1706. Where Will the Ball Fall

Medium | Depth-First Search | Array | Dynamic Programming | Matrix | Simulation — Leetcode Link

Problem Description

In the given problem, we have a box represented by a 2-D grid of size `m x n`, where `m` is the number of rows and `n` is the number of columns. There are `n` balls that we will be dropping from each column's top. Each cell in the grid has a diagonal board that can either redirect the ball to the right or to the left. The boards are represented by either a `1` (redirects to the right) or a `-1` (redirects to the left). Our objective is to determine the final column where each ball lands at the bottom, or decide that the ball is stuck in the box. A ball is considered stuck if it encounters a 'V' shaped pattern formed by two adjacent cells with diagonal boards pointing towards each other or if it is redirected into a wall.

To summarize, we need to simulate the path of each ball dropped from the top and see where, if at all, it comes out at the bottom. We must return an array where each index contains the column number where the corresponding ball exits, or `-1` if the ball gets stuck.

Intuition

When thinking about the solution, it seems evident that we need to track the journey of each ball from the moment it is dropped until it reaches the bottom or gets stuck. We can do this using Depth-First Search (DFS). The DFS will begin at the top of each column and move step by step, analyzing the board's direction in the current cell.

The intuition behind the solution is as follows:

- Start from the `0th` row of every column and attempt to trace the path of the ball to the bottom.
- If the ball is directed right (`1`), we need to ensure that the next column is also directing the ball right, otherwise, it will get stuck.
- If the ball is directed left (`-1`), we must check that the previous column also keeps directing the ball left, for the same reason as above.
- If the ball is on the leftmost or rightmost side, check if the direction of the board does not push it into the wall; if it does, the ball gets stuck.
- If none of these conditions are met, the ball proceeds to the next row in whichever direction the board is pointing.
- If the ball reaches the bottom (`mth` row), we have found the column where the ball exits.

The `dfs` function takes care of these considerations for each ball, and it's called for each column at the top of the grid. The result of these `dfs` calls is stored in an array, which is then returned as the solution.

The code uses recursion in the `dfs` function to trace the path of the ball until it either reaches the bottom (and the column is returned) or gets stuck (and `-1` is returned).

Solution Approach

The solution implements a recursive Depth-First Search (DFS) function `dfs` which takes two parameters, `i` and `j`, corresponding to the current cell's row and column indexes. It attempts to follow the ball's path from the top of the column where it is dropped, until it either falls out of the grid or gets stuck according to the rules described in the problem.

Here is a step-by-step walkthrough of the recursive `dfs` function implemented in the solution:

- Base Case: If `i` equals the number of rows `m`, it means the ball has reached the bottom of the grid. The function then returns the column index `j`, indicating the exit column.

- Boundary Check: If the ball is at the first column and the current board directs to the left (`grid[i][j] == -1`), or it's at the last column and the board directs to the right (`grid[i][j] == 1`), the ball is stuck because it will collide with the wall. In these cases, the function returns `-1`.

- V-shaped Pattern Check: If the current board directs the ball to the right (`grid[i][j] == 1`) but the next column's board directs to the left (`grid[i][j + 1] == -1`), the ball is stuck in a 'V' shaped pattern. The same check applies if the current board directs left (`grid[i][j] == -1`) and the previous column's board directs to the right (`grid[i][j - 1] == 1`). Again, the function returns `-1` if any of these scenarios occur.

- Recursive Case: If none of these conditions apply, the ball can continue to the next row. The direction is determined by the current board: if `grid[i][j] == 1`, the ball will go to the cell on the right (`j+1`), while if `grid[i][j] == -1`, it will go to the cell on the left (`j-1`). It then calls itself with the new cell's coordinates and continues the path tracing.

- Loop Through Columns: The function `findBall` sets up a loop that goes through all columns (for each ball). "dfs(0, j)" calls the recursive `dfs` function from the top row `(0)` of each column `(j)`. The results are added to the return list, which eventually contains the outcome for each ball's path.

It's important to note that the recursion in this solution serves as backtracking through the grid. At any point, if a ball cannot move downwards, it returns `-1`, effectively "backtracking" and ending that path. The use of the list comprehension `[dfs(0, j) for j in range(n)]` succinctly executes this process for each ball and compiles the results into the final list.

In summary, the algorithm employs DFS and checks for boundary conditions as well as path continuity (no 'V' pattern obstruction) to determine the ball's final position. It leverages recursion to elegantly handle the traversal of each possible pathway.

Example Walkthrough

Let's take a small grid example to illustrate the solution approach to see how the Depth-First Search (DFS) would work.

Consider a 2×3 grid (2 rows and 3 columns):

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

There will be 3 balls dropped, each aligning with the columns from left to right (column indexes 0, 1, and 2).

Ball 1 (Column 0):

- Start at `grid[0][0]`, the board is `1`, so move right to `grid[0][1]`.
- At `grid[0][1]`, the board is `-1`, so move left to `grid[1][0]`.
- At `grid[1][0]`, the board is `-1`, and we are at the leftmost column so the ball can't move and gets stuck. The result is `-1`.

Ball 2 (Column 1):

- Start at `grid[0][1]`, the board is `-1`, so move left to `grid[0][0]`.
- At `grid[0][0]`, the board is `1`, so move right to `grid[1][1]`.
- At `grid[1][1]`, the board is `1`, so move right to `grid[1][2]`.
- At `grid[1][2]`, the board is `-1`, and it's a bottom row so the ball exits from column 2. The result is `2`.

Ball 3 (Column 2):

- Start at `grid[0][2]`, the board is `1`, so the ball tries to move right but hits the wall. It gets stuck. The result is `-1`.

The DFS function traces these paths:

- For Ball 1, it gets stuck as it cannot move left from `grid[1][0]`.
- For Ball 2, it successfully exits the grid at column 2.
- For Ball 3, it hits the wall at `grid[0][2]` and gets stuck.

The return array for this example should be [-1, 2, -1], indicating the outcomes for the balls corresponding to each column.

To recap, the function `findBall` applies DFS starting from the first row of each column, determining where the ball will end. It repeatedly calls `dfs(0, j)` for each starting position `j`. The conditions in the DFS function account for situations where the ball may get stuck against a wall, in a 'V' pattern, or exit successfully. The return array is built by compiling the results of the DFS for each ball, which in this scenario is `[-1, 2, -1]`.

Python Solution

```python
from typing import List

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

        # Helper function to simulate the ball drop
        def drop_ball(row: int, col: int) -> int:
            # Base case: ball has reached the bottom row
            if row == rows:
                return col
            # Edge case: ball is at the leftmost position and is directed left
            if col == 0 and grid[row][col] == -1:
                return -1
            # Edge case: ball is at the rightmost position and is directed right
            if col == cols - 1 and grid[row][col] == 1:
                return -1
            # V-shape case: ball is directed into a wall
            if grid[row][col] == 1 and grid[row][col + 1] == -1:
                return -1
            # Inverted V-shape case: ball is directed into a wall
            if grid[row][col] == -1 and grid[row][col - 1] == 1:
                return -1
            # Recursive case: keep the ball moving according to the direction
            # If the direction is right (1), move the ball to the next row and next column
            # If the direction is left (-1), move the ball to the next row and previous column
            return drop_ball(row + 1, col + grid[row][col])

        # Simulate the drop for each ball positionally indexed from 0 to cols-1
        # and store their outcomes in a list
        return [drop_ball(0, col) for col in range(cols)]

```

Java Solution

```java
class Solution {
    // Dimensions of the grid
    private int rowCount;
    private int columnCount;
    // The grid to operate on
    private int[][] grid;

    // Finds the eventual positions of the balls
    public int[] findBall(int[][] grid) {
        // Initialize the dimensions based on the input grid
        rowCount = grid.length;
        columnCount = grid[0].length;
        // Store the grid in the field for easy access
        this.grid = grid;

        // Result array to store the final position of each ball
        int[] result = new int[columnCount];
        // Simulate the path for each ball starting from the top row
        for (int col = 0; col < columnCount; ++col) {
            result[col] = dropBall(0, col);
        }
        return result;
    }

    // Runs the simulation for a single ball drop, returning its final position
    private int dropBall(int row, int col) {
        // If the ball has reached the bottom row, return the column index
        if (row == rowCount) {
            return col;
        }
        // Check for out-of-bound scenarios or cases where the ball gets stuck
        if (col == 0 && grid[row][col] == -1 || col == columnCount - 1 && grid[row][col] == 1 ||
            grid[row][col] == 1 && grid[row][col + 1] == -1 ||
            grid[row][col] == -1 && grid[row][col - 1] == 1) {
            return -1; // Ball is stuck or fell out of the grid
        }
        // Recursively drop the ball into the next slot based on the current direction
        return grid[row][col] == 1 ? dropBall(row + 1, col + 1) : dropBall(row + 1, col - 1);
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // Declaring class member variables for the grid dimensions.
    int rows, cols;
    vector<vector<int>> board;

    vector<int> findBall(vector<vector<int>>& grid) {
        // Initialize the member variables with the input grid.
        this->board = grid;
        rows = grid.size();
        cols = grid[0].size();

        // Prepare the answer vector which will hold the final positions of the balls.
        vector<int> finalPositions(cols);

        // Process each ball starting from the top of each column.
        for (int col = 0; col < cols; ++col) {
            finalPositions[col] = dropBall(0, col);
        }

        // Return the computed final positions of the balls.
        return finalPositions;
    }

    int dropBall(int row, int col) {
        // If we reach the bottom row, return the current column as the ball's final position.
        if (row == rows) return col;

        // Handle the cases where the ball is stuck:
        // - at the left edge with a left-tilted cell
        // - at the right edge with a right-tilted cell
        // - in a 'V' shaped position
        // - in an inverted 'V' shaped position
        if ((col == 0 && board[row][col] == -1) ||
            (col == cols - 1 && board[row][col] == 1) ||
            (board[row][col] == 1 && board[row][col + 1] == -1) ||
            (board[row][col] == -1 && board[row][col - 1] == 1)) {
            return -1; // Ball is stuck and does not have a final position.
        }

        // If none of the above cases apply,
        // continue the recursion in the direction that the cell tilts.
        return board[row][col] == 1 ? dropBall(row + 1, col + 1) : dropBall(row + 1, col - 1);
    }
};

```

Typescript Solution

```typescript
function findBall(grid: number[][]): number[] {
    // Get the number of rows (m) and columns (n) from the grid dimensions.
    const rows = grid.length;
    const columns = grid[0].length;

    // Initialize the result array with column positions where the ball will exit.
    const result = new Array(columns).fill(-1); // Preset with -1 to indicate that the ball is stuck.

    // Helper function to perform deep-first search to find the ball's path.
    const findBallPathDFS = (rowIndex: number, colIndex: number): number => {
        // If the ball reached the bottom row, return the current column index.
        if (rowIndex === rows) {
            return colIndex;
        }
        // If the ball is in a V-shaped path (pointing to the right)...
        if (grid[rowIndex][colIndex] === 1) {
            // If the ball is at the rightmost column or has a wall to the right, it's stuck.
            if (colIndex === columns - 1 || grid[rowIndex][colIndex + 1] === -1) {
                return -1;
            }
            // Otherwise, move the ball down and right.
            return findBallPathDFS(rowIndex + 1, colIndex + 1);
        } else {
            // If the ball is in a V-shaped path (pointing to the left)...
            // If the ball is at the leftmost column or has a wall to the left, it's stuck.
            if (colIndex === 0 || grid[rowIndex][colIndex - 1] === 1) {
                return -1;
            }
            // Otherwise, move the ball down and left.
            return findBallPathDFS(rowIndex + 1, colIndex - 1);
        }
    };

    // Iterate over each column index to simulate the falling of the ball from that column.
    for (let columnIndex = 0; columnIndex < columns; columnIndex++) {
        // Store the result of each ball's path in the result array.
        result[columnIndex] = findBallPathDFS(0, columnIndex);
    }

    // Return the updated result array with each ball's exit column (or -1 if stuck).
    return result;
}

```

Time and Space Complexity

Time Complexity

The function `findBall` entails a depth-first search (DFS) algorithm that recursively computes the path of each ball within the grid until it reaches the bottom or encounters a blockage.

Since each ball is independent of the others, the complexity for a single ball is dependent on the depth of the recursion, which in this case is the number of rows `m`. The DFS does not necessarily traverse every row for every ball, as it can be terminated early in case the ball gets stuck. However, in the worst-case scenario where no early termination happens, the recursion depth would be equivalent to `m`.

Given that we are performing this operation independently for each of the `n` columns, the overall worst-case time complexity scales linearly with the number of balls, leading to a final time complexity of `O(m * n)`.

Space Complexity

The space complexity primarily stems from the depth of the recursive calls which can go as deep as the number of rows `m`. In the worst case where the recursion is not terminated early by a blockage, the recursion stack may grow to a depth of `m`. Therefore, the worst-case space complexity is `O(m)` due to the recursion stack.

The non-recursion related space consumption is constant as there is a fixed number of variables being used, and no additional data structures that scale with the input size are being maintained. Thus, aside from the space taken up by the input grid itself which we do not usually count in space complexity analysis, the extra space used by the algorithm is proportional to the recursive depth.