**835. Image Overlap**

Medium

527719Add to ListShare

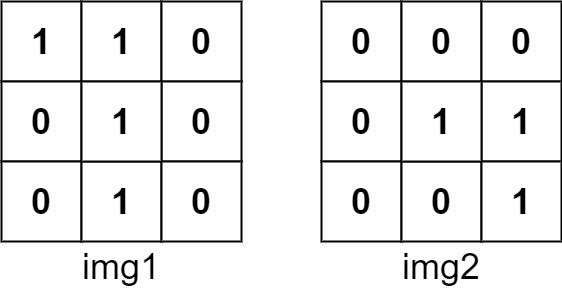
You are given two images img1 and img2 both of size n x n, represented as binary, square matrices of the same size. (A binary matrix has only 0s and 1s as values.)

We translate one image however we choose (sliding it left, right, up, or down any number of units), and place it on top of the other image.  After, the *overlap* of this translation is the number of positions that have a 1 in both images.

(Note also that a translation does **not** include any kind of rotation.)

What is the largest possible overlap?

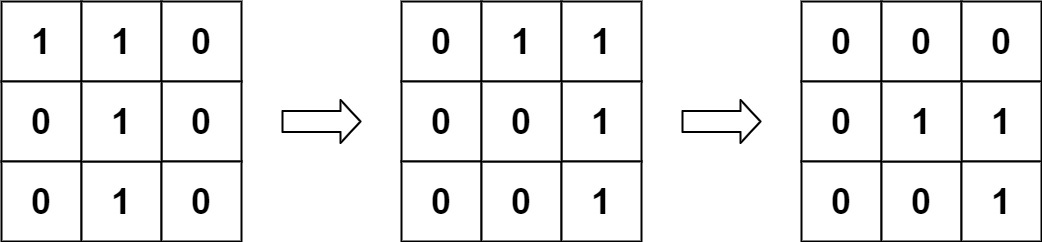
**Example 1:**



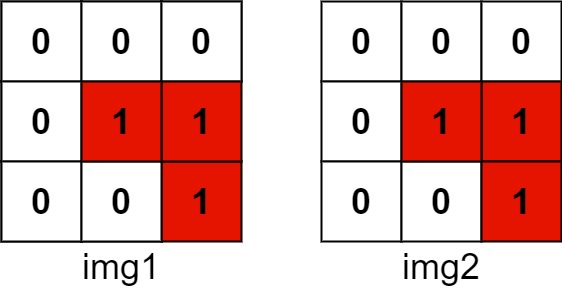
**Input:** img1 = [[1,1,0],[0,1,0],[0,1,0]], img2 = [[0,0,0],[0,1,1],[0,0,1]]

**Output:** 3

**Explanation:** We slide img1 to right by 1 unit and down by 1 unit.



The number of positions that have a 1 in both images is 3. (Shown in red)



**Example 2:**

**Input:** img1 = [[1]], img2 = [[1]]

**Output:** 1

**Example 3:**

**Input:** img1 = [[0]], img2 = [[0]]

**Output:** 0

**Constraints:**

* n == img1.length
* n == img1[i].length
* n == img2.length
* n == img2[i].length
* 1 <= n <= 30
* img1[i][j] is 0 or 1.
* img2[i][j] is 0 or 1.

Approach 2: Linear Transformation

**Intuition**

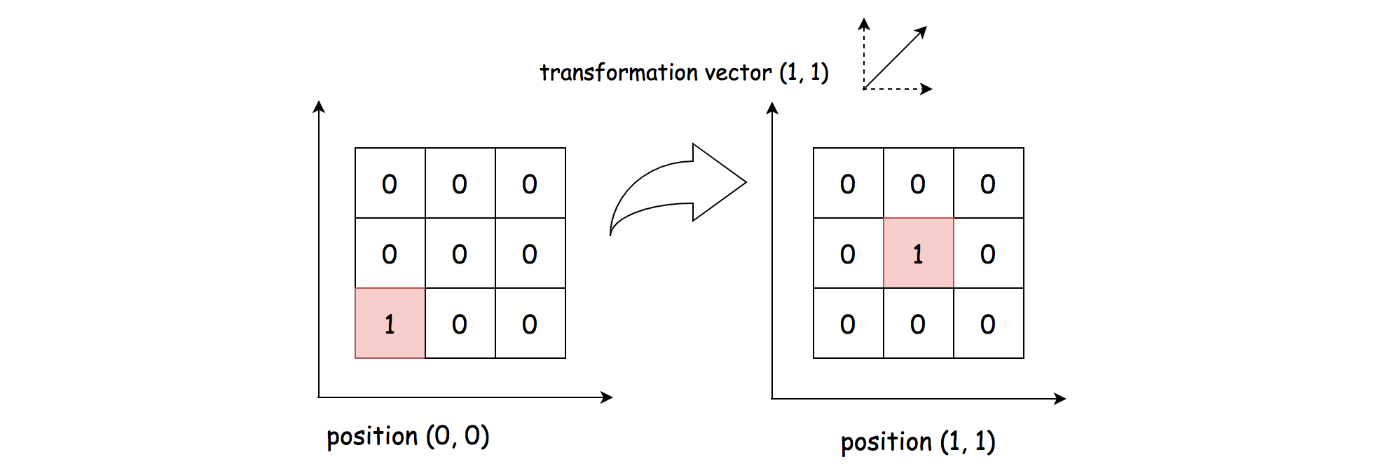
One drawback of the above algorithm is that we would scan through those zones that are filled with zeros over and over, even though these zones are not of our interests.

Because for those cells filled with zero, no matter how we *shift*, they would not *add up* to the final solutions. As a follow-up question, we could ask ourselves that, *can we****focus****on those cells with ones?*

The answer is yes. The idea is that we filter out those cells with ones in both matrices, and then we apply the ***linear transformation*** to *align* the cells.

First of all, we define a 2-dimension coordinate, via which we could assign a unique coordinate to each cell in the matrix, *e.g.* a cell can be indexed as I = (X\_i, Y\_i)*I*=(*Xi*​,*Yi*​).

Then to shift a cell, we can obtain the new position of the cell by applying a *linear transformation*. For example, to shift the cell to the right by one and to the up side by one is to apply the linear transformation vector of V = (1, 1)*V*=(1,1). The new index of the cell can be obtained by I + V = (X\_i + 1, Y\_i + 1)*I*+*V*=(*Xi*​+1,*Yi*​+1).



Furthermore, given two matrices, we have two non-zero cells respectively in the matrices as P\_a =(X\_a, Y\_a)*Pa*​=(*Xa*​,*Ya*​) and P\_b = (X\_b, Y\_b)*Pb*​=(*Xb*​,*Yb*​). To **align** these cells together, we would need a transformation vector as V\_{ab} = (X\_b - X\_a, Y\_b - Y\_a)*Vab*​=(*Xb*​−*Xa*​,*Yb*​−*Ya*​), so that P\_a + V\_{ab} = P\_b*Pa*​+*Vab*​=*Pb*​.

Now, the key insight is that all the cells in the **same** overlapping zone would share the **same** linear transformation vector.

Based on the above insight, we can then use the transformation vector V\_{ab}*Vab*​ as a key to **group** all the non-zero cells alignments between two matrices. Each group represents an overlapping zone. Naturally, the size of the overlapping zone would be the size of the group as well.

**Algorithm**

The algorithm can be implemented in two overall steps.

* First, we filter out those non-zero cells in each matrix respectively.
* Second, we do a cartesian product on the non-zero cells. For each pair of the products, we calculate the corresponding linear transformation vector as V\_{ab} = (X\_b - X\_a, Y\_b - Y\_a)*Vab*​=(*Xb*​−*Xa*​,*Yb*​−*Ya*​). Then, we count the number of the pairs that have the same transformation vector, which is also the number of ones in the overlapping zone.

Here are some sample implementation which are inspired from the user [TejPatel18](https://leetcode.com/problems/image-overlap/discuss/150504/Python-Easy-Logic) in the discussion forum.

**Complexity Analysis**

Let M\_a, M\_b*Ma*​,*Mb*​ be the number of non-zero cells in the matrix A and B respectively. Let N*N* be the width of the matrix.

* Time Complexity: \mathcal{O}(N^4)O(*N*4).
  + In the first step, we filter out the non-zero cells in each matrix, which would take \mathcal{O}(N^2)O(*N*2) time.
  + In the second step, we enumerate the cartesian product of non-zero cells between the two matrices, which would take \mathcal{O}(M\_a \cdot M\_b)O(*Ma*​⋅*Mb*​) time. In the worst case, both M\_a*Ma*​ and M\_b*Mb*​ would be up to N^2*N*2, *i.e.* matrix filled with ones.
  + To sum up, the overall time complexity of the algorithm would be \mathcal{O}(N^2) + \mathcal{O}(N^2 \cdot N^2) = \mathcal{O}(N^4)O(*N*2)+O(*N*2⋅*N*2)=O(*N*4).
  + Although this approach has the same time complexity as the previous approach, it should run faster in practice, since we ignore those zero cells.
* Space Complexity: \mathcal{O}(N^2)O(*N*2)
  + We kept the indices of non-zero cells in both matrices. In the worst case, we would need the \mathcal{O}(N^2)O(*N*2) space for the matrices filled with ones.