**Codewriting**

Given an array of positive integers **a**, your task is to calculate the sum of every possible **a[i] ∘ a[j]**, where **a[i] ∘ a[j]** is the concatenation of the string representations of **a[i]** and **a[j]** respectively.

Example

* For **a = [10, 2]**, the output should be **solution(a) = 1344**.
  + **a[0] ∘ a[0] = 10 ∘ 10 = 1010**,
  + **a[0] ∘ a[1] = 10 ∘ 2 = 102**,
  + **a[1] ∘ a[0] = 2 ∘ 10 = 210**,
  + **a[1] ∘ a[1] = 2 ∘ 2 = 22**.

So the sum is equal to **1010 + 102 + 210 + 22 = 1344**.

* For **a = [8]**, the output should be **solution(a) = 88**.

There is only one number in **a**, and **a[0] ∘ a[0] = 8 ∘ 8 = 88**, so the answer is **88**.

* For **a = [1, 2, 3]**, the output should be **solution(a) = 198**.
  + **a[0] ∘ a[0] = 1 ∘ 1 = 11**,
  + **a[0] ∘ a[1] = 1 ∘ 2 = 12**,
  + **a[0] ∘ a[2] = 1 ∘ 3 = 13**,
  + **a[1] ∘ a[0] = 2 ∘ 1 = 21**,
  + **a[1] ∘ a[1] = 2 ∘ 2 = 22**,
  + **a[1] ∘ a[2] = 2 ∘ 3 = 23**,
  + **a[2] ∘ a[0] = 3 ∘ 1 = 31**,
  + **a[2] ∘ a[1] = 3 ∘ 2 = 32**,
  + **a[2] ∘ a[2] = 3 ∘ 3 = 33**.

The total result is **11 + 12 + 13 + 21 + 22 + 23 + 31 + 32 + 33 = 198**.

Input/Output

* **[execution time limit] 4 seconds (py)**
* **[input] array.integer a**

A non-empty array of positive integers.

*Guaranteed constraints:*  
**1 ≤ a.length ≤ 105**,  
**1 ≤ a[i] ≤ 106**.

* **[output] integer64**

The sum of all **a[i] ∘ a[j]**s. It's guaranteed that the answer is less than **253**.

**[Python 2] Syntax Tips**

You are given two arrays of integers **a** and **b** of the same length, and an integer **k**. We will be iterating through array **a** from left to right, and simultaneously through array **b** from right to left, and looking at pairs **(x, y)**, where **x** is from **a** and **y** is from **b**. Such a pair is called *tiny* if the concatenation **xy** is strictly less than **k**.

Your task is to return the number of *tiny* pairs that you'll encounter during the simultaneous iteration through **a** and **b**.

def solution(numbers):

    sum = 0

    current\_index\_looking\_at = 0

    for i in numbers:

        for j in numbers:

            temp = str(i)+str(j)

            sum += int(temp)

    return sum

Example

* For **a = [1, 2, 3]**, **b = [1, 2, 3]**, and **k = 31**, the output should be  
  **solution(a, b, k) = 2**.

We're considering the following pairs during iteration:

* + **(1, 3)**. Their concatenation equals **13**, which is less than **31**, so the pair is *tiny*;
  + **(2, 2)**. Their concatenation equals **22**, which is less than **31**, so the pair is *tiny*;
  + **(3, 1)**. Their concatenation equals **31**, which is not less than **31**, so the pair is not *tiny*.

As you can see, there are **2** *tiny* pairs during the iteration, so the answer is **2**.

* For **a = [16, 1, 4, 2, 14]**, **b = [7, 11, 2, 0, 15]**, and **k = 743**, the output should be  
  **solution(a, b, k) = 4**.

We're considering the following pairs during iteration:

* + **(16, 15)**. Their concatenation equals **1615**, which is greater than **743**, so the pair is not *tiny*;
  + **(1, 0)**. Their concatenation equals **10**, which is less than **743**, so the pair is *tiny*;
  + **(4, 2)**. Their concatenation equals **42**, which is less than **743**, so the pair is *tiny*.
  + **(2, 11)**. Their concatenation equals **211**, which is less than **743**, so the pair is *tiny*;
  + **(14, 7)**. Their concatenation equals **147**, which is less than **743**, so the pair is *tiny*.

There are **4** *tiny* pairs during the iteration, so the answer is **4**.

Input/Output

* **[execution time limit] 4 seconds (py)**
* **[input] array.integer a**

An array of non-negative integers.

*Guaranteed constraints:*  
**0 ≤ a.length ≤ 105**,  
**0 ≤ a[i] ≤ 104**.

* **[input] array.integer b**

An array of non-negative integers.

*Guaranteed constraints:*  
**b.length = a.length**,  
**0 ≤ b[i] ≤ 104**.

* **[input] integer k**

An integer to compare concatenated pairs with.

*Guaranteed constraints:*  
**0 ≤ k ≤ 109**.

* **[output] integer**

The number of *tiny* pairs during the iteration.

**[Python 2] Syntax Tips**

Let's say a triple **(a, b, c)** is a *zigzag* if either **a < b > c** or **a > b < c**.

Given an array of integers **numbers**, your task is to check all the triples of its consecutive elements for being a *zigzag*. More formally, your task is to construct an array of length **numbers.length - 2**, where the **ith** element of the output array equals **1** if the triple **(numbers[i], numbers[i + 1], numbers[i + 2])** is a *zigzag*, and **0** otherwise.

Example

* For **numbers = [1, 2, 1, 3, 4]**, the output should be **solution(numbers) = [1, 1, 0]**.
  + **(numbers[0], numbers[1], numbers[2]) = (1, 2, 1)** **is** a *zigzag*, because **1 < 2 > 1**;
  + **(numbers[1], numbers[2] , numbers[3]) = (2, 1, 3)** **is** a *zigzag*, because **2 > 1 < 3**;
  + **(numbers[2], numbers[3] , numbers[4]) = (1, 3, 4)** **is not** a *zigzag*, because **1 < 3 < 4**;
* For **numbers = [1, 2, 3, 4]**, the output should be **solution(numbers) = [0, 0]**;

Since all the elements of **numbers** are increasing, there are no *zigzags*.

* For **numbers = [1000000000, 1000000000, 1000000000]**, the output should be **solution(numbers) = [0]**.

Since all the elements of **numbers** are the same, there are no *zigzags*.

function solution(a, b, k) {

    let pairs = 0;

    let arr = [];

    b.reverse()

    for (num in a) {

        for (num in b) {

            result = String(a[num]) + String(b[num])

            if (result < k) {

                if ((arr.findIndex(e => e === result)) === -1) {

                    arr.push(String(result));

                    pairs++

                }

            }

        }

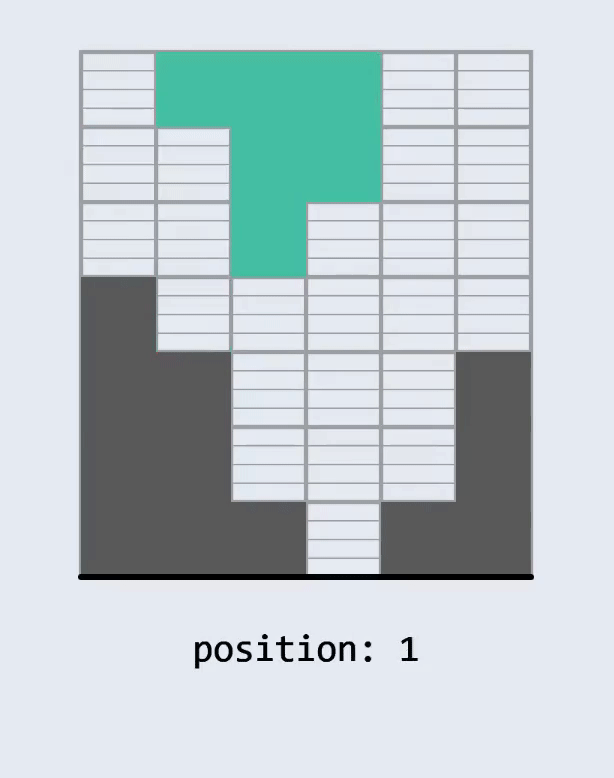
    }

    return pairs

}

You are given a matrix of integers **field** of size **n × m** representing a game field, and also a matrix of integers **figure** of size **3 × 3** representing a figure. Both matrices contain only **0**s and **1**s, where **1** means that the cell is occupied, and **0** means that the cell is free.

You choose a position at the top of the game field where you put the figure and then drop it down. The figure falls down until it either reaches the ground (bottom of the field) or lands on an occupied cell, which blocks it from falling further. After the figure has stopped falling, some of the rows in the field may become fully occupied.



Your task is to find the dropping position such that at least one full row is formed. As a dropping position you should consider the **column index of the cell** in game field which matches the top left corner of the figure **3 × 3** matrix. If there are multiple dropping positions satisfying the condition, feel free to return any of them. If there are no such dropping positions, return **-1**.

*Note: When falling, the****3 × 3****matrix of the figure must be entirely inside the game field, even if the figure matrix is not totally occupied.*

Example

* For
* **field = [[0, 0, 0],**
* **[0, 0, 0],**
* **[0, 0, 0],**
* **[1, 0, 0],**
* **[1, 1, 0]]**

**Explain**

and

**figure = [[0, 0, 1],**

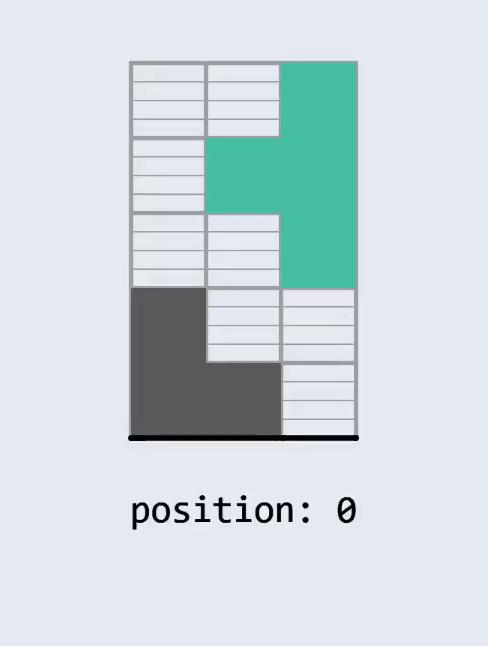
**[0, 1, 1],**

**[0, 0, 1]]**

**Explain**

the output should be **solution(field, figure) = 0**.

The figure can be dropped only from position **0**. When the figure stops falling, two fully occupied rows are formed, so dropping position **0** satisfies the condition.



* For
* **field = [[0, 0, 0, 0, 0],**
* **[0, 0, 0, 0, 0],**
* **[0, 0, 0, 0, 0],**
* **[1, 1, 0, 1, 0],**
* **[1, 0, 1, 0, 1]]**

**Explain**

and

**figure = [[1, 1, 1],**

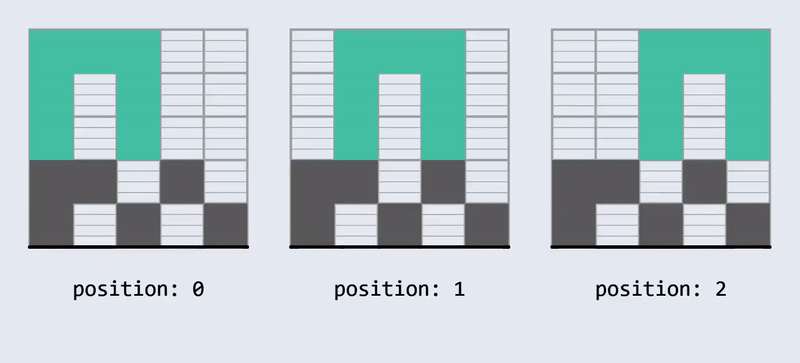
**[1, 0, 1],**

**[1, 0, 1]]**

**Explain**

the output should be **solution(field, figure) = 2**.

The figure can be dropped from three positions - **0**, **1**, and **2**. As you can see below, a fully occupied row will be formed only when dropping from position **2**:



* For
* **field = [[0, 0, 0, 0],**
* **[0, 0, 0, 0],**
* **[0, 0, 0, 0],**
* **[1, 0, 0, 1],**
* **[1, 1, 0, 1]]**

**Explain**

and

**figure = [[1, 1, 0],**

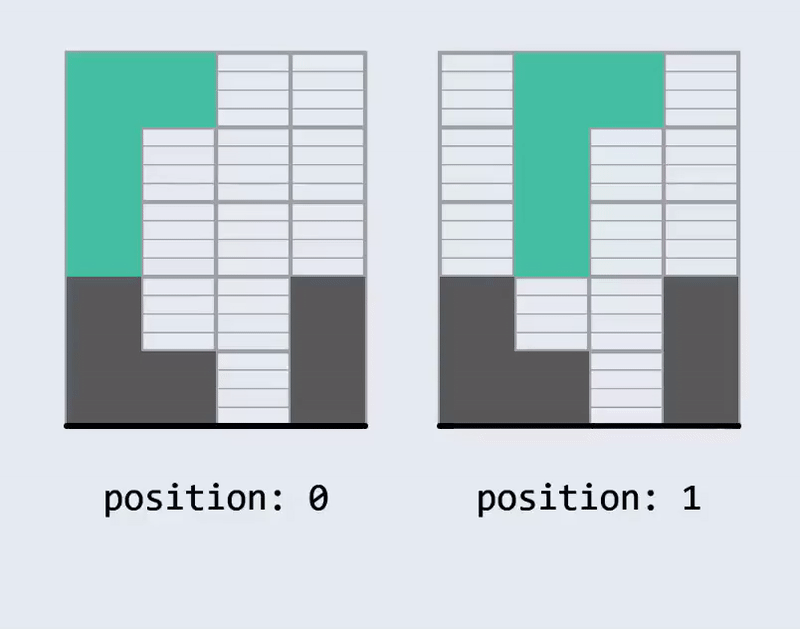
**[1, 0, 0],**

**[1, 0, 0]]**

**Explain**

the output should be **solution(field, figure) = -1**.

The figure can be dropped from two positions - **0** and **1**, and in both cases, a fully occupied line won't be obtained:

  
Note that the figure could technically form a full row if it was dropped one position further to the right, but in that case the figure matrix wouldn't be fully contained with the field.

Input/Output

* **[execution time limit] 4 seconds (py)**
* **[input] array.array.integer field**

A matrix of integers representing the game field. It's guaranteed that at the beginning there are no fully occupied rows on the game field and that the top three rows are fully free.

*Guaranteed constraints:*  
**4 ≤ field.length ≤ 100**,  
**3 ≤ field[i].length ≤ 100**,  
**0 ≤ field[i][j] ≤ 1**.

* **[input] array.array.integer figure**

A matrix of integers representing the figure. It's guaranteed that the figure's occupied cells are pairwise connected and there is at least one occupied cell at the bottom row of the figure.

*Guaranteed constraints:*  
**figure.length = 3**,  
**figure[i].length = 3**,  
**0 ≤ figure[i][j] ≤ 1**.

* **[output] integer**

The dropping position such that a full row is formed. If there are multiple possible positions, return any of them. If there is no such position, return **-1**.

**[Python 2] Syntax Tips**

def solution(field, figure):

   height = len(field)

   width = len(field[0])

   figure\_size = len(figure)

   for column in range(width - figure\_size + 1):

       row = 1

       while row < height - figure\_size + 1:

           can\_fit = True

           for dx in range(figure\_size):

               for dy in range(figure\_size):

                   if field[row + dx][column + dy] == 1 and figure[dx][dy] == 1:

                       can\_fit = False

           if not can\_fit:

               break

           row += 1

       row -= 1

       for dx in range(figure\_size):

           row\_filled = True

           for column\_index in range(width):

            if not (field[row + dx][column\_index] == 1 or

                    (column <= column\_index < column + figure\_size and\

                  figure[dx][column\_index - column] == 1)):

                row\_filled = False

           if row\_filled:

               return column

   return -1