3047. Find the Largest Area of Square Inside Two Rectangles

Medium Geometry Array Math

Problem Description

In this problem, we are given `n` rectangles on a 2D plane. Each rectangle's bottom-left and top-right coordinates are provided via two separate 2D arrays: `bottomLeft` and `topRight`. Our task is to identify the largest area of a square that can fit within an intersection of any two of these rectangles.

If no rectangles intersect, then the area of the largest square that can be fitted is zero by definition. Otherwise, we are to find and return the largest possible square area that can be inscribed within such an intersection.

Intuition

To find the solution, we apply an enumeration method. What this means is we will be checking all possible pairs of rectangles to find intersections. For each pair of rectangles, we determine the coordinates of their overlapping area. Specifically, we look at the x and y coordinates of their bottom left and the top right corners to do so.

Rectangles intersect if and only if one rectangle's bottom left is less than the other's top right for both the x and y axes. If they do intersect, the dimensions of the intersection can be calculated using max and min functions. The intersection's bottom-left x coordinate is the larger of the two rectangles' bottom-left x coordinates, and similarly, the bottom-left y coordinate is the larger of the two rectangles' bottom-left y coordinates. The top-right x coordinate is the smaller of the top-right x coordinates, and the top-right y coordinate is the smaller of the y coordinates.

Once we have the dimensions of the intersecting area, we need to check if they can form a square. A square is constrained by its smallest dimension, so we take the smaller of the intersection's width or height as the potential size of the square. Then we calculate the area of the square (side length squared) and keep track of the largest area found during our enumeration. If a square can be placed within the intersection region, we update our answer with the area of that square if it's larger than the previously recorded answer.

Solution Approach

The solution approach involves iterating through all possible pairs of rectangles and calculating their possible intersection. This is a brute-force approach powered by the combinatorial power of the `combinations` function from the Python standard library's `itertools` module. This function generates all unique pairs of rectangles so that we can consider their potential intersection.

Once we have a pair of rectangles, the implementation computes the width `w` and height `h` of their intersection using the following formulas:

- For width `w`:

  ```
  w = min(x2, x4) - max(x1, x3)
  ```

  The `min` function is used to find the least top-right x-coordinate between the two rectangles, and the `max` function finds the greatest bottom-left x-coordinate. The difference gives the horizontal span of the intersection.

- For height `h`:

  ```
  h = min(y2, y4) - max(y1, y3)
  ```

  Similarly, the `min` function is used to determine the least top-right y-coordinate, and the `max` function locates the greatest bottom-left y-coordinate. The difference gives the vertical span of the intersection.

If either `w` or `h` is negative, it means that there is no positive area of intersection between the two rectangles, and we skip over that pair.

After finding the intersection dimensions, the following steps are carried out:

1. We then find the side length of the largest possible inscribed square by taking the minimum of `w` and `h`. This is because a square's sides must be equal, and the intersection's smallest dimension limits the side's length.

   ```
   e = min(w, h)
   ```

2. We ensure that the side length is positive (`e > 0`). If it is, we have a valid square, and we compute its area (`e * e`).
3. The solution keeps updating the variable `ans` with the maximum area found so far. This is done using another `max` function:

   ```
   ans = max(ans, e * e)
   ```

At the end of the iteration through all rectangle pairs, the value of `ans` represents the largest square area that can be inscribed in any intersection of two input rectangles, which is what the function returns.

Example Walkthrough

Let's say we are given the following pairs of bottom-left (`bottomLeft`) and top-right (`topRight`) coordinates for `n = 3` rectangles:

```
bottomLeft = [[1, 2], [3, 5], [6, 7]]
topRight = [[4, 8], [6, 10], [9, 9]]
```

Using the solution approach outlined above, let's walk through the steps to find the largest square that can fit in an intersection of any two of these rectangles.

1. Generate all unique pairs of rectangles from our `n = 3` rectangles. We will have the following three pairs to check for intersections and possible inscribed squares:
   - Pair 1: Rectangle 1 and Rectangle 2
   - Pair 2: Rectangle 1 and Rectangle 3
   - Pair 3: Rectangle 2 and Rectangle 3
2. For each of these pairs, we calculate the width and height of their potential intersection. Let's enumerate through them:
   - For Pair 1 (Rectangles 1 & 2):
     - The width `w` of the overlapping area is calculated as `min(x2[0], x4[0]) - max(x1[0], x3[0])`, which for their given coordinates is `min(4, 6) - max(1, 3)`, resulting in `w = 1`.
     - The height `h` is `min(y2[1], y4[1]) - max(y1[1], y3[1])`, which gives us `h = min(8, 10) - max(2, 5)`, so `h = 3`.
     - The possible square side length `e = min(w, h) = 1`, so the area of the square is `e * e = 1`.
   - For Pair 2 (Rectangles 1 & 3 - no intersection as they are far apart):
     - Similarly, for width `w`, we get `w = min(4, 9) - max(1, 6)` which results in a negative value, indicating no positive area of intersection. We skip over this pair.
   - For Pair 3 (Rectangles 2 & 3):
     - The width `w` is `min(x2[0], x4[0]) - max(x3[0], x3[0])` giving us `w = min(6, 9) - max(3, 6)`, which results in `w = 0` (edges touching, but no area of intersection).
     - Calculating height `h` similarly would not matter as we already have `w = 0`. We skip over this pair.
3. As we iterate over these pairs, we keep track of the maximum square area `ans`. Initially, `ans = 0`. After checking Pair 1, we update `ans` to `max(0, 1)` which is `1`.
4. Since Pair 2 and Pair 3 do not contribute any area (no valid intersections for a square), the `ans` does not change and remains `1`.
5. At the end of the iteration, we return the largest square area found which is `1`. This is the largest square that can fit within an intersection of any of the given rectangles.

Solution Implementation

Python

```python
from typing import List
from itertools import combinations

class Solution:
    def largestSquareArea(self, bottom_left: List[List[int]], top_right: List[List[int]]) -> int:
        """
        Calculate the largest square area from overlapping rectangles.

        :param bottom_left: List of [x, y] coordinates for the bottom left corner of rectangles.
        :param top_right: List of [x, y] coordinates for the top right corner of rectangles.
        :return: Area of the largest square that can be formed within the overlapping area of rectangles.
        """

        # Initialize the variable to store the maximum square area found.
        max_square_area = 0

        # Generate all pairs of rectangles to check for overlapping.
        for ((x1, y1), (x2, y2)), ((x3, y3), (x4, y4)) in combinations(zip(bottom_left, top_right), 2):

            # Calculate the width of the overlapping area.
            width = min(x2, x4) - max(x1, x3)

            # Calculate the height of the overlapping area.
            height = min(y2, y4) - max(y1, y3)

            # The edge of the largest square inside the overlapping area
            # is the minimum of width and height.
            edge = min(width, height)

            # If there is an overlapping area, calculate its square area.
            if edge > 0:
                max_square_area = max(max_square_area, edge * edge)

        # Return the largest square area found.
        return max_square_area
```

Java

```java
class Solution {
    public long largestSquareArea(int[][] bottomLeftCorners, int[][] topRightCorners) {
        long maxSquareArea = 0; // Initialize the maximum square area to 0

        // Iterate through each rectangle
        for (int i = 0; i < bottomLeftCorners.length; ++i) {
            // Get the bottom left and top right coordinates of the first rectangle
            int x1 = bottomLeftCorners[i][0], y1 = bottomLeftCorners[i][1];
            int x2 = topRightCorners[i][0], y2 = topRightCorners[i][1];

            // Compare the first rectangle with every other rectangle
            for (int j = i + 1; j < bottomLeftCorners.length; ++j) {
                // Get the bottom left and top right coordinates of the second rectangle
                int x3 = bottomLeftCorners[j][0], y3 = bottomLeftCorners[j][1];
                int x4 = topRightCorners[j][0], y4 = topRightCorners[j][1];

                // Calculate the width and height of the overlapping rectangle
                int overlapWidth = Math.min(x2, x4) - Math.max(x1, x3);
                int overlapHeight = Math.min(y2, y4) - Math.max(y1, y3);

                // Find the maximum square edge for the overlapping area
                int maxSquareEdge = Math.min(overlapWidth, overlapHeight);

                // If an overlap exists (positive edge length), calculate the possible square area
                if (maxSquareEdge > 0) {
                    // Update the maximum square area if the current square area is greater
                    maxSquareArea = Math.max(maxSquareArea, 1L * maxSquareEdge * maxSquareEdge);
                }
            }
        }

        // Return the maximum square area found
        return maxSquareArea;
    }
}
```

C++

```cpp
class Solution {
public:
    long long largestSquareArea(vector<vector<int>>& bottomLeft, vector<vector<int>>& topRight) {
        long long maxSquareArea = 0; // Initialize the maximum square area to 0

        // Loop through each rectangle
        for (int i = 0; i < bottomLeft.size(); ++i) {
            // Get the coordinates of the bottom left and top right corners of the first rectangle
            int x1 = bottomLeft[i][0], y1 = bottomLeft[i][1];
            int x2 = topRight[i][0], y2 = topRight[i][1];

            // Now compare the first rectangle with every other rectangle
            for (int j = i + 1; j < bottomLeft.size(); ++j) {
                // Get the coordinates of the bottom left and top right corners of the second rectangle
                int x3 = bottomLeft[j][0], y3 = bottomLeft[j][1];
                int x4 = topRight[j][0], y4 = topRight[j][1];

                // Calculate width and height of the overlapping area between two rectangles
                int overlapWidth = min(x2, x4) - max(x1, x3);
                int overlapHeight = min(y2, y4) - max(y1, y3);

                // The edge length of the largest square is limited by the smaller of the two dimensions
                int squareEdge = min(overlapWidth, overlapHeight);

                // If there's a valid overlapping area, calculate its square area
                if (squareEdge > 0) {
                    maxSquareArea = max(maxSquareArea, static_cast<long long>(squareEdge) * squareEdge);
                }
            }
        }

        // Return the largest square area found
        return maxSquareArea;
    }
};
```

TypeScript

```typescript
function largestSquareArea(bottomLeftCorners: number[][], topRightCorners: number[][]): number {
    let maximumArea = 0; // Initialize the maximum area to 0

    // Iterate through each rectangle by using bottom-left and top-right corners
    for (let i = 0; i < bottomLeftCorners.length; ++i) {
        const [x1, y1] = bottomLeftCorners[i]; // Bottom left corner (x1, y1)
        const [x2, y2] = topRightCorners[i];   // Top right corner (x2, y2)

        // Continue iterating to find the overlapping areas
        for (let j = i + 1; j < bottomLeftCorners.length; ++j) {
            const [x3, y3] = bottomLeftCorners[j]; // Compare with next bottom left corner (x3, y3)
            const [x4, y4] = topRightCorners[j];   // Compare with next top right corner (x4, y4)

            // Calculate the overlap width and height
            const width = Math.min(x2, x4) - Math.max(x1, x3);
            const height = Math.min(y2, y4) - Math.max(y1, y3);

            // Edge of the largest possible square within the overlap
            const edgeSize = Math.min(width, height);

            // Check if there is a valid overlapping square
            if (edgeSize > 0) {
                // Calculate the area of the square and update the maximum area if necessary
                maximumArea = Math.max(maximumArea, edgeSize * edgeSize);
            }
        }
    }

    // Return the largest square's area found in the overlaps
    return maximumArea;
}
```

```python
from typing import List
from itertools import combinations

class Solution:
    def largestSquareArea(self, bottom_left: List[List[int]], top_right: List[List[int]]) -> int:
        """
        Calculate the largest square area from overlapping rectangles.

        :param bottom_left: List of [x, y] coordinates for the bottom left corner of rectangles.
        :param top_right: List of [x, y] coordinates for the top right corner of rectangles.
        :return: Area of the largest square that can be formed within the overlapping area of rectangles.
        """

        # Initialize the variable to store the maximum square area found.
        max_square_area = 0

        # Generate all pairs of rectangles to check for overlapping.
        for ((x1, y1), (x2, y2)), ((x3, y3), (x4, y4)) in combinations(zip(bottom_left, top_right), 2):

            # Calculate the width of the overlapping area.
            width = min(x2, x4) - max(x1, x3)

            # Calculate the height of the overlapping area.
            height = min(y2, y4) - max(y1, y3)

            # The edge of the largest square inside the overlapping area
            # is the minimum of width and height.
            edge = min(width, height)

            # If there is an overlapping area, calculate its square area.
            if edge > 0:
                max_square_area = max(max_square_area, edge * edge)

        # Return the largest square area found.
        return max_square_area
```

Time and Space Complexity

The time complexity of the given code is $O(n^2)$, where n is the number of rectangles. This is due to the use of the `combinations` function with `2` as the second argument, which generates all pairs of rectangles to determine the area of the potential square. Since for n elements there are $n*(n-1)/2$ combinations, this operation is bound by a quadratic function of n.

The space complexity of the code is $O(1)$. This refers to the constant extra space used by the variables within the function. The algorithm's space requirements do not grow with the input size; `ans`, `w`, `h`, and `e` variables use a fixed amount of space regardless of the number of rectangles.