2209. Minimum White Tiles After Covering With Carpets

Hard String Dynamic Programming Prefix Sum Leetcode Link

Problem Description

In this LeetCode problem, we have a binary string `floor` representing the colors of tiles on a floor, where '0' indicates a black tile and '1' indicates a white tile. We are given a certain number of black carpets (`numCarpets`) and the length of each carpet (`carpetLen`). The goal is to cover as many white tiles as possible with these carpets, minimizing the number of white tiles that remain visible. Carpets can overlap, and we need to calculate the minimum number of white tiles that will be left uncovered after optimally placing the carpets.

Intuition

To solve this problem, we need to find an optimal way to place the carpets to cover the maximum number of white tiles. Since carpets can overlap, we have to make decisions at each step regarding where to place a carpet and whether to use it at the current position or save it for later.

Given the nature of the problem, we can think of a dynamic programming approach where we keep track of the number of carpets left (`j`) and which position we are at (`i`) in the string. As we encounter a white tile (represented by '1'), we have two choices: either place a carpet here or skip this tile. If we decide to place a carpet, all tiles from the current position to the length of the carpet will be covered. If we skip, we move on to the next tile.

The solution uses a depth-first search (DFS) function with memoization (achieved through the `@cache` decorator) to remember the results of subproblems. The DFS function calculates the minimum number of white tiles revealed for each scenario and returns the minimum of covering the current tile or moving ahead. Thus, this approach uses a top-down dynamic programming strategy to optimize carpet placement and achieve the desired result.

The additional array `s` is maintained to keep track of the cumulative sum of white tiles up to any given position for quick calculation of white tiles left when we decide to skip placing a carpet. This setup aids in optimizing the process by avoiding the need to recount white tiles in subproblems repeatedly.

After computing the answer using the DFS approach, the `cache_clear()` method is called to clear the cache, ensuring that the cache is not filled with stale values from previous test cases.

Solution Approach

The implementation of the solution involves dynamic programming combined with depth-first search (DFS) and memoization. Here are the steps and algorithms used in the implementation:

1. **DFS with Memoization**: The main algorithm employed is a recursive DFS function `dfs(i, j)` that takes two parameters—`i`, the current tile index in the `floor`, and `j`, the number of carpets remaining. The recursion explores different scenarios of placing or not placing carpets and recalls optimal substructure results via memoization to prevent redundant calculations (`@cache` decorator).

2. **Base Case Handling**: The base cases are when we've considered all the tiles (i.e., `i >= n`, where `n` is the length of `floor`), or when there are no carpets left (`j == 0`). The former case returns `0` since there are no more tiles to cover, and the latter returns the total number of white tiles remaining starting from index `i` to the end (`s[-1] - s[i]`).

3. **Making Decisions**: When at a black tile ('0'), we can simply move to the next tile (`dfs(i + 1, j)`). When at a white tile ('1'), there's a choice to make: either cover it with a carpet (and evaluate `dfs(i + carpetLen, j - 1)`, where `i + carpetLen` represents skipping all covered tiles and `j - 1` decrements the available carpets) or leave it uncovered (and evaluate `1 + dfs(i + 1, j)` where `1` represents the current white tile left uncovered). The recursive DFS function returns the minimum of these two scenarios.

4. **Cumulative Sum Array**: The array `s` serves to pre-compute the prefix sum of white tiles at each index, where `s[i + 1] = s[i] + int(floor[i] == '1')`. This allows fast calculation of white tiles over a range of indices, optimizing the process when determining how many white tiles would remain if no carpet is placed starting from a particular index.

5. **Calling DFS and Clearing Cache**: The solution ultimately calls `dfs(0, numCarpets)` to start the process from the beginning of the floor with all carpets available. Once the result is computed, `dfs.cache_clear()` is called to reset the memoization before handling the next test case.

The intelligent use of depth-first search with memoization to remember previous results, combined with cumulative sum logic for efficiency, brings the complexity of an otherwise exponential brute force solution down to a manageable level, enabling the solving of larger inputs effectively.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Suppose the binary string `floor` is `"1100101"`, we are given `numCarpets = 2`, and `carpetLen = 2`. We aim to cover as many white tiles as possible with these carpets.

1. **Initialization**: We first create a cumulative sum array `s` with an additional `0` at the beginning to handle zero-indexed arrays easily. For the string `"1100101"`, `s = [0, 1, 2, 2, 2, 3, 4, 5]`. This array will help in quickly calculating the number of white tiles we have between any two given indices.

2. **Recursive DFS Call**: We start the DFS process from index `0` with all carpets available (`dfs(0, 2)`).

3. **Making Decisions**:
   - At index `0`, we have a white tile. We have two choices:
     - Place a carpet and move two tiles ahead, `dfs(2, 1)`.
     - Skip the tile, leaving it uncovered, `1 + dfs(1, 2)`.
   - Let's say we decide to place a carpet. Now at index `2`, we have a choice again:
     - Place a carpet from `2` to `4` and move carpet length ahead, `dfs(4, 0)`.
     - Skip the tile, `s[-1] - s[2] + dfs(3, 1)` (since `j` is not `0`, we perform `s[-1] - s[2]` to count the remaining white tiles starting from index `2`).
4. **Evaluating scenarios with memoization**: Each of these scenarios is evaluated, and the DFS function will keep track of the scenario yielding the minimum number of white tiles left uncovered.

5. **Base Cases**: Whenever `i >= n` (beyond last index) or `j == 0`, we handle the base cases as previously detailed.

After making optimal choices at each step and using memoization to avoid repeated calculations, the recursive function will yield the minimum number of white tiles that will remain uncovered. For the above example, two carpets will cover tiles from indices `0` to `1` and `2` to `3` or `4` to `5`, depending on the decision made at index `2`. In both cases, there will be `2` white tiles showing—the minimum for this configuration.

Upon finishing the evaluation, we call `dfs.cache_clear()` to prepare the memoization for the next test case if needed, without interference from the current state. Following this approach results in an efficient solution to the problem.

Python Solution

```python
from functools import lru_cache

class Solution:
    def minimumWhiteTiles(self, floor: str, num_carpets: int, carpet_len: int) -> int:
        # Using LRU cache to memorize results reducing time complexity
        @lru_cache(maxsize=None)
        def dfs(index, remaining_carpets):
            # If we've considered all tiles or have no remaining carpets
            if index >= n or remaining_carpets == 0:
                # Return the sum of white tiles starting from the current index
                return suffix_sum[n] - suffix_sum[index]

            # If the current tile is white, we need to consider it
            if floor[index] == '1':
                # We have the choice to place a carpet or leave the tile exposed
                return min(dfs(index + 1, remaining_carpets), # Leave tile exposed
                           dfs(index + carpet_len, remaining_carpets - 1)) # Place a carpet
            else:
                # If the current tile is already black, just move to the next tile
                return dfs(index + 1, remaining_carpets)

        n = len(floor)

        # Precompute the prefix sum of white tiles to use later in calculation
        suffix_sum = [0] * (n + 1)
        for i in range(n):
            suffix_sum[i + 1] = suffix_sum[i] + (floor[i] == '1')

        # Calculate the minimum number of white tiles using DFS starting from tile index 0 with all carpets available
        min_white_tiles = dfs(0, num_carpets)

        # Clear the cache for the lru_cache decorator as it's a good practice to free up memory after use
        dfs.cache_clear()
        return min_white_tiles

# Example usage (uncomment to run):
# sol = Solution()
# print(sol.minimumWhiteTiles("10101", 2, 2)) # Should output the minimum number of white tiles
```

Java Solution

```java
class Solution {
    private int[][] memo; // memoization table
    private int[] prefixSums; // prefix sums array for '1's in the floor string
    private int floorLength; // length of the 'floor' string
    private int carpetLength; // the length of a single carpet

    public int minimumWhiteTiles(String floor, int numCarpets, int carpetLen) {
        floorLength = floor.length();
        memo = new int[floorLength][numCarpets + 1];

        // Initialize the memoization table with -1 to indicate uncalculated states
        for (int[] row : memo) {
            Arrays.fill(row, -1);
        }

        // Precompute the prefix sums of '1's in the floor string
        prefixSums = new int[floorLength + 1];
        for (int i = 0; i < floorLength; ++i) {
            prefixSums[i + 1] = prefixSums[i] + (floor.charAt(i) == '1' ? 1 : 0);
        }

        carpetLength = carpetLen;

        // Start the recursive depth-first search from position 0 with all carpets available
        return dfs(0, numCarpets);
    }

    private int dfs(int position, int remainingCarpets) {
        // If we have reached past the end of the floor, no more white tiles to cover
        if (position >= floorLength) {
            return 0;
        }
        // If we have no carpets left, return the number of white tiles until the end
        if (remainingCarpets == 0) {
            return prefixSums[floorLength] - prefixSums[position];
        }
        // If the result has been computed before, return it from the memoization table
        if (memo[position][remainingCarpets] != -1) {
            return memo[position][remainingCarpets];
        }
        // If the current floor tile is not white, go to next tile
        if (prefixSums[position + 1] == prefixSums[position]) {
            return dfs(position + 1, remainingCarpets);
        }
        // Consider two scenarios:
        // 1. Cover the current tile with a carpet and move the position by carpetLength
        // 2. Leave the current tile white and move to the next tile
        int minWhiteTiles = Math.min(
            1 + dfs(position + 1, remainingCarpets), // not using a carpet here
            dfs(position + carpetLength, remainingCarpets - 1) // using a carpet
        );
        // Save the result to the memoization table
        memo[position][remainingCarpets] = minWhiteTiles;
        return minWhiteTiles;
    }
}
```

C++ Solution

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

using namespace std;

class Solution {
public:
    int minimumWhiteTiles(string floor, int numCarpets, int carpetLen) {
        int n = floor.size();

        // Create a memoization table with an initial value of -1.
        vector<vector<int>> dp(n, vector<int>(numCarpets + 1, -1));

        // Create a prefix sum array 's' where s[i] indicates the number of white tiles up to index i-1.
        vector<int> prefixSum(n + 1);
        for (int i = 0; i < n; ++i) {
            prefixSum[i + 1] = prefixSum[i] + (floor[i] == '1');
        }

        // Declare the dfs function to be used for the memoization.
        function<int(int, int)> dfs;

        dfs = [&](int pos, int remainingCarpets) {
            if (pos >= n) {
                // Base case: if we've covered the entire floor, no white tiles are left.
                return 0;
            }
            if (remainingCarpets == 0) {
                // If no carpets are left, return the count of remaining white tiles.
                return prefixSum[n] - prefixSum[pos];
            }
            if (dp[pos][remainingCarpets] != -1) {
                // Return the result from the memoization table if already computed.
                return dp[pos][remainingCarpets];
            }
            if (prefixSum[pos + 1] == prefixSum[pos]) {
                // If the current position has no white tile, move to the next position.
                return dfs(pos + 1, remainingCarpets);
            }

            // Recurrence relation:
            // min(putting a carpet here and moving forward by carpetLen, not putting a carpet here
            // and considering the current white tile uncovered)
            int ans = min(dfs(pos + carpetLen, remainingCarpets - 1), 1 + dfs(pos + 1, remainingCarpets));

            // Save the result in the memoization table before returning.
            dp[pos][remainingCarpets] = ans;

            return ans;
        };

        // Start the dfs from the first position with all carpets available.
        return dfs(0, numCarpets);
    }
};
```

Typescript Solution

```typescript
// Importing functionalities from the standard library (instead of #include which is C++ syntax)
import { memoize } from 'lodash';

// Global variable declarations (every variable used matches Typescript's syntax)
let n: number;
let carpetLen: number;
let prefixSum: number[];
const dp: number[][] = [];

// Utility function to compute the prefix sum array.
// This function calculates the cumulative sum of white tiles up to each index.
function computePrefixSum(floor: string) {
    prefixSum = new Array(n + 1).fill(0);
    for (let i = 0; i < n; ++i) {
        prefixSum[i + 1] = prefixSum[i] + (floor[i] === '1' ? 1 : 0);
    }
}

// The memoization of dfs using a higher-order function - this would be typical in Typescript to handle previous state.
// Since there's no direct equivalent of `std::function` from C++, we use Typescript function types.
const dfs: (pos: number, remainingCarpets: number) => number = memoize(
    (pos: number, remainingCarpets: number): number => {
        if (pos >= n) {
            return 0; // Base case: if we've covered the entire floor, no white tiles are left.
        }
        if (remainingCarpets === 0) {
            return prefixSum[n] - prefixSum[pos]; // No carpets left: return count of white tiles.
        }
        if (dp[pos][remainingCarpets] !== -1) {
            return dp[pos][remainingCarpets]; // Return memoized result if present
        }
        if (prefixSum[pos + 1] === prefixSum[pos]) {
            return dfs(pos + 1, remainingCarpets); // No white tile at current, move to next
        }

        // Decision to put or not put a carpet
        let result = Math.min(
            dfs(pos + carpetLen, remainingCarpets - 1), // Putting a carpet here
            1 + dfs(pos + 1, remainingCarpets)          // Not putting carpet here
        );

        dp[pos][remainingCarpets] = result; // Update memoization table
        return result;
    }
);

// This function initializes the dp array and computes the minimum number of white tiles after placing the carpets.
// This is the equivalent of the `minimumWhiteTiles` method in the provided C++ solution.
function minimumWhiteTiles(floor: string, numCarpets: number, carpetLength: number): number {
    n = floor.length; // Size of the floor
    carpetLen = carpetLength; // Length of one carpet
    dp.length = 0; // Resetting dp if already filled

    // Initializing memoization table with initial value of -1
    for (let i = 0; i < n; i++) {
        dp.push(new Array(numCarpets + 1).fill(-1));
    }

    // Compute prefix sum only once at the beginning to use throughout
    computePrefixSum(floor);

    // Start the dfs from the first position with all carpets available
    return dfs(0, numCarpets);
}
```

Time and Space Complexity

Time Complexity

The overall time complexity of the given algorithm is determined by the number of states the dynamic programming needs to compute and the time it takes to compute each state. The algorithm uses a top-down dynamic programming (DFS) approach with memoization.

- The function `dfs` is a recursive function with two parameters `i` and `j`, which represent the current index in the string `floor` and the number of carpets left to use, respectively.
- `i` can have a maximum of `n` different states, where `n` is the length of the `floor` string.
- `j` can have a maximum of `numCarpets + 1` different states (ranging from `0` to `numCarpets`).
- For each state, `dfs` makes at most two recursive calls, representing the two choices available: either place a carpet at the current position or not.

Map this into time complexity, assuming `n` is the length of the string `floor` and `c` is the number of carpets `numCarpets`:

`T(n, c)` = `T(n - 1, c)` (move to the next tile without placing a carpet) + `T(n - carpetLen, c - 1)` (place a carpet and skip `carpetLen` tiles)

Solving this, we have `O(n * c * c2)` time complexity, where `c2` is the work done for each state. Hence, the total time complexity is `O(n * c * 2)`.

Space Complexity

The space complexity of the algorithm includes the space required for memoization and the depth of the recursive call stack.

- Memoization requires `O(n * c)` space since it stores a result for each possible state (`i`, `j`).
- The recursion depth can go as deep as `n` because we might go down one level deeper for each tile.

Therefore, the overall space complexity is `O(n * c)` + `O(n)`. Since `O(n * c)` is the dominating term, the simplified space complexity is also `O(n * c)`.