741. Cherry Pickup

Hard | Array | Dynamic Programming | Matrix

Leetcode Link

Problem Description

In this problem, we have a grid in the form of an `n x n` matrix, where each cell can hold a value of 0, 1, or -1, each representing an empty cell, a cherry, or a thorn respectively. Our goal is to collect as many cherries as possible by following these rules:

1. Start at the top-left corner of the grid `(0, 0)` and move to the bottom-right corner `(n - 1, n - 1)` only moving right or down.
2. After reaching `(n - 1, n - 1)`, return to `(0, 0)` by moving left or up.
3. While passing through a cell that contains a cherry `(1)`, collect it, turning the cell into an empty cell `(0)`.
4. If there's a thorn `(-1)` in a cell, you cannot pass through it.
5. If there's no valid path from `(0, 0)` to `(n - 1, n - 1)`, you cannot collect any cherries.

Based on these rules, we have to determine the maximum number of cherries that can be collected.

Intuition

The solution to this problem utilizes dynamic programming. We think of two paths taken simultaneously: one going from `(0, 0)` to `(n - 1, n - 1)` and the other moving back to the start. Since both paths are traversed simultaneously and each move for both paths is either down or right, the total number of moves for both paths will be the same at every step. We use `k` to represent the total number of steps taken so far by each path.

We can represent the state of our traversal using a 3D DP array `dp[k][i1][i2]`, where `k` is the total number of steps each path has taken, `i1` is the row of the first path, and `i2` is the row of the second path. By knowing the row of each path and the total number of steps, we can easily determine the column for each path (`j1` and `j2`) since `j1 = k - i1` and `j2 = k - i2`.

At every step, we attempt to transition from the previous state by deciding from which previous cell each path could have come from: either from the left or from above for each path. This gives us four possible previous states to consider. We calculate the maximum number of cherries collected among these four possibilities while ensuring that we don't step on cells with thorns or move outside the grid and we update the current state with the maximum cherries collected.

Note that we only add the cherries from grid cell `grid[i1][j1]` once if both paths are at the same cell since one cherry cannot be picked twice.

In the end, the maximum number of cherries collected will be the maximum value contained in the DP array for the state representing both paths returning to `(0, 0)`.

Solution Approach

The solution implements a 3D dynamic programming (DP) approach. The key idea is to create a DP table that will keep track of the maximum cherries collected at every step for every position of two simultaneous paths. The DP table is represented as `dp[k][i1][i2]`, where `k` is the total number of steps taken, `i1` is the row position in the grid of the first path, and `i2` is the row position of the second path.

Implementation Steps:

1. Initialize a 3D list `dp` with dimensions `'two times n minus one'` by `n` by `n`, filled with `-inf` to represent the worst case (not being able to reach certain positions). The reason for the first dimension to be `'two times n minus one'` is because the maximum number of steps `k` it would take to reach from `(0, 0)` to `(n - 1, n - 1)` and back is `2n - 2`, accounting for each possible step state.

2. Set the starting position `dp[0][0][0]` to the value of the starting cell `grid[0][0]`, because that is where both paths start and the initial number of cherries collected.

3. Iterate over `k` from `1` to `2n - 2` to cover all possible step states.

4. For each `k`, iterate over all possible row positions `i1` and `i2` for both paths. We calculate the corresponding column positions `j1` and `j2` by `k - i1` and `k - i2` as per the constraint that `i + j` equals the steps taken since both paths started at `(0, 0)`.

5. For each `i1` and `i2` pair, check if the corresponding `j1`, `j2` are within bounds and the cells are not blocked by thorns (`grid[i1][j1] != -1` and `grid[i2][j2] != -1`).

6. Compute the cherry count `t` that could be picked up by moving to this position for both paths. Add the values of `grid[i1][j1]` and `grid[i2][j2]` if the destinations are different, else add any one of them once.

7. Update the current state `dp[k][i1][i2]` by considering all the possible previous states from which the paths could have come: `(i1 - 1, j1)` and `(i1, j1 - 1)` for the first path, `(i2 - 1, j2)` and `(i2, j2 - 1)` for the second one. Take the maximum of the current state and the cherry count `t` plus the previous state value `dp[k - 1][x1][x2]`.

8. After filling the DP table, the answer will be the value at `dp[-1][-1][-1]`, representing the maximum cherries that can be collected by both paths after they've returned to `(0, 0)`. If no path exists, the result should not be negative, so we return the maximum of `0` and this value.

By using dynamic programming, this approach avoids recalculating the result for overlapping subproblems, as we would in a naive recursive approach, thereby significantly reducing the time complexity.

The algorithm capitalizes on the insight that since the paths can be traversed simultaneously, we essentially are looking for the best paths from `(0, 0)` to `(n - 1, n - 1)` and back that cover the most cells containing cherries without crossing into any cells with thorns.

Example Walkthrough

Let's consider a small `3x3` grid to illustrate the solution approach:

```
[
 [ 1,  1, -1],
 [ 1, -1,  1],
 [ 1,  1,  1]
]
```

Each value represents a cell in the grid; `1` is a cherry, `-1` is a thorn, and cells with value `0` would represent empty cells, though there are none in this example.

Step-by-Step Solution:

1. **Initialize the DP table**: We create a `DP` table with dimensions `5x3x3` since `k` can range from `0` to `2 * 3 - 2 = 4`. Initially, the `DP` table is filled with `-inf`, but we set `dp[0][0][0]` to `1` as this is the value of the starting cell `grid[0][0]`.

2. **Iterate over all steps `k`**: We begin with `k=1` and proceed to `k=4`.

3. **Iterate over all pairs of row positions `(i1, i2)`**:

   - For `k=1`, there are two possible positions for each path, `(0, 1)` or `(1, 0)` for both `i1` and `i2`. However, due to thorns, `(0, 1)` isn't viable for either path. So we only consider `(1, 0)` (down for both paths).
   - Assume the upper path moves down to `[1][0]` and the lower path also moves down to `[1][0]`. The total cherries collected would be `1` from this step, making `dp[1][1][1]` equal to `2` (including the cherry from `dp[0][0][0]`).
4. **Compute cherry count `t`**:

   - Continuing with the steps, update `dp[k][i1][i2]` accordingly based on the viable previous positions (ignoring positions with thorns).
   - For example, when `k=3`, if one path is at `[2][1]` (moved right from `[2][0]`) and the other is at `[1][2]` (moved down to `[2][0]` and then moved right), we would have `j1 = 2` and `j2 = 1`. This position is particularly interesting since both paths can potentially add cherries. If both paths hadn't collected the cherry at `[2][0]` initially, they could both collect it now.
5. **Update the DP table**:

   - At each step `k`, update the `dp[k][i1][i2]` entry with the maximum number of cherries collected based on the previous positions of the paths.
   - For instance, `dp[3][2][1]` would be based on the maximum of:
     - `dp[2][1][1]` (both paths moved down from `[1][1]` and `[1][0]`),
     - `dp[2][1][0]` + grid[2][1] (upper path moved down, lower path moved right),
     - `dp[2][2][0]` (invalid since `j1` would be out of bounds),
     - `dp[2][2][1]` (both paths moved right).
6. **Return the result**: After we have filled up the DP table, we look at `dp[-1][-1][-1]`, which is `dp[4][2][2]`. This value represents the maximum number of cherries collected by the two simultaneous paths. Since cells with thorns are non-viable, if all paths to `(n - 1, n - 1)` are blocked, the DP cell will be `-inf`, and thus we return the maximum of this value and `0`.

By following this grid and tracing the steps stated above, we can visualize how the DP table is filled at each step `k` and how we account for paths that can't move due to thorns or have already picked cherries. The final DP table will give us the maximum number of cherries (in viable scenarios) that we could collect by traversing the grid twice using two paths according to the given rules.

Python Solution

```python
from typing import List

class Solution:
    def cherry_pickup(self, grid: List[List[int]]) -> int:
        n = len(grid)

        # Initialize the dynamic programming table with negative infinity
        dp = [[[-float('inf')] * n for _ in range(n)] for _ in range(2 * n - 1)]
        dp[0][0][0] = grid[0][0]  # Starting point

        # Iterate over all possible steps `k` from top-left to bottom-right
        for k in range(1, 2 * n - 1):
            for i1 in range(n):
                for i2 in range(n):
                    # Calculate positions for both paths on step k
                    j1, j2 = k - i1, k - i2

                    # Continue if out of bounds or hits a thorn
                    if (
                        not 0 <= j1 < n
                        or not 0 <= j2 < n
                        or grid[i1][j1] == -1
                        or grid[i2][j2] == -1
                    ):
                        continue

                    # Collect cherries from the current cell(s)
                    cherries = grid[i1][j1]
                    if i1 != i2:
                        cherries += grid[i2][j2]

                    # Check all previous step combinations to get the maximum cherries
                    for prev_i1 in range(i1 - 1, i1 + 1):
                        for prev_i2 in range(i2 - 1, i2 + 1):
                            if prev_i1 >= 0 and prev_i2 >= 0:
                                dp[k][i1][i2] = max(
                                    dp[k][i1][i2], dp[k - 1][prev_i1][prev_i2] + cherries
                                )

        # Return the maximum cherries collectible, ensuring non-negative result
        return max(0, dp[-1][-1][-1])

```

Java Solution

```java
class Solution {

    // Method to maximize the number of cherries collected by two persons from grid
    public int cherryPickup(int[][] grid) {
        int n = grid.length; // n represents the dimension of the grid

        // dp[k][i1][i2] represents the max cherries two persons can collect
        // where both persons have walked k steps in total and they are at positions
        // (i1, k-i1) and (i2, k-i2) respectively
        int[][][] dp = new int[n * 2 - 1][n][n];

        // Initializing the starting point
        dp[0][0][0] = grid[0][0];

        // Iterate over total number of steps
        for (int k = 1; k <= 2 * n - 2; ++k) {
            // Traverse grid for person 1 at (i1, j1)
            for (int i1 = 0; i1 < n; ++i1) {
                // Traverse grid for person 2 at (i2, j2)
                for (int i2 = 0; i2 < n; ++i2) {
                    int j1 = k - i1, j2 = k - i2; // Calculate j1 and j2 based on i1, i2, and k
                    dp[k][i1][i2] = Integer.MIN_VALUE; // Initialize with min value

                    // Check if out of bounds or if cell is a thorn
                    if (j1 < 0 || j1 >= n || j2 < 0 || j2 >= n || grid[i1][j1] == -1 || grid[i2][j2] == -1) {
                        continue;
                    }

                    int cherries = grid[i1][j1];

                    // If the persons are at different positions, collect cherries from both positions
                    if (i1 != i2) {
                        cherries += grid[i2][j2];
                    }

                    // Check each possibility of a step for both persons, and choose the one which yields max cherries
                    for (int prevI1 = i1 - 1; prevI1 <= i1; ++prevI1) {
                        for (int prevI2 = i2 - 1; prevI2 <= i2; ++prevI2) {
                            if (prevI1 >= 0 && prevI2 >= 0) {
                                dp[k][i1][i2] = Math.max(dp[k][i1][i2], dp[k - 1][prevI1][prevI2] + cherries);
                            }
                        }
                    }
                }
            }
        }

        // Return max cherries collected or 0 if none could be collected
        return Math.max(0, dp[2 * n - 2][n - 1][n - 1]);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int cherryPickup(vector<vector<int>>& grid) {
        int size = grid.size();
        // 3D dp array initialized with very small negative values
        vector<vector<vector<int>>> dp(2 * size, vector<vector<int>>(size, vector<int>(size, INT_MIN)));
        dp[0][0][0] = grid[0][0]; // Starting cell

        // 'k' is the sum of the indices (i+j) of each step, determining the "time" step diagonal
        for (int k = 1; k < 2 * size - 1; ++k) {
            for (int i1 = 0; i1 < size; ++i1) {
                for (int i2 = 0; i2 < size; ++i2) {
                    // Calculate the second coordinate of the robots' positions based on k
                    int j1 = k - i1, j2 = k - i2;
                    // Skip out-of-bounds coordinates or thorns
                    if (j1 < 0 || j1 >= size || j2 < 0 || j2 >= size || grid[i1][j1] == -1 || grid[i2][j2] == -1) continue;

                    // Cherries picked up by both robots, don't double count if same cell
                    int cherries = grid[i1][j1];
                    if (i1 != i2) cherries += grid[i2][j2];

                    // Check all combinations of previous steps
                    for (int prevI1 = i1 - 1; prevI1 <= i1; ++prevI1) {
                        for (int prevI2 = i2 - 1; prevI2 <= i2; ++prevI2) {
                            if (prevI1 >= 0 && prevI2 >= 0) { // Boundary check
                                // Update the dp value for the maximum cherries collected
                                dp[k][i1][i2] = max(dp[k][i1][i2], dp[k - 1][prevI1][prevI2] + cherries);
                            }
                        }
                    }
                }
            }
        }
        // Return the max of 0 and the final cell to account for negative values
        return max(0, dp[2 * size - 2][size - 1][size - 1]);
    }
};

```

Typescript Solution

```typescript
// Define the function cherryPickup to calculate the maximum number of cherries that can be collected.
/**
 * @param grid A 2-D grid representing the field with cherries, where -1 is a thorn, and other cells contain cherries.
 * @returns The maximum number of cherries that can be collected.
 */
function cherryPickup(grid: number[][]): number {
    const gridSize: number = grid.length;
    const maxK: number = gridSize * 2 - 1;
    const dp: number[][][] = new Array(maxK);

    // Initialize the dp (dynamic programming) array with very small numbers to indicate unvisited paths.
    for (let k = 0; k < maxK; ++k) {
        dp[k] = new Array(gridSize);
        for (let i = 0; i < gridSize; ++i) {
            dp[k][i] = new Array(gridSize).fill(-1e9);
        }
    }

    // Start from the top-left corner
    dp[0][0][0] = grid[0][0];

    // Iterate through all possible steps
    for (let k = 1; k < maxK; ++k) {
        for (let i1 = 0; i1 < gridSize; ++i1) {
            for (let i2 = 0; i2 < gridSize; ++i2) {
                const j1: number = k - i1;
                const j2: number = k - i2;

                // Avoid invalid and thorny paths
                if (
                    j1 < 0 ||
                    j1 >= gridSize ||
                    j2 < 0 ||
                    j2 >= gridSize ||
                    grid[i1][j1] == -1 ||
                    grid[i2][j2] == -1
                ) {
                    continue;
                }

                let cherriesPicked: number = grid[i1][j1];
                // Avoid double counting if both paths converge on a cell
                if (i1 !== i2) {
                    cherriesPicked += grid[i2][j2];
                }

                // Explore all possible moves from previous steps
                for (let x1 = i1 - 1; x1 <= i1; ++x1) {
                    for (let x2 = i2 - 1; x2 <= i2; ++x2) {
                        if (x1 >= 0 && x2 >= 0) {
                            dp[k][i1][i2] = Math.max(dp[k][i1][i2], dp[k - 1][x1][x2] + cherriesPicked);
                        }
                    }
                }
            }
        }
    }

    // The final result is the maximum cherries collected, ensuring at least 0 if no path is found.
    return Math.max(0, dp[maxK - 1][gridSize - 1][gridSize - 1]);
}

```

Time and Space Complexity

The given Python code represents a dynamic programming approach to solve the problem of picking cherries in a grid, where two persons start from the top-left corner and move to the bottom-right corner collecting cherries and avoiding thorns.

Time Complexity:

The time complexity is determined by the number of states in the dynamic programming matrix and the number of operations needed to compute each state. There are 3 nested loops:

1. The outer `k` loop which goes from `1` to `2n - 2`. This gives us a `O(2n - 1)` complexity factor.
2. The two inner `i1` and `i2` loops, each ranging from `0` to `n - 1`. This gives us a `O(n^2)` complexity factor.
3. Finally, the two most inner loops iterate over `x1` and `x2` which have a constant range of `-1` to `1`. This results in a constant time complexity, which in this case can be considered `O(1)` since it doesn't scale with `n`.

Multiplying these together gives us a time complexity of `O((2n - 1) * n^2 * 1)`, which simplifies to `O(2n^3 - n^2)`. Since we're interested in big-O notation, we drop the constant coefficients and lower order terms, simplifying further to `O(n^3)`.

Space Complexity:

The space complexity is determined by the size of the `dp` array. The `dp` array has dimensions `len(grid) << 1 - 1` by `n` by `n`, which results in space complexity of `O((2n - 1) * n^2)`. This simplifies to `O(2n^3 - n^2)`. With big-O notation, we simplify this further to `O(n^3)` since the squared term is negligible compared to the cubic term for large `n`.

In conclusion, both the time and space complexities are `O(n^3)`.