1725. Number Of Rectangles That Can Form The Largest Square

Easy Array [Leetcode Link]

Problem Description

In the given problem, we are presented with an array named `rectangles`, where each element `rectangles[i]` represents the dimensions of the `i`th rectangle as a pair `[l_i, w_i]` indicating its length `l_i` and width `w_i`.

We are allowed to make cuts to these rectangles in order to form squares. A square can be formed from a rectangle if its side length `k` is less than or equal to both the length and the width of the rectangle. For instance, from a rectangle `[4,6]`, we can cut out a square with a maximum side length of `4`.

The objective is to find the side length of the largest square, `maxLen`, that can be formed from these rectangles. Then we need to count the number of rectangles in the array that are capable of forming a square with this maximum side length.

To summarize, the task is to first determine the largest possible square side length that can be cut out from any of the rectangles, and then to count how many rectangles in the list are large enough to provide a square of that size.

Intuition

The solution involves a straightforward approach. Here's the process to understand how the solution works:

1. Initialize two variables `ans` and `mx` to keep track of the count of rectangles that can produce the largest square, and to store the maximum square side length, respectively.
2. Traverse through each rectangle in the `rectangles` array and for each rectangle, determine the side length of the largest possible square, which would be the minimum of the length and width of the rectangle as we are bound by the lesser dimension. This value is stored in a temporary variable `t`.
3. If we find a square side length `t` greater than the current `mx`, we know that we have found a new, larger square side length. So, we update `mx` to this new maximum side length `t`, and reset `ans` to `1` since this is the first rectangle that can form a square of side length `t`.
4. If `t` is equal to `mx`, it means the current rectangle can also form a square of the largest side length found so far. Therefore, we increment `ans` by `1` to reflect this additional rectangle.
5. If `t` is less than `mx`, we do nothing, as the rectangle cannot produce a square of side length `maxLen`.
6. Once all rectangles have been processed, `ans` will contain the final count of rectangles that can produce squares of side length `mx`, and we return `ans`.

This solution works well because it efficiently iterates through the list only once, determines the maximum square side length in the process, and simultaneously counts the qualifying rectangles.

Solution Approach

The solution takes advantage of simple mathematical concepts and logical comparison within a single-pass for-loop to achieve the desired result.

Here's a step-by-step breakdown of the implementation details with reference to the solution code:

1. Initialize two variables, `ans` to store the number of rectangles that can form the largest square, initially set to 0, and `mx` to store the maximum square side length possible, also initially set to 0.

```
ans = mx = 0
```

2. We begin iterating over each rectangle provided in the `rectangles` list. Within the loop, we're performing the following steps for each rectangle:

```
for l, w in rectangles:
```

3. Calculate the largest possible square side length `t` from the current rectangle by taking the minimum of its length `l` and width `w`. This relies on the constraint that a square's side length must be less than or equal to the lengths of both sides of the rectangle.

```
t = min(l, w)
```

4. Compare the calculated square side length `t` with the maximum found so far, `mx`. If `t` is greater, we've discovered a new maximum square size. We then update `mx` to this new value, and set `ans` to 1 since this is the first occurrence of such a large square.

```
if mx < t:
    mx, ans = t, 1
```

5. If `t` equals the current maximum `mx`, it means another rectangle was found which can form a square of the same maximum size. Hence we increment `ans` by 1 without changing `mx`.

```
elif mx == t:
    ans += 1
```

6. If `t` is less than `mx`, that indicates the current rectangle cannot contribute to a square of side length `mx`, so no operation is performed.
7. After the loop completes, the variable `ans` contains the count of rectangles which can form the largest square, and this value is returned.

```
return ans
```

In terms of algorithms and data structures, this solution is a straight-forward linear scan (O(n) complexity) without the need for any additional complex data structures. The use of basic variables and a single for-loop makes the code efficient and easy to understand. This elegant simplicity arises from the observation that we're interested only in the counts related to the maximum square size which can be maintained and updated on-the-fly as we inspect each rectangle.

Example Walkthrough

Let's consider the following array of rectangles: `rectangles = [[3, 5], [6, 6], [5, 2], [4, 4]]`. We need to determine the largest square side length we can cut from any rectangle and then count how many rectangles can form a square of that size.

1. We initialize `ans` and `mx` to `0`. These will keep track of the count and size of the largest square, respectively.
2. We begin iterating over each rectangle:
   - For the first rectangle `[3, 5]`, the largest square side length `t` we can cut is `min(3, 5) = 3`. Since `3` is greater than the current `mx` (0), we update `mx` to `3` and set `ans` to `1`.
   - Moving on to the second rectangle `[6, 6]`, `t = min(6, 6) = 6`. This is greater than the current `mx` (3), so `mx` is updated to `6` and `ans` is reset to `1`.
   - For the third rectangle `[5, 2]`, `t = min(5, 2) = 2`. This is less than the current `mx` (6), so no changes are made to `mx` or `ans`.
   - The last rectangle `[4, 4]` also provides a square with `t = min(4, 4) = 4`, which is still less than our `mx` (6), so again no changes are made to `mx` or `ans`.
3. After completing the loop, `mx` is `6` and `ans` is `1`, since only one rectangle, `[6, 6]`, could produce a square with the largest possible side length. Hence, the function would return `1`.

This example demonstrates the straightforward linear scan through the array and the simple tracking of the maximum square side length and the count of rectangles that could form a square of that maximum size.

Python Solution

```python
from typing import List

class Solution:
    def countGoodRectangles(self, rectangles: List[List[int]]) -> int:
        # Initialize maximum square length and count of good rectangles
        max_square_length = 0
        good_rectangles_count = 0

        # Loop through each rectangle to find the maximum square length
        for length, width in rectangles:
            # Find the smaller of the two sides to get the largest possible square
            side_length = min(length, width)

            # If the current square length is greater than the previous maximum,
            # update max_square_length and reset good_rectangles_count
            if side_length > max_square_length:
                max_square_length = side_length
                good_rectangles_count = 1
            # If the current square length is equal to the maximum,
            # increment the count of good rectangles
            elif side_length == max_square_length:
                good_rectangles_count += 1

        # Return the total count of rectangles that can produce the largest square
        return good_rectangles_count

```

Java Solution

```java
class Solution {
    public int countGoodRectangles(int[][] rectangles) {
        int countOfMaxSquares = 0; // This will keep track of how many maximum squares we can cut.
        int maxSize = 0; // This will keep the maximum square size we can get.

        // Loop through each rectangle.
        for (int[] rectangle : rectangles) {
            // Find the size of the largest square that fits within the rectangle.
            // This is the smaller side of the rectangle.
            int largestSquareSize = Math.min(rectangle[0], rectangle[1]);

            // If we found a larger square size than we've seen so far,
            // update the maximum size and reset the count of max squares.
            if (maxSize < largestSquareSize) {
                maxSize = largestSquareSize;
                countOfMaxSquares = 1;
            } else if (maxSize == largestSquareSize) {
                // If we found a square with a size equal to the current maximum,
                // increment the count of max size squares.
                ++countOfMaxSquares;
            }
        }

        // After checking all rectangles, return the count of the largest squares.
        return countOfMaxSquares;
    }
}

```

C++ Solution

```cpp
#include <vector>  // Include necessary library for vector
#include <algorithm> // Include for access to the min function

class Solution {
public:
    int countGoodRectangles(std::vector<std::vector<int>>& rectangles) {
        int maxSquareLen = 0;     // Variable to store the maximum square length
        int goodRectanglesCount = 0;  // Variable to count the number of good rectangles

        // Iterate over each rectangle in the input vector
        for (const std::vector<int>& rect : rectangles) {
            // Calculate the maximum square length that can be cut out of the current rectangle
            int squareLen = std::min(rect[0], rect[1]);

            // If the current square length is greater than maxSquareLen, update maxSquareLen
            // and reset goodRectanglesCount since we have found a larger square
            if (squareLen > maxSquareLen) {
                maxSquareLen = squareLen;
                goodRectanglesCount = 1;  // Start counting from one as this is the new largest square
            }
            // If the current square length is equal to maxSquareLen, increment the count
            else if (squareLen == maxSquareLen) {
                ++goodRectanglesCount;
            }
        }

        // After iterating through all rectangles, return the count of good rectangles
        return goodRectanglesCount;
    }
};

```

Typescript Solution

```typescript
// Function to count the number of rectangles that can form a square with the largest side
function countGoodRectangles(rectangles: number[][]): number {
    let maxSquareSide = 0; // Initialize a variable to keep track of the maximum square side length
    let squareCount = 0; // Initialize a counter to keep track of the number of squares with maxSquareSide

    // Loop through each rectangle
    for (let [length, width] of rectangles) {
        // Find the minimum side to determine the size of the largest square
        let largestSquareSide = Math.min(length, width);

        // If the current square's side is equal to the maximum found so far, increment the count
        if (largestSquareSide == maxSquareSide) {
            squareCount++;
        } else if (largestSquareSide > maxSquareSide) {
            // If current square's side is larger, update the maximum side and reset the count
            maxSquareSide = largestSquareSide;
            squareCount = 1;
        }
    }

    // After looping through all rectangles, return the count of squares with the largest side
    return squareCount;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the number of rectangles in the input list `rectangles`. This is because the code iterates through each rectangle exactly once, and for each rectangle, it performs a constant number of operations: calculating the minimum of the length and width, comparison, and possibly incrementing the answer or updating the maximum length of a square.

Space Complexity

The space complexity of the code is `O(1)`. This is because the code only uses a fixed amount of additional space: two variables (`ans` and `mx`). The amount of space used does not depend on the input size, so the space complexity is constant.