Solution

Overview

First of all, this is a really fun problem to solve, as one would discover later. In addition, it could be a practical problem in real world. For instance, if one can find the maximal overlapping zone between two images, one could *clip* the images to make them smaller and more focused.

In this article, we will cover three approaches as follows:

* We could solve the problem intuitively by enumerating all the possible overlapping zones.
* Or more efficiently, we can apply some knowledge of ***linear algebra*** (or geometry), as we will present another solution in the second approach.
* Finally, we could even solve it with the conception of ***convolution***, as in Convolution Neural Network (*a.k.a.* CNN), which is the backbone operation for the image recognition models nowadays.

Approach 1: Shift and Count

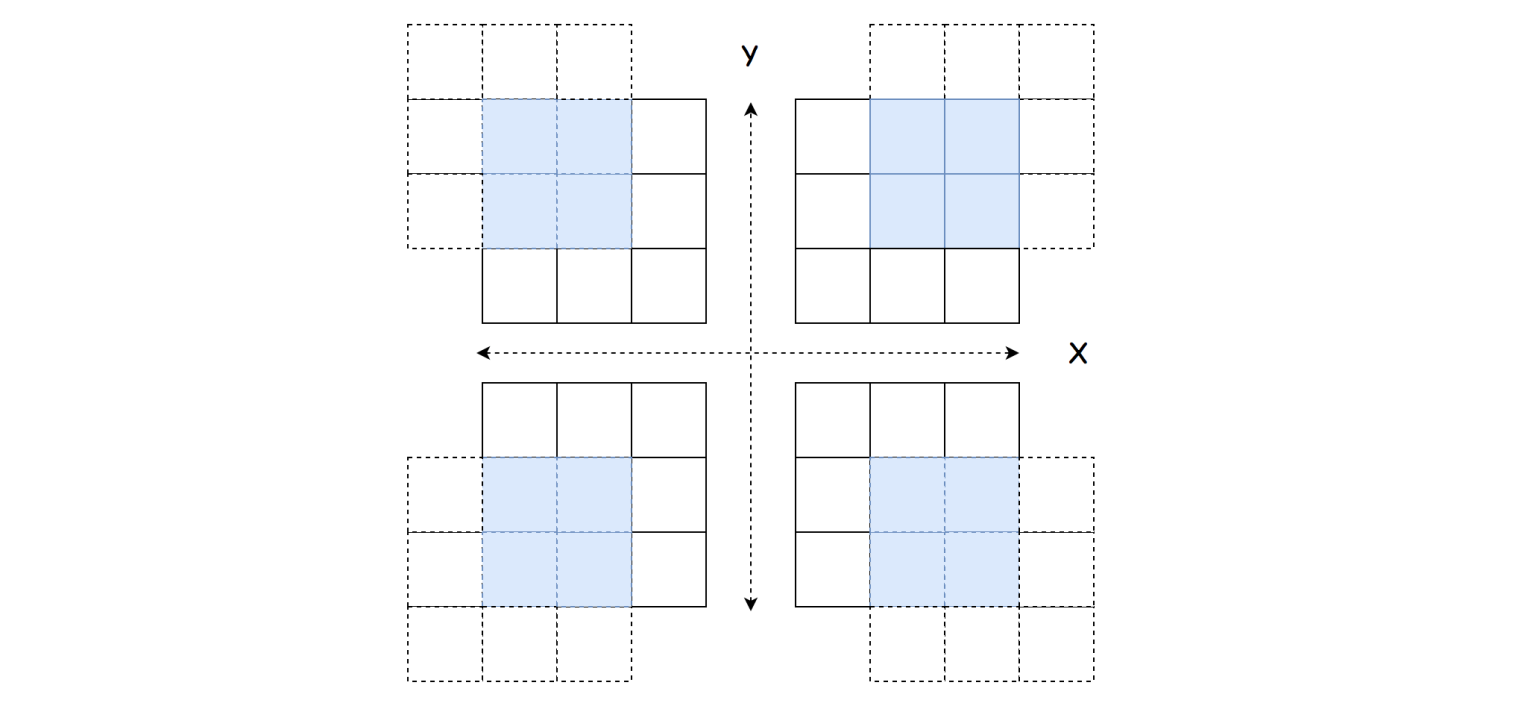
**Intuition**

As stated in the problem description, in order to calculate the number of ones in the overlapping zone, we should first ***shift*** one of the images. Once the image is shifted, it is intuitive to ***count*** the numbers.

Therefore, a simple idea is that one could come up all possible overlapping zones, by *shifting* the image matrix, and then simply count within each overlapping zone.

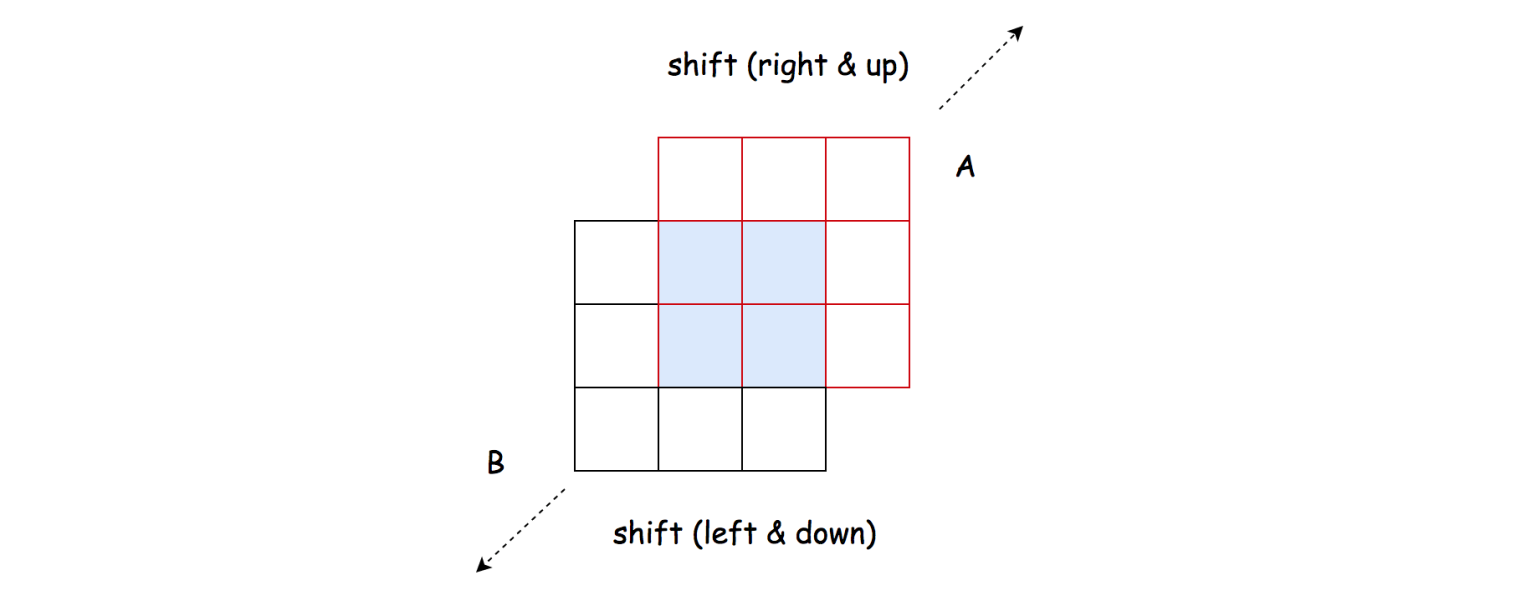
The image matrix could be shifted in four directions, *i.e.* left, right, up and down.

We could represent the shifting with a 2-axis coordinate as follows, where the X-axis indicates the shifting on the directions of *left* and *right* and the Y-axis indicates the shifting of *up* and *down*.



For instance, the coordinate of (1, 1) represents that we shift the matrix to the right by one and to the up side by one as well.

One important insight is that shifting one matrix to a direction is **equivalent** to shifting the other matrix to the *opposite* direction, in the sense that we would have the same overlapping zone at the end.

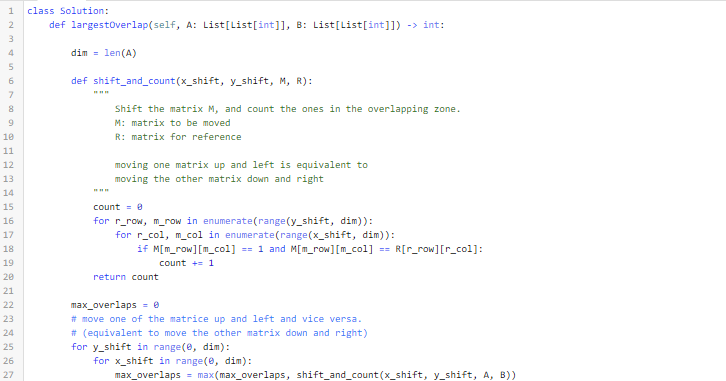


For example, by shifting the matrix A to one step on the right, is same as shifting the matrix B to the left by one step.

**Algorithm**

Based on the above intuition, we could implement the solution step by step. First we define the function shift\_and\_count(x\_shift, y\_shift, M, R) where we shift the matrix M with reference to the matrix R with the shifting coordinate (x\_shift, y\_shift) and then we count the overlapping ones in the overlapping zone.

* The algorithm is organized as a loop over all possible combinations of shifting coordinates (x\_shift, y\_shift).
* More specifically, the ranges of x\_shift and y\_shift are both [0, N-1] where N*N* is the width of the matrix.
* At each iteration, we invoke the function shift\_and\_count() twice to shift and count the overlapping zone, first with the matrix B as the reference and vice versa.





**Complexity Analysis**

Let N*N* be the width of the matrix.

First of all, let us calculate the number of all possible shiftings, (*i.e.* the number of overlapping zones).

For a matrix of length N*N*, we have 2(N-1)2(*N*−1) possible offsets along each axis to shift the matrix. Therefore, there are in total 2(N-1) \cdot 2(N-1) = 4(N-1)^22(*N*−1)⋅2(*N*−1)=4(*N*−1)2 possible overlapping zones to calculate.

* Time Complexity: \mathcal{O}(N^4)O(*N*4)
  + As discussed before, we have in total 4(N-1)^24(*N*−1)2 possible overlapping zones.
  + The size of the overlapping zone is bounded by \mathcal{O}(N^2)O(*N*2).
  + Since we iterate through each overlapping zone to find out the overlapping ones, the overall time complexity of the algorithm would be 4(N-1)^2 \cdot \mathcal{O}(N^2) = \mathcal{O}(N^4)4(*N*−1)2⋅O(*N*2)=O(*N*4).
* Space Complexity: \mathcal{O}(1)O(1)
  + As one can see, a constant space is used in the above algorithm.

Approach 2: Linear Transformation

**Intuition**

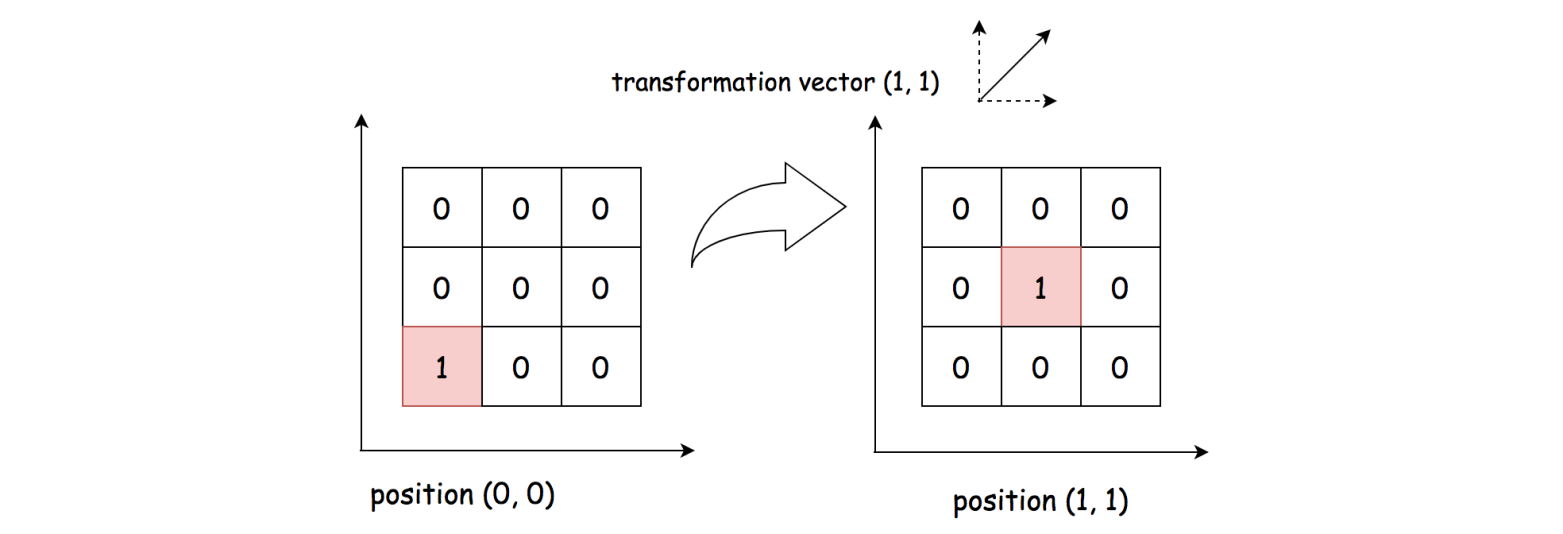
One drawback of the above algorithm is that we would scan through those zones that are filled with zeros over and over, even though these zones are not of our interests.

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.

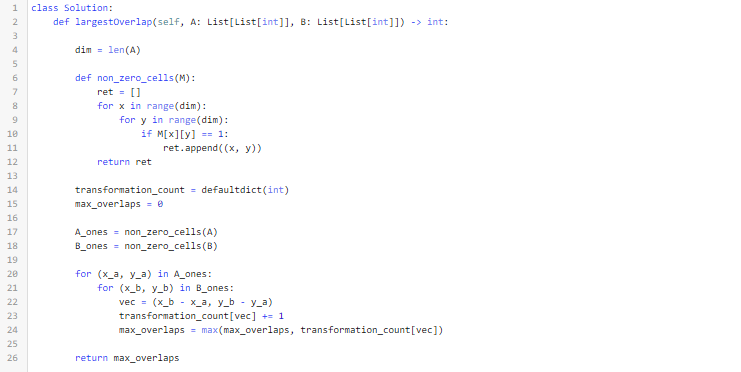
Based on the above insight, we can then use the transformation vector V\_{ab}*Vab*​ as a key to **group** all the non-zero cells alignments between two matrices. Each group represents an overlapping zone. Naturally, the size of the overlapping zone would be the size of the group as well.

**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**

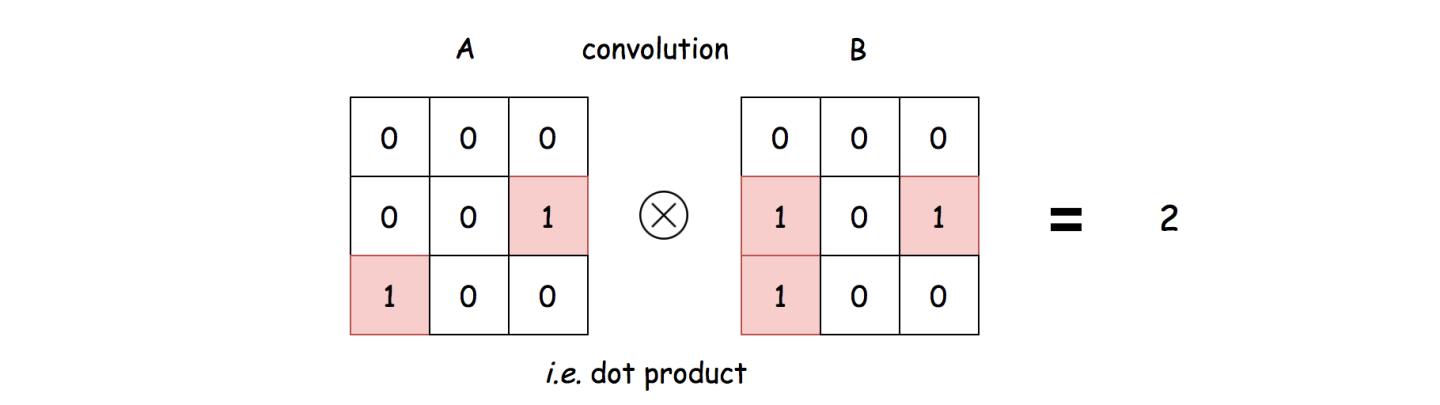
Let M\_a, M\_b*Ma*​,*Mb*​ be the number of non-zero cells in the matrix A and B respectively. Let N*N* be the width of the matrix.

* 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.

Approach 3: Imagine Convolution

**Intuition**

As it turns out, the number of overlapped ones in an overlapping zone is equal to the result of performing a [convolution operation](https://en.wikipedia.org/wiki/Kernel_(image_processing)) between two matrices.

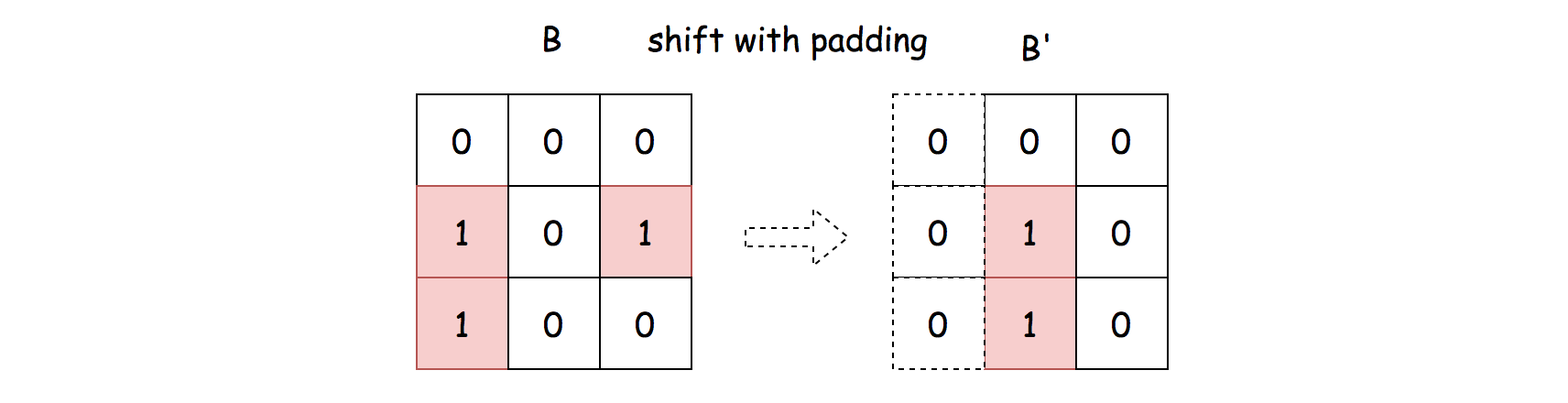


To put it simple, the convolution between two matrices is to perform a **dot product** between them, *i.e.* \sum\_{x}^{N}\sum\_{y}^{N}\big(V\_{xy}(A) \cdot V\_{xy}(B)\big)∑*xN*​∑*yN*​(*Vxy*​(*A*)⋅*Vxy*​(*B*)) where V\_{xy}(A)*Vxy*​(*A*) and is the value of the cell in the matrix A at the position of (x, y)(*x*,*y*).

From the above formulas, one can clearly see why the result of convolution is the number of overlapping ones.

Usually, the image convolution is performed between an image and a specific ***kernel*** matrix, in order to obtain certain effects such as blurring or sharpening. In our case, we would perform the convolution between the matrix A and the ***shifted*** matrix B which serves as a kernel.

More importantly, we should shift the matrix with truncation and ***zero padding***, in order to obtain a proper kernel for convolution.

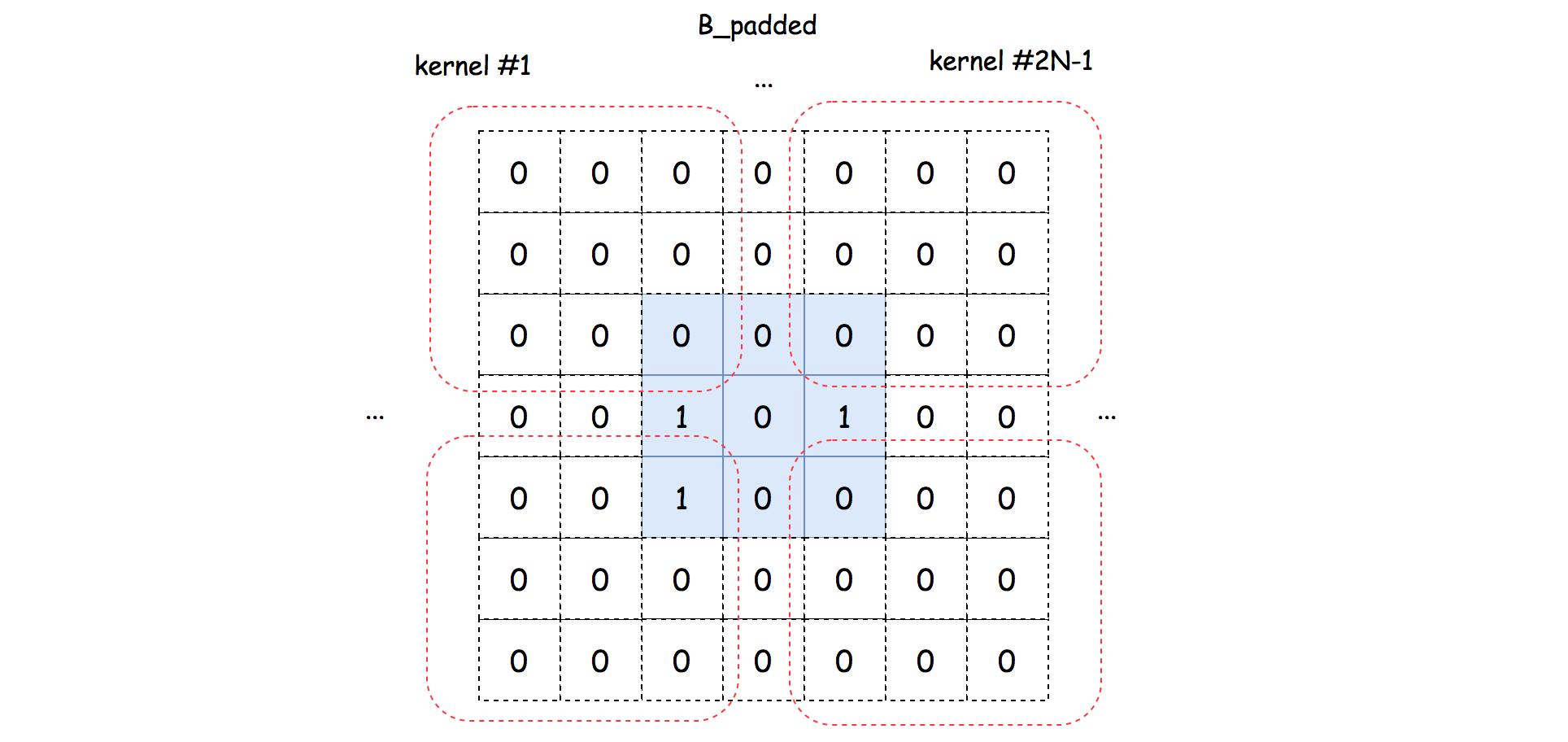


As a result, rather than manually counting the number of overlapping ones, we could perform the convolution operation instead.

**Algorithm**

Overall, we enumerate all possible kernels (by shifting the matrix B), and then perform the convolution operation to count the overlapping ones. The algorithm can be broke down into the following steps:

* First of all, we extend both the width and height of the matrix B to N + 2(N-1) = 3N-2*N*+2(*N*−1)=3*N*−2, and pad the extended cells with zeros, as follows:

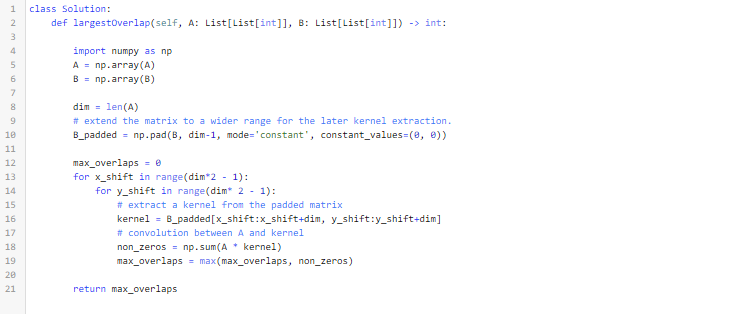


* From the extended and padded matrix B, we then can extract the kernel one by one.
* For each kernel, we perform the convolution operation with the matrix A to count the number of overlapping ones.
* At the end, we return the maximal value of overlapping ones.

Here are some sample implementations that are inspired from the solution of user [HeroKillerEver](https://leetcode.com/problems/image-overlap/discuss/131344/An-interesting-SOLUTION%3A-Let-us-think-about-it-as-%22Convolution%22) in the discussion forum.

Note that, in the Python solution, we utilise the numpy package, which is a well-known library in the tasks of data processing and machine learning.

The numpy library is highly *optimized* for the matrix operations, which is why it runs faster than the Approach #1, although they have the same time complexity.



**Complexity Analysis**

Let N*N* be the width of the matrix.

* Time Complexity: \mathcal{O}(N^4)O(*N*4)
  + We iterate through (2N-1) \cdot (2N-1)(2*N*−1)⋅(2*N*−1) number of kernels.
  + For each kernel, we perform a convolution operation, which takes \mathcal{O}(N^2)O(*N*2) time.
  + To sum up, the overall time complexity of the algorithm would be (2N-1) \cdot (2N-1) \cdot \mathcal{O}(N^2) = \mathcal{O}(N^4)(2*N*−1)⋅(2*N*−1)⋅O(*N*2)=O(*N*4).
* Space Complexity: \mathcal{O}(N^2)O(*N*2)
  + We extend the matrix B to the size of (3N-2) \cdot (3N-2)(3*N*−2)⋅(3*N*−2), which would require the space of \mathcal{O}(N^2)O(*N*2).