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 0s and 1s. The task is to determine the size of the largest square

subgrid where all the cells on its border are filled with 1s. 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 Intuition

# grid structure.

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 1s

The solution is built upon a dynamic programming approach, where we prepare auxiliary arrays down and right to store the maximum

number of consecutive 1s below each cell (including the cell itself) and to the right of each cell (including the cell itself), respectively.

- 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 1s 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
- We also need to ensure that the bottom-left and top-right corners of the potential subgrid have at least k consecutive 1s upwards and leftwards to ensure the border is made entirely of 1s. 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).

square, both the number of consecutive 1s downward and rightward from this cell is at least 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 1s 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 1s 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 1s to the right (and including) grid cell (i, j), extending all the way to the right side of the

# Here is how the implementation approach breaks down:

grid or until a 0 is encountered.

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

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

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

the largest to the smallest possible size, ensuring that the first valid square found is also the largest.

build the down and right matrices by looking at the cells that have already been processed.

decrease the size until we either find a valid square or finish checking all sizes.

down[i][j] should be greater than or equal to k.

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

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

 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

return 0. This approach uses the dynamic programming paradigm to build solutions to subproblems (the number of consecutive 1s extending

downward and rightward from each cell) and uses those solutions to solve the bigger problem (finding the largest square with 1s on

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

Example Walkthrough

5 0 1 1 1 1 Following the steps of the implementation approach:

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

## 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.

1 1 3 3 1 0

1 4 3 2 1 0

2 4 3 2 2 1

3 3 2 0 2 1

4 2 1 1 1 1

5 1 1 1 1 1

conditions:

o down[0][1] >= 3

6. If no valid square is found

class Solution:

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

34

3

3. Fill the down and right matrices.

values for each cell.

its border).

1 1 1 1 1 0

down matrix:

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

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

5 0 1 1 1 1 right matrix:

Starting with k = 4 (the largest possible square based on the smallest grid dimension), check if a square of that size can exist

4. Iterate over all potential squares

beginning from each cell (i, j).

is 3, so we can't form a square of size 4.

o right[0][1] >= 3 o right[0 + 3 - 1][1] (bottom-left corner) = right[2][1] >= 3 o 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.

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

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

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

(3, 1) has a down value of 1, which is less than 3, this cell does not form a valid square of size 3.

found, we would return 0. In this example, we found a square, so we return 9.

def largest1BorderedSquare(self, grid: List[List[int]]) -> int:

# Initialize dynamic programming tables to store the count

# Iterate from the largest possible square to the smallest

if (down[row][col] >= max\_side and

return max\_side \* max\_side

int rows = grid.length; // the number of rows in the grid

int largest1BorderedSquare(std::vector<std::vector<int>>& grid) {

// Fill the down and right arrays with the consecutive 1's count

// Loop from the maximum possible size of the square to the smallest

for (int maxSize = std::min(rows, cols); maxSize > 0; --maxSize) {

down[i][j] = (i + 1 < rows) ? down[i + 1][j] + 1 : 1;

// Check each possible position for the top-left corner of the square

if (down[i][j] >= maxSize && right[i][j] >= maxSize &&

// If a valid square is found, return its area

right[i + maxSize - 1][j] >= maxSize &&

down[i][j + maxSize - 1] >= maxSize) {

// Check if both vertical and horizontal sides have at least 'maxSize' 1's

right[i][j] = (j + 1 < cols) ? right[i][j + 1] + 1 : 1;

// Create 2D arrays to store the count of continuous 1's in the down and right directions

int rows = grid.size(), cols = grid[0].size();

right[row][col] >= max\_side and

for col in range(cols - max\_side + 1):

# of continuous '1's until the current cell from the top and from the left.

# Check every possible position for the top-left corner of the square

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

# Get the dimensions of the grid

rows, cols = len(grid), len(grid[0])

for row in range(rows -1, -1, -1):

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 col in range(cols -1, -1, -1):

for max\_side in range(min(rows, cols), 0, -1):

for row in range(rows - max\_side + 1):

public int largest1BorderedSquare(int[][] grid) {

if grid[row][col] == 1:

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

Through this process, we efficiently determined the largest square subgrid with all 1s 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

# 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)

# Check if there are enough '1's to form the sides of the square

right[row][col] = 1 + (right[row][col + 1] if col + 1 < cols else 0)

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

#### 29 30 31 32 # Return 0 if no 1-bordered square is found 33 return 0

Java Solution

1 class Solution {

C++ Solution

1 #include <vector>

5 class Solution {

6 public:

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

42

2 #include <algorithm>

#include <cstring>

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));

for (int i = rows - 1; i >= 0; --i) {

if (grid[i][j] == 1) {

for (int  $j = cols - 1; j >= 0; --j) {$ 

for (int i = 0; i <= rows - maxSize; ++i) {</pre>

for (int j = 0; j <= cols - maxSize; ++j) {</pre>

return maxSize \* maxSize;

```
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) {
 9
10
                for (int j = cols - 1; j >= 0; --j) {
11
                    if (grid[i][j] == 1) { // If the cell has a 1, calculate the consecutive 1's
12
                        bottom[i][j] = (i + 1 < rows) ? bottom[i + 1][j] + 1 : 1;
13
                        right[i][j] = (j + 1 < cols) ? right[i][j + 1] + 1 : 1;
14
15
16
17
18
           // Iterate through all possible square sizes from large to small
19
           for (int size = Math.min(rows, cols); size > 0; --size) {
20
               // Check for a square with the current size
               for (int i = 0; i <= rows - size; ++i) {
21
22
                    for (int j = 0; j <= cols - size; ++j) {
23
                        // Verify if the current position can form a square of given size
24
                        if (bottom[i][j] >= size && right[i][j] >= size &&
25
                            right[i + size -1][j] >= size && bottom[i][j + size -1] >= size) {
26
                            return size * size; // Return the area of the square
27
28
29
30
31
32
           return 0; // if no 1-bordered square is found, return 0
33
34
35
```

### 36 37 38 39 40 41

```
43
             // Return 0 if no 1-bordered square is found
 44
             return 0;
 45
 46 };
 47
Typescript Solution
    function largest1BorderedSquare(grid: number[][]): number {
         const rows = grid.length;
         const cols = grid[0].length;
  4
  5
         // 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));
  6
         const right: number[][] = Array.from({ length: rows }, () => new Array(cols).fill(0));
  8
  9
         // Fill the down and right arrays with the consecutive 1s count
         for (let i = rows - 1; i >= 0; i--) {
 10
             for (let j = cols - 1; j >= 0; j--) {
 11
                 if (grid[i][j] === 1) {
 12
                     down[i][j] = (i + 1 < rows) ? down[i + 1][j] + 1 : 1;
 13
 14
                     right[i][j] = (j + 1 < cols) ? right[i][j + 1] + 1 : 1;
 15
 16
 17
 18
         // Check for the largest possible square by looping from the maximum possible size to the smallest
 19
 20
         for (let maxSize = Math.min(rows, cols); maxSize > 0; maxSize--) {
             for (let i = 0; i <= rows - maxSize; i++) {</pre>
 21
 22
                 for (let j = 0; j <= cols - maxSize; j++) {</pre>
 23
                     // Validate if both vertical and horizontal sides have at least 'maxSize' 1s
 24
                     if (down[i][j] >= maxSize && right[i][j] >= maxSize &&
 25
                         right[i + maxSize - 1][j] >= maxSize &&
 26
                         down[i][j + maxSize - 1] >= maxSize) {
 27
                         // If a valid square is found, return its area
 28
                         return maxSize * maxSize;
 29
 30
 31
 32
 33
 34
         // Return 0 if no 1-bordered square is found
        return 0;
 36
 37
 38 // Usage
    const grid = [
         [1, 1, 1],
        [1, 0, 1],
         [1, 1, 1]
 42
```

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

has a complexity of 0(m \* n).

Time Complexity:

n.

Time and Space Complexity

1;

43

46

• 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 0(m \* n \* min(m, n)).

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

const largestSquare = largest1BorderedSquare(grid);

console.log(largestSquare); // Outputs the area of the largest 1-bordered square

- Therefore, the overall time complexity is  $0(m * n) + 0(m * n * min(m, n)) \Rightarrow 0(m * n * min(m, n))$ , which is dominated by the triple nested loop.
- Space Complexity: The space complexity can be analyzed by considering the additional space used:
  - 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 0(m \* n) because the space used for the down and right matrices dominates the space

Two auxiliary matrices down and right, each of size m \* n, are created. Hence, the space required for these matrices is 2 \* m \*

• 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

requirement.