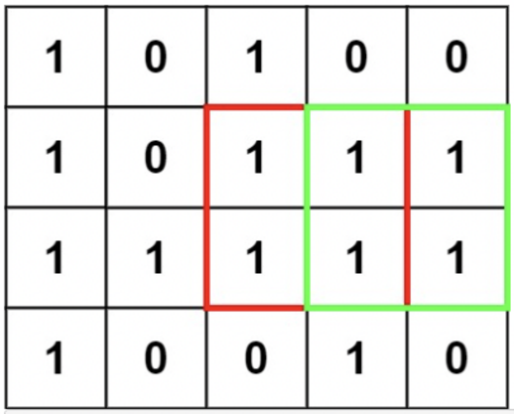
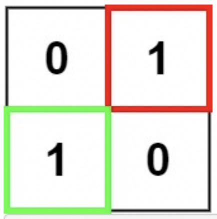
Given an `m x n` binary `matrix` filled with `0`'s and `1`'s, *find the largest square containing only* `1`'s *and return its area*.

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'`.

Summary

We need to find the largest square comprising of all ones in the given $m \times n$ matrix. In other words we need to find the largest set of connected ones in the given matrix that forms a square.

Solution

Approach #1 Brute Force [Accepted]

The simplest approach consists of trying to find out every possible square of 1's that can be formed from within the matrix. The question now is – how to go for it?

We use a variable to contain the size of the largest square found so far and another variable to store the size of the current, both initialized to 0. Starting from the left uppermost point in the matrix, we search for a 1. No operation needs to be done for a 0. Whenever a 1 is found, we try to find out the largest square that can be formed including that 1. For this, we move diagonally (right and downwards), i.e. we increment the row index and column index temporarily and then check whether all the elements of that row and column are 1 or not. If all the elements happen to be 1, we move diagonally further as previously. If even one element turns out to be 0, we stop this diagonal movement and update the size of the largest square. Now we, continue the traversal of the matrix from the element next to the initial 1 found, till all the elements of the matrix have been traversed.

Java

Java

```java
public class Solution {
    public int maximalSquare(char[][] matrix) {
        int rows = matrix.length, cols = rows > 0 ? matrix[0].length : 0;
        int maxsqlen = 0;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (matrix[i][j] == '1') {
                    int sqlen = 1;
                    boolean flag = true;
                    while (sqlen + i < rows && sqlen + j < cols && flag) {
                        for (int k = j; k <= sqlen + j; k++) {
                            if (matrix[i + sqlen][k] == '0') {
                                flag = false;
                                break;
                            }
                        }
                        for (int k = i; k <= sqlen + i; k++) {
                            if (matrix[k][j + sqlen] == '0') {
                                flag = false;
                                break;
                            }
                        }
                        if (flag)
                            sqlen++;
                    }
                    if (maxsqlen < sqlen) {
                        maxsqlen = sqlen;
                    }
                }
            }
        }
        return maxsqlen * maxsqlen;
    }
}
```

Complexity Analysis

- Time complexity : $O((mn)^2)$. In worst case, we need to traverse the complete matrix for every 1.
- Space complexity : $O(1)$. No extra space is used.

Approach #2 (Dynamic Programming) [Accepted]

Algorithm

We will explain this approach with the help of an example.

```
0 1 1 1 0
1 1 1 1 1
0 1 1 1 1
0 1 1 1 1
0 0 1 1 1
```

We initialize another matrix (dp) with the same dimensions as the original one initialized with all 0's.

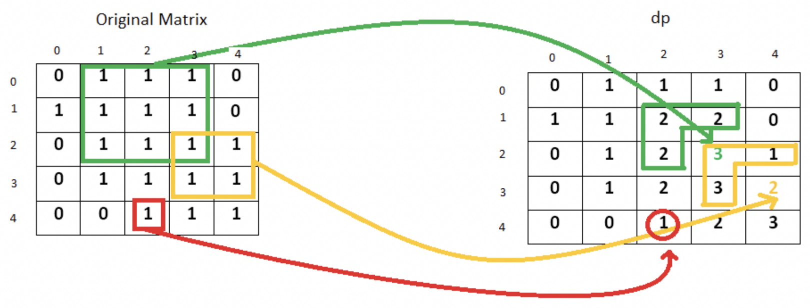
dp(i,j) represents the side length of the maximum square whose bottom right corner is the cell with index (i,j) in the original matrix.

Starting from index (0,0), for every 1 found in the original matrix, we update the value of the current element as

$\mathrm{dp}(i,j) = \min\left(\mathrm{dp}(i-1,j), \mathrm{dp}(i-1,j-1), \mathrm{dp}(i,j-1)\right) + 1.$

We also remember the size of the largest square found so far. In this way, we traverse the original matrix once and find out the required maximum size. This gives the side length of the square (say $maxsqlen$). The required result is the area $maxsqlen^2$.

To understand how this solution works, see the figure below.



An entry 2 at $(1,3)$ implies that we have a square of side 2 up to that index in the original matrix. Similarly, a 2 at $(1,2)$ and $(2,2)$ implies that a square of side 2 exists up to that index in the original matrix. Now to make a square of side 3, only a single entry of 1 is pending at $(2,3)$. So, we enter a 3 corresponding to that position in the dp array.

Now consider the case for the index $(3,4)$. Here, the entries at index $(3,3)$ and $(2,3)$ imply that a square of side 3 is possible up to their indices. But, the entry 1 at index $(2,4)$ indicates that a square of side 1 only can be formed up to its index. Therefore, while making an entry at the index $(3,4)$, this element obstructs the formation of a square having a side larger than 2. Thus, the maximum sized square that can be formed up to this index is of size 2×2.

Java

```java
public class Solution {
    public int maximalSquare(char[][] matrix) {
        int rows = matrix.length, cols = rows > 0 ? matrix[0].length : 0;
        int[][] dp = new int[rows + 1][cols + 1];
        int maxsqlen = 0;
        for (int i = 1; i <= rows; i++) {
            for (int j = 1; j <= cols; j++) {
                if (matrix[i-1][j-1] == '1'){
                    dp[i][j] = Math.min(Math.min(dp[i][j - 1], dp[i - 1][j]),
dp[i - 1][j - 1]) + 1;
                    maxsqlen = Math.max(maxsqlen, dp[i][j]);
                }
            }
        }
        return maxsqlen * maxsqlen;
    }
}
```

Complexity Analysis

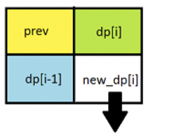
- Time complexity : $O(mn)$. Single pass.
- Space complexity : $O(mn)$. Another matrix of same size is used for dp.

Approach #3 (Better Dynamic Programming) [Accepted]

Algorithm

In the previous approach for calculating dp of i^{th} row we are using only the previous element and the $(i-1)^{th}$ row. Therefore, we don't need 2D dp matrix as 1D dp array will be sufficient for this.

Initially the dp array contains all 0's. As we scan the elements of the original matrix across a row, we keep on updating the dp array as per the equation $dp[j] = min(dp[j-1], dp[j], prev)$, where prev refers to the old $dp[j-1]$. For every row, we repeat the same process and update in the same dp array.



min(prev,dp[i-1],dp[i])

java

```java
public class Solution {
    public int maximalSquare(char[][] matrix) {
        int rows = matrix.length, cols = rows > 0 ? matrix[0].length : 0;
        int[] dp = new int[cols + 1];
        int maxsqlen = 0, prev = 0;
        for (int i = 1; i <= rows; i++) {
            for (int j = 1; j <= cols; j++) {
                int temp = dp[j];
                if (matrix[i - 1][j - 1] == '1') {
                    dp[j] = Math.min(Math.min(dp[j - 1], prev), dp[j]) + 1;
                    maxsqlen = Math.max(maxsqlen, dp[j]);
                } else {
                    dp[j] = 0;
                }
                prev = temp;
            }
        }
        return maxsqlen * maxsqlen;
    }
```

Complexity Analysis

- Time complexity : $O(mn)$. Single pass.
- Space complexity : $O(n)$. Another array which stores elements in a row is used for dp.