2943. Maximize Area of Square Hole in Grid

Medium Array Sorting

Problem Description

In this LeetCode problem, we're tasked with finding the largest possible square-shaped hole that can be created in a grid by removing certain bars. The grid consists of `n + 2` horizontal bars and `m + 2` vertical bars, forming `1 x 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 `hBars`, and vertical bars at positions listed 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

Firstly, we need to understand that the maximum square hole we can make in the grid is limited by the number of consecutive 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 `hBars` and `vBars` 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 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 square hole need one more unit space to be formed).

After calculating this for both `hBars` and `vBars`, 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:

1. Sort both arrays, `hBars` and `vBars`. Sorting brings all consecutive sequences into adjacent positions in the array, which makes it easy to count them in the next step.
2. 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.
3. Loop through the `nums` list starting from the second element (indexed at `1`), and compare each element with its predecessor.
   - 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 start counting a new sequence from this point.
4. 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).
5. Apply the helper function to both `hBars` and `vBars` to get the size of the longest consecutive sequence for both horizontal and vertical bars.
6. 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 be a square, therefore, it is limited by the shorter side.
7. Calculate the area of the largest square hole by squaring the side length found in the previous step. The square of the minimum consecutive sequence length provides us with the maximum square hole area after removing the bars, which is the required output.

Using the steps from the approach above, the code neatly encapsulates the logic within one main function `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 `O(n log n)` for each array.

Example Walkthrough

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

- `n = 2` = 5 horizontal bars (indexed from 1 to 5)
- `m = 2` = 5 vertical bars (indexed from 1 to 5)
- `hBars` = [3, 4]
- `vBars` = [2, 4]

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

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

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

Step 5: Repeat steps 3 and 4 for `vBars`:

- `cnt` starts at `1`, and `ans` starts at `1`.
- Comparing `vBars[1]` to `vBars[0]`, we see that `4 != 2 + 1`, so these are not consecutive.
- The bounds with `ans` still equal to `1`.

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.

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 bars, which allow for a side length of 2.

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 x 2 = 4`.

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

Solution Implementation

Python

```python
class Solution:
    def maximizeSquareHoleArea(self, n: int, m: int, hBars: List[int], vBars: List[int]) -> int:
        # Define a helper function to find the largest square hole along one direction (either horizontally or vertically).
        def find_largest_consecutive_sequence(nums: List[int]) -> int:
            nums.sort() # Sort the array to find consecutive numbers.
            max_sequence_length = current_sequence_length = 1 # Initialize counters for sequences.
            for i in range(1, len(nums)):
                if nums[i] == nums[i - 1] + 1:
                    current_sequence_length += 1 # Increase the sequence length if consecutive.
                    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.

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

Java

```java
class Solution {
    public int maximizeSquareHoleArea(int n, int m, int[] horizontalBars, int[] verticalBars) {
        // Find the maximum consecutive bars for both the horizontal and vertical arrays
        int maxConsecutiveBars = Math.min(findMaxConsecutiveBars(horizontalBars), findMaxConsecutiveBars(verticalBars));
        // The area of the largest square hole is the square of the number of maximum consecutive bars
        return maxConsecutiveBars * maxConsecutiveBars;
    }

    // Helper method to find the length of the maximum set of consecutive bars
    private int findMaxConsecutiveBars(int[] bars) {
        // Sort the array to easily find consecutive numbers
        Arrays.sort(bars);
        // Initialize variables to store the current count of consecutive numbers and the maximum found so far
        int maxConsecutive = 1;
        int currentConsecutiveCount = 1;

        // Iterate through the sorted array to find the maximum set of consecutive numbers
        for (int i = 1; i < bars.length; ++i) {
            // If the current bar is consecutive to the previous one, increase the count
            if (bars[i] == bars[i - 1] + 1) {
                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;
    }
}
```

C++

```cpp
#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 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) {
                if (bars[i] == bars[i - 1] + 1) {
                    // If consecutive, increment count
                    largestSquareSide = max(largestSquareSide, ++consecutive);
                } else {
                    // Reset count if not consecutive
                    consecutive = 1;
                }
            }
            return largestSquareSide + 1; // Add 1 to get the side length of the hole
        };

        // Call the lambda function for both horizontal and vertical bars
        int maxHorizontal = findLargestSquare(horizontalBars);
        int maxVertical = findLargestSquare(verticalBars);

        // Find the minimum of max horizontal and vertical squares to form a square hole
        int minSideLength = min(maxHorizontal, maxVertical);

        // Calculate and return the area of the largest square hole
        return minSideLength * minSideLength;
    }
};
```

TypeScript

```typescript
// Function to find the maximum size for a square hole that can be formed
// by removing certain horizontal and vertical bars.
// n: The total number of horizontal bars
// m: The total number of vertical bars
// hBars: List of positions of removable horizontal bars
// vBars: List of positions of removable vertical bars
function maximizeSquareHoleArea(n: number, m: number, hBars: number[], vBars: number[]): number {
    // Helper function to calculate the maximum number of consecutive bars
    const getMaxConsecutiveBars = (bars: number[]): number => {
        // Sort the bars array to count consecutive numbers
        bars.sort((a, b) => a - b);
        let maxSize = 1; // Initialize maximum size to 1
        let consecutiveCount = 1; // Count of consecutive bars, start with 1 as we count the first bar

        // Iterate over the bars to find the maximum consecutive sequence
        for (let i = 1; i < bars.length; ++i) {
            // If the current bar is consecutive to the previous one
            if (bars[i] === bars[i - 1] + 1) {
                consecutiveCount++; // Increase count
                maxSize = Math.max(maxSize, consecutiveCount); // Update the max size found
            } else {
                // Reset count when no longer consecutive
                consecutiveCount = 1;
            }
        }

        // Return the size including the space without bars
        return maxSize + 1;
    };

    // Calculate the maximum size for both horizontal and vertical bars
    const horizontalMaxSize = getMaxConsecutiveBars(hBars);
    const verticalMaxSize = getMaxConsecutiveBars(vBars);

    // 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;
}
```

```python
class Solution:
    def maximizeSquareHoleArea(self, n: int, m: int, hBars: List[int], vBars: List[int]) -> int:
        # Define a helper function to find the largest square hole along one direction (either horizontally or vertically).
        def find_largest_consecutive_sequence(nums: List[int]) -> int:
            nums.sort() # Sort the array to find consecutive numbers.
            max_sequence_length = current_sequence_length = 1 # Initialize counters for sequences.
            for i in range(1, len(nums)):
                if nums[i] == nums[i - 1] + 1:
                    current_sequence_length += 1 # Increase the sequence length if consecutive.
                    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.

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

Time and Space Complexity

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 subsequent loop runs in linear time, `O(n)`, once 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 `hBars` and once for `vBars`, and assuming `n` refers to the longer of the two lists for worst-case analysis, then the overall time complexity remains `O(n log n)` since these two calls are sequential, not nested.

Space Complexity

The space complexity of the function `f` is `O(n)` because of the space required to sort the list. The sort operation may require additional space to hold elements temporarily during the sort. The variables `ans` and `cnt` use constant space and do not scale with input size.

Overall, since we consider the larger of the two lists `hBars` and `vBars`, the total space complexity of the `maximizeSquareHoleArea` function is `O(n)`, where `n` is the length of the larger input list.