2943. Maximize Area of Square Hole in Grid

Medium <u>Array</u> <u>Sorting</u>

## **Problem Description**

removing certain bars. The grid consists of n + 2 horizontal bars and m + 2 vertical bars, forming  $1 \times 1$  unit cells. The bars are 1-indexed for reference. To create a hole, we have the option of removing horizontal bars at positions listed in hears, and vertical bars at positions listed

In this LeetCode problem, we're tasked with finding the largest possible square-shaped hole that can be created in a grid by

in vBars. Each array contains distinct positions, and they fall within certain ranges—hBars values are within [2, n + 1], and vBars within [2, m + 1]. Our goal is to return the maximum area of the square hole. To visualize this, imagine removing consecutive bars to create the

largest possible square void in the grid's structure.

Intuition

square hole need one more unit space to be formed).

makes it easy to count them in the next step.

start counting a new sequence from this point.

## bars we can remove either horizontally or vertically. Therefore, the key to this problem is figuring out the longest sequence of

consecutive bar numbers in the hars and vars that we're permitted to take out. Let's consider any sequence of removable bars. The longest consecutive run of bar numbers in either direction will ultimately determine the maximum size of the square-shaped hole we can achieve. For instance, if we can remove three consecutive

horizontal bars and two consecutive vertical bars, the largest square hole we can create is 2 x 2, since the square cannot

Firstly, we need to understand that the maximum square hole we can make in the grid is limited by the number of consecutive

exceed the smaller side. The solution approach here is quite clever. We sort each bar array to make it easier to find consecutive sequences. Then, using a helper function f(nums), we traverse through the sorted array and count the length of the longest increasing consecutive subsequence. This function returns the longest length plus one (to account for the extra space at the end of the last bar, as a

After calculating this for both hars and vars, the side length of the largest possible square hole is the minimum of these two lengths. Since square area is equal to the side length squared, we finally return the side length squared (calculated minimum) as the answer. Solution Approach

The implementation relies on a straightforward but effective sorting and counting approach. Here's a step-by-step breakdown:

Sort both arrays, hars and vars. Sorting brings all consecutive sequences into adjacent positions in the array, which

## Define a helper function f(nums) which takes a sorted list of bar positions. Initialize two variables: ans and cnt to 1. Here,

ans will store the length of the longest consecutive sequence we find, and cnt is a counter for the current consecutive sequence.

Loop through the nums list starting from the second element (indexed at 1), and compare each element with its predecessor.

and vertical bars.

required output.

**Example Walkthrough** 

• n + 2 = 5 horizontal bars (indexed from 1 to 5)

Step 1: Sort both arrays, hBars = [3, 4], vBars = [2, 4].

Step 3: Let's apply function f(nums) on hBars:

• cnt starts at 1, and ans starts at 1.

• The loop ends with ans still equal to 1.

bars, which allow for a side length of 2.

for i in range(1, len(nums)):

**if** nums[i] == nums[i - 1] + 1:

return maxConsecutiveBars \* maxConsecutiveBars;

// Sort the array to easily find consecutive numbers

private int findMaxConsecutiveBars(int[] bars) {

for (int i = 1; i < bars.length; ++i) {</pre>

if (bars[i] == bars[i - 1] + 1) {

Arrays.sort(bars):

int maxConsecutive = 1;

} else {

**}**;

**}**;

**TypeScript** 

consecutive = 1;

return minSideLength \* minSideLength;

// by removing certain horizontal and vertical bars.

// hBars: List of positions of removable horizontal bars

// vBars: List of positions of removable vertical bars

for (let i = 1; i < bars.length; ++i) {</pre>

consecutiveCount = 1;

for i in range(1, len(nums)):

else:

return max\_hole\_size \*\* 2

Time and Space Complexity

if nums[i] == nums[i - 1] + 1:

# we square the maximum hole size to get the answer.

if (bars[i] === bars[i - 1] + 1) {

// n: The total number of horizontal bars

bars.sort((a, b) => a - b);

} else {

return maxSize + 1;

// m: The total number of vertical bars

// Reset count if not consecutive

int maxHorizontal = findLargestSquare(horizontalBars);

int maxVertical = findLargestSquare(verticalBars);

int minSideLength = min(maxHorizontal, maxVertical);

// Calculate and return the area of the largest square hole

// Function to find the maximum size for a square hole that can be formed

// Helper function to calculate the maximum number of consecutive bars

// Iterate over the bars to find the maximum consecutive sequence

// If the current bar is consecutive to the previous one

const getMaxConsecutiveBars = (bars: number[]): number => {

// Sort the bars array to count consecutive numbers

consecutiveCount++; // Increase count

// Return the size including the space without bars

const horizontalMaxSize = getMaxConsecutiveBars(hBars);

const verticalMaxSize = getMaxConsecutiveBars(vBars);

// Reset count when no longer consecutive

// Calculate the maximum size for both horizontal and vertical bars

def find largest consecutive sequence(nums: List[int]) -> int:

nums.sort() # Sort the array to find consecutive numbers.

# by finding the smallest of the two maximum consecutive sequences.

let maxSize = 1; // Initialize maximum size to 1

// Call the lambda function for both horizontal and vertical bars

return largestSquareSide + 1; // Add 1 to get the side length of the hole

// Find the minimum of max horizontal and vertical squares to form a square hole

function maximizeSquareHoleArea(n: number, m: number, hBars: number[], vBars: number[]): number {

let consecutiveCount = 1; // Count of consecutive bars, start with 1 as we count the first bar

maxSize = Math.max(maxSize, consecutiveCount); // Update the max size found

def maximizeSquareHoleArea(self, n: int, m: int, hBars: List[int], vBars: List[int]) -> int:

# Calculate the maximum square hole size for both horizontal and vertical bars

# Since the problem is about finding the area of the largest square hole,

max sequence length = current\_sequence\_length = 1 # Initialize counters for sequences.

max\_sequence\_length = max(max\_sequence\_length, current\_sequence\_length)

current sequence length += 1 # Increase the sequence length if consecutive.

current sequence length = 1 # Reset the sequence length if not consecutive.

max\_hole\_size = min(find\_largest\_consecutive\_sequence(hBars), find\_largest\_consecutive\_sequence(vBars))

return max\_sequence\_length + 1 # Add 1 to include the space on both ends of the sequence.

int currentConsecutiveCount = 1;

Solution Implementation

**Python** 

Java

class Solution {

class Solution:

If the current element is exactly one more than the previous (nums[i] == nums[i - 1] + 1), it means that we've found consecutive bars. Increment cnt to extend the current consecutive sequence. Update ans to hold the maximum value between ans and the current cnt.

If the current element is not consecutive, reset cnt to 1 because we've hit a break in the sequence, and we'll need to

- When the loop finishes, add 1 to the final ans. This is because the square hole size is actually one unit larger than the number of bars removed (the last bar removed implies there is one extra unit of open space that completes the square). Apply the helper function to both hars and vars to get the size of the longest consecutive sequence for both horizontal
- be a square, therefore, it is limited by the shorter side. Calculate the area of the largest square hole by squaring the side length found in the previous step. The square of the

The side length of the largest square hole we can create is the minimum of these two sizes. This is because the shape has to

minimum consecutive sequence length provides us with the maximum square hole area after removing the bars, which is the

maximizeSquareHoleArea and one helper function f. The use of sorting and a single pass counting mechanism ensures an efficient solution, with the overall time complexity being dominated by the sorting step, which is 0(n log n) for each array.

To illustrate the solution approach, let's walk through a small example. Suppose we have the following input:

Using the steps from the approach above, the code neatly encapsulates the logic within one main function

 m + 2 = 5 vertical bars (indexed from 1 to 5) • hBars = [3, 4]• vBars = [2, 4]

## • We start with ans = 1 and cnt = 1. Loop through hBars, starting from the second element:

Step 5: Repeat steps 3 and 4 for vBars:

• Comparing vBars[1] to vBars[0], we see that 4 != 2 + 1, so these are not consecutive.

Compare hBars[1] to hBars[0]. Since 4 == 3 + 1, they are consecutive, so increase cnt to 2.

Since we are at the end of the array, we finish the loop with ans being the value of cnt, which is 2.

Step 4: Add 1 to the final ans, yielding 2 + 1 = 3. This means horizontally, we can fit a square hole of side length 3.

The arrays are already sorted in this example, so we can proceed directly to the next step.

Step 2: We will use a helper function f(nums) to find the length of the longest consecutive sequence.

Step 6: Add 1 to the final ans, yielding 1 + 1 = 2. This means vertically, the largest square hole we can form has a side length of 2.

def maximizeSquareHoleArea(self, n: int, m: int, hBars: List[int], vBars: List[int]) -> int:

# Calculate the maximum square hole size for both horizontal and vertical bars

public int maximizeSquareHoleArea(int n, int m, int[] horizontalBars, int[] verticalBars) {

// Iterate through the sorted array to find the maximum set of consecutive numbers

// If the current bar is consecutive to the previous one, increase the count

// Find the maximum consecutive bars for both the horizontal and vertical arrays

// Helper method to find the length of the maximum set of consecutive bars

def find largest consecutive sequence(nums: List[int]) -> int:

nums.sort() # Sort the array to find consecutive numbers.

Step 8: The area of the square is the side length squared, so in this example, the largest square hole we can create has an area of  $2 \times 2 = 4$ .

Following this approach, the maximizeSquareHoleArea function would return the value 4 as the answer for this example.

max\_sequence\_length = max(max\_sequence\_length, current\_sequence\_length) else: current sequence length = 1 + Reset the sequence length if not consecutive. return max\_sequence\_length + 1 # Add 1 to include the space on both ends of the sequence.

int maxConsecutiveBars = Math.min(findMaxConsecutiveBars(horizontalBars), findMaxConsecutiveBars(verticalBars));

# Define a helper function to find the largest square hole along one direction (either horizontally or vertically).

Step 7: The comparison of the two sizes (3 and 2) shows that the maximum square hole we can create is limited by the vertical

# by finding the smallest of the two maximum consecutive sequences. max\_hole\_size = min(find\_largest\_consecutive\_sequence(hBars), find\_largest\_consecutive\_sequence(vBars)) # Since the problem is about finding the area of the largest square hole, # we square the maximum hole size to get the answer. return max\_hole\_size \*\* 2

// The area of the largest square hole is the square of the number of maximum consecutive bars

// Initialize variables to store the current count of consecutive numbers and the maximum found so far

max sequence length = current\_sequence\_length = 1 # Initialize counters for sequences.

current sequence length += 1 # Increase the sequence length if consecutive.

```
currentConsecutiveCount++;
               // Update maxConsecutive with the maximum value found so far
               maxConsecutive = Math.max(maxConsecutive, currentConsecutiveCount);
           } else {
               // Reset the count if the current bar is not consecutive
               currentConsecutiveCount = 1;
       // Return the maximum consecutive count plus one (to account for the space between bars)
       return maxConsecutive + 1;
#include <vector>
#include <algorithm>
using namespace std;
class Solution {
public:
   // Method to maximize the square hole area given horizontal and vertical bars
   int maximizeSquareHoleArea(int n, int m, vector<int>& horizontalBars, vector<int>& verticalBars) {
       // Lambda function to calculate the largest square hole from a sequence of bars
       auto findLargestSquare = [](vector<int>& bars) {
            int largestSquareSide = 1; // Initial largest square side length
            int consecutive = 1:
                                   // Count of consecutive bars
            sort(bars.begin(), bars.end()); // Sort the bars to find consecutive numbers
            // Iterate through the sorted bars to find the max count of consecutive numbers
            for (int i = 1; i < bars.size(); ++i) {</pre>
               if (bars[i] == bars[i - 1] + 1) {
                    // If consecutive, increment count
                    largestSquareSide = max(largestSquareSide, ++consecutive);
```

```
// The size of the square hole will be the minimum of the two sizes squared
// because the hole must be square, and its sides depend on the minimum of horizontal and vertical spacing
return Math.min(horizontalMaxSize, verticalMaxSize) ** 2;
```

class Solution:

**}**;

The function maximizeSquareHoleArea includes two calls to the helper function f, once for hBars and once for vBars. The complexity analysis will be the same for both calls since the operations performed by f are identical for both lists. **Time Complexity** 

The time complexity is dominated by the sorting operation inside the function f. Since Python's sort method typically uses

Timsort, which has an average and worst-case time complexity of O(n log n) where n is the length of the list being sorted. The

# Define a helper function to find the largest square hole along one direction (either horizontally or vertically).

subsequent loop runs in linear time, O(n), since it iterates through the sorted list once.

Therefore, the time complexity of the function f is  $O(n \log n) + O(n)$  which simplifies to  $O(n \log n)$  because the n log n term dominates. Since we call the function f twice, once for hars and once for vars, and assuming n refers to the longer of the two lists for

**Space Complexity** The space complexity of the function f is 0(n) because of the space required to sort the list. The sort operation may require

worst-case analysis, then the overall time complexity remains 0(n log n) since these two calls are sequential, not nested.

with input size.

additional space to hold elements temporarily during the sort. The variables ans and cnt use constant space and do not scale Overall, since we consider the larger of the two lists hars and vars, the total space complexity of the maximizeSquareHoleArea function is O(n), where n is the length of the larger input list.