Outline

- Maximal Square
- Space Optimization for Maximal Square

Maximal Square Problem

The Maximal Square Problem involves finding the largest square that can be formed in a given binary matrix. The binary matrix consists of 0s and 1s, where 1s represent parts of the square and 0s represent empty spaces.

Solution

To solve this problem, dynamic programming can be used. Create a 2D array dp where dp[i][j] represents the side length of the largest square whose bottom-right corner is at (i, j). The relation for filling the dp array is:

If matrix[i][j] is ‘1’, then :

$$dp[i][j]=\min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1.$$

If matrix[i][j] is '0', then :

dp[i][j] = 0.

This is because the largest square at position (i, j) is constrained by the squares ending at positions (i-1, j), (i, j-1), and (i-1, j-1).

Dynamic programming Properties

Optimal Substructure Property : The maximal square submatrix that ends at any position (i, j) can be determined by solving smaller subproblems — specifically, the maximal square that ends at (i-1, j), (i, j-1), and (i-1, j-1).

Overlapping Subproblems: The value of dp[i][j] depends on the previously computed values of dp[i-1][j], dp[i][j-1], and dp[i-1][j-1]. These values are reused multiple times in the calculation of other squares, leading to overlapping subproblems.

Steps for filling of 2D DP array

Initialize a 2D DP array of the same dimensions as the input matrix, with all values set to 0.

First row and First column of dp table :
We fill in the dp values for the first row and first column based on the corresponding values from the matrix. If matrix[0][j] is 1, then dp[0][j] will be 1. Otherwise, dp[0][j] will remain 0 (this is already the default value for the dp array).

Rest of the dp table:
For all other cells (i > 0 and j > 0), we apply the dynamic programming formula to find the largest square that can end at that cell. This is done by checking the minimum of the neighboring cells (above, left, and top-left diagonal), and adding 1 to it.

Final result: The maximum value in the dp table will give the side length of the largest square. We return the area of that square by squaring the value of max_side

Algorithm

```
Function MaximalSquare(matrix):
   Initialize rows = number of rows in matrix
   Initialize cols = number of columns in matrix
   Initialize max_side = 0

   Create a 2D array dp of size rows x cols, initialized to 0

   // Fill the first row of dp table
   For j = 0 to cols - 1:
      If matrix[0][j] is "1":
         dp[0][j] = 1  // The value in the first row of dp is the same as the value in matrix[0][j]

   // Fill the first column of dp table
   For i = 0 to rows - 1:
      If matrix[i][0] is "1":
         dp[i][0] = 1  // The value in the first column of dp is the same as the value in matrix[i][0]

   // Fill the rest of the dp table
   For i = 1 to rows - 1:
      For j = 1 to cols - 1:
         If matrix[i][j] is "1":
            dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1  // Square size based on neighboring values
            max_side = max(max_side, dp[i][j])  // Update the largest side length

   Return max_side * max_side  // Return the area of the largest square
End Function
```

Time & Space Complexity

Time Complexity:
The time complexity of this solution is O(m * n), where m is the number of rows and n is the number of columns in the matrix.

Space Complexity:
The space complexity is O(m * n), because we are using an additional 2D DP array of the same size as the input matrix.

Example

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

Initialize the dp array: We will initialize a dp array of the same size as the input matrix (5x4) and fill it with zeros

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

Example

Fill the first row and first column of the dp table from the given matrix:

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

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

Fill the rest of the dp table using the dynamic programming formula:

i = 1, j = 1:
matrix[1][1] = 0, so dp[1][1] = 0

i = 1, j = 2:
matrix[1][2] = 1, so dp[1][2] = min(dp[0][2], dp[1][1], dp[0][1]) + 1 = min(1, 0, 0) + 1 = 1

i = 1, j = 3:
matrix[1][3] = 1, so dp[1][3] = min(dp[0][3], dp[1][2], dp[0][2]) + 1 = min(0, 1, 1) + 1 = 1

i = 2, j = 1:
matrix[2][1] = 1, so dp[2][1] = min(dp[1][1], dp[2][0], dp[1][0]) + 1 = min(0, 1, 1) + 1 = 1

i = 2, j = 2:
matrix[2][2] = 1, so dp[2][2] = min(dp[1][2], dp[2][1], dp[1][1]) + 1 = min(1, 1, 0) + 1 = 1

Example

i = 2, j = 3:
matrix[2][3] = 1, so dp[2][3] = min(dp[1][3], dp[2][2], dp[1][2]) + 1 = min(1, 2, 0) + 1 = 2

i = 3, j = 1:
matrix[3][1] = 0, so dp[3][1] = 0

i = 3, j = 2:
matrix[3][2] = 0, so dp[3][2] = 0

i = 3, j = 3:
matrix[3][3] = 1, so dp[3][3] = min(dp[2][3], dp[3][2], dp[2][2]) + 1 = min(2, 0, 2) + 1 = 1

i = 4, j = 1:
matrix[4][1] = 0, so dp[4][1] = 0

i = 4, j = 2:
matrix[4][2] = 1, so dp[4][2] = min(dp[3][2], dp[4][1], dp[3][1]) + 1 = min(0, 0, 0) + 1 = 1

i = 4, j = 3:
matrix[4][3] = 0, so dp[4][3] = 0

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

Find the maximum square:
The largest value in the dp table is 2 (dp[2][3]).Therefore, the side length of the largest square is 2, and the area is 2 * 2 = 4

Space Optimised Maximal Square Problem

Solution

To optimize space for the Maximal Square Problem, we can reduce the space complexity **from O(m * n)** (where m is the number of rows and n is the number of columns) **to O(n)** (where n is the number of columns).

Insight:

In the DP solution, the value of dp[i][j] depends only on the values in the previous row (dp[i-1][j]), the current row (dp[i][j-1]), and the diagonal (dp[i-1][j-1]).

Therefore, instead of storing the entire dp table, we only need to store the current row and the previous row of the dp table

Algorithm

```
Function MaximalSquare(matrix):

  Initialize rows = number of rows in matrix
  Initialize cols = number of columns in matrix
  prev = Array of size (cols + 1), initialized to 0  // This represents the row before current
  curr = Array of size (cols + 1), initialized to 0  // This represents the current row

  Initialize max_side = 0  // To track the size of the largest square

  // Loop through each row of the matrix
  For i = 0 to rows - 1:
    // Loop through each column of the matrix
    For j = 0 to cols - 1:
      If matrix[i][j] is "1":
         curr[j + 1] = min(prev[j + 1], curr[j], prev[j]) + 1
         max_side = max(max_side, curr[j + 1])
      Else:
        curr[j + 1] = 0  // If the matrix cell is "0", no square can end here

    // After processing the current row, move current row to previous row
    prev = curr  // Now the current row becomes the previous row for the next iteration

 Return max_side * max_side   // Return the area of the largest square
End Function
```

These arrays are both of size cols + 1 to handle the boundary condition (dp[0][j] and dp[i][0])