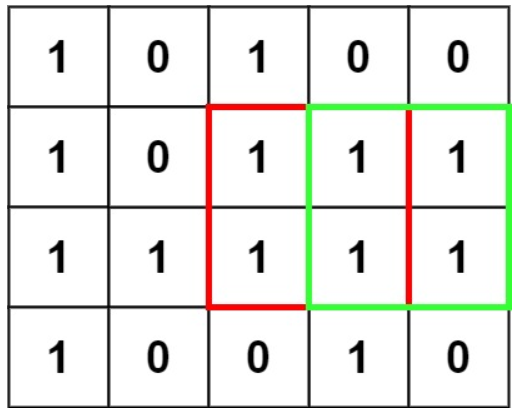
<https://leetcode.com/problems/maximal-square/>

Given an m x n binary matrix filled with 0's and 1's, *find the largest square containing only* 1's *and return its area*.

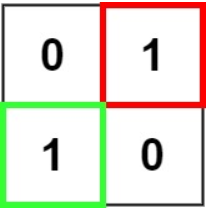
**Example 1:**



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

Output: 4

**Example 2:**



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

Output: 1

**Example 3:**

Input: matrix = [["0"]]

Output: 0

**Constraints:**

* m == matrix.length
* n == matrix[i].length
* 1 <= m, n <= 300
* matrix[i][j] is '0' or '1'.

**Attempt 1: 2023-08-28**

**Solution 1:  Brute Force (10 min, TLE 76/78, check all sizes starting at all points)**

class Solution {

public int maximalSquare(char[][] matrix) {

int rows = matrix.length;

int cols = matrix[0].length;

int n = Math.min(rows, cols);

// Assume a candidate maximum square length to begin with

for(int maxLen = n; maxLen > 0; maxLen--) {

// The top right corner is the start position to create

// a square, and this start position pick up range is

// also a rectangle inside [0,0] to [rows - maxLen, cols - maxLen]

// if the maximum square length define as 'maxLen'

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

for(int j = 0; j <= cols - maxLen; j++) {

// Need to check if any cell as [p,q] = '0' happen

// in the candidate maximal square

int p;

for(p = i; p < i + maxLen; p++) {

int q;

for(q = j; q < j + maxLen; q++) {

if(matrix[p][q] == '0') {

break;

}

}

if(q < j + maxLen) {

break;

}

}

if(p == i + maxLen) {

return maxLen \* maxLen;

}

}

}

}

return 0;

}

}

Time Complexity : O(M\*N\*min(M,N)^3)

Space Complexity : O(1), only constant extra space is being used

**Refer to**

<https://leetcode.com/problems/maximal-square/solutions/61805/evolve-from-brute-force-to-dp/>

Brute force O(n^5), check all sizes starting at all points

public int maximalSquare(char[][] matrix) {

int r=matrix.length;

if(r==0) return 0;

int c=matrix[0].length, n=Math.min(r,c);

for(int s=n;s>0;s--)

for(int i=0;i<=r-s;i++)

for(int j=0;j<=c-s;j++) {

int p;

for(p=i;p<i+s;p++) {

int q;

for(q=j;q<j+s;q++)

if(matrix[p][q]=='0') break;

if(q<j+s) break;

}

if(p==i+s) return s\*s;

}

return 0;

}

**Refer to**

<https://leetcode.com/problems/maximal-square/solutions/1632376/c-python-6-simple-solution-w-explanation-optimizations-from-brute-force-to-dp/>

✔️ ***Solution - I (Brute-Force)***

To start with brute-force approach, we can simply consider each possible starting cell (row, col) and side length (sideLen) of square starting at that cell. For each cell and sideLen, we will check if the corresponding square inside the matrix is valid or not (i.e, all cells are "1" or not). After checking each possible squares, we will return the one with maximum area. We can slightly optimize the code by running from sideLen = min(m, n) down to 1 instead of the other way around. This ensures that we can return the area of sqaure as soon as we find the 1st valid square since that square would be the 1st valid square of maximum side length.

class Solution {

public:

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

auto isValidSquare = [&](int i, int j, int side) {

return all\_of(begin(M)+i, begin(M)+i+side, [&](auto& R){

return all\_of(begin(R)+j, begin(R)+j+side, [&](auto cell) { return cell == '1'; });

});

};

int m = size(M), n = size(M[0]);

for(int sideLen = min(m, n); sideLen; sideLen--)

for(int row = 0; row <= m-sideLen; row++)

for(int col = 0; col <= n-sideLen; col++)

if(isValidSquare(row, col, sideLen))

return sideLen\*sideLen;

return 0;

}

};

***Time Complexity :*** O(M\*N\*min(M,N)^3)

***Space Complexity :*** O(1), only constant extra space is being used

**Solution 2:  Brute Force (360 min, check if a square contains all 1s can be improved to constant by preprocessing)**

class Solution {

// Check if a square contains all 1s can be improved to constant by preprocessing.

// ones[i][j] is the number of 1s in matrix(0, 0, i - 1, j - 1)

public int maximalSquare(char[][] matrix) {

int rows = matrix.length;

int cols = matrix[0].length;

int n = Math.min(rows, cols);

int[][] ones = new int[rows + 1][cols + 1];

for(int i = 1; i <= rows; i++) {

for(int j = 1; j <= cols; j++) {

ones[i][j] = matrix[i - 1][j - 1] - '0' + ones[i][j - 1] + ones[i - 1][j] - ones[i - 1][j - 1];

}

}

for(int maxLen = n; maxLen > 0; maxLen--) {

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

for(int j = 0; j <= cols - maxLen; j++) {

if(ones[i + maxLen][j + maxLen] - ones[i + maxLen][j] - ones[i][j + maxLen] + ones[i][j] == maxLen \* maxLen) {

return maxLen \* maxLen;

}

}

}

}

return 0;

}

}

Time Complexity : O(M\*N\*min(M,N))

Space Complexity : O(M \* N)

**Refer to**

<https://leetcode.com/problems/maximal-square/solutions/61805/evolve-from-brute-force-to-dp/>

O(n^3), check if a square contains all 1s can be improved to constant by preprocessing. ones[i][j] is the number of 1s in matrix(0,0,i-1,j-1)

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

int r=matrix.size();

if(!r) return 0;

int c=matrix[0].size(), n=min(r,c);

vector<vector<int>> ones(r+1,vector<int>(c+1));

for(int i=1;i<=r;i++)

for(int j=1;j<=c;j++) ones[i][j] = matrix[i-1][j-1]-'0' + ones[i-1][j]+ones[i][j-1]-ones[i-1][j-1];

for(int s=n;s>0;s--)

for(int i=0;i<=r-s;i++)

for(int j=0;j<=c-s;j++) if(ones[i+s][j+s]-ones[i+s][j]-ones[i][j+s]+ones[i][j] == s\*s) return s\*s;

return 0;

}

***Time Complexity :*** O(M\*N\*min(M,N))

***Space Complexity :*** O(M\*N)

**Solution 3: Native DFS (10 min, TLE 62/78)**

Increase i, j style (we can assume every start point {i, j} as bottom right corner of its calculating range rectangle)

class Solution {

public int maximalSquare(char[][] matrix) {

int maxLen = 0;

int m = matrix.length;

int n = matrix[0].length;

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

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

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

maxLen = Math.max(maxLen, helper(i, j, matrix));

}

}

}

return maxLen \* maxLen;

}

private int helper(int i, int j, char[][] matrix) {

// No need check on i < 0 or j < 0 case, because we only increase i, j in recursion call

//if(i < 0 || i >= matrix.length || j < 0 || j >= matrix[0].length || matrix[i][j] == '0') {

if(i == matrix.length || j == matrix[0].length || matrix[i][j] == '0') {

return 0;

}

return Math.min(helper(i + 1, j + 1, matrix), Math.min(helper(i + 1, j, matrix), helper(i, j + 1, matrix))) + 1;

}

}

==============================================================================================

Decrease i, j style (we can assume every start point {i, j} as top left corner of its calculating range rectangle)

class Solution {

public int maximalSquare(char[][] matrix) {

int maxLen = 0;

int m = matrix.length;

int n = matrix[0].length;

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

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

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

maxLen = Math.max(maxLen, helper(i, j, matrix));

}

}

}

return maxLen \* maxLen;

}

private int helper(int i, int j, char[][] matrix) {

// No need check on i == matrix.length or matrix[0].length case, because we only decrease i, j in recursion call

//if(i < 0 || i >= matrix.length || j < 0 || j >= matrix[0].length || matrix[i][j] == '0') {

if(i < 0 || j < 0 || matrix[i][j] == '0') {

return 0;

}

return Math.min(helper(i - 1, j - 1, matrix), Math.min(helper(i - 1, j, matrix), helper(i, j - 1, matrix))) + 1;

}

}

**Refer to**

<https://leetcode.com/problems/maximal-square/solutions/955685/Java-Recursive-(TLE)-greater-Memoization-greater-2D-Bottom-Up-greater-1D-Bottom-Up/>

- For each of the cell 'r,c' with the value of 1

- We can treat this cell as the top left corner of a rectangle

- We will first need to recursively check the length of the maximal square located to the 'right, bottom, bottom right'

- Then we can generate a new rectangle with length 'min(right, bottom, bottom right) + 1'

- We will find the length of the largest square

- Then square the length to find the area

public class MaximalSquareRecursiveApproach {

public int maximalSquare(char[][] matrix) {

int maxLength = 0;

for (int r = 0; r < matrix.length; r++) {

for (int c = 0; c < matrix[r].length; c++) {

maxLength = Math.max(maxLength, getMaxLength(r, c, matrix));

}

}

return maxLength \* maxLength;

}

private int getMaxLength(int r, int c, char[][] matrix) {

if (r < 0 || r >= matrix.length || c < 0 || c >= matrix[r].length || matrix[r][c] == '0') return 0;

return Math.min(

getMaxLength(r + 1, c + 1, matrix),

Math.min(getMaxLength(r, c + 1, matrix), getMaxLength(r + 1, c, matrix))

) + 1;

}

}

**Solution 4: DFS + Memoization (10 min)**

Increase i, j style (we can assume every start point {i, j} as bottom right corner of its calculating range rectangle)

class Solution {

public int maximalSquare(char[][] matrix) {

int maxLen = 0;

int m = matrix.length;

int n = matrix[0].length;

Integer[][] memo = new Integer[m][n];

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

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

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

maxLen = Math.max(maxLen, helper(i, j, matrix, memo));

}

}

}

return maxLen \* maxLen;

}

private int helper(int i, int j, char[][] matrix, Integer[][] memo) {

// No need check on i < 0 or j < 0 case, because we only increase i, j in recursion call

//if(i < 0 || i >= matrix.length || j < 0 || j >= matrix[0].length || matrix[i][j] == '0') {

if(i == matrix.length || j == matrix[0].length || matrix[i][j] == '0') {

return 0;

}

if(memo[i][j] != null) {

return memo[i][j];

}

return memo[i][j] = Math.min(helper(i + 1, j + 1, matrix, memo), Math.min(helper(i + 1, j, matrix, memo), helper(i, j + 1, matrix, memo))) + 1;

}

}

==============================================================================================

Decrease i, j style (we can assume every start point {i, j} as top left corner of its calculating range rectangle)

class Solution {

public int maximalSquare(char[][] matrix) {

int maxLen = 0;

int m = matrix.length;

int n = matrix[0].length;

Integer[][] memo = new Integer[m][n];

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

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

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

maxLen = Math.max(maxLen, helper(i, j, matrix, memo));

}

}

}

return maxLen \* maxLen;

}

private int helper(int i, int j, char[][] matrix, Integer[][] memo) {

// No need check on i == matrix.length or j == matrix[0].length case, because we only increase i, j in recursion call

//if(i < 0 || i >= matrix.length || j < 0 || j >= matrix[0].length || matrix[i][j] == '0') {

if(i < 0 || j < 0 || matrix[i][j] == '0') {

return 0;

}

if(memo[i][j] != null) {

return memo[i][j];

}

return memo[i][j] = Math.min(helper(i - 1, j - 1, matrix, memo), Math.min(helper(i - 1, j, matrix, memo), helper(i, j - 1, matrix, memo))) + 1;

}

}

Time Complexity : O(M\*N)

Space Complexity : O(M\*N)

**Refer to**

<https://leetcode.com/problems/maximal-square/solutions/955685/Java-Recursive-(TLE)-greater-Memoization-greater-2D-Bottom-Up-greater-1D-Bottom-Up/>

public class MaximalSquareMemoizationApproach {

public int maximalSquare(char[][] matrix) {

if (matrix.length == 0) return 0;

int m = matrix.length, n = matrix[0].length, maxLength = 0;

int[][] memo = new int[m][n];

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

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

maxLength = Math.max(maxLength, getMaxLength(r, c, matrix, memo));

}

}

return maxLength \* maxLength;

}

private int getMaxLength(int r, int c, char[][] matrix, int[][] memo) {

if (r < 0 || r >= matrix.length || c < 0 || c >= matrix[r].length || matrix[r][c] == '0') return 0;

if (memo[r][c] != 0) return memo[r][c];

return memo[r][c] = Math.min(

getMaxLength(r + 1, c + 1, matrix, memo),

Math.min(getMaxLength(r, c + 1, matrix, memo), getMaxLength(r + 1, c, matrix, memo))

) + 1;

}

}

***Time Complexity :*** O(M\*N)

***Space Complexity :*** O(M\*N)

**Solution 5: 2D DP (10 min)**

**Style 1: The 2D DP array initialize as one more column and one more row**

Decrease i, j style (we can assume every start point {i, j} as bottom right corner of its calculating range rectangle)

class Solution {

public int maximalSquare(char[][] matrix) {

int maxLen = 0;

int m = matrix.length;

int n = matrix[0].length;

// Why we have to add one more column and one more row ?

// Because the recurrence formula will show have two potential way:

// (1) Derive {i, j} from top right corner

// dp[i][j] = Math.max(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1])

// The {i, j} closely relate to its previous row or column

// (2) Derive {i, j} from bottom left corner

// dp[i][j] = Math.max(Math.min(dp[i + 1][j], dp[i][j + 1]), dp[i + 1][j + 1])

// The {i, j} closely relate to its next row or column

// For calculating the first or last column and row on original matrix

// convenient, we have to add one more column and one more row onto the

// original matrix, otherwise we have to initialize the first or last

// column and row with special handling rather then other columns and rows

int[][] dp = new int[m + 1][n + 1];

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

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

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

dp[i][j] = 1 + Math.min(Math.min(dp[i + 1][j], dp[i][j + 1]), dp[i + 1][j + 1]);

}

// Condition when matrix[i][j] == '0' should not ignore, but since when

// initialize 2D DP array, any dp[i][j] cell is 0, no need to reset to 0

// but the reset dp[i][j] back to 0 when matrix[i][j] == '0' is a required

// statement and cannot comment out when 2D DP downgrade to 1D DP, because

// in 1D DP array, after removing row dimension, the 1D dp[j] and dpPrev[j]

// array will iteratively exchange values, the value of dp[j] is dynamically

// inheried from previous dpPrev[j] and vise verse, no default dp[i][j] = 0

// any more, if no reset condition, will bring wrong dp status to next iteration

//else {

// dp[i][j] = 0;

//}

maxLen = Math.max(maxLen, dp[i][j]);

}

}

return maxLen \* maxLen;

}

}

==============================================================================================

Decrease i, j style (we can assume every start point {i, j} as top left corner of its calculating range rectangle)

class Solution {

public int maximalSquare(char[][] matrix) {

int maxLen = 0;

int m = matrix.length;

int n = matrix[0].length;

// Why we have to add one more column and one more row ?

// Because the recurrence formula will show have two potential way:

// (1) Derive {i, j} from top right corner

// dp[i][j] = Math.max(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1])

// The {i, j} closely relate to its previous row or column

// (2) Derive {i, j} from bottom left corner

// dp[i][j] = Math.max(Math.min(dp[i + 1][j], dp[i][j + 1]), dp[i + 1][j + 1])

// The {i, j} closely relate to its next row or column

// For calculating the first or last column and row on original matrix

// convenient, we have to add one more column and one more row onto the

// original matrix, otherwise we have to initialize the first or last

// column and row with special handling rather then other columns and rows

int[][] dp = new int[m + 1][n + 1];

for(int i = 1; i <= m; i++) {

for(int j = 1; j <= n; j++) {

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

dp[i][j] = 1 + Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]);

}

// Condition when matrix[i][j] == '0' should not ignore, but since when

// initialize 2D DP array, any dp[i][j] cell is 0, no need to reset to 0

// but the reset dp[i][j] back to 0 when matrix[i][j] == '0' is a required

// statement and cannot comment out when 2D DP downgrade to 1D DP, because

// in 1D DP array, after removing row dimension, the 1D dp[j] and dpPrev[j]

// array will iteratively exchange values, the value of dp[j] is dynamically

// inheried from previous dpPrev[j] and vise verse, no default dp[i][j] = 0

// any more, if no reset condition, will bring wrong dp status to next iteration

//else {

// dp[i][j] = 0;

//}

maxLen = Math.max(maxLen, dp[i][j]);

}

}

return maxLen \* maxLen;

}

}

Time Complexity : O(M\*N)

Space Complexity : O(M\*N)

**Style 2: The 2D DP array initialize as same size as original matrix**

Decrease i, j style (we can assume every start point {i, j} as bottom right corner of its calculating range rectangle)

class Solution {

public int maximalSquare(char[][] matrix) {

int maxLen = 0;

int m = matrix.length;

int n = matrix[0].length;

// Why we have to add one more column and one more row ?

// Because the recurrence formula will show have two potential way:

// (1) Derive {i, j} from top right corner

// dp[i][j] = Math.max(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1])

// The {i, j} closely relate to its previous row or column

// (2) Derive {i, j} from bottom left corner

// dp[i][j] = Math.max(Math.min(dp[i + 1][j], dp[i][j + 1]), dp[i + 1][j + 1])

// The {i, j} closely relate to its next row or column

// For calculating the first or last column and row on original matrix

// convenient, we have to add one more column and one more row onto the

// original matrix, otherwise we have to initialize the first or last

// column and row with special handling rather then other columns and rows

//int[][] dp = new int[m + 1][n + 1];

int[][] dp = new int[m][n];

// Initialize last row (for last row, no dp[i + 1][j] or dp[i + 1][j + 1],

// only dp[i][j + 1] when j < matrix[0].length - 1, but since we try to

// find square on last row, the 'maxLen' on last row will be only 1 if the

// cell {m - 1, j} is '1')

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

dp[m - 1][j] = matrix[m - 1][j] == '1' ? 1 : 0;

maxLen = Math.max(maxLen, dp[m - 1][j]);

}

// Initialize last column (for last column, no dp[i][j + 1] or dp[i + 1][j + 1],

// only dp[i + 1][j] when i < matrix.length - 1, but since we try to find

// square on last column, the 'maxLen' on last column will be only 1 if the

// cell {i, n - 1} is '1')

for(int i = 0; i < m - 1; i++) {

dp[i][n - 1] = matrix[i][n - 1] == '1' ? 1 : 0;

maxLen = Math.max(maxLen, dp[i][n - 1]);

}

for(int i = m - 2; i >= 0; i--) {

for(int j = n - 2; j >= 0; j--) {

// If 2D DP array change from dp[i][j], then condition change from

// matrix[i][j] == '1' to matrix[i - 1][j - 1] == '1'

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

dp[i][j] = 1 + Math.min(Math.min(dp[i + 1][j], dp[i][j + 1]), dp[i + 1][j + 1]);

}

// Condition when matrix[i][j] == '0' should not ignore, but since when

// initialize 2D DP array, any dp[i][j] cell is 0, no need to reset to 0

// but the reset dp[i][j] back to 0 when matrix[i][j] == '0' is a required

// statement and cannot comment out when 2D DP downgrade to 1D DP, because

// in 1D DP array, after removing row dimension, the 1D dp[j] and dpPrev[j]

// array will iteratively exchange values, the value of dp[j] is dynamically

// inheried from previous dpPrev[j] and vise verse, no default dp[i][j] = 0

// any more, if no reset condition, will bring wrong dp status to next iteration

//else {

// dp[i][j] = 0;

//}

maxLen = Math.max(maxLen, dp[i][j]);

}

}

return maxLen \* maxLen;

}

}

==============================================================================================

Decrease i, j style (we can assume every start point {i, j} as top left corner of its calculating range rectangle)

class Solution {

public int maximalSquare(char[][] matrix) {

int maxLen = 0;

int m = matrix.length;

int n = matrix[0].length;

// Why we have to add one more column and one more row ?

// Because the recurrence formula will show have two potential way:

// (1) Derive {i, j} from top right corner

// dp[i][j] = Math.max(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1])

// The {i, j} closely relate to its previous row or column

// (2) Derive {i, j} from bottom left corner

// dp[i][j] = Math.max(Math.min(dp[i + 1][j], dp[i][j + 1]), dp[i + 1][j + 1])

// The {i, j} closely relate to its next row or column

// For calculating the first or last column and row on original matrix

// convenient, we have to add one more column and one more row onto the

// original matrix, otherwise we have to initialize the first or last

// column and row with special handling rather then other columns and rows

//int[][] dp = new int[m + 1][n + 1];

int[][] dp = new int[m][n];

// Initialize first row (for first row, no dp[i - 1][j] or dp[i - 1][j - 1],

// only dp[i][j - 1] when j > 0, but since we try to find square on first row,

// the 'maxLen' on first row will be only 1 if the cell {0, j} is '1')

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

dp[0][j] = matrix[0][j] == '1' ? 1 : 0;

maxLen = Math.max(maxLen, dp[0][j]);

}

// Initialize first column (for first column, no dp[i][j - 1] or dp[i - 1][j - 1],

// only dp[i - 1][j] when i > 0, but since we try to find square on first column,

// the 'maxLen' on first column will be only 1 if the cell {i, 0} is '1')

for(int i = 1; i < m; i++) {

dp[i][0] = matrix[i][0] == '1' ? 1 : 0;

maxLen = Math.max(maxLen, dp[i][0]);

}

for(int i = 1; i < m; i++) {

for(int j = 1; j < n; j++) {

// If 2D DP array change from dp[m + 1][n + 1], then condition change from

// matrix[i - 1][j - 1] == '1' to matrix[i][j] == '1'

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

dp[i][j] = 1 + Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]);

}

// Condition when matrix[i][j] == '0' should not ignore, but since when

// initialize 2D DP array, any dp[i][j] cell is 0, no need to reset to 0

// but the reset dp[i][j] back to 0 when matrix[i][j] == '0' is a required

// statement and cannot comment out when 2D DP downgrade to 1D DP, because

// in 1D DP array, after removing row dimension, the 1D dp[j] and dpPrev[j]

// array will iteratively exchange values, the value of dp[j] is dynamically

// inheried from previous dpPrev[j] and vise verse, no default dp[i][j] = 0

// any more, if no reset condition, will bring wrong dp status to next iteration

//else {

// dp[i][j] = 0;

//}

maxLen = Math.max(maxLen, dp[i][j]);

}

}

return maxLen \* maxLen;

}

}

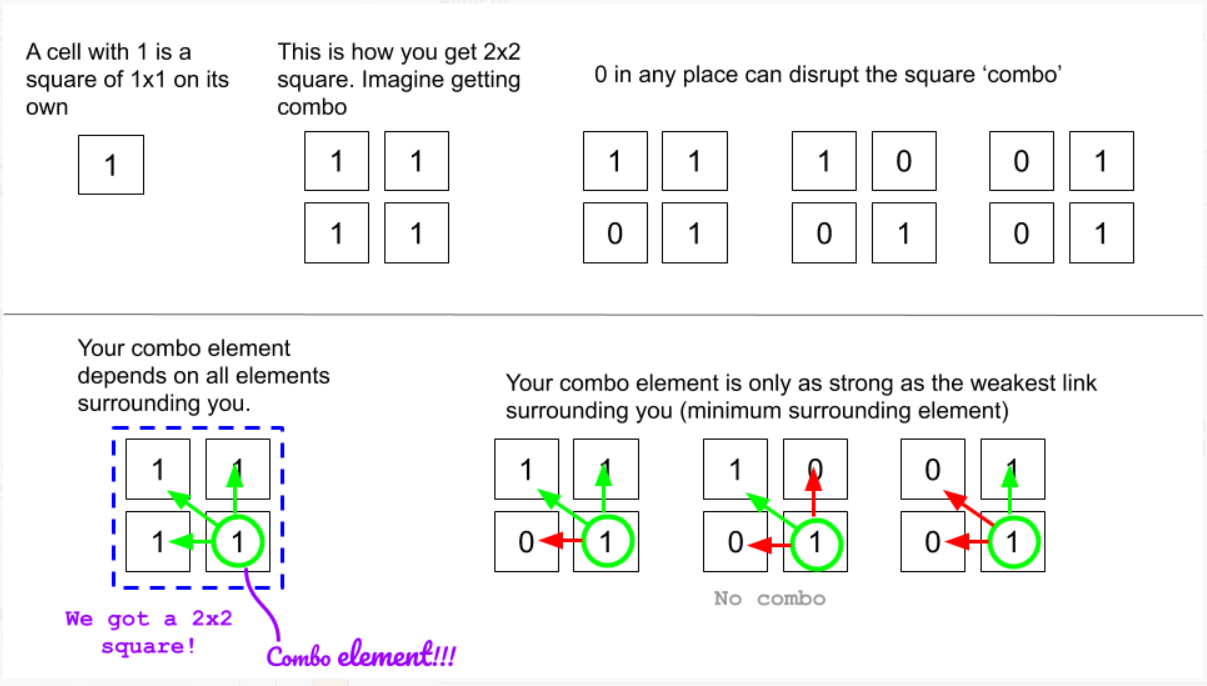
Time Complexity : O(M\*N)

Space Complexity : O(M\*N)

**Refer to**

<https://leetcode.com/problems/maximal-square/solutions/600149/python-thinking-process-diagrams-dp-approach/>

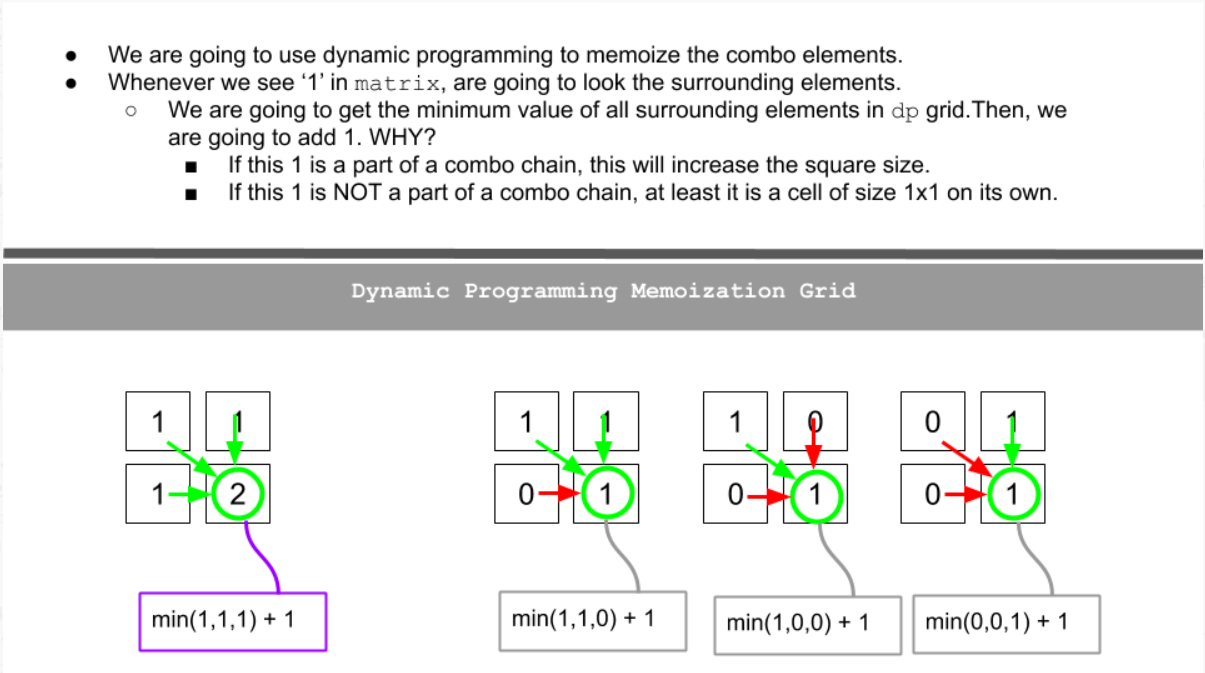
**Understanding basics**



* Here I want to mention that we are drawing squares from top left corner to bottom right corner. Therefore, when I mention, "surrounding elements", I am saying cells above the corner cell and the cells on the left of the corner cell.

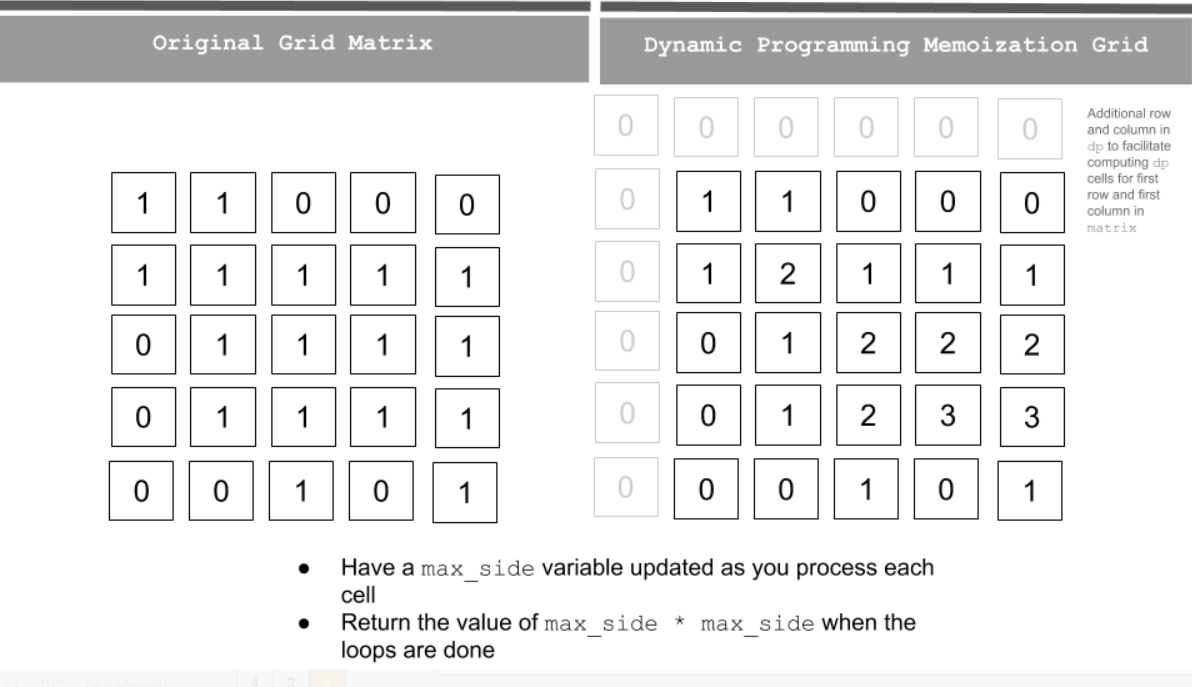
**Building DP grid to memoize**

* We are going to create a dp grid with initial values of 0.
* We are going to update dp as described in the following figure.



**Bigger Example**

* Let's try to see a bigger example.
* We go over one cell at a time row by row in the matrix and then update our dp grid accordingly.
* Update max\_side with the maximum dp cell value as you update.



In the code, I create a dp grid which has one additional column and one additional row. The reason is to facilitate the index dp[r-1][c] dp[r][c-1] and dp[r-1][c-1] for cells in first row and first column in matrix.

class Solution:

def maximalSquare(self, matrix: List[List[str]]) -> int:

if matrix is None or len(matrix) < 1:

return 0

rows = len(matrix)

cols = len(matrix[0])

dp = [[0]\*(cols+1) for \_ in range(rows+1)]

max\_side = 0

for r in range(rows):

for c in range(cols):

if matrix[r][c] == '1':

dp[r+1][c+1] = min(dp[r][c], dp[r+1][c], dp[r][c+1]) + 1 # Be careful of the indexing since dp grid has additional row and column

max\_side = max(max\_side, dp[r+1][c+1])

return max\_side \* max\_side

**Complexity Analysis**

Time complexity : O(mn). Single pass - row x col (m=row; n=col)Space complexity : O(mn). Additional space for dp grid (don't need to worry about additional 1 row and col).

**Follow up**

Space can be optimized as we don't need to keep the whole dp grid as we progress down the rows in matrix.

**Refer to**

<https://leetcode.com/problems/maximal-square/solutions/61805/evolve-from-brute-force-to-dp/>

public int maximalSquare(char[][] matrix) {

int r=matrix.length;

if(r==0) return 0;

int c=matrix[0].length,edge=0;

int[][] dp=new int[r+1][c+1];

for(int i=1;i<=r;i++)

for(int j=1;j<=c;j++) {

if(matrix[i-1][j-1]=='0') continue;

dp[i][j]=1+Math.min(dp[i-1][j],Math.min(dp[i-1][j-1],dp[i][j-1]));

edge=Math.max(edge,dp[i][j]);

}

return edge\*edge;

}

**Refer to**

<https://leetcode.com/problems/maximal-square/solutions/955685/Java-Recursive-(TLE)-greater-Memoization-greater-2D-Bottom-Up-greater-1D-Bottom-Up/>

public class MaximalSquareBottomUp2DApproach {

public int maximalSquare(char[][] matrix) {

if (matrix.length == 0) return 0;

int m = matrix.length, n = matrix[0].length, maxLength = 0;

int[][] length = new int[m + 1][n + 1];

for (int r = m - 1; r >= 0; r--) {

for (int c = n - 1; c >= 0; c--) {

if (matrix[r][c] == '0') continue;

length[r][c] = Math.min(

length[r + 1][c + 1], Math.min(length[r + 1][c], length[r][c + 1])

) + 1;

maxLength = Math.max(maxLength, length[r][c]);

}

}

return maxLength \* maxLength;

}

}

***Time Complexity :*** O(M\*N)

***Space Complexity :*** O(M\*N)

**Solution 6: 2 rows DP (10 min)**

Decrease i, j style (we can assume every start point {i, j} as bottom right corner of its calculating range rectangle)

class Solution {

public int maximalSquare(char[][] matrix) {

int maxLen = 0;

int m = matrix.length;

int n = matrix[0].length;

//int[][] dp = new int[m + 1][n + 1];

int[] dpPrev = new int[n + 1];

// The 'dpPrev' will be initialized as last row, which is

// the extra added row at bottom based on original matrix,

// just fill entire row as 0 is fine

//for(int i = 0; i <= n; i++) {

// dpPrev[i] = 0;

//}

int[] dp = new int[n + 1];

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

// Since the recurrence formula depends on next column,

// the 'dp' need to initialize based on the extra added

// last column cell(nth column cell), just fill it as 0

//dp[n] = 0;

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

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

//dp[i][j] = 1 + Math.min(Math.min(dp[i + 1][j], dp[i][j + 1]), dp[i + 1][j + 1]);

dp[j] = 1 + Math.min(Math.min(dpPrev[j], dp[j + 1]), dpPrev[j + 1]);

} else {

// Condition when matrix[i][j] == '0' should not ignore, but since when

// initialize 2D DP array, any dp[i][j] cell is 0, no need to reset to 0

// but the reset dp[i][j] back to 0 when matrix[i][j] == '0' is a required

// statement and cannot comment out when 2D DP downgrade to 1D DP, because

// in 1D DP array, after removing row dimension, the 1D dp[j] and dpPrev[j]

// array will iteratively exchange values, the value of dp[j] is dynamically

// inheried from previous dpPrev[j] and vise verse, no default dp[i][j] = 0

// any more, if no reset condition, will bring wrong dp status to next iteration

//dp[i][j] = 0;

dp[j] = 0;

}

//maxLen = Math.max(maxLen, dp[i][j]);

maxLen = Math.max(maxLen, dp[j]);

}

dpPrev = dp.clone();

}

return maxLen \* maxLen;

}

}

==============================================================================================

Decrease i, j style (we can assume every start point {i, j} as top left corner of its calculating range rectangle)

class Solution {

public int maximalSquare(char[][] matrix) {

int maxLen = 0;

int m = matrix.length;

int n = matrix[0].length;

//int[][] dp = new int[m + 1][n + 1];

int[] dpPrev = new int[n + 1];

// The 'dpPrev' will be initialized as first row, which is

// the extra added row at top based on original matrix,

// just fill entire row as 0 is fine

//for(int i = 0; i <= n; i++) {

// dpPrev[i] = 0;

//}

int[] dp = new int[n + 1];

for(int i = 1; i <= m; i++) {

// Since the recurrence formula depends on previous column,

// the 'dp' need to initialize based on the extra added

// first column cell(0th column cell), just fill it as 0

//dp[0] = 0;

for(int j = 1; j <= n; j++) {

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

//dp[i][j] = 1 + Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]);

dp[j] = 1 + Math.min(Math.min(dpPrev[j], dp[j - 1]), dpPrev[j - 1]);

} else {

// Condition when matrix[i][j] == '0' should not ignore, but since when

// initialize 2D DP array, any dp[i][j] cell is 0, no need to reset to 0

// but the reset dp[i][j] back to 0 when matrix[i][j] == '0' is a required

// statement and cannot comment out when 2D DP downgrade to 1D DP, because

// in 1D DP array, after removing row dimension, the 1D dp[j] and dpPrev[j]

// array will iteratively exchange values, the value of dp[j] is dynamically

// inheried from previous dpPrev[j] and vise verse, no default dp[i][j] = 0

// any more, if no reset condition, will bring wrong dp status to next iteration

//dp[i][j] = 0;

dp[j] = 0;

}

//maxLen = Math.max(maxLen, dp[i][j]);

maxLen = Math.max(maxLen, dp[j]);

}

dpPrev = dp.clone();

}

return maxLen \* maxLen;

}

}

Time Complexity : O(2\*N)

Space Complexity : O(M\*N)

**Refer to**

<https://leetcode.com/problems/maximal-square/solutions/61803/C++-space-optimized-DP/>

In the above code, it uses O(mn) space. Actually each time when we update dp[i][j], we only need dp[i-1][j-1], dp[i-1][j] (the previous row) and dp[i][j-1] (the current row). So we may just keep two rows.

class Solution {

public:

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

if (matrix.empty()) {

return 0;

}

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

vector<int> pre(n, 0), cur(n, 0);

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

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

if (!i || !j || matrix[i][j] == '0') {

cur[j] = matrix[i][j] - '0';

} else {

cur[j] = min(pre[j - 1], min(pre[j], cur[j - 1])) + 1;

}

sz = max(cur[j], sz);

}

fill(pre.begin(), pre.end(), 0);

swap(pre, cur);

}

return sz \* sz;

}

};

**Refer to**

<https://leetcode.com/problems/maximal-square/solutions/1632376/c-python-6-simple-solution-w-explanation-optimizations-from-brute-force-to-dp/>

✔️ ***Solution - V (Space-Optimized Dynamic Programming)***

We can see that we are only ever accessing the current row and next row of dp. Thus we dont need to store every row of it and can do away with only storing two rows.

A common and easy way to convert 2D dp to linear space usage is by defining 2 rows in dp and alternating between those rows for each computation. This basically ensures we are using previous computed row to compute the current row and we dont even need to change the previous solution by much. We can simply alternate between rows using the mod 2(%2) or AND 1 (& 1) operations.

Thus, we can optimize on space as below -

**C++**

class Solution {

public:

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

int m = size(M), n = size(M[0]), ans = 0;

vector<vector<int>> dp(2, vector<int>(n+1));

for(int i = m-1; ~i; i--)

for(int j = n-1; ~j; j--)

dp[i&1][j] = (M[i][j] == '1' ? 1 + min({dp[(i+1)&1][j], dp[i&1][j+1], dp[(i+1)&1][j+1]}) : 0),

ans = max(ans, dp[i&1][j]);

return ans \* ans;

}

};

***Time Complexity :*** O(2\*N)

***Space Complexity :*** O(M\*N)

**Solution 7: 1 row DP (10 min)**

Decrease i, j style (we can assume every start point {i, j} as bottom right corner of its calculating range rectangle)

class Solution {

public int maximalSquare(char[][] matrix) {

int maxLen = 0;

int m = matrix.length;

int n = matrix[0].length;

// Only keep one array as 'dpPrev'

int[] dpPrev = new int[n + 1];

// The 'dpPrev' will be initialized as last row, which is

// the extra added row at bottom based on original matrix,

// just fill entire row as 0 is fine

//for(int i = 0; i <= n; i++) {

// dpPrev[i] = 0;

//}

// Instead of 'dp' array, we create a variable 'prev' to record dpPrev[j + 1]

int prev = 0;

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

// Since the recurrence formula depends on next column,

// the 'dp' need to initialize based on the extra added

// last column cell(nth column cell), just fill it as 0

//dp[n] = 0;

// Actually below two lines not required because dpPrev[n] always default as 0,

// but to strictly follow pattern also explained in L72.Edit Distance, still

// keep the same

prev = dpPrev[n];

dpPrev[n] = 0;

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

// Reserve 'dpPrev[j]' because it gonna be update to new value since

// single array deployed only

int temp = dpPrev[j];

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

//dp[j] = 1 + Math.min(Math.min(dpPrev[j], dp[j + 1]), dpPrev[j + 1]);

// -> 'prev' replace dpPrev[j + 1]

dpPrev[j] = 1 + Math.min(Math.min(dpPrev[j], dpPrev[j + 1]), prev);

} else {

// Condition when matrix[i][j] == '0' should not ignore, but since when

// initialize 2D DP array, any dp[i][j] cell is 0, no need to reset to 0

// but the reset dp[i][j] back to 0 when matrix[i][j] == '0' is a required

// statement and cannot comment out when 2D DP downgrade to 1D DP, because

// in 1D DP array, after removing row dimension, the 1D dp[j] and dpPrev[j]

// array will iteratively exchange values, the value of dp[j] is dynamically

// inheried from previous dpPrev[j] and vise verse, no default dp[i][j] = 0

// any more, if no reset condition, will bring wrong dp status to next iteration

//dp[j] = 0;

dpPrev[j] = 0;

}

//maxLen = Math.max(maxLen, dp[j]);

maxLen = Math.max(maxLen, dpPrev[j]);

prev = temp;

}

//dpPrev = dp.clone();

}

return maxLen \* maxLen;

}

}

==============================================================================================

Decrease i, j style (we can assume every start point {i, j} as top left corner of its calculating range rectangle)

class Solution {

public int maximalSquare(char[][] matrix) {

int maxLen = 0;

int m = matrix.length;

int n = matrix[0].length;

//int[][] dp = new int[m + 1][n + 1];

int[] dpPrev = new int[n + 1];

// The 'dpPrev' will be initialized as first row, which is

// the extra added row at top based on original matrix,

// just fill entire row as 0 is fine

//for(int i = 0; i <= n; i++) {

// dpPrev[i] = 0;

//}

// Instead of 'dp' array, we create a variable 'prev' to record dpPrev[j - 1]

int prev = 0;

for(int i = 1; i <= m; i++) {

// Since the recurrence formula depends on previous column,

// the 'dp' need to initialize based on the extra added

// first column cell(0th column cell), just fill it as 0

//dp[0] = 0;

// Actually below two lines not required because dpPrev[0] always default as 0,

// but to strictly follow pattern also explained in L72.Edit Distance, still

// keep the same

prev = dpPrev[0];

dpPrev[0] = 0;

for(int j = 1; j <= n; j++) {

// Reserve 'dpPrev[j]' because it gonna be update to new value since

// single array deployed only

int temp = dpPrev[j];

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

//dp[j] = 1 + Math.min(Math.min(dpPrev[j], dp[j - 1]), dpPrev[j - 1]);

// -> 'prev' replace dpPrev[j - 1]

dpPrev[j] = 1 + Math.min(Math.min(dpPrev[j], dpPrev[j - 1]), prev);

} else {

// Condition when matrix[i][j] == '0' should not ignore, but since when

// initialize 2D DP array, any dp[i][j] cell is 0, no need to reset to 0

// but the reset dp[i][j] back to 0 when matrix[i][j] == '0' is a required

// statement and cannot comment out when 2D DP downgrade to 1D DP, because

// in 1D DP array, after removing row dimension, the 1D dp[j] and dpPrev[j]

// array will iteratively exchange values, the value of dp[j] is dynamically

// inheried from previous dpPrev[j] and vise verse, no default dp[i][j] = 0

// any more, if no reset condition, will bring wrong dp status to next iteration

//dp[j] = 0;

dpPrev[j] = 0;

}

//maxLen = Math.max(maxLen, dp[j]);

maxLen = Math.max(maxLen, dpPrev[j]);

prev = temp;

}

//dpPrev = dp.clone();

}

return maxLen \* maxLen;

}

}

Time Complexity : O(N)

Space Complexity : O(M\*N)

**Refer to**

<https://leetcode.com/problems/maximal-square/solutions/61803/C++-space-optimized-DP/>

Furthermore, we may only use just one vector (thanks to @stellari for sharing the idea).

class Solution {

public:

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

if (matrix.empty()) {

return 0;

}

int m = matrix.size(), n = matrix[0].size(), sz = 0, pre;

vector<int> cur(n, 0);

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

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

int temp = cur[j];

if (!i || !j || matrix[i][j] == '0') {

cur[j] = matrix[i][j] - '0';

} else {

cur[j] = min(pre, min(cur[j], cur[j - 1])) + 1;

}

sz = max(cur[j], sz);

pre = temp;

}

}

return sz \* sz;

}

};

**Refer to**

<https://leetcode.com/problems/maximal-square/solutions/61805/evolve-from-brute-force-to-dp/>

public class MaximalSquareBottomUp1DApproach {

public int maximalSquare(char[][] matrix) {

if (matrix.length == 0) return 0;

int m = matrix.length, n = matrix[0].length, maxLength = 0;

int[] length = new int[n + 1];

for (int r = m - 1; r >= 0; r--) {

int prev = 0;

for (int c = n - 1; c >= 0; c--) {

if (matrix[r][c] == '0') {

prev = length[c];

length[c] = 0;

continue;

}

int cur = length[c];

length[c] = Math.min(prev, Math.min(length[c], length[c + 1])) + 1;

prev = cur;

maxLength = Math.max(maxLength, length[c]);

}

}

return maxLength \* maxLength;

}

}

***Time Complexity :*** O(N)

***Space Complexity :*** O(M\*N)