1139. Largest 1-Bordered Square

Medium | Array | Dynamic Programming | Matrix | Leetcode Link

Problem Description

The problem presents us with a 2-dimensional grid made up of `0`s and `1`s. The task is to determine the size of the largest square subgrid where all the cells on its border are filled with `1`s. The size of a subgrid is defined by the number of elements it contains. If no such subgrid exists, we are to return a size of `0`. This is a classic problem for exploring the concepts of dynamic programming in a 2D grid structure.

Intuition

The solution is built upon a dynamic programming approach, where we prepare auxiliary arrays `down` and `right` to store the maximum number of consecutive `1`s below each cell (including the cell itself) and to the right of each cell (including the cell itself), respectively.

The process is as follows:

- We iterate over the grid from bottom to top and from right to left. This reverse order helps us easily calculate the consecutive `1`s for the `down` and `right` arrays as we can simply add `1` to the count found in the next row or column.
- After populating the `down` and `right` arrays, we look for the largest square subgrid with all `1`s on its border. We do this by trying to find a square of size `k` (starting from the largest possible size and decreasing the size) where in the top left corner of this square, both the number of consecutive `1`s downward and rightward from this cell is at least `k`.
- We also need to ensure that the bottom-left and top-right corners of the potential subgrid have at least `k` consecutive `1`s upwards and leftwards to ensure the border is made entirely of `1`s.
- We do this by iterating over all possible positions of the grid where a square of size `k` can fit. If we find such a square, we immediately return its size (which is `k*k`).
- If no such square is found after checking all potential positions for all potential sizes, we return `0`.

Using this approach, the algorithm efficiently identifies the largest square subgrid with all `1`s on its border without checking every possible subgrid exhaustively, thus reducing the time complexity significantly compared to a naive approach.

Solution Approach

The implementation of the solution employs dynamic programming. We create two matrices called `down` and `right`. Both matrices have the same dimensions as the original grid. Each entry in `down[i][j]` represents the number of consecutive `1`s below (and including) grid cell `(i, j)`, extending all the way down to the bottom of the grid or until a `0` is encountered. Similarly, `right[i][j]` represents the number of consecutive `1`s to the right (and including) grid cell `(i, j)`, extending all the way to the right side of the grid or until a `0` is encountered.

Here is how the implementation approach breaks down:

1. Iterate over the grid from the bottom-right corner to the top-left corner. This iteration order is critical because it allows us to build the `down` and `right` matrices by looking at the cells that have already been processed.

2. For each cell `(i, j)`, we check if it contains a `1`. If it does, we set `down[i][j]` to `down[i + 1][j] + 1` and `right[i][j]` to `right[i][j + 1] + 1`. If the cell is at the last row or column, we simply set `down[i][j]` or `right[i][j]` to `1`. This is because there are no cells below or to the right to continue the consecutive sequence.

3. Once the `down` and `right` matrices are filled, we iterate over all potential squares starting from the largest possible size `k` and decrease the size until we either find a valid square or finish checking all sizes.

4. We look for a square starting at cell `(i, j)` with size `k`. To verify if the square's borders consist entirely of `1`s, we check four conditions:

   - `down[i][j]` should be greater than or equal to `k`.
   - `right[i][j]` should be greater than or equal to `k`.
   - `right[i + k - 1][j]` (bottom-left corner of the square) should be greater than or equal to `k`.
   - `down[i][j + k - 1]` (top-right corner of the square) should be greater than or equal to `k`.

5. If a square satisfying these conditions is found, we immediately return the size of the square, `k * k`, since we are iterating from the largest to the smallest possible size, ensuring that the first valid square found is also the largest.

6. If no valid square is found after checking all possibilities, which means no square with borders made entirely of `1`s exists, we return `0`.

This approach uses the dynamic programming paradigm to build solutions to subproblems (the number of consecutive `1`s extending downward and rightward from each cell) and uses those solutions to solve the bigger problem (finding the largest square with `1`s on its border).

Example Walkthrough

Let's illustrate the solution approach with a small example grid:

```
1 1 1 1 0
1 1 1 1 1
1 1 0 1 1
1 1 1 1 1
0 1 1 1 1
```

Following the steps of the implementation approach:

1. **Iterate over the grid from the bottom-right corner to the top-left corner.**

   We start from the bottom-right cell `(4, 4)` of the grid and move towards the top-left cell `(0, 0)`, calculating the `down` and `right` values for each cell.

2. **For each cell `(i, j)`, we check if it contains a `1`.**

   For instance, at cell `(4, 4)`, it contains a `1`. Since it's in the last row and last column, `down[4][4]` and `right[4][4]` will both be set to `1`.

3. **Fill the `down` and `right` matrices.**

   Here's what the matrices look like after being filled based on step 2:

   `down` matrix:

   ```
   1 3 3 1 0
   1 2 2 1 1
   1 1 0 1 1
   1 1 1 1 1
   0 1 1 1 1
   ```

   `right` matrix:

   ```
   4 3 2 1 0
   4 3 2 2 1
   3 2 0 2 1
   2 1 1 1 1
   1 1 1 1 1
   ```

4. **Iterate over all potential squares**

   Starting with `k = 4` (the largest possible square based on the smallest grid dimension), check if a square of that size can exist beginning from each cell `(i, j)`.

   From the `right` and `down` matrices, we can see that there's no cell that has both `down` and `right` values of `4`. The maximum value is `3`, so we can't form a square of size `4`.

   Decrease `k` to `3` and repeat the process. Now we have multiple cells with both `down` and `right` values of at least `3`. We choose one, say cell `(1, 1)`, and check the conditions for the square's bottom-left and top-right corners as well. Since the bottom-left `(3, 1)` has a `down` value of `1`, which is less than `3`, this cell does not form a valid square of size `3`.

   We continue to check other potential top-left corners of squares of size `3`, eventually finding that cell `(0, 1)` satisfies all conditions:

   - `down[0][1] >= 3`
   - `right[0][1] >= 3`
   - `right[0 + 3 - 1][1] (bottom-left corner) = right[2][1] >= 3`
   - `down[0][1 + 3 - 1] (top-right corner) = down[0][3] >= 3`

5. **Return the size of the found square**

   Since we've found a valid square of size `3` whose all border cells are `1`, we return the size of this square: `3 * 3` which is `9`.

6. **If no valid square is found**

   However, if we were unable to find a valid square of size `3`, we would decrease `k` to `2`, then to `1`, and if still no valid square is found, we would return `0`. In this example, we found a square, so we return `9`.

Through this process, we efficiently determined the largest square subgrid with all `1`s on its border without needing to inspect every possible subgrid. This dynamic programming approach reduces the time complexity and provides a methodical way to find the solution.

Python Solution

```python
class Solution:
    def largest1BorderedSquare(self, grid: List[List[int]]) -> int:
        # Get the dimensions of the grid
        rows, cols = len(grid), len(grid[0])

        # Initialize dynamic programming tables to store the count
        # of continuous '1's until the current cell from the top and from the left.
        down = [[0] * cols for _ in range(rows)]
        right = [[0] * cols for _ in range(rows)]

        # Fill in the tables with the count of '1's
        for row in range(rows - 1, -1, -1):
            for col in range(cols - 1, -1, -1):
                if grid[row][col] == 1:
                    # If on the bottom or right edge, count is 1, otherwise add from accumulator
                    down[row][col] = 1 + (down[row + 1][col] if row + 1 < rows else 0)
                    right[row][col] = 1 + (right[row][col + 1] if col + 1 < cols else 0)

        # Iterate from the largest possible square to the smallest
        for max_side in range(min(rows, cols), 0, -1):
            # Check every possible position for the top-left corner of the square
            for row in range(rows - max_side + 1):
                for col in range(cols - max_side + 1):
                    # Check if there are enough '1's to form the sides of the square
                    if (down[row][col] >= max_side and
                        right[row][col] >= max_side and
                        right[row + max_side - 1][col] >= max_side and
                        down[row][col + max_side - 1] >= max_side):
                        # If a square is found, return its area
                        return max_side * max_side

        # Return 0 if no 1-bordered square is found
        return 0

```

Java Solution

```java
class Solution {
    public int largest1BorderedSquare(int[][] grid) {
        int rows = grid.length; // the number of rows in the grid
        int cols = grid[0].length; // the number of columns in the grid
        int[][] bottom = new int[rows][cols]; // DP matrix to store consecutive 1's from current cell to bottom
        int[][] right = new int[rows][cols]; // DP matrix to store consecutive 1's from current cell to right

        // Populate the bottom and right matrix with consecutive counts of 1's
        for (int i = rows - 1; i >= 0; --i) {
            for (int j = cols - 1; j >= 0; --j) {
                if (grid[i][j] == 1) { // If the cell has a 1, calculate the consecutive 1's
                    bottom[i][j] = (i + 1 < rows) ? bottom[i + 1][j] + 1 : 1;
                    right[i][j] = (j + 1 < cols) ? right[i][j + 1] + 1 : 1;
                }
            }
        }

        // Iterate through all possible square sizes from large to small
        for (int size = Math.min(rows, cols); size > 0; --size) {
            // Check for a square with the current size
            for (int i = 0; i <= rows - size; ++i) {
                for (int j = 0; j <= cols - size; ++j) {
                    // Verify if the current position can form a square of given size
                    if (bottom[i][j] >= size && right[i][j] >= size &&
                        right[i + size - 1][j] >= size && bottom[i][j + size - 1] >= size) {
                        return size * size; // Return the area of the square
                    }
                }
            }
        }

        return 0; // if no 1-bordered square is found, return 0
    }
}

```

C++ Solution

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

class Solution {
public:
    int largest1BorderedSquare(std::vector<std::vector<int>>& grid) {
        int rows = grid.size(), cols = grid[0].size();

        // Create 2D arrays to store the count of continuous 1's in the down and right directions
        int down[rows][cols];
        int right[rows][cols];
        // Initialize the arrays with zeros
        std::memset(down, 0, sizeof(down));
        std::memset(right, 0, sizeof(right));

        // Fill the down and right arrays with the consecutive 1's count
        for (int i = rows - 1; i >= 0; --i) {
            for (int j = cols - 1; j >= 0; --j) {
                if (grid[i][j] == 1) {
                    down[i][j] = (i + 1 < rows) ? down[i + 1][j] + 1 : 1;
                    right[i][j] = (j + 1 < cols) ? right[i][j + 1] + 1 : 1;
                }
            }
        }

        // Loop from the maximum possible size of the square to the smallest
        for (int maxSize = std::min(rows, cols); maxSize > 0; --maxSize) {
            // Check each possible position for the top-left corner of the square
            for (int i = 0; i <= rows - maxSize; ++i) {
                for (int j = 0; j <= cols - maxSize; ++j) {
                    // Check if both vertical and horizontal sides have at least 'maxSize' 1's
                    if (down[i][j] >= maxSize && right[i][j] >= maxSize &&
                        right[i + maxSize - 1][j] >= maxSize &&
                        down[i][j + maxSize - 1] >= maxSize) {
                        // If a valid square is found, return its area
                        return maxSize * maxSize;
                    }
                }
            }
        }

        // Return 0 if no 1-bordered square is found
        return 0;
    }
};

```

Typescript Solution

```typescript
function largest1BorderedSquare(grid: number[][]): number {
    const rows = grid.length;
    const cols = grid[0].length;

    // Create 2D arrays to store the count of continuous 1s in the downward and rightward directions
    const down: number[][] = Array.from({ length: rows }, () => new Array(cols).fill(0));
    const right: number[][] = Array.from({ length: rows }, () => new Array(cols).fill(0));

    // Fill the down and right arrays with the consecutive 1s count
    for (let i = rows - 1; i >= 0; i--) {
        for (let j = cols - 1; j >= 0; j--) {
            if (grid[i][j] === 1) {
                down[i][j] = (i + 1 < rows) ? down[i + 1][j] + 1 : 1;
                right[i][j] = (j + 1 < cols) ? right[i][j + 1] + 1 : 1;
            }
        }
    }

    // Check for the largest possible square by looping from the maximum possible size to the smallest
    for (let maxSize = Math.min(rows, cols); maxSize > 0; maxSize--) {
        for (let i = 0; i <= rows - maxSize; i++) {
            for (let j = 0; j <= cols - maxSize; j++) {
                // Validate if both vertical and horizontal sides have at least 'maxSize' 1s
                if (down[i][j] >= maxSize && right[i][j] >= maxSize &&
                    right[i + maxSize - 1][j] >= maxSize &&
                    down[i][j + maxSize - 1] >= maxSize) {
                    // If a valid square is found, return its area
                    return maxSize * maxSize;
                }
            }
        }
    }

    // Return 0 if no 1-bordered square is found
    return 0;
}

// Usage
const grid = [
    [1, 1, 1],
    [1, 0, 1],
    [1, 1, 1]
];
const largestSquare = largest1BorderedSquare(grid);
console.log(largestSquare);  // Outputs the area of the largest 1-bordered square

```

Time and Space Complexity

The given Python code is for finding the largest 1-bounded square in a 2D binary grid.

Time Complexity:

The time complexity of the code can be analyzed as follows:

- First, we have two nested loops that iterate over the entire grid of size `m * n` to populate the `down` and `right` matrices. This step has a complexity of `O(m * n)`.
- Next, we have a triple nested loop where the outer loop decreases `k` from `min(m, n)` to 1, and the two inner loops iterate over the `m` and `n` dimensions. In the worst case where `k` equals `min(m, n)`, the complexity is `O(m * n * min(m, n))`.

Therefore, the overall time complexity is `O(m * n) + O(m * n * min(m, n)) => O(m * n * min(m, n))`, which is dominated by the triple nested loop.

Space Complexity:

The space complexity can be analyzed by considering the additional space used:

- Two auxiliary matrices `down` and `right`, each of size `m * n`, are created. Hence, the space required for these matrices is `2 * m * n`.
- Apart from these two matrices, only a constant amount of additional space is used for indices and temporary variables.

Thus, the overall space complexity is `O(m * n)` because the space used for the `down` and `right` matrices dominates the space requirement.