21 Largest square submatrix of ones

You are given a matrix where elements are either 0 or 1. Return the number of ones in the largest square-shaped region of the matrix which contains only ones.

Example:

```
matrix = [
    [ 1,  1,  0,  1,  0],
    [ 1,  1,  0,  0,  1],
    [ 0,  1,  1,  1,  0],
    [ 0,  1,  1,  1,  0],
    [ 1,  1,  1,  1,  1],
]
```

The answer is 9 for the 3×3 square with top-left corner on the third row and second column.

Clarification questions

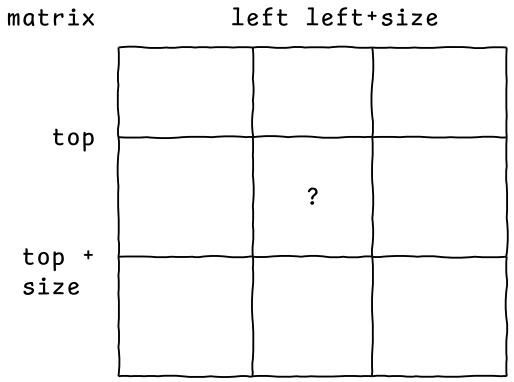
Q: How large can the matrix be?
A: Up to 1000 rows/columns.

Q: What answer should be given if the matrix is empty?
A: 0.

Q: Is a single 1 considered a square?
A: Yes.

Solution 1: brute-force, $O(n^5)$

The straightforward solution is to enumerate all the possible squares that contain only ones, computing the area for each one and keeping track of the maximum. We can enumerate all squares by first enumerating all the possible positions of their top-left corner, then iterating over all the possible sizes:



Suppose we write a helper function `square_has_only_ones(top, left, size)` to check if a square contains only ones. Then the search for the largest square can be written as:

```
def find_largest_square(matrix):
    def square_has_only_ones(top, left, size):
        ...

    num_rows = len(matrix)
    num_cols = len(matrix[0])
    largest = 0
    for top in range(num_rows):
        for left in range(num_cols):
            # Make sure we do not go past the edges of the matrix.
            max_size = min(num_rows - top, num_cols - left)
            for size in range(1, max_size + 1):
                if square_has_only_ones(top, left, size):
                    largest = max(largest, size * size)
    return largest
```

The helper function can be implemented as:

```
def find_largest_square(matrix):
    def square_has_only_ones(top, left, size):
        for row in range(top, top + size):
            for col in range(left, left + size):
                if matrix[row][col] == 0:
                    return False
        return True
    ...
```

We also need to handle the special case of the empty matrix. Putting it all together:

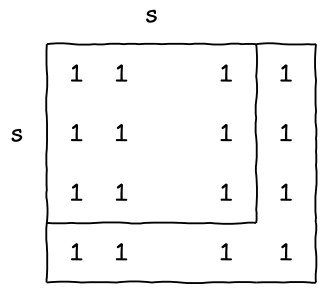
```
def find_largest_square(matrix):
    if not matrix:
        return 0
    num_rows = len(matrix)
    num_cols = len(matrix[0])

    def square_has_only_ones(top, left, size):
        for row in range(top, top + size):
            for col in range(left, left + size):
                if matrix[row][col] == 0:
                    return False
        return True

    largest = 0
    for top in range(num_rows):
        for left in range(num_cols):
            max_size = min(num_rows - top, num_cols - left)
```

```
            for size in range(1, max_size + 1):
                if square_has_only_ones(top, left, size):
                    largest = max(largest, size * size)
    return largest
```

The time complexity is $O(n^5)$ due to the 5 nested loops that iterate up to n steps.

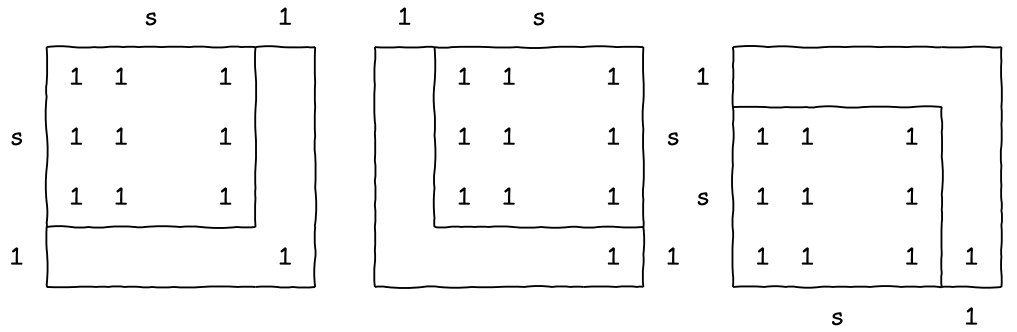
Solution 2: dynamic programming, $O(n^2)$

We can reduce complexity by checking if squares are filled with ones in a more efficient way. An idea is to do this incrementally:



If we already checked that an $s \times s$ square is filled with ones, we can extend it by one row and column. We only have to check if the row and column are filled with ones to determine if the new larger square is also filled with ones. This reduces the complexity of checking that a new square is filled with ones from $O(n^2)$ to $O(n)$.

However we can do better if we look at the structure of the larger square. Any $(s+1) \times (s+1)$ square contains three overlapping $s \times s$ squares:



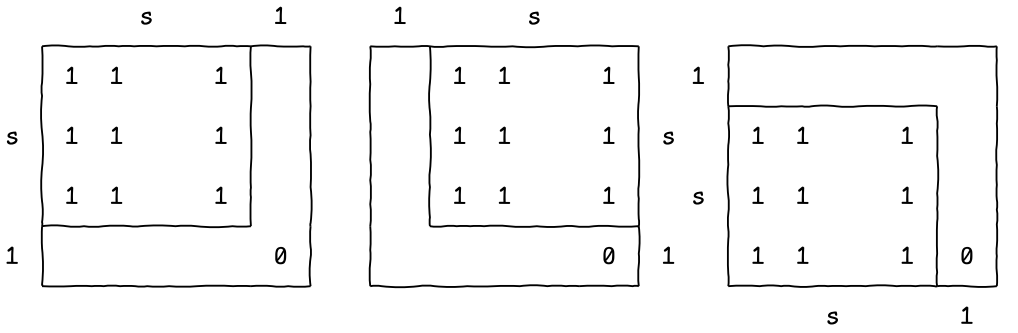
To check if an $(s + 1) \times (s + 1)$ square is filled with ones, it is sufficient to:

- Check if the bottom-right corner contains a one;
- Check if the $s \times s$ square one row above is filled with ones;
- Check if the $s \times s$ square one column to the left is filled with ones;
- Check if the $s \times s$ square one row above and one column to the left is filled with ones.

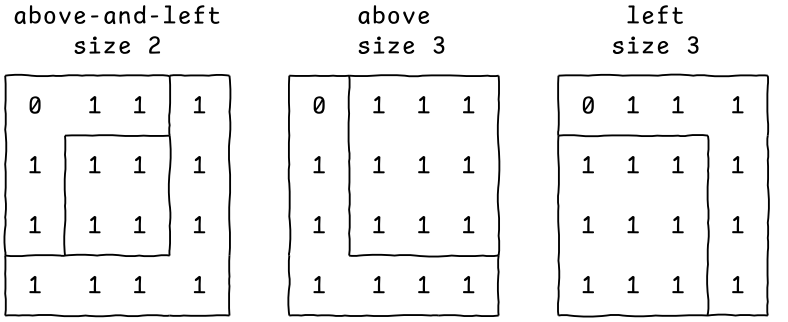
If the squares on rows above and on columns to the left have already been checked, we can check the $(s + 1) \times (s + 1)$ square incrementally in just $O(1)$ time.

We still have to decide how to handle the cases where some of the checks are failing.

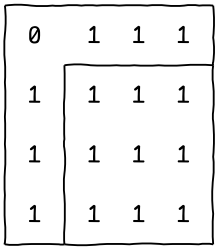
If the the bottom-right corner contains a zero, we cannot form a square filled with ones:



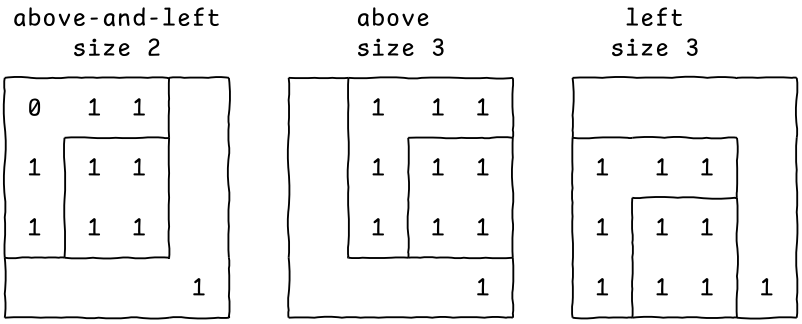
If one or more of the three smaller neighboring squares are not of the same size:



In this case we cannot form a square of size 4 filled with ones, since the top-left corner contains a zero. However we can form a square of size 3:



This is equivalent to growing incrementally squares of size 2:



Given the above, let's refine the procedure:

- We iterate over the matrix elements row by row, column by column;
- If the current cell contains a zero, no square filled with ones can be formed there;
- If the current cell contains a one:
  - We read the size `s1` of the largest square with bottom-right corner above the cell;
  - We read the size `s2` of the largest square with bottom-right corner to the left of the cell;
  - We read the size `s3` of the largest square with bottom-right corner above and to the left of the cell;
  - We compute the size of the largest square filled with ones that has bottom-right corner in the cell as `min(s1, s2, s3)`.

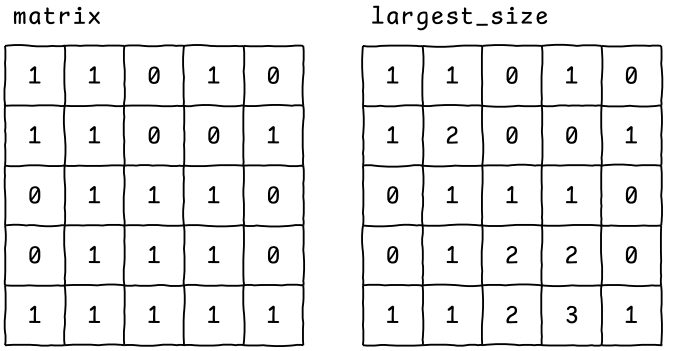
We can implement this as:

```
def find_largest_square(matrix):
    if not matrix:
        return 0
    num_rows = len(matrix)
    num_cols = len(matrix[0])
    # largest_size[row][col] = size of the largest square filled with ones
    #                          having bottom-right corner at (row, col)
    largest_size = [[0] * len(row) for row in matrix]
    largest = 0
    for row in range(num_rows):
        for col in range(num_cols):
            if matrix[row][col] == 0:
                continue
            size_above = largest_size[row - 1][col] if row else 0
            size_left = largest_size[row][col - 1] if col else 0
            size_above_left = (largest_size[row - 1][col - 1]
                               if row and col else 0)
            largest_size[row][col] = 1 + min(size_above,
                                             size_left,
                                             size_above_left)
            largest = max(largest, largest_size[row][col])
```

```
    return largest * largest
```

The time complexity is $O(n^2)$. The space complexity is $O(n^2)$ as well due to the additional matrix `largest_size` used to store square sizes. It is possible to reduce space complexity to $O(n)$, since we only need to store the last row of `largest_size`.

Here is an example of `largest_size` computed for the sample input:



The largest cell of `largest_size` has value 3, giving a square size of 9.