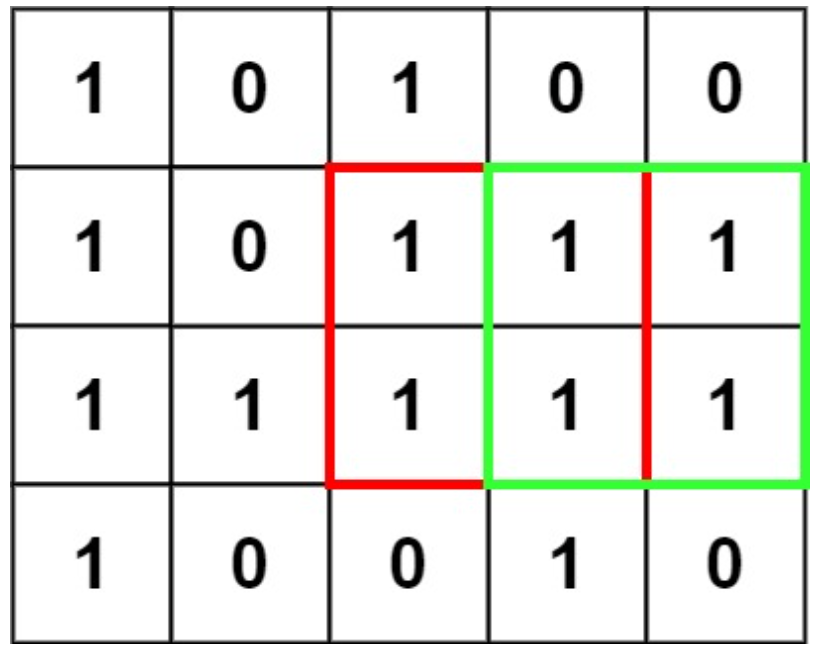
# 题目

在一个由 '0' 和 '1' 组成的二维矩阵内，找到只包含 '1' 的最大正方形，并返回其面积。

示例 1：



输入：

matrix = [["1","0","1","0","0"],

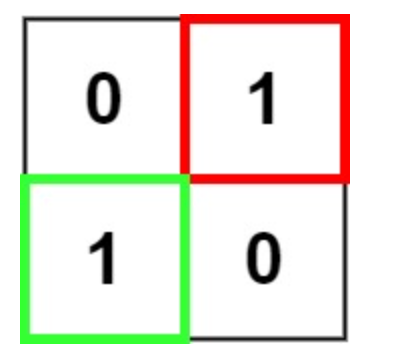
["1","0","1","1","1"],

["1","1","1","1","1"],

["1","0","0","1","0"]]

输出：4

示例 2：



输入：matrix = [["0","1"],["1","0"]]

输出：1

示例 3：

输入：matrix = [["0"]]

输出：0

提示：

m == matrix.length

n == matrix[i].length

1 <= m, n <= 300

matrix[i][j] 为 '0' 或 '1'

# 分析

## 方法一：暴力法

由于正方形的面积等于边长的平方，因此要找到最大正方形的面积，首先需要找到最大正方形的边长，然后计算最大边长的平方即可。

暴力法是最简单直观的做法，具体做法如下：

1、遍历矩阵中的每个元素，每次遇到 1，则将该元素作为正方形的左上角；

2、确定正方形的左上角后，根据左上角所在的行和列计算可能的最大正方形的边长（正方形的范围不能超出矩阵的行数和列数），在该边长范围内寻找只包含 1 的最大正方形；

3、每次在下方新增一行以及在右方新增一列，判断新增的行和列是否满足所有元素都是 1。

代码：

class Solution {

public:

int maximalSquare(vector<vector<char>>& matrix) {

if (matrix.size() == 0 || matrix[0].size() == 0) {

return 0;

}

int maxSide = 0;

int rows = matrix.size(), columns = matrix[0].size();

for (int i = 0; i < rows; i++) {

for (int j = 0; j < columns; j++) {

if (matrix[i][j] == '1') {

// 遇到一个 1 作为正方形的左上角

maxSide = max(maxSide, 1);

// 计算可能的最大正方形边长

int currentMaxSide = min(rows - i, columns - j);

for (int k = 1; k < currentMaxSide; k++) {

// 判断新增的一行一列是否均为 1

bool flag = true;

if (matrix[i + k][j + k] == '0') {

break;

}

for (int m = 0; m < k; m++) {

if (matrix[i + k][j + m] == '0' || matrix[i + m][j + k] == '0') {

flag = false;

break;

}

}

if (flag) {

maxSide = max(maxSide, k + 1);

} else {

break;

}

}

}

}

}

int maxSquare = maxSide \* maxSide;

return maxSquare;

}

};

复杂度分析：

时间复杂度：O(mnmin(m,n)2)，其中 m 和 n 是矩阵的行数和列数。

需要遍历整个矩阵寻找每个 1，遍历矩阵的时间复杂度是 O(mn)。

对于每个可能的正方形，其边长不超过 m 和 n 中的最小值，需要遍历该正方形中的每个元素判断是不是只包含 1，遍历正方形时间复杂度是 O(min(m,n)2)。

总时间复杂度是 O(mnmin(m,n)2)。

空间复杂度：O(1)。额外使用的空间复杂度为常数。

## 方法二：动态规划

要解决“找到二维矩阵中只包含‘1’的最大正方形”问题，我们可以利用动态规划（DP）高效求解。核心思路是通过DP数组记录每个位置能构成的最大正方形边长，从而避免重复计算，降低时间复杂度。

解题思路

1、问题分析：正方形的特性是“边长相等”，且由‘1’构成的最大正方形，其右下角的‘1’必然依赖于左上角、上方、左侧三个位置的正方形边长。例如，若matrix[i][j] = '1'，则该位置能构成的最大正方形边长，等于其左上角（i-1,j-1）、上方（i-1,j）、左侧（i,j-1）三个位置的最小边长加1（若其中任一位置为0，则当前位置最大边长为1）。

2、DP数组定义：设dp[i][j]表示以matrix[i][j]为右下角的正方形的最大边长。

3、状态转移方程：

若matrix[i][j] == '0'：dp[i][j] = 0（无法构成正方形）。

若matrix[i][j] == '1'：dp[i][j] = min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]) + 1（依赖三个相邻位置的最小边长，确保能构成正方形）。

4、边界处理：

第一行或第一列的位置（i=0或j=0）：若matrix[i][j] == '1'，则dp[i][j] = 1（最多只能构成边长为1的正方形）。

5、结果计算：遍历DP数组，记录最大边长max\_side，最终面积为max\_side \* max\_side。

代码实现（空间优化前）：

class Solution {

public:

int maximalSquare(vector<vector<char>>& matrix) {

if (matrix.empty() || matrix[0].empty()) return 0;

int m = matrix.size(); // 矩阵行数

int n = matrix[0].size(); // 矩阵列数

vector<vector<int>> dp(m, vector<int>(n, 0)); // DP数组

int max\_side = 0; // 最大正方形边长

for (int i = 0; i < m; ++i) {

for (int j = 0; j < n; ++j) {

if (matrix[i][j] == '1') {

// 边界情况：第一行或第一列，边长最多为1

if (i == 0 || j == 0) {

dp[i][j] = 1;

} else {

// 状态转移：取三个相邻位置的最小边长加1

dp[i][j] = min({dp[i-1][j-1], dp[i-1][j], dp[i][j-1]}) + 1;

}

// 更新最大边长

max\_side = max(max\_side, dp[i][j]);

}

}

}

return max\_side \* max\_side; // 返回面积

}

};

空间优化（O(n)空间）：

观察状态转移方程可知，dp[i][j]仅依赖于上一行的dp[i-1][j-1]、dp[i-1][j]和当前行的dp[i][j-1]。因此，可将二维DP数组优化为一维数组，降低空间复杂度：

- 设dp[j]表示当前行第j列的最大边长。

- 用变量prev记录上一行的dp[j-1]（即dp[i-1][j-1]），避免被当前行的dp[j-1]覆盖。

优化后代码：

class Solution {

public:

int maximalSquare(vector<vector<char>>& matrix) {

if (matrix.empty() || matrix[0].empty()) return 0;

int m = matrix.size();

int n = matrix[0].size();

vector<int> dp(n, 0); // 一维DP数组

int max\_side = 0;

int prev = 0; // 记录上一行的dp[j-1]（即dp[i-1][j-1]）

for (int i = 0; i < m; ++i) {

prev = 0; // 每行开始时，上一行的dp[0]（j=0时无j-1，prev初始化为0）

for (int j = 0; j < n; ++j) {

int temp = dp[j]; // 暂存当前行的dp[j]，下一轮作为prev（上一行的dp[j-1]）

if (matrix[i][j] == '1') {

if (i == 0 || j == 0) {

dp[j] = 1;

} else {

// 状态转移：prev是dp[i-1][j-1]，dp[j]是dp[i-1][j]，dp[j-1]是当前行的dp[i][j-1]

dp[j] = min({prev, dp[j], dp[j-1]}) + 1;

}

max\_side = max(max\_side, dp[j]);

} else {

dp[j] = 0; // 当前位置为0，边长重置为0

}

prev = temp; // 更新prev，为下一列的j+1做准备

}

}

return max\_side \* max\_side;

}

};

代码解释

1、空间优化前：

- 二维DP数组dp[m][n]完整记录每个位置的最大边长，逻辑清晰，适合理解。

- 时间复杂度O(m\*n)（遍历整个矩阵一次），空间复杂度O(m\*n)（存储DP数组）。

2、空间优化后：

- 一维DP数组dp[n]复用每行的空间，prev变量记录上一行的关键值，避免信息丢失。

- 时间复杂度仍为O(m\*n)，空间复杂度降至O(n)（仅存储一行的DP值），更适合大规模矩阵（如题目中m,n≤300的场景）。