1463. Cherry Pickup II

Hard | Array | Dynamic Programming | Matrix | Leetcode Link

Problem Description

In this problem, we're given a 2D grid representing a field of cherries, where each cell contains a certain number of cherries. We have two robots with the goal to collect as many cherries as possible. One robot starts at the top-left corner of the grid, and the other starts at the top-right corner. The objective is to move both robots to the bottom row, collecting cherries along the way, and to maximize the total number of cherries collected by both robots combined.

The robots can only move to the cell directly below them or diagonally to the left or right below them. This means from any cell (i, j), the next move can be to cells (i+1, j-1), (i+1, j), or (i+1, j+1). Additionally, if both robots end up on the same cell, only one of them can collect the cherries there, preventing double-counting.

The challenge is to find the path for both robots such that the total amount of cherries collected is maximized, considering they must follow the grid paths and cannot move outside the grid boundaries.

Intuition

To find the maximum number of cherries both robots can collect, we can use dynamic programming. This approach typically involves breaking down the problem into smaller subproblems that we solve and combine to find the final solution.

Here, we define a three-dimensional dynamic programming array `dp[i][j1][j2]` where `i` represents the current row, and `j1` and `j2` represent the columns of Robot 1 and Robot 2, respectively. The value stored in `dp[i][j1][j2]` is the maximum number of cherries both robots can collect starting from row `i` to the bottom of the grid, with Robot 1 at column `j1` and Robot 2 at column `j2`.

Since the robots can move to three different cells in the next row from their current position, we will consider all possible combinations of movements for both robots. We'll take the maximum of the valid moves, ensuring we avoid cells outside the grid boundaries.

The solution also considers that if both robots end up in the same cell, we only add the cherries from that cell once. After considering all possible movements for each row, the algorithm gradually builds up the solution, ending with the maximum count of cherries when the robots reach the bottom row.

The optimal value will be the maximum value found in `dp[m-1][j1][j2]`, which represents the last row of the grid for any column positions `j1` and `j2` of the two robots.

Solution Approach

The solution provided revolves around the principle of dynamic programming, where the problem's complexity is reduced by breaking it down into simpler subproblems and caching intermediate results for future reference.

Here, we follow these steps to implement the solution:

1. Initialization: We create two 2D arrays `f` and `g` with dimensions `n x n` where `n` is the number of columns in the grid. They will hold the current and next state's maximum cherry counts respectively. `f[0][n - 1]` is initialized with the sum of cherries from the starting positions of both robots (top-left and top-right).
2. Iteration: We iterate through each row starting from the second row, as the first row is already covered by our initialization. During each iteration through rows `i` from `1` to `m - 1` (where `m` is the number of rows), we iterate through all columns `j1` and `j2` for Robot 1 and Robot 2's positions, respectively.
3. Cherry Collection (Nested Loop): For every possible position `(i, j1, j2)` of the robots, we calculate the cherry count `x` that can be collected on row `i` considering both robots' positions, taking care not to double-count if they share the same cell.
4. Transitions (Nested Loop within Nested Loop): Within the same loop, we check all possible previous positions `(i-1, y1, y2)` where the robots could have moved from. This involves looking at the rows right above and considering three possible positions for each robot.
5. State Update: We update our next state `g[j1][j2]` with the maximum of the current value in `g[j1][j2]` and the sum of `f[y1][y2]` and `x`. This represents the maximum cherries from the previous step plus the current step's cherries.
6. Transition to Next State: Once we've computed the values for all possible positions of `g`, we swap `f` and `g` so that `f` holds our next state's values when the next row iteration begins and `g` is reset to be filled again.
7. Final Result: Upon completing the row iterations, we have the maximum number of cherries that can be collected stored in the `f` array. To find the maximum, we iterate through `f` using `product(range(n), range(n))` to produce all combinations of `j1` and `j2` locations and finding the maximum entry.

It's important to understand that dynamic programming is efficient here because we avoid recomputing maximum cherry counts for each subproblem, which would be the case in a naive recursive implementation. By storing these values in the 2D arrays, we only compute each subproblem once.

We use Python's tuple assignment to efficiently swap `f` and `g` arrays. The product function from itertools module is used to generate cartesian product of the column indices for the final result calculation. The algorithm's overall time complexity is O(mn^29), considering the size of the grid and the number of possible movements.

Example Walkthrough

To illustrate the solution approach, let's consider a small example of a grid with 3 rows and 3 columns with the following number of cherries:

```
grid = [
  [3, 1, 1],
  [2, 5, 1],
  [1, 5, 5]
]
```

Robot 1 (R1) starts at the top left (cell with 3 cherries), and Robot 2 (R2) starts at the top right (cell with 1 cherry).

Initial State:

- Initialize `f` and `g` with dimensions `3 x 3` (since we have 3 columns), and set `f[0][2]` with 4 (sum of cherries at R1's and R2's starting positions).

First Iteration (i=1, second row):

- Consider all combinations `(j1, j2)`, where `j1` and `j2` are the column indices for R1 and R2, respectively.
- R1 can move to positions `(1,0)`, `(1,1)`, and `(1,2)`, and R2 can move to positions `(1,1)`, `(1,2)`, and `(1,3)`. Note: `(1,3)` is invalid due to grid boundaries.
- For each `(j1, j2)`, collect cherries `x` without double-counting if on the same cell and update `g[j1][j2]` with the maximum cherry count considering the previous positions `(0, y1, y2)`.

Second Iteration (i=2, third row):

- Similar to the first iteration, we perform the nested loops and check all combinations of R1 and R2 moves from row 2 to row 3.
- Cherry collection and state update steps are similar but now for the third row.

Transitions and Final Result:

- After iteration through all rows, swap `f` and `g` to prepare for the next row or to conclude with the final result.
- The final result is the maximum number out of all entries in `f` to find the maximum cherries that both robots can collect.

Example Iteration Details:

1. After the first step, `f` array has `f[0][2] = 4` since both robots start at opposite ends of the first row, 3+1 cherries collected.
2. For the next iteration (second row), let's assume `j1 = 1` and `j2 = 1` (both robots are at the center). We collected 5 cherries since they both end up in the same cell.
   - When coming from `f`, check all possible `y1, y2` to ensure we're adding the max previous value of cherries, so `f[0][2]` + cherries collected (5) for these positions and update `g[1][1]` accordingly.
   - After considering all combinations, `g` may look like this:
     ```
     [0, 3, 0]
     [7, 0, 0]
     [0, 0, 0]
     ```
3. After iterating through all rows, the robots will have collected a maximum number of cherries. In the final iteration, robots could end up at `f[2][0]` and `f[2][2]` with the maximum sums obtained.
   - `f[2][0]` could have a sum of cherries collected from the first robot and `f[2][2]` for the second robot.
   - Find the maximum entry in `f` to get the result.

Through this method, we have avoided double counting, and have systematically considered each possibility to ensure that the total number of cherries collected is maximized by the end when both robots have reached the bottom row. The result we'd obtain from the `f` array would give us the max cherries collected.

Python Solution

```python
from itertools import product

class Solution:
    def cherry_pickup(self, grid: List[List[int]]) -> int:
        # Get dimensions of the grid
        rows, cols = len(grid), len(grid[0])

        # Initialize the DP table f with -1, to store the cherries picked up for each (j1, j2) pair
        f = [[-1] * cols for _ in range(cols)]

        # Initialize temporary DP table to store the next row values
        temp_dp = [[-1] * cols for _ in range(cols)]

        # Base case initialization: starting at the first row
        f[0][cols - 1] = grid[0][0] + grid[0][cols - 1]

        # Start filling the DP table from the second row
        for row in range(1, rows):
            for j1 in range(cols):
                for j2 in range(cols):
                    # Collect cherries for current positions (j1, j2), avoid double collecting if j1 == j2
                    cherries = grid[row][j1] + (0 if j1 == j2 else grid[row][j2])

                    # Transition from previous positions (y1, y2) to current (j1, j2)
                    for y1 in range(max(j1 - 1, 0), min(j1 + 2, cols)):
                        for y2 in range(max(j2 - 1, 0), min(j2 + 2, cols)):
                            if f[y1][y2] != -1:  # Valid previous state
                                temp_dp[j1][j2] = max(temp_dp[j1][j2], f[y1][y2] + cherries)

            # Swap the tables: make temp_dp the new DP table and reset temp_dp for the next iteration
            f, temp_dp = temp_dp, [[-1] * cols for _ in range(cols)]

        # Find the maximum cherries that can be collected for all (j1, j2) pairs in the last row
        return max(f[j1][j2] for j1, j2 in product(range(cols), range(cols)))
```

Java Solution

```java
import java.util.Arrays;

class Solution {
    public int cherryPickup(int[][] grid) {
        // m is the number of rows, n is the number of columns in the grid
        int m = grid.length;
        int n = grid[0].length;

        // f[] stores the max cherries collected till the previous row
        // g[] is used as a temporary array to store the results for the current row
        int[][] dpPrevious = new int[n][n];
        int[][] dpCurrent = new int[n][n];

        // Initialize both arrays with -1 to indicate unvisited cells
        for (int i = 0; i < n; ++i) {
            Arrays.fill(dpPrevious[i], -1);
            Arrays.fill(dpCurrent[i], -1);
        }

        // Since the two people start at opposite ends, we add their starting points
        dpPrevious[0][n - 1] = grid[0][0] + grid[0][n - 1];

        // Iterate through all rows starting from the second row
        for (int i = 1; i < m; ++i) {
            // Go through all possible positions of the two people (j1 and j2)
            for (int j1 = 0; j1 < n; ++j1) {
                for (int j2 = 0; j2 < n; ++j2) {
                    // x represents the cherries picked by both people;
                    // If they are at the same cell, only count the cherries once
                    int cherries = grid[i][j1] + (j1 == j2 ? 0 : grid[i][j2]);

                    // Consider all possible combinations for the previous positions of the two people
                    for (int y1 = j1 - 1; y1 <= j1 + 1; ++y1) {
                        for (int y2 = j2 - 1; y2 <= j2 + 1; ++y2) {
                            // Check if the previous positions are within the grid bounds and have been visited
                            if (y1 >= 0 && y1 < n && y2 >= 0 && y2 < n && dpPrevious[y1][y2] != -1) {
                                // Calculate the maximum cherries that can be collected for the current positions
                                dpCurrent[j1][j2] = Math.max(dpCurrent[j1][j2], dpPrevious[y1][y2] + cherries);
                            }
                        }
                    }
                }
            }

            // Swap the references of dpPrevious and dpCurrent for the next iteration
            int[][] temp = dpPrevious;
            dpPrevious = dpCurrent;
            dpCurrent = temp;

            // Clear the secondary array after swapping to prepare for the next iteration
            for (int i = 0; i < n; ++i) {
                Arrays.fill(dpCurrent[i], -1);
            }
        }

        // Answer variable to find the maximum of all dpPrevious values after completion
        int maxCherries = 0;
        for (int j1 = 0; j1 < n; ++j1) {
            for (int j2 = 0; j2 < n; ++j2) {
                maxCherries = Math.max(maxCherries, dpPrevious[j1][j2]);
            }
        }

        // Return the maximum cherries that can be collected
        return maxCherries;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int cherryPickup(vector<vector<int>>& grid) {
        int rows = grid.size(), cols = grid[0].size();

        // 'dpCurrent' holds the max cherries picked up to the i-th row for all column pairs (j1, j2).
        vector<vector<int>> dpCurrent(cols, vector<int>(cols, -1));

        // 'dpNext' to hold the temporary results for the next row computations.
        vector<vector<int>> dpNext(cols, vector<int>(cols, -1));

        // Initialize the first row situation where both persons start at the corners.
        dpCurrent[0][cols - 1] = grid[0][0] + grid[0][cols - 1];

        // Iterate over all rows of the grid starting from the second row (i=1).
        for (int i = 1; i < rows; ++i) {
            // Try all possible column positions for the first person (j1).
            for (int j1 = 0; j1 < cols; ++j1) {
                // Try all possible column positions for the second person (j2).
                for (int j2 = 0; j2 < cols; ++j2) {
                    // Cherries picked up by the both persons - If on the same cell, don't double count.
                    int cherries = grid[i][j1] + (j1 == j2 ? 0 : grid[i][j2]);

                    // Consider all possible moves from previous row to current row for both persons
                    for (int prevJ1 = j1 - 1; prevJ1 <= j1 + 1; ++prevJ1) {
                        for (int prevJ2 = j2 - 1; prevJ2 <= j2 + 1; ++prevJ2) {
                            // If both previous positions are within bounds and a valid number of cherries was picked
                            if (prevJ1 >= 0 && prevJ1 < cols && prevJ2 >= 0 && prevJ2 < cols && dpCurrent[prevJ1][prevJ2] != -1) {
                                // Take the max between the current number of cherries and the newly computed value
                                dpNext[j1][j2] = max(dpNext[j1][j2], dpCurrent[prevJ1][prevJ2] + cherries);
                            }
                        }
                    }
                }
            }
            // Update 'dpCurrent' with 'dpNext' and reset 'dpNext' for the next iteration.
            swap(dpCurrent, dpNext);
            fill(dpNext.begin(), dpNext.end(), vector<int>(cols, -1));
        }

        // Find the maximum number of cherries that can be picked by traversing the last row's 'dpCurrent'.
        int maxCherries = 0;
        for (int j1 = 0; j1 < cols; ++j1) {
            for (int j2 = 0; j2 < cols; ++j2) {
                maxCherries = max(maxCherries, dpCurrent[j1][j2]);
            }
        }
        return maxCherries;
    }
};
```

Typescript Solution

```typescript
function cherryPickup(grid: number[][]): number {
    const rowCount = grid.length; // Number of rows in the grid
    const colCount = grid[0].length; // Number of columns in the grid

    // dp1 stores the temporary results for cherry pickup
    let dp1: number[][] = new Array(colCount).fill(0).map(() => new Array(colCount).fill(-1));

    // dp2 is used for swapping with dp1 in each iteration
    let dp2: number[][] = new Array(colCount).fill(0).map(() => new Array(colCount).fill(-1));

    // Initial cherry count from both starting positions (0,0) and (0,colCount-1)
    dp1[0][colCount - 1] = grid[0][0] + grid[0][colCount - 1];

    // Iterate through each row
    for (let row = 1; row < rowCount; ++row) {
        // Go through all possible columns j1 and j2 for person 1 and 2 respectively
        for (let col1 = 0; col1 < colCount; ++col1) {
            for (let col2 = 0; col2 < colCount; ++col2) {
                // Calculate the cherry count for the current positions col1 and col2
                const cherryCount = grid[row][col1] + (col1 === col2 ? 0 : grid[row][col2]);
                // Check all neighboring positions from the previous row
                for (let prevCol1 = col1 - 1; prevCol1 <= col1 + 1; ++prevCol1) {
                    for (let prevCol2 = col2 - 1; prevCol2 <= col2 + 1; ++prevCol2) {
                        // If the new positions are within bounds and have a valid previous cherry count
                        if (prevCol1 >= 0 && prevCol1 < colCount &&
                            prevCol2 >= 0 && prevCol2 < colCount &&
                            dp1[prevCol1][prevCol2] !== -1) {
                            // Update the dp2 array with max cherry count for current positions
                            dp2[col1][col2] = Math.max(dp2[col1][col2], dp1[prevCol1][prevCol2] + cherryCount);
                        }
                    }
                }
            }
        }
        // Swap dp1 and dp2 for the next iteration
        [dp1, dp2] = [dp2, dp1];
    }

    let maxCherries = 0; // Variable to store the max cherries collected
    // Loop to find the maximum value in the last row of the dp1 array
    for (let col1 = 0; col1 < colCount; ++col1) {
        for (let col2 = 0; col2 < colCount; ++col2) {
            maxCherries = Math.max(maxCherries, dp1[col1][col2]);
        }
    }
    return maxCherries; // Return the maximum number of cherries collected
}
```

Time and Space Complexity

Time Complexity

The time complexity of the algorithm is determined by the number of loops and the operations that occur inside them. There are three nested loops with respect to `m`, `n`, `j1`, and `j2`, resulting in a time complexity of `O(m * n^2)`. Within the nested loops for `j1` and `j2`, there are two more loops for `y1` and `y2`. Each of these run through at most 3 iterations, which contributes a constant factor, not changing the asymptotic behavior. Therefore, the overall time complexity remains `O(m * n^4)`.

Space Complexity

The space complexity of the algorithm comes from the `f` and `g` 2D arrays that are used to store intermediate results. Both arrays have dimensions `n x n`, so each array requires `O(n^2)` space. Since there are two such arrays, you might initially think that this doubles the space requirement, but because they are swapped (`f, g = g, f`), the total space complexity remains `O(n^2)` as no additional space is required for the swapping operation.