835. Image Overlap

Matrix

# **Problem Description**

<u>Array</u>

Medium

determine the largest overlap between these two images when one image is translated (slid in any direction) over the other. The overlap is quantified by counting the number of positions where both images have a 1. A key constraint is that rotation is not allowed, and any 1's that move outside the matrix bounds are discarded. Our goal is to find the maximum overlap after trying all possible translations. Intuition

The problem presents us with two binary square matrices, img1 and img2, each containing only 0's and 1's. Our task is to

To solve this problem, we should think about how to track the overlap between the two images effectively. One way to do this is to consider each 1 in img1 and match it against each 1 in img2. For each pairing, we can calculate the translation vector (the difference in x and y coordinates) required to move the 1 from img1 directly onto the 1 in img2.

We approach the solution by iterating over all elements of img1 and img2. For each 1 we find in img1, we iterate through img2, and for every 1 in img2, we calculate the translation vector. This vector is represented as a tuple (delta\_row, delta\_col), where delta\_row is the difference in the row indices and delta\_col is the difference in the column indices between a 1 in img1 and a 1 in img2.

All these vectors are stored in a counter (a dictionary with tuples as keys and their counts as values), which is used to count how many times each vector occurs. The highest count in this counter, after considering all pairs of 1 bits, would represent the most overlapping translation, giving us the largest possible overlap between img1 and img2.

The provided solution employs a brute-force approach, utilizing nested loops to iterate through each element of both img1 and

## We start by obtaining the size of the given images n, assuming both images are of size $n \times n$ .

**Solution Approach** 

img2. Here's a step-by-step explanation:

We create a Counter from the collections module to keep track of all translation vectors and how frequently they result in an overlap. A translation vector is represented as a tuple (delta\_i, delta\_j), which corresponds to the difference in rows

- (delta\_i) and columns (delta\_j) needed to align a 1 from img1 with a 1 from img2.
- We then proceed with four nested loops: The outer two loops iterate over all the cells of img1: ■ For each position (i, j) in img1 that contains a 1, we then iterate over every position in img2 using two more loops.
- For each position (h, k) in img2 that contains a 1, we calculate the translation vector needed to move the 1 from (i, j) in img1 to (h,

complexity of  $O(n^4)$ , where n is the size of one dimension of the image.

The inner two loops iterate over all the cells of img2:

- k) in img2. The calculated translation vector (i - h, j - k) is then used to increment the count in our Counter. This effectively counts
- how many 1 bits from img1 would overlap with 1 bits in img2 when img1 is translated according to this vector. After evaluating all possible pairs of 1s in both images, we find the maximum count from our Counter. This count represents the largest number of overlapping 1s for the best translation. We return this maximum count as the solution.

In the worst case, every element of img1 (which could be a 1) has to be compared with every element of img2, leading us to a time

- Finally, the last line of the function accounts for the possibility of no overlap at all. If cnt (our Counter) remains empty because there are no 1 bits overlapping in any translation, we return 0. Otherwise, we return the maximum value found in cnt, which is the result of max(cnt.values()).
- Let's illustrate the solution approach with an example where img1 and img2 are both 3×3 binary matrices: img1 = [[1,0,0],[0,1,0], [0,0,0]

We create a Counter to keep track of the translation vectors - delta\_i and delta\_j, which denote the row and column

## Now, let's walk through the solution approach:

[0,1,0],

[0,0,1]

differences, respectively.

which is (0, 0).

Solution Implementation

from typing import List

class Solution:

from collections import Counter

size = len(A)

for i in range(size):

for j in range(size):

**if** A[i][j] == **1**:

for h in range(size):

public int largestOverlap(int[][] img1, int[][] img2) {

int n = img1.length; // dimension of the images

// Iterate through the first image

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

if (img1[i][j] == 1) {

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

int maxOverlap = 0; // To track the maximum overlap

for k in range(size):

**if** B[h][k] == 1:

return max(overlap\_count.values()) if overlap\_count else 0

Map<List<Integer>, Integer> shiftCount = new HashMap<>();

for (int h = 0; h < size; ++h) {</pre>

function largestOverlap(image1: number[][], image2: number[][]): number {

// Proceed if this cell of the first image has a '1'

for (let row2 = 0; row2 < size; ++row2) {</pre>

// Now, iterate through each cell of the second image

for (let col2 = 0; col2 < size; ++col2) {</pre>

// Return the maximum overlap count found

// Determine the dimension of the square images

// Create a map to hold the counts of the overlaps

// Initialize the maximum number of overlaps to 0

// Iterate through each cell of the first image

if (image1[row1][col1] === 1) {

for (let col1 = 0; col1 < size; ++col1) {</pre>

for (let row1 = 0; row1 < size; ++row1) {</pre>

const overlapCount: Map<number, number> = new Map();

return max0verlap;

const size = image1.length;

let maxOverlap = 0;

for (int k = 0; k < size; ++k) {

if (image2[h][k] == 1) {

overlapCount[translationVector]++;

img2 = [[0,0,0],

**Example Walkthrough** 

Then, we iterate over each cell in img1:

∘ For cell (0,0) in img1, which contains a 1, we check every cell in img2. Only cells (1,1) and (2,2) contain a 1 in img2.

○ To overlap (0,0) from img1 onto (1,1) in img2, we calculate the translation vector as (1 - 0, 1 - 0) which is (1, 1).

For each 1 in img2, we calculate the translation vectors necessary to overlap this 1 with the 1 at (0,0) in img1:

We first identify the size n of the square matrices, which in this case is 3.

○ To overlap (0,0) from img1 onto (2,2) in img2, the translation vector is (2 - 0, 2 - 0) which is (2, 2). We add and/or increment the count of these vectors in our Counter.

○ There is one 1 at (1,1) in img1 which corresponds directly to the 1 at (1,1) in img2. The translation vector in this case is (1 - 1, 1 - 1)

Finally, we scan through our Counter to determine the maximum overlap, which is 1 in this case since each translation only

So, the largest overlap by translating img1 over img2 or vice versa is a single 1. The steps above demonstrate the brute-force

We increment the count of this vector in our Counter. Now our Counter has counted translation vectors with counts as follows: • (1, 1):1 count

# Obtain the size of the square images

We repeat steps 3-5 for the other 1 at position (1,1) in img1:

overlaps one 1 in both images. The result, therefore, is 1.

o (2, 2):1 count (0, 0):1 count

approach systematically considering all translation vectors and assessing the overlaps.

def largestOverlap(self, A: List[List[int]], B: List[List[int]]) -> int:

# If we have a 1 (active pixel) in the first image

# Check against every pixel in the second image

overlap\_count[(i - h, j - k)] += 1

// Using a map to store the count of overlaps with particular vector shifts

// Only interested in cells that are 1 (part of the image)

- **Python**
- # Initialize a counter to keep track of overlaps overlap\_count = Counter() # Go through every pixel in first image

# Count overlaps by storing the delta (difference) of positions

# If there's a 1 in the same position of the second image

```
# Find the maximum number of overlaps (if there are any), otherwise return 0
```

Java

class Solution {

```
// Iterate through the second image
                    for (int h = 0; h < n; ++h) {
                        for (int k = 0; k < n; ++k) {
                            // Only interested in cells that are 1 (part of the image)
                            if (img2[h][k] == 1) {
                                // Calculate the vector shift needed to overlay img1's 1 over img2's 1
                                List<Integer> shift = List.of(i - h, j - k);
                                // Using merge to count occurrences of each shift
                                int count = shiftCount.merge(shift, 1, Integer::sum);
                                // Keeping track of the max overlap by comparing and choosing the larger count
                                maxOverlap = Math.max(maxOverlap, count);
       // Return the max overlap found during iteration
       return maxOverlap;
C++
#include <vector>
#include <map>
#include <utility>
#include <algorithm>
class Solution {
public:
    int largestOverlap(std::vector<std::vector<int>>& image1, std::vector<std::vector<int>>& image2) {
        int size = image1.size();
        std::map<std::pair<int, int>, int> overlapCount; // Holds the count of overlaps per translation vector
        int maxOverlap = 0; // Keeps track of the maximum overlap count
       // Iterate over all the pixels in image1
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                // Check if there is an image part at the current pixel in image1
                if (image1[i][j] == 1) {
                    // Iterate over all the pixels in image2
```

// Check if there is an image part at the current pixel in image2

// Increment the overlap count for this translation vector

std::pair<int, int> translationVector = std::make\_pair(i - h, j - k);

// Update the maxOverlap if the current overlap count is greater

maxOverlap = std::max(maxOverlap, overlapCount[translationVector]);

// Calculate the translation vector from the current pixel in image2 to the current pixel in imag

**Time Complexity** 

**TypeScript** 

```
// Look for '1's in the second image to count overlaps
                          if (image2[row2][col2] === 1) {
                              // Calculate the translation vector as a unique key (assuming size < 200)
                              const translationKey = (row1 - row2) * 200 + (col1 - col2);
                              // Use the translation key to update and get the count of this overlap
                              const currentCount = (overlapCount.get(translationKey) ?? 0) + 1;
                              overlapCount.set(translationKey, currentCount);
                              // Keep track of the max number of overlaps
                              maxOverlap = Math.max(maxOverlap, currentCount);
      // Return the maximum number of overlapping ones
      return max0verlap;
from typing import List
from collections import Counter
class Solution:
   def largestOverlap(self, A: List[List[int]], B: List[List[int]]) -> int:
       # Obtain the size of the square images
       size = len(A)
       # Initialize a counter to keep track of overlaps
       overlap_count = Counter()
       # Go through every pixel in first image
        for i in range(size):
           for j in range(size):
               # If we have a 1 (active pixel) in the first image
               if A[i][j] == 1:
                   # Check against every pixel in the second image
                    for h in range(size):
                       for k in range(size):
                           # If there's a 1 in the same position of the second image
                           if B[h][k] == 1:
                               # Count overlaps by storing the delta (difference) of positions
                               overlap count[(i - h, j - k)] += 1
       # Find the maximum number of overlaps (if there are any), otherwise return 0
       return max(overlap_count.values()) if overlap_count else 0
Time and Space Complexity
```

### The time complexity of the function is governed by the four nested loops: • The two outer loops iterate over each cell of the img1 matrix, resulting in n^2 iterations, where n is the length of the matrix side. • The two inner loops iterate over each cell of the img2 matrix, also resulting in n^2 iterations.

possible directions. Here is the analysis of its time complexity and space complexity:

For each '1' in img1, the inner loops iterate through the entire img2 matrix. In the worst case, this results in n^2 iterations for the

possible shifts range from (-n+1, -n+1) to (n-1, n-1).

- '1's in img1. Combining these factors, the worst-case time complexity is  $0(n^2 * n^2)$ , which simplifies to  $0(n^4)$ .
- **Space Complexity**

The space complexity is determined by the additional space used by the program which is mainly due to the Counter object used

to count the number of overlaps for each shift. The number of different shifts is bounded by the size of the matrix (n \* n), as the

The given code snippet is designed to find the largest overlap of two given binary matrices img1 and img2 by shifting img2 in all

Therefore, in the worst case, the Counter could have up to n^2 entries if every possible shift occurs at least once. Thus, the space complexity is  $0(n^2)$ .

In summary, the time complexity of the code is  $0(n^4)$  and the space complexity is  $0(n^2)$ .