221. Maximal Square

Medium Array Dynamic Programming Matrix

Problem Description

The problem presents a scenario where you are given a matrix composed of only `0`s and `1`s, which corresponds to a binary grid. Your objective is to discover the largest square formed entirely of `1`s within this matrix and then return the area of that square. This is fundamentally a problem of pattern recognition and optimization, as one needs to efficiently navigate the matrix to recognize the largest square pattern without having to examine every possible square explicitly.

Intuition

The intuitive leap for solving this problem lies in dynamic programming, which allows us to store and reuse the results of subproblems to build up to the final solution effectively. The principle is to traverse the matrix once, and at each cell that contains a `1`, determine the largest square that can be formed ending at that cell. The key observation is that the size of the largest square ending at a cell is limited by the smallest of the adjacent cells to the top, left, and top-left (diagonal). If all of these are parts of squares of `1`s, then the current cell can extend those squares by one more layer.

To achieve this, we initialize an auxiliary matrix `dp` with the same dimensions as the input matrix plus an extra row and column for padding, filled with zeros. As we iterate through each cell in the original matrix, we update the corresponding cell in the `dp` matrix. If the current cell in the original matrix is a `1`, we look at the `dp` values of the adjacent cells mentioned previously – top, left, and top-left – and find the minimum value among them. The `dp` value for the current cell is one more than this minimum value, which reflects the size of the largest square that could be formed up to that cell.

Throughout this process, we track the maximum `dp` value seen, which corresponds to the size of the largest square of `1`s found. Once the entire matrix has been traversed, this maximum value is squared to give the final area of the largest square since the area is the side length squared, and the side length is what the `dp` matrix stores.

Solution Approach

The implementation of the solution involves initializing a dynamic programming (DP) table named `dp`. This table `dp` is a 2D array with the same number of rows and columns as the input `matrix`, plus one extra for each to provide padding. The padding helps to simplify the code, as it allows us not to have special cases for the first row and first column.

Step-by-Step Implementation:

1. **Initialization:** Create a 2D list `dp` with `n + 1` rows and `n + 1` columns filled with zeros, where `n` and `n` are the row and column counts of the input `matrix`, respectively. Also, initialize a variable `mx` to zero; this will hold the length of the largest square's side found during the DP table fill-up.

2. **Iterate through matrix:** Using two nested loops, iterate through the `matrix`. The outer loop goes through each row `i`, and the inner loop goes through each column `j`.

3. **DP table update:**
   - If the current cell `matrix[i][j]` is a `'1'` (a character, not the number 1), update the DP table at `dp[i + 1][j + 1]`. The reason for `i + 1` and `j + 1` is to account for the padding; we're essentially shifting the index to ensure the top row and leftmost column in the `dp` are all zeros).
   - The update is done by taking the minimum of the three adjacent cells — `dp[i][j + 1]`, `dp[i + 1][j]`, and `dp[i][j]` — and adding 1 to it. This represents the side length of the largest square ending at `matrix[i][j]`.
4. **Track the maximum square side:** Update the `mx` variable with the maximum value of the current `dp[i + 1][j + 1]` and `mx`. This keeps track of the largest square side length found so far.

5. **Compute final area:** After completing the iteration over the entire matrix, the maximum side length of a square with only `1`s is stored in `mx`. To find the area, simply return `mx * mx`, which squares the side length to give the area.

Code Analysis:

- **DP table as memoization**: The `dp` matrix is a form of memoization that allows the algorithm to refer to previously computed results and build upon them, which dramatically reduces time complexity from exponential to polynomial.
- **Time and Space Complexity**: The time complexity of this solution is `O(m * n)` since it processes each cell exactly once. The space complexity is also `O(m * n)` due to the extra DP table used for storing intermediate results.

By applying these steps, the solution leverages dynamic programming to effectively solve the problem in a manageable time frame while ensuring that we do not perform redundant calculations.

Example Walkthrough

Let's consider a small example to illustrate the solution approach given above. Suppose we have the following binary grid as our input matrix:

```
matrix = [
  [1, 0, 1, 0, 0],
  [1, 0, 1, 1, 1],
  [1, 1, 1, 1, 1],
  [1, 0, 0, 1, 0]
]
```

1. **Initialization:** We initialize our `dp` table to have dimensions `5 x 6` (since the original matrix is `4 x 5`, we add one for padding). It looks like this after initialization:

```
dp = [
  [0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0]
]
```

And we set `mx = 0`.

2. **Iterate through matrix:** We start iterating with `i = 0` and `j = 0`. We find `matrix[0][0]` is `1`, so we need to update `dp` at `[i+1][j+1]`, which is `dp[1][1]`.

3. **DP table update:** Since `dp[1][1]`'s adjacent cells (`dp[0][1]`, `dp[1][0]`, and `dp[0][0]`) are all zeros, we take the minimum (which is `0`) and add `1` to it.

```
dp = [
  [0, 0, 0, 0, 0, 0],
  [0, 1, 0, 0, 0, 0], // dp[1][1] updated
  [0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0]
]
```

4. **Track the maximum square side:** `mx` is updated to `1`, as `1` is larger than `0` (previous `mx` value).

Continuing in this manner for all `1`s in the original matrix:

```
Final dp table after iterating through the entire matrix:

dp = [
  [0, 0, 0, 0, 0, 0],
  [0, 1, 0, 1, 0, 0],
  [0, 1, 0, 1, 1, 1],
  [0, 1, 1, 1, 2, 2],
  [0, 1, 0, 0, 1, 0]
]

Maximum square size found, mx = 2
```

5. **Compute final area:** Finally, we compute the area of the largest square found by squaring `mx`. Thus, we get `2 * 2 = 4`.

In our example, the largest square composed entirely of `1`s has a side length of `2`, and the area of that square is `4`. The solution correctly identifies this through the methodical updating of the `dp` table and maintains the `mx` variable as it iterates through the given matrix.

Python Solution

```python
from typing import List

class Solution:
    def maximalSquare(self, matrix: List[List[str]]) -> int:
        rows, cols = len(matrix), len(matrix[0]) # Get the dimensions of the matrix
        dp = [[0] * (cols + 1) for _ in range(rows + 1)] # Initialize DP table with extra row and column
        max_side_length = 0 # Maximum side length of a square of '1's

        for row in range(rows):
            for col in range(cols):
                # Check if the current element is a '1'
                if matrix[row][col] == '1':
                    # Update the DP table by considering the top, left, and top-left neighbors
                    dp[row + 1][col + 1] = min(
                        dp[row][col + 1],    # Top
                        dp[row + 1][col],    # Left
                        dp[row][col]         # Top-Left
                    ) + 1
                    # Update the max side length found so far
                    max_side_length = max(max_side_length, dp[row + 1][col + 1])

        # Return the area of the largest square
        return max_side_length * max_side_length
```

Java Solution

```java
class Solution {
    public int maximalSquare(char[][] matrix) {
        // Find the dimensions of the matrix.
        int rows = matrix.length;
        int cols = matrix[0].length;

        // Initialize a DP (Dynamic Programming) table with extra row and column.
        int[][] dp = new int[rows + 1][cols + 1];

        // Initialize the variable to store the size of the maximum square.
        int maxSquareSize = 0;

        // Loop through each cell in the matrix.
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // If the cell contains a '1', it is a potential part of a square.
                if (matrix[i][j] == '1') {
                    // The size of the square ending at (i, j) is 1 plus the minimum of
                    // the size of the squares above, to the left, and diagonally above and to the left.
                    dp[i + 1][j + 1] = Math.min(Math.min(dp[i][j + 1], dp[i + 1][j]), dp[i][j]) + 1;

                    // Update the maximum size encountered so far.
                    maxSquareSize = Math.max(maxSquareSize, dp[i + 1][j + 1]);
                }
            }
        }

        // Return the area of the largest square found.
        return maxSquareSize * maxSquareSize;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // for std::min and std::max

class Solution {
public:
    int maximalSquare(vector<vector<char>>& matrix) {
        // Get the number of rows (n) and columns (n) in the matrix.
        int numRows = matrix.size();
        int numCols = matrix[0].size();

        // Create a 2D DP (dynamic programming) table with an extra row and column set to 0.
        vector<vector<int>> dp(numRows + 1, vector<int>(numCols + 1, 0));

        int maxSize = 0; // Initialize the maximum square size found to 0.

        // Iterate through the matrix, starting from the top-left corner.
        for (int i = 0; i < numRows; ++i) {
            for (int j = 0; j < numCols; ++j) {
                // If the current element is '1', calculate the size of the square.
                if (matrix[i][j] == '1') {
                    // The size of the square ending at (i, j) is the minimum of the three
                    // neighboring squares plus 1.
                    dp[i + 1][j + 1] = std::min(std::min(dp[i][j + 1], dp[i + 1][j]), dp[i][j]) + 1;
                    // Update the maximum size found so far.
                    maxSize = std::max(maxSize, dp[i + 1][j + 1]);
                }
            }
        }

        // Return the area of the largest square found.
        return maxSize * maxSize;
    }
};
```

Typescript Solution

```typescript
function maximalSquare(matrix: string[][]): number {
    // Get the number of rows (numRows) and columns (numCols) in the matrix.
    const numRows = matrix.length;
    const numCols = matrix[0].length;

    // Create a 2D DP (dynamic programming) table with an extra row and column set to 0.
    let dp: number[][] = Array.from({ length: numRows + 1 }, () => Array(numCols + 1).fill(0));

    let maxSize: number = 0; // Initialize the maximum square size found to 0.

    // Iterate through the matrix, starting from the top-left corner.
    for (let i = 0; i < numRows; i++) {
        for (let j = 0; j < numCols; j++) {
            // If the current element is '1', calculate the size of the square.
            if (matrix[i][j] === '1') {
                // The size of the square ending at (i, j) is the minimum of the three
                // neighboring squares plus 1.
                dp[i + 1][j + 1] = Math.min(Math.min(dp[i][j + 1], dp[i + 1][j]), dp[i][j]) + 1;

                // Update the maximum size found so far.
                maxSize = Math.max(maxSize, dp[i + 1][j + 1]);
            }
        }
    }

    // Return the area of the largest square found.
    return maxSize * maxSize;
}
```

Time and Space Complexity

The time complexity of the provided code is `O(m * n)`, where `m` is the number of rows in the matrix and `n` is the number of columns. This is because the code contains two nested loops, each of which iterate over the rows and the columns of the input matrix, respectively.

The space complexity of the code is `O(m * n)`, since a 2D list `dp` of size `(m + 1) x (n + 1)` is created to store the size of the largest square ending at each position in the matrix. Each element of the matrix contributes to one cell in the `dp` array, hence the space complexity is proportional to the size of the input matrix.