2742. Painting the Walls

Hard Array Dynamic Programming Leetcode Link

Problem Description

In this problem, you are tasked with calculating the minimum cost to paint a number of walls, given two painters with different properties. The first painter is a **paid painter** who can paint each wall at a certain cost and within a certain amount of time as specified by arrays `cost` and `time`. Specifically, the `i<sup>th</sup>` wall can be painted by the paid painter at the cost of `cost[i]` and will take `time[i]` units of time to finish. Conversely, the second painter is a **free painter** who can paint any wall without any cost, but each wall takes exactly `1` unit of time to paint. The catch is that the free painter can only work while the paid painter is occupied.

Your goal is to find the strategy of using these two painters such that all `n` walls are painted at the minimum possible cost.

Intuition

The problem can be approached as a dynamic programming problem where the main idea is to decide, for each wall, whether it should be painted by the paid painter or the free painter. Since each painter's time and cost differ, we're looking for an optimal cost that minimizes the overall expenditure.

Our solution uses a depth-first search (DFS) with memoization to avoid recalculating the same scenarios, which would be an inefficient use of time and resources. We create a function `dfs(i, j)` representing the minimum cost of painting from the `i`th wall while having `j` units of free painter's available working time left. Memoization helps us store the results of our calculations for different `(i, j)` pairs, ensuring that each pair is only calculated once.

In the recursive `dfs` function, if we have more free painting time (`j`) than walls left to paint (`n - i`), we can paint them all for free, and the cost is zero. If we are trying to paint beyond the `nth` wall (`i >= n`), it's not a valid scenario, so we return infinity which denotes an impossibly high cost. Lastly, for every wall, we consider the minimum cost between these two options:

- If the paid painter paints the wall, we add the cost of painting that wall and call `dfs` for the next wall, with the free painter's available time increased by the `time[i]` it takes to paint the current wall.
- If the free painter paints the wall, the cost is zero, but since the free painter only works when the paid painter is also working, we reduce the free painter's available time by 1 and call `dfs` for the next wall.

We kickstart the process with `dfs(0, 0)`, meaning we start from the 0th wall with no free painting time available initially. The computational process can be optimized in languages other than Python by offsetting the `j` parameter to keep it within a positive range. This dynamic programming approach helps break down a complex problem into smaller, manageable subproblems, efficiently arriving at the minimum overall cost to paint all walls.

Solution Approach

The solution approach uses a combination of Depth-First Search (DFS) and memorization (a common technique in dynamic programming) to explore all possible sequences of assigning walls to either the paid or free painter while ensuring that the decision leads to the least cost possible.

The `dfs` function is a recursive function that takes two parameters: `i`, which is the index of the current wall under consideration, and `j`, which is the remaining free painting time available at the current step. `n`, determined outside the `dfs` function, represents the total number of walls to be painted. The function uses two base conditions:

- When `n - i <= j`, it means we have more or equal free painting time compared to the number of walls left. Hence, all the remaining walls can be painted for free, resulting in no additional cost (`0` cost).
- When `i >= n`, it signifies we have passed all the walls, and there is no further action possible. In this scenario, the function returns infinity (`inf`), which acts as a cost prohibitive placeholder ensuring this path isn't selected.

For each wall, we have two choices:

- We can either have the paid painter paint the current wall, in which case we incur the cost (`cost[i]`) and have more free painting time (`j + time[i]`) since the free painter can only work when the paid painter is working. This leads to the recursive call `dfs(i + 1, j + time[i]) + cost[i]`.
- Alternatively, we can have the free painter paint the current wall at no cost but lose a unit of free paint time (`j - 1`), resulting in the recursive call `dfs(i + 1, j - 1)`.

The solution applies memoization by decorating the `dfs` function with Python's `@cache` decorator to automatically save the results of function calls with the same arguments (`i` and `j`). This way, when a state is revisited, the function returns the saved result immediately instead of recalculating it, vastly reducing the number of recursive calls needed.

The `dfs(0, 0)` initiates the recursive search from the first wall with no free time available for the free painter at the start. As the search unfolds, the `dfs` function will construct and explore different sequences of decisions and find the minimum cost possible, which the function returns to the caller, `paintWalls`.

Through this implementation, the algorithm successfully explores the decision space to find an optimal painting cost with efficient time complexity and optimal use of resources by avoiding unnecessary calculations.

Example Walkthrough

Let's say we have `3` walls to be painted, and the arrays representing the paid painter's cost and time are `[3, 5, 1]` and `[1, 2, 2]` respectively. We need to decide the order of walls to be assigned to the paid or free painter to minimize the total cost.

We start our DFS at the first wall with `dfs(0, 0)` which means index `0` and `0` free painting time available.

1. **Wall 1**: We're at `dfs(0, 0)`. We have two choices:

   a. Paid painter paints: We incur a cost of `3`, and the free painting time becomes `1` (since it takes `1` time unit for the paid painter to paint wall 1). We then move to `dfs(1, 1)`.

   b. Free painter paints: No cost is incurred (`0` cost), but since the free painter can only work while the paid painter is also working, we move to `dfs(1, -1)`. However, because the free time cannot be negative, it is implied that the paid painter will need to paint the next wall as well; hence, we won't consider this scenario further.

2. **Wall 2**: Let's assume the paid painter painted the first wall. We are now at `dfs(1, 1)`. The choices:

   a. Paid painter paints: We incur an additional cost of `5`, and the free painting time becomes `3` (since it takes `2` time units for the paid painter to paint wall 2). We then move to `dfs(2, 3)`.

   b. Free painter paints: No additional cost is incurred, and the free painting time becomes `0` (`1` free time from before - `1` for painting wall 2). We move to `dfs(2, 0)`.

3. **Wall 3**: Depending on previous choices, we have different scenarios:

   a. If the paid painter painted the first two walls, we're at `dfs(2, 3)` now.

   - Paid painter: We incur an additional cost of `1`, and the free painting time becomes `5` (since it takes `2` time units for the paid painter to paint wall 3). This is unnecessary as this is the last wall. So, we just consider the cost and move to `dfs(3, 5)`.
   - Free painter: No additional cost incurred, and the free painting time becomes `2` (`3` - `1`). This is the last wall, and no additional walls are left. We move to `dfs(3, 2)`.

   b. If the free painter painted wall 2, we are now at `dfs(2, 0)`.

   - Paid painter: We incur the cost of `1` and move to `dfs(3, 2)` (since no free time was left before, the free painter cannot work here).
   - Free painter: This scenario is again invalid as we don't have free painting time remaining (`0` - `1` would be negative).

4. **Conclusion**: We can now calculate the total costs for these scenarios and select the minimum:

   - If the paid painter paints all walls, total cost will be `3 + 5 + 1 = 9`.
   - If the paid painter paints walls 1 and 2, and free painter paints wall 3, total cost will be `3 + 5 + 0 = 8`.

The minimum cost for painting all 3 walls, therefore, is `8` using the strategy of the paid painter painting the first two walls and the free painter painting the last one while the paid painter is painting the second wall.

This illustrates how the `dfs` function works to minimize the total painting cost by considering all possible scenarios and selecting the one with the lowest total cost, further optimized by the use of memoization to avoid recalculating the same states.

Python Solution

```python
from typing import List
from functools import lru_cache

class Solution:
    def paintWalls(self, costs: List[int], times: List[int]) -> int:
        # Get the total number of walls to be painted
        num_walls = len(costs)

        # Define the memoization decorator to cache results of the recursive function
        @lru_cache(None)
        def dfs(wall_index: int, time_remaining: int) -> int:
            # Base case 1: If there are fewer walls left than the current time,
            # the cost is 0 as no more painting is needed.
            if num_walls - wall_index <= time_remaining:
                return 0

            # Base case 2: If all walls are considered, return infinity because
            # exceeding time is not permissible.
            if wall_index >= num_walls:
                return float('inf')

            # Recursively consider two options and take the minimum:
            # 1. Paint the current wall (increment time and add the cost)
            # 2. Skip the current wall (decrement the time but no cost)
            return min(
                dfs(wall_index + 1, time_remaining + times[wall_index]) + costs[wall_index],
                dfs(wall_index + 1, time_remaining - 1)
            )

        # Start the recursive function with wall index 0 and time remaining 0
        return dfs(0, 0)

# Example usage:
# solution = Solution()
# Minimum cost to paint all walls:
# print(solution.paintWalls([1, 2, 3, 4], [2, 1, 1, 1]))

```

Java Solution

```java
class Solution {
    // Declare private member variables
    private int numberOfWalls;
    private int[] paintingCost;
    private int[] paintingTime;
    private Integer[][] memoization;

    // Public method to start the process of calculating minimum cost to paint walls
    public int paintWalls(int[] cost, int[] time) {
        numberOfWalls = cost.length;
        this.paintingCost = cost;
        this.paintingTime = time;
        // Initialize memoization array with size [numberOfWalls][2 * numberOfWalls + 1]
        memoization = new Integer[numberOfWalls][numberOfWalls << 1 | 1];
        // Start the depth-first search from the first wall and a time budget equal to the number of walls
        return dfs(0, numberOfWalls);
    }

    // Helper method to perform depth-first search and calculate minimum cost via dynamic programming
    private int dfs(int wallIndex, int timeBudget) {
        // Base condition: if there are fewer or equal walls remaining than the time budget, no cost is needed
        if (numberOfWalls - wallIndex <= timeBudget - numberOfWalls) {
            return 0;
        }

        // Base condition: if we have considered all walls, return a large number to denote an infeasible solution
        if (wallIndex >= numberOfWalls) {
            return Integer.MAX_VALUE / 2; // Use a large value to prevent integer overflow when added
        }

        // Check if the result for the current state has already been computed
        if (memoization[wallIndex][timeBudget] == null) {
            // Recursively calculate the minimum cost of the two possible choices:
            // 1. Paint the current wall and increase the time budget by the time it takes to paint it
            int costIfPaint = dfs(wallIndex + 1, timeBudget + paintingTime[wallIndex]) + paintingCost[wallIndex];
            // 2. Don't paint the current wall and decrease the time budget by 1
            int costIfNotPaint = dfs(wallIndex + 1, timeBudget - 1);

            // Take the minimum of both choices and store it in the memoization array
            memoization[wallIndex][timeBudget] = Math.min(costIfPaint, costIfNotPaint);
        }

        // Return the computed minimum cost for the current state
        return memoization[wallIndex][timeBudget];
    }
}

```

C++ Solution

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

using namespace std;

class Solution {
public:
    // Method to calculate the minimum cost to paint all walls.
    int paintWalls(vector<int>& cost, vector<int>& time) {
        int n = cost.size(); // Number of walls
        // f is a memoization table where f[i][j] stores the minimum cost to paint walls from i to n, given j days.
        int f[n][n << 1 | 1];
        memset(f, -1, sizeof(f)); // Initialize table to -1 representing values not computed yet.

        // Recursive function to calculate minimum cost
        // It is a lambda capturing all external variables by reference.
        function<int(int, int)> dfs = [&](int i, int j) -> int {
            if (n - i <= j - n) {
                // Base case: If the number of days left is at least as many as the number of walls left, cost is 0
                return 0;
            }
            if (i >= n) {
                // Base case: If we have gone past the last wall, return a large number to represent infeasibility
                return 1 << 30;
            }
            if (f[i][j] == -1) {
                // If the cost hasn't been computed yet, calculate it by considering the following two scenarios:
                // 1. Paint the current wall on day j and then proceed to the next wall on day j + time[i]
                // 2. Do not paint any wall on day j and proceed to day j - 1
                f[i][j] = min(dfs(i + 1, j + time[i]) + cost[i], dfs(i + 1, j - 1));
            }
            return f[i][j]; // return the minimum cost found
        };

        // Start the recursive function with the first wall and n days available
        return dfs(0, n);
    }
};

```

Typescript Solution

```typescript
// Importing necessary functionality from array-fills library in JavaScript.
// You will need to include "@types/array-fill" for proper TypeScript typings if you are in a TypeScript environment.
import arrayFill from "array-fill";

// Define the function to calculate the minimum cost to paint all walls.
function paintWalls(cost: number[], time: number[]): number {
    const n: number = cost.length; // Number of walls

    // Initialize a memoization table to store the minimum cost to paint walls from i to n, given j days.
    const f: number[][] = Array.from({ length: n }, () => arrayFill(new Array((n << 1) | 1), -1));

    // Define the recursive function to calculate the minimum cost.
    function dfs(i: number, j: number): number {
        if (n - i <= j - n) {
            // Base case: If there are as many days left as there are walls left, the cost is 0.
            return 0;
        }
        if (i >= n) {
            // Base case: If we have gone past the last wall, return a large number to represent infeasibility.
            return 1 << 30;
        }
        if (f[i][j] === -1) {
            // If the cost hasn't been computed yet, calculate it considering two scenarios:
            // 1. Paint the current wall on day j and then proceed to the next wall on day j + time[i].
            // 2. Proceed to the next wall on day j, but without painting any wall today.
            f[i][j] = Math.min(dfs(i + 1, j + time[i]) + cost[i], dfs(i, j - 1));
        }
        return f[i][j]; // Return the minimum cost found.
    }

    // Start the recursive function with the first wall and n days available.
    return dfs(0, n);
}

// The following is an example of how to use the paintWalls function:
const wallCosts = [1, 2, 3];
const wallTimes = [1, 2, 3];
const minimumCost = paintWalls(wallCosts, wallTimes);
console.log(`The minimum cost to paint all walls is: ${minimumCost}`);

```

Time and Space Complexity

The given Python code is designed to solve a problem related to painting walls with certain costs and times. Below is an analysis of the time and space complexity of this recursive solution with memoization.

Time Complexity:

The time complexity is `O(n^2)` because the `dfs` function is called with various combinations of i and j, where `i` ranges from 0 to `n-1` and `j` ranges from 0 to `n-1` as well. Despite the overlapping subproblems, due to the memoization with `@cache`, each unique state `(i, j)` is solved only once. Since there are `n` possible values for `i` and approximately `n` values for `j` to consider in the worst case, the total number of states that can be stored and computed is `n * n`, leading to `n^2` distinct calls to `dfs`.

Space Complexity:

The space complexity is `O(n^2)` as well. This is because of the memoization that caches results for every unique call to `dfs(i, j)`. Since there can be `n` different values for `i` and up to `n` different values for `j`, the cache might need to store all `n * n` possible combinations of arguments to `dfs`. Hence, the space used by the cache contributes to a quadratic space complexity with respect to the length of the `cost` and `time` lists.