1861. Rotating the Box

Medium Array Two Pointers Matrix

Leetcode Link

Problem Description

In this problem, we are provided with a 2D matrix `box`, which represents the side view of a box containing stones (`'#'`), stationary obstacles (`'*'`), and empty spaces (`'.'`). Our goal is to simulate what happens to the stones when the box is rotated 90 degrees clockwise. Importantly, gravity comes into play and the stones may fall down towards the new bottom of the box until they either hit the bottom, an obstacle, or another stone. Unlike the stones, obstacles do not move due to gravity. The problem guarantees that before the rotation, each stone is stable, meaning it's either on the bottom, on an obstacle, or on another stone.

The task is to return a new matrix that represents the final state of the box after rotation and gravity have affected the position of the stones.

Intuition

The solution approach can be divided into two key parts - rotation and gravity handling:

1. Rotate the box by 90 degrees clockwise: This step requires transforming the coordinates of each cell in the original box. The original cell at `(i, j)` will move to `(j, m - i - 1)` in the rotated matrix, where `m` is the number of rows in the original matrix.

2. Apply gravity to stones: This step simulates the effect of gravity on the stones. After rotation, we need to process each new column (which corresponds to a row in the original box from bottom to top) and let each stone fall down. This is done by keeping track of empty spaces where a stone could fall. We can use a queue to remember the position of the empty spaces. When we encounter a stone, we try to let it fall to the lowest available empty space tracked by our queue. If we encounter an obstacle, we reset the queue as stones cannot pass through obstacles.

By sequentially applying these two steps to the initial box, we achieve the desired final state.

Solution Approach

The solution is implemented in a few clear steps:

1. **Rotation**: To perform the 90-degree clockwise rotation of the `m x n` matrix, we need to create a new `n x m` matrix. For each cell in the original matrix, which is at position `(i, j)`, we place it in the new matrix at position `(j, m - i - 1)`. This effectively rotates the matrix while preserving the relative positions of stones, obstacles, and empty spaces.

2. **Gravity simulation**: After the rotation, we need to simulate gravity. This is where an understanding of the new orientation is crucial. What were originally the rows of the original box now become columns, and gravity will act 'downwards' along these new columns.

   To handle the gravity, we start at the bottom-most cell of each new column (which corresponds to the right-most cells of the original rows before rotation) and move upward:

   a. If we find a stone (`'#'`), we check if we have previously encountered any empty space (which would be `deque` in the example). If we have, we place the stone in the lowest encountered empty space position, which we can get by popping from the left side of the `deque`. Then we mark the original stone position as empty.

   b. If we find an obstacle (`'*'`), we clear our `deque` because stones cannot pass through obstacles, so any encountered empty spaces above the obstacle can no longer be filled by stones from below.

   c. If we find an empty space (`'.'`), we add its position to our `deque`, as it's a potential space where a stone could fall if a stone is encountered later.

By sequentially rotating the box and then applying gravity, we correctly simulate the effect of the box's rotation on the stones. The use of a `deque` is integral to tracking the potential 'fall points' for the stones and ensures that we process them in the right order (the 'lowest' empty space the stone can fall to). Once all columns have been processed, the `ans` matrix fully represents the state of the box after rotation and gravity have been applied.

Example Walkthrough

Let's illustrate the solution approach using a small example:

Suppose we have the following `box` matrix:

```
[
  ['#', '.', '#'],
  ['#', '*', '.'],
  ['#', '#', '.']
]
```

This box is a `3x3` matrix where the first column contains stones stacked on top of each other, the second column has a stone on top, an obstacle in the middle, and an empty space at the bottom, and the third column has two stones on top with an empty space at the bottom.

Step 1: Rotation

If we rotate the box 90 degrees clockwise, the positions change as follows:

- The top row of the original matrix (`['#', '.', '#']`) becomes the **first column** of the rotated matrix.
- The middle row of the original matrix (`['#', '*', '.']`) becomes the **second column** of the rotated matrix.
- The bottom row of the original matrix (`['#', '#', '.']`) becomes the **third column** of the rotated matrix.

So after rotation, our matrix `ans` looks like this:

```
[
  ['#', '#', '#'],
  ['.', '*', '#'],
  ['#', '.', '.']
]
```

Step 2: Gravity simulation

Now we simulate gravity on the rotated matrix:

- For the **first column**, no stones are above empty spaces, so no changes occur after simulating gravity.
- For the **second column**, the stone at position `[1, 1]` (`'*'`) is an obstacle, and it doesn't fall. There are no stones above it, so this column remains unchanged as well.
- For the **third column**, we have one stone at `[1, 2]` which is above two empty spaces. It will fall to the bottom-most empty space at `[2, 2]`. After this stone falls, the column looks like `['.', '.', '#']`.

Finally, after applying gravity to each column, the matrix `ans` is:

```
[
  ['#', '#', '#'],
  ['.', '*', '.'],
  ['#', '.', '#']
]
```

This matrix represents the final state of the box after rotation and the effect of gravity on the stones. The stone from the third column of the rotated matrix has fallen down, and we now have a stone at the bottom of the last column.

Python Solution

```python
from collections import deque

class Solution:
    def rotate_the_box(self, box: List[List[str]]) -> List[List[str]]:
        # First, get the dimensions of the box
        rows, cols = len(box), len(box[0])

        # Initialize the answer matrix with None, rotated 90 degrees
        rotated_box = [[None] * rows for _ in range(cols)]

        # Rotate the box 90 degrees clockwise to the right
        for row in range(rows):
            for col in range(cols):
                rotated_box[col][rows - row - 1] = box[row][col]

        # For each column of the rotated box (which were rows in the original box)
        for col in range(rows):
            queue = deque()  # Initialize a queue to store the positions of empty '.' slots
            # Start from bottom and go upwards
            for row in reversed(range(cols)):
                # When we see an obstacle '*', we clear the queue as it can't be passed
                if rotated_box[row][col] == '*':
                    queue.clear()
                # If it's empty '.', add this position to the queue
                elif rotated_box[row][col] == '.':
                    queue.append(row)
                # When we find a stone '#', and there is an available position below it (recorded in the queue)
                elif queue:
                    new_pos = queue.popleft()  # Take the lowest available position
                    rotated_box[new_pos][col] = '#'  # Move the stone to the new position
                    rotated_box[row][col] = '.'  # Update the old position to empty '.'
                    queue.append(row)  # The old position is now a new empty position
        return rotated_box

```

Java Solution

```java
class Solution {

    // Method to rotate the box and let the stones fall down due to gravity
    public char[][] rotateTheBox(char[][] box) {
        int rows = box.length; // Number of rows in the box
        int cols = box[0].length; // Number of columns in the box
        char[][] rotatedBox = new char[cols][rows]; // Resultant array after rotation

        // Rotate the box by 90 degrees clockwise. The bottom becomes the right side.
        for (int row = 0; row < rows; ++row) {
            for (int col = 0; col < cols; ++col) {
                rotatedBox[col][rows - row - 1] = box[row][col];
            }
        }

        // Simulate gravity after rotation. Stones fall down to the bottom (right side of the original box).
        for (int col = 0; col < rows; ++col) {
            Deque<Integer> emptySpaces = new ArrayDeque<>(); // Queue to track empty spaces (.)
            for (int row = cols - 1; row >= 0; --row) {
                // If there's an obstacle, clear the queue as we can't move stones across obstacles.
                if (rotatedBox[row][col] == '*') {
                    emptySpaces.clear();
                }
                // If the cell is empty, keep track of the row index.
                else if (rotatedBox[row][col] == '.') {
                    emptySpaces.offer(row);
                }
                // If there's a stone, move it down if there's an empty space under it.
                else if (!emptySpaces.isEmpty()) {
                    rotatedBox[emptySpaces.pollFirst()][col] = '#'; // Move stone to the next empty space.
                    rotatedBox[row][col] = '.'; // Make the original place of the stone empty.
                    emptySpaces.offer(row); // Now the old place of the stone is an empty space.
                }
            }
        }
        return rotatedBox; // Return the box after rotation and gravity simulation
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<vector<char>> rotateTheBox(vector<vector<char>>& box) {
        // Get dimensions of the box
        int rows = box.size(), cols = box[0].size();

        // Initialize the answer matrix with dimensions swapped (rotated)
        vector<vector<char>> rotatedBox(cols, vector<char>(rows));

        // Rotate the box clockwise by 90 degrees
        for (int row = 0; row < rows; ++row) {
            for (int col = 0; col < cols; ++col) {
                // Assign values from the original box to the rotated box
                rotatedBox[col][rows - row - 1] = box[row][col];
            }
        }

        // Handle gravity - make stones fall down after rotation
        for (int col = 0; col < rows; ++col) {
            queue<int> emptySpaces;  // Queue to keep track of empty spaces where stones can fall

            for (int row = cols - 1; row >= 0; --row) {
                if (rotatedBox[row][col] == '*') {
                    // Encountered an obstacle; clear the queue of empty spaces
                    queue<int> tempQueue;
                    swap(tempQueue, emptySpaces);
                } else if (rotatedBox[row][col] == '.') {
                    // Found an empty space; add its position to the queue
                    emptySpaces.push(row);
                } else if (!emptySpaces.empty()) {
                    // We found a stone and there's an empty space below it
                    // Move the stone down to the nearest empty space
                    rotatedBox[emptySpaces.front()][col] = '#';
                    // Mark the old position as empty
                    rotatedBox[row][col] = '.';
                    // Update the queue for the next available empty space
                    emptySpaces.push(row);
                    // Remove the space just filled from the queue
                    emptySpaces.pop();
                }
            }
        }

        // Return the rotated box with stones fallen due to gravity
        return rotatedBox;
    }
};

```

Typescript Solution

```typescript
type Box = char[][];  // Define type alias for readability

// Rotate the box by 90 degrees and let the stones fall with gravity
function rotateTheBox(box: Box): Box {
    // Get the dimensions of the box
    const rows: number = box.length;
    const cols: number = box[0].length;

    // Initialize the answer matrix with swapped dimensions (rotated box)
    let rotatedBox: Box = new Array(cols).fill(null).map(() => new Array(rows).fill('.'));

    // Rotate the box clockwise by 90 degrees
    for (let row = 0; row < rows; ++row) {
        for (let col = 0; col < cols; ++col) {
            // Assign values from the original box to the rotated box
            rotatedBox[col][rows - row - 1] = box[row][col];
        }
    }

    // Process gravity in the rotated box
    for (let col = 0; col < rows; ++col) {
        let emptySpaces: number[] = [];  // Track empty spaces where stones can fall

        for (let row = cols - 1; row >= 0; --row) {
            if (rotatedBox[row][col] === '*') {
                // Encountered an obstacle, reset the list of empty spaces
                emptySpaces = [];
            } else if (rotatedBox[row][col] === '.') {
                // Found an empty space, add its position to the list
                emptySpaces.push(row);
            } else if (emptySpaces.length > 0) {
                // Found a stone and there's an empty space below it
                // Move the stone down to the nearest empty space
                rotatedBox[emptySpaces[0]][col] = '#';
                // Mark the old position as empty
                rotatedBox[row][col] = '.';
                // Add the position of the moved stone to the list of empty spaces and remove the one just filled
                emptySpaces.shift();
                emptySpaces.push(row);
            }
        }
    }

    // Return the rotated box with stones affected by gravity
    return rotatedBox;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed in two main parts: rotation and gravity simulation.

- **Rotation**: The nested loops used for rotating the box iterate `m` times for each of the `n` columns, which results in a time complexity of `O(m * n)` for this part.

- **Gravity simulation**: The inner loop for gravity simulation also iterates `n` times for each of the `m` columns. However, while the `deque` operations (`popleft` and `append`) are `O(1)` in average cases, clearing the `deque` can also be considered to have an amortized time complexity of `O(1)` since each stone (`#`) or obstacle (`*`) needs to be moved or evaluated only once across the entire process.

Hence, the time complexity of the gravity simulation is also `O(m * n)`.

Combining both parts, the overall time complexity of the algorithm remains `O(m * n)` since these parts are sequential and not nested on top of each other.

Space Complexity

The space complexity can be primarily attributed to the space needed for the rotated `ans` array and the `deque` used for gravity simulation.

- The `ans` array is of size `m * n`, which is the same as the input box size, hence giving us a space complexity of `O(m * n)` for the `ans` array.

- The space used by the `deque` could at most hold `n` positions in the worst case, which would occur if there was a column filled with empty spaces followed by a stone at the bottom. Therefore, the space complexity for the `deque` is `O(n)`.

Since `O(m * n)` is the dominating term, the overall space complexity of the given code is `O(m * n)`.

In summary:

- Time Complexity: `O(m * n)`
- Space Complexity: `O(m * n)`