ways modulo MOD, which represents the maximum number of unique tilings.
            return ways % MOD

        # Define the modulo constant to prevent large number arithmetic issues.
        MOD = 10**9 + 7
        # Call the helper function with the initial states of the board (0 tiles placed).
        return count_ways(0, 0)

```

Java Solution

```java
class Solution {
    public int numTilings(int n) {
        // Initialize a DP array to store tiling counts for 4 states
        // f[0]: full covering, f[1]: top row is missing, f[2]: bottom row is missing, f[3]: transitional state (both rows are missir
        long[] dp = {1, 0, 0, 0};

        // Modulo value to prevent overflow
        int mod = (int) 1e9 + 7;

        // Iterate over the sequence from 1 to n
        for (int i = 1; i <= n; ++i) {
            // Temporary array to hold the new states counts
            long[] newDp = new long[4];

            // New full covering can be achieved from any previous state
            newDp[0] = (dp[0] + dp[1] + dp[2] + dp[3]) % mod;

            // New top missing can be achieved from state when previously bottom was missing and the transitional state
            newDp[1] = (dp[2] + dp[3]) % mod;

            // New bottom missing can be achieved from state when previously top was missing and the transitional state
            newDp[2] = (dp[1] + dp[3]) % mod;

            // New transitional state comes solely from previous full covering state
            newDp[3] = dp[0];

            // Update the dp array for the next iteration
            dp = newDp;
        }

        // After completing all iterations, the count of fully covered tilings is in dp[0]
        return (int) dp[0];
    }
}

```

C++ Solution

```cpp
#include <cstring> // For memcpy

class Solution {
public:
    // MOD constant to be used for taking the remainder after division.
    static const int MOD = 1e9 + 7;

    int numTilings(int n) {
        // An array to hold the number of ways to tile for different subproblems.
        // f[0]: full cover, f[1]: top row is missing one, f[2]: bottom row is missing one, f[3]: both top and bottom are missing o
        long tilingWays[4] = {1, 0, 0, 0};

        // Iterate through all subproblem sizes from 1 to n
        for (int i = 1; i <= n; ++i) {
            // Temporary array to compute the new number of tiling ways for the current subproblem.
            long newTilingWays[4] = {0, 0, 0, 0};

            // Full cover is obtained by adding one 2x2 tile or two 2x1 tiles to any of the four previous states.
            newTilingWays[0] = (tilingWays[0] + tilingWays[1] + tilingWays[2] + tilingWays[3]) % MOD;
            // Top row missing one can be obtained by adding a 2x1 tile to the previous state of bottom row missing one or both top
            newTilingWays[1] = (tilingWays[2] + tilingWays[3]) % MOD;
            // Bottom row missing one can be obtained by adding a 2x1 tile to the previous state of top row missing one or both top
            newTilingWays[2] = (tilingWays[1] + tilingWays[3]) % MOD;
            // Both top and bottom missing one can only be obtained by placing a 2x2 tile in the full cover state.
            newTilingWays[3] = tilingWays[0];

            // Update tilingWays array with the new computed values.
            memcpy(tilingWays, newTilingWays, sizeof(newTilingWays));
        }

        // Return the number of ways to fully cover a 2xN board.
        return tilingWays[0];
    }
};

```

Typescript Solution

```typescript
// MOD constant to be used for taking the remainder after division.
const MOD: number = 1e9 + 7;

// Calculates the number of ways to tile a 2xN board with dominoes.
function numTilings(n: number): number {
    // An array to hold the number of ways to tile for different subproblems:
    // - tilingWays[0]: full cover,
    // - tilingWays[1]: top row is missing one,
    // - tilingWays[2]: bottom row is missing one,
    // - tilingWays[3]: both top and bottom are missing one (in L-shape).
    let tilingWays: number[] = [1, 0, 0, 0];

    // Iterate through all subproblem sizes from 1 to n.
    for (let i = 1; i <= n; ++i) {
        // Temporary array to compute the new number of tiling ways for the current subproblem.
        let newTilingWays: number[] = [0, 0, 0, 0];

        // Full cover is obtained by adding one 2x2 tile or two 2x1 tiles to any of the four previous states.
        newTilingWays[0] = (tilingWays[0] + tilingWays[1] + tilingWays[2] + tilingWays[3]) % MOD;
        // Top row missing one can be obtained by adding a 2x1 tile to the previous state of bottom row missing one or both top and
        newTilingWays[1] = (tilingWays[2] + tilingWays[3]) % MOD;
        // Bottom row missing one can be obtained by adding a 2x1 tile to the previous state of top row missing one or both top and
        newTilingWays[2] = (tilingWays[1] + tilingWays[3]) % MOD;
        // Both top and bottom missing one can only be obtained by placing a 2x2 tile in the full cover state.
        newTilingWays[3] = tilingWays[0];

        // Update tilingWays array with the new computed values.
        tilingWays = [...newTilingWays];
    }

    // Return the number of ways to fully cover a 2xN board.
    return tilingWays[0];
}

// The numTilings function can be exported or used directly, depending on the context of the TypeScript project.

```

Time and Space Complexity

The provided Python function `numTilings` is intended for finding the number of ways to tile a 2xN board with 2×1 dominos and trominoes (L-shaped tiles). This is a dynamic programming problem often posed on platforms like LeetCode.

Time Complexity

The time complexity of the function is difficult to frame precisely due to the complex recursive structure. On every call to `dfs`, potentially up to 4 recursive calls can be made, and since this algorithm calls the `dfs` with arguments up to `n` (both `i` and `j`), the time complexity is loosely bounded by a function of the form `O(4^m)`, where `m` represents the number of recursive steps needed to reach the base case from the current step.

However, due to memoization (caching of results), repeated states are not recalculated, significantly reducing the number of recursive calls. Memoization effectively ensures that each state of (i, j) where 0 <= i, j <= n is computed only once. Since `i` and `j` can take on `n+1` possible values each, the actual time complexity can be approximated as `O(n^2)`.

Space Complexity

The space complexity consists of the space used by the memoization cache and the stack space used by the recursion.

The cache will have an entry for each unique (i, j) pair, so it will consume `O(n^2)` space. The stack space, in the worst case, could be as deep as the maximum depth of recursive calls, which is `O(n)` because in the worst case we increase one of `i` or `j` by 1 until `n` is reached.

Therefore, the overall space complexity of the `numTilings` function can be considered as `O(n^2)` due to the memoization cache, which is the dominating factor here.

So, the final analysis is:

- Time Complexity: `O(n^2)`, considering memoization
- Space Complexity: `O(n^2)`, due to the caching and the call stack