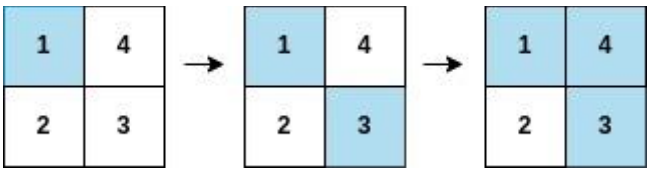
1. **First Completely Painted Row or Column**

You are given a 0-indexed integer array arr, and an m x n integer matrix mat. arr and mat both contain all the integers in the range [1, m * n].

Go through each index i in arr starting from index 0 and paint the cell in mat containing the integer arr[i].

Return the smallest index i at which either a row or a column will be completely painted in mat.

Example 1:

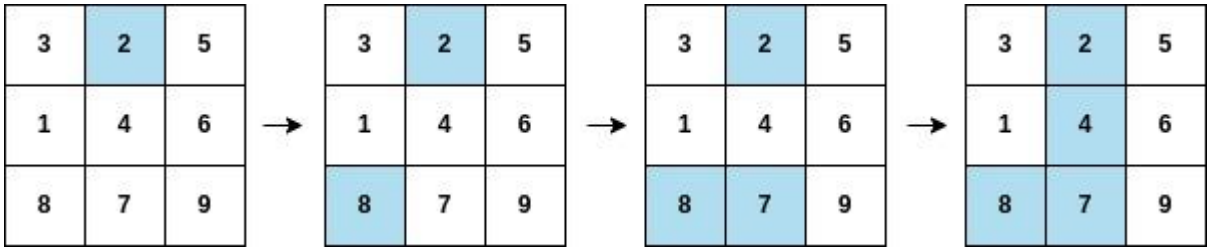


Input: arr = [1,3,4,2], mat = [[1,4],[2,3]]

Output: 2

Explanation: The moves are shown in order, and both the first row and second column of the matrix become fully painted at arr[2].

Example 2:



Input: arr = [2,8,7,4,1,3,5,6,9], mat = [[3,2,5],[1,4,6],[8,7,9]]

Output: 3

Explanation: The second column becomes fully painted at arr[3].

Constraints:

m == mat.length

n = mat[i].length

arr.length == m * n

1 <= m, n <= 105

1 <= m * n <= 105

1 <= arr[i], mat[r][c] <= m * n

All the integers of arr are unique.

All the integers of mat are unique.

Solution:

Imagine you have a grid (a table) with rows and columns, like this:

```
1 2 3
4 5 6
7 8 9
```

Each box in the grid has a number, and you also have a list of numbers, like this:

```
arr = [4, 2, 8, 1, 7, 3, 6, 5, 9]
```

The Game:

1. One by one, you take a number from the list arr.
2. You look at the grid and "paint" the box that has that number. Painting means marking it as done.
3. Your goal is to find out when **one full row** or **one full column** of the grid is completely painted.

How It Works (with an Example):

Let’s start painting step by step.

Step 1: Take the first number in arr, which is 4.
Look at the grid. The number 4 is here:

```
1 2 3
[4] 5 6
7 8 9
```

Now paint it. The grid looks like this:

```
1 2 3
[X] 5 6
7 8 9
```

No row or column is fully painted yet.

Step 2: Take the next number in arr, which is 2.
Paint the box with 2:

1 [X] 3
[X] 5 6
7 8 9

Still, no row or column is fully painted.

Step 3: Take the next number in arr, which is 8.
Paint the box with 8:

1 [X] 3
[X] 5 6
7 [X] 9

Still, no row or column is fully painted.

Step 4: Take the next number in arr, which is 1.
Paint the box with 1:

[X] [X] 3
[X] 5 6
7 [X] 9

Still, no row or column is fully painted.

Step 5: Take the next number in arr, which is 7.
Paint the box with 7:

[X] [X] 3
[X] 5 6
[X] [X] 9

Now, **the first column** (the first boxes in every row) is fully painted. □

The Answer:

The smallest index in arr where this happens is 4 (because 7 is the 5th number in arr, and we start counting from 0).

So, the answer is **4**.

Why Is This Fun?

It's like a puzzle where you keep marking boxes and try to be the first to paint a full row or column. The challenge is to find out when that happens!

```
public class FirstCompletelyPainted {
    public static int firstCompleteIndex(int[] arr, int[][] mat) {
        int m = mat.length;      // Number of rows
        int n = mat[0].length;   // Number of columns
        // Arrays to track how many cells are painted in each row and column
        int[] rowCount = new int[m];
        int[] colCount = new int[n];
        // Find the position of each number in the matrix
        int[][] positions = new int[m * n + 1][2];
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                positions[mat[i][j]] = new int[]{i, j};
            }
        }
        // Go through each number in arr
        for (int i = 0; i < arr.length; i++) {
            int row = positions[arr[i]][0];
            int col = positions[arr[i]][1];
            // Paint the row and column
            rowCount[row]++;
            colCount[col]++;
            // Check if a row or column is completely painted
            if (rowCount[row] == n || colCount[col] == m) {
                return i; // Return the index
            }
```

```
        }
        return -1; // No row or column is fully painted
    }
    public static void main(String[] args) {
        int[] arr = {4, 2, 8, 1, 7, 3, 6, 5, 9};
        int[][] mat = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };
        int result = firstCompleteIndex(arr, mat);
        System.out.println("The first completely painted row or column is at index: " + result);
    }
}
```

Couples Holding Hands

2. **There are n couples sitting in 2n seats arranged in a row and want to hold hands.**

The people and seats are represented by an integer array row where row[i] is the ID of the person sitting in the ith seat. The couples are numbered in order, the first couple being (0, 1), the second couple being (2, 3), and so on with the last couple being (2n - 2, 2n - 1).

Return the minimum number of swaps so that every couple is sitting side by side. A swap consists of choosing any two people, then they stand up and switch seats.

Example 1:

Input: row = [0,2,1,3]

Output: 1

Explanation: We only need to swap the second (row[1]) and third (row[2]) person.

Example 2:

Input: row = [3,2,0,1]

Output: 0

Explanation: All couples are already seated side by side.

Constraints:

2n == row.length

2 <= n <= 30

n is even.

0 <= row[i] < 2n

All the elements of row are unique.

Let’s break it down in a fun and simple way, step by step!

Imagine This Scenario

- You have **n couples** sitting in a row.
- Each person has an **ID** (a number) written on their T-shirt.
- Couples are numbered as:
    - Couple 0: IDs 0 and 1
    - Couple 1: IDs 2 and 3
    - Couple 2: IDs 4 and 5, and so on.

Now, the people are sitting in a row, but not all couples are sitting next to each other. Your job is to make sure that **all couples sit side by side** by swapping their positions.

The Rules

1. A **couple** is defined as two numbers that are consecutive (like 0 and 1, 2 and 3, etc.).
2. You can only swap two people at a time.
3. The goal is to find the **smallest number of swaps** to fix the seating so all couples sit together.

Example

Let’s say we have $n = 3$ couples, and the seating is like this:

row = [3, 1, 4, 0, 2, 5]

Here’s what’s happening:

- Seat 0: Person 3
- Seat 1: Person 1
- Seat 2: Person 4
- Seat 3: Person 0
- Seat 4: Person 2
- Seat 5: Person 5

Step-by-Step Fixing

1. **Look at the first pair (Seats 0 and 1)**:
    - People sitting here: 3 and 1.
    - Couple for 3 is 2, but 2 is not sitting next to 3. They need to swap places!

    Swap **Person 1** (Seat 1) with **Person 2** (Seat 4).

    After the swap:

    row = [3, 2, 4, 0, 1, 5]

2. **Move to the next pair (Seats 2 and 3)**:
    - People sitting here: 4 and 0.
    - Couple for 4 is 5, but 5 is not sitting next to 4. They need to swap places!

    Swap **Person 0** (Seat 3) with **Person 5** (Seat 5).

    After the swap:

    row = [3, 2, 4, 5, 1, 0]

3. **Move to the next pair (Seats 4 and 5)**:
    - People sitting here: 1 and 0.
    - Couple for 1 is 0, and they are already sitting together! No need to swap.

Result

Now all couples are sitting together:

- Couple 0: 3 and 2
- Couple 1: 4 and 5
- Couple 2: 1 and 0

Answer

It took **2 swaps** to fix the seating.

Why Is This Fun?

It's like playing a puzzle! You're trying to minimize the number of swaps while making sure everyone is happy sitting next to their partner.

Solution:

```
public class CouplesHoldingHands {

    public static int minSwapsCouples(int[] row) {
        int swaps = 0;

        for (int i = 0; i < row.length; i += 2) {
            int firstPerson = row[i];
            int expectedPartner = (firstPerson % 2 == 0) ? firstPerson + 1 : firstPerson - 1;

            if (row[i + 1] != expectedPartner) {
                swaps++;

                // Find and swap the expected partner
                for (int j = i + 1; j < row.length; j++) {
                    if (row[j] == expectedPartner) {
                        // Swap row[j] with row[i + 1]
                        int temp = row[j];
                        row[j] = row[i + 1];
                        row[i + 1] = temp;
                        break;
                    }
                }
            }
        }

        return swaps;
    }

    public static void main(String[] args) {
        int[] row = {3, 1, 4, 0, 2, 5};
        int result = minSwapsCouples(row);
        System.out.println("Minimum swaps needed: " + result);
    }
}
```

How It Works:

1. Iterate through the row two seats at a time (`i` and `i+1`).
2. Check if the second person (`row[i+1]`) is the expected partner of the first person (`row[i]`).
3. If not:
    - Find the expected partner in the row.
    - Swap the expected partner with the person sitting next to the first person.
4. Count each swap.

This is straightforward and easy to understand!