Maximal Square

Given a 2D binary matrix filled with 0's and 1's, find the largest square containing all 1's and return its area.

For example, given the following matrix:

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

Return 4.

Credits:

Special thanks to @Freezen for adding this problem and creating all test cases.

Solution 1

Well, this problem desires for the use of dynamic programming. They key to any DP problem is to come up with the state equation. In this problem, we define the state to be **the maximal size of the square that can be achieved at point** `(i, j)`, denoted as `P[i][j]`. Remember that we use **size** instead of square as the state (`square = size^2`).

Now let's try to come up with the formula for `P[i][j]`.

First, it is obvious that for the topmost row (`i = 0`) and the leftmost column (`j = 0`), `P[i][j] = matrix[i][j]`. This is easily understood. Let's suppose that the topmost row of `matrix` is like `[1, 0, 0, 1]`. Then we can immediately know that the first and last point can be a square of size `1` while the two middle points cannot make any square, giving a size of `0`. Thus, `P = [1, 0, 0, 1]`, which is the same as `matrix`. The case is similar for the leftmost column. Till now, the boundary conditions of this DP problem are solved.

Let's move to the more general case for `P[i][j]` in which `i > 0` and `j > 0`. First of all, let's see another simple case in which `matrix[i][j] = 0`. It is obvious that `P[i][j] = 0` too. Why? Well, since `matrix[i][j] = 0`, no square will contain `matrix[i][j]`. According to our definition of `P[i][j]`, `P[i][j]` is also `0`.

Now we are almost done. The only unsolved case is `matrix[i][j] = 1`. Let's see an example.

Suppose `matrix = [[0, 1], [1, 1]]`, it is obvious that `P[0][0] = 0, P[0][1] = P[1][0] = 1`, what about `P[1][1]`? Well, to give a square of size larger than `1` in `P[1][1]`, all of its three neighbors (left, up, left-up) should be non-zero, right? In this case, the left-up neighbor `P[0][0] = 0`, so `P[1][1]` can only be 1, which means that it contains the square of itself.

Now you are near the solution. In fact, `P[i][j] = min(P[i - 1][j], P[i][j - 1], P[i - 1][j - 1]) + 1` in this case.

Taking all these together, we have the following state equations.

1. `P[0][j] = matrix[0][j]` (topmost row);
2. `P[i][0] = matrix[i][0]` (leftmost column);
3. For `i > 0` and `j > 0`: if `matrix[i][j] = 0`, `P[i][j] = 0`; if `matrix[i][j] = 1`, `P[i][j] = min(P[i - 1][j], P[i][j - 1], P[i - 1][j - 1]) + 1`.

Putting them into codes, and maintain a variable `maxsize` to record the maximum size of the square we have seen, we have the following (unoptimized) solution.

```
int maximalSquare(vector<vector<char>>& matrix) {
    int m = matrix.size();
    if (!m) return 0;
    int n = matrix[0].size();
    vector<vector<int> > size(m, vector<int>(n, 0));
    int maxsize = 0;
    for (int j = 0; j < n; j++) {
        size[0][j] = matrix[0][j] - '0';
        maxsize = max(maxsize, size[0][j]);
    }
    for (int i = 1; i < m; i++) {
        size[i][0] = matrix[i][0] - '0';
        maxsize = max(maxsize, size[i][0]);
    }
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            if (matrix[i][j] == '1') {
                size[i][j] = min(size[i - 1][j - 1], min(size[i - 1][j], size[i][
j - 1])) + 1;
                maxsize = max(maxsize, size[i][j]);
            }
        }
    }
    return maxsize * maxsize;
}
```

Now let's try to optimize the above solution. As can be seen, each time when we update `size[i][j]`, we only need `size[i][j - 1], size[i - 1][j - 1]` (at the previous left column) and `size[i - 1][j]` (at the current column). So we do not need to maintain the full `m*n` matrix. In fact, keeping two columns is enough. Now we have the following optimized solution.

```
int maximalSquare(vector<vector<char>>& matrix) {
    int m = matrix.size();
    if (!m) return 0;
    int n = matrix[0].size();
    vector<int> pre(m, 0);
    vector<int> cur(m, 0);
    int maxsize = 0;
    for (int i = 0; i < m; i++) {
        pre[i] = matrix[i][0] - '0';
        maxsize = max(maxsize, pre[i]);
    }
    for (int j = 1; j < n; j++) {
        cur[0] = matrix[0][j] - '0';
        maxsize = max(maxsize, cur[0]);
        for (int i = 1; i < m; i++) {
            if (matrix[i][j] == '1') {
                cur[i] = min(cur[i - 1], min(pre[i - 1], pre[i])) + 1;
                maxsize = max(maxsize, cur[i]);
            }
        }
        swap(pre, cur);
        fill(cur.begin(), cur.end(), 0);
    }
    return maxsize * maxsize;
}
```

Now you see the solution is finished? In fact, it can still be optimized! In fact, we need not maintain two vectors and one is enough. If you want to explore this idea, please refer to the answers provided by @stellari below. Moreover, in the code above, we distinguish between the `0`-th row and other rows since the `0`-th row has no row above it. In fact, we can make all the `m` rows the same by padding a `0` row on the top (in the following code, we pad a `0` on top of `dp`). Finally, we will have the following short code :) If you find it hard to understand, try to run it using your pen and paper and notice how it realizes what the two-vector solution does using only one vector.

```
int maximalSquare(vector<vector<char>>& matrix) {
    if (matrix.empty()) return 0;
    int m = matrix.size(), n = matrix[0].size();
    vector<int> dp(m + 1, 0);
    int maxsize = 0, pre = 0;
    for (int j = 0; j < n; j++) {
        for (int i = 1; i <= m; i++) {
            int temp = dp[i];
            if (matrix[i - 1][j] == '1') {
                dp[i] = min(dp[i], min(dp[i - 1], pre)) + 1;
                maxsize = max(maxsize, dp[i]);
            }
            else dp[i] = 0;
            pre = temp;
        }
    }
    return maxsize * maxsize;
}
```

This solution, since posted, has been suggested various improvements by kind people. For a more comprehensive collection of the solutions, please visit my technical blog.

written by jianchao.li.fighter original link here

Solution 2

```
public int maximalSquare(char[][] a) {
    if(a.length == 0) return 0;
    int m = a.length, n = a[0].length, result = 0;
    int[][] b = new int[m+1][n+1];
    for (int i = 1 ; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if(a[i-1][j-1] == '1') {
                b[i][j] = Math.min(Math.min(b[i][j-1] , b[i-1][j-1]), b[i-1][j])
+ 1;
                result = Math.max(b[i][j], result); // update result
            }
        }
    }
    return result*result;
}
```

written by andywhite original link here

Solution 3

```
public int maximalSquare(char[][] a) {
  if (a == null || a.length == 0 || a[0].length == 0)
    return 0;

  int max = 0, n = a.length, m = a[0].length;

  // dp(i, j) represents the length of the square
  // whose lower-right corner is located at (i, j)
  // dp(i, j) = min{ dp(i-1, j-1), dp(i-1, j), dp(i, j-1) }
  int[][] dp = new int[n + 1][m + 1];

  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
      if (a[i - 1][j - 1] == '1') {
        dp[i][j] = Math.min(dp[i - 1][j - 1], Math.min(dp[i - 1][j], dp[i][j - 1]
)) + 1;
        max = Math.max(max, dp[i][j]);
      }
    }
  }

  // return the area
  return max * max;
}
```

written by jeantimex original link here