# 836. Rectangle Overlap

Easy Geometry Math

Leetcode Link

## Problem Description

The problem provides a representation of two axis-aligned rectangles. Each rectangle is described with a list `[x1, y1, x2, y2]`, where `(x1, y1)` is the coordinate of its bottom-left corner and `(x2, y2)` is the coordinate of its top-right corner. This means that the edges of the rectangle are parallel to the X and Y axes, with horizontal edges running left to right and vertical edges running bottom to top.

The task is to determine whether these two rectangles overlap with each other. Overlapping, in this context, means that the area where the two rectangles cover is more than zero. If they just touch at the edges or corners, it is not considered an overlap. The output should be `true` if there is an overlap, otherwise `false`.

## Intuition

To determine if the two rectangles overlap, we look at the cases when they definitely don't overlap and invert the logic to find when they do. Rectangles do not overlap if one is completely to the left, right, above, or below the other.

- **Completely to the left:** This happens when the right edge of one rectangle (`x2` or `x4`) is to the left of the left edge of the other (`x1` or `x3`). That means `x2 <= x3` or `x4 <= x1`.
- **Completely to the right:** Similarly, if the left edge of one rectangle is to the right of the right edge of the other, they don't overlap. So, `x3 >= x2` or `x1 >= x4`.
- **Completely above:** If the bottom edge of one rectangle is above the top edge of the other, there's no overlap. Thus, `y3 >= y2` or `y1 >= y4`.
- **Completely below:** Conversely, if the top edge of one is below the bottom edge of the other, they do not overlap. Hence, `y2 <= y3` or `y4 <= y1`.

To resolve these conditions into a single check for non-overlapping, we can take the `not` of each comparison and combine them with logical `or`. Now, if any of these non-overlap conditions are `true`, we have a `not` operator in front which will make the whole condition `false`. Hence, if none of the non-overlap conditions is met, the rectangles must overlap and `return not (...)` will return `true`. The solution code effectively checks for these conditions to return the correct overlap status.

## Solution Approach

The solution given follows a straightforward approach that leverages computational geometry concepts. The key idea is to determine whether two rectangles overlap without actually calculating the intersection area.

The code defines a method `isRectangleOverlap` which accepts two lists, `rec1` and `rec2`, each containing the coordinates of the bottom-left and top-right corners of a rectangle. Within this method, these coordinates are unpacked into variables `x1, y1, x2, y2` for `rec1` and `x3, y3, x4, y4` for `rec2`.

To determine if the rectangles overlap, the function checks the inverse of the four possible ways that the rectangles could not overlap:

1. One rectangle is to the left of the other: `x3 >= x2` or `x4 <= x1`.
2. One rectangle is to the right of the other: `x2 <= x3` or `x1 >= x4`.
3. One rectangle is above the other: `y3 >= y2` or `y1 >= y4`.
4. One rectangle is below the other: `y2 <= y3` or `y4 <= y1`.

For non-overlap, at least one of these conditions must be true. The code checks the negative condition (`not`) to see if none of these four non-overlapping conditions is true. If none of them holds, the rectangles must overlap, which logically means that the two rectangles do have an area of intersection.

This is accomplished in a single return statement in Python:

```
return not (y3 >= y2 or y4 <= y1 or x3 >= x2 or x4 <= x1)
```

This line directly translates the absence of non-overlapping conditions into a boolean result indicating whether or not the rectangles overlap.

No additional data structures or advanced algorithmic patterns are needed for this task. The solution's elegance lies in its simplicity and its O(1) time complexity, as it only involves simple arithmetic operations and logical comparisons.

### Example Walkthrough

Let's consider two rectangles given by their bottom-left and top-right points:

- Rectangle 1 (`rec1`): `[1, 1, 3, 3]`
- Rectangle 2 (`rec2`): `[2, 2, 4, 4]`

We need to determine if these rectangles overlap using the approach described above. First, we unpack the coordinates:

- For `rec1`, we have `x1 = 1`, `y1 = 1`, `x2 = 3`, and `y2 = 3`.
- For `rec2`, we have `x3 = 2`, `y3 = 2`, `x4 = 4`, and `y4 = 4`.

Now, we apply the four checks to see if there is a non-overlapping condition that applies:

1. One rectangle is to the left of the other:
   This would mean that the right edge of `rec1` is to the left of the left edge of `rec2` (`x2 <= x3`), or the right edge of `rec2` is to the left of the left edge of `rec1` (`x4 <= x1`). Checking these conditions:
   - `x2 <= x3` is `3 <= 2`, which is `false`.
   - `x4 <= x1` is `4 <= 1`, which is `false`.
2. One rectangle is to the right of the other:
   This would be `x3 >= x2` or `x1 >= x4`.
   Checking these:
   - `x3 >= x2` is `2 >= 3`, which is `false`.
   - `x1 >= x4` is `1 >= 4`, which is `false`.
3. One rectangle is above the other:
   This would be `y3 >= y2` or `y1 >= y4`.
   Checking these:
   - `y3 >= y2` is `2 >= 3`, which is `false`.
   - `y1 >= y4` is `1 >= 4`, which is `false`.
4. One rectangle is below the other:
   This would be `y2 <= y3` or `y4 <= y1`.
   Checking these:
   - `y2 <= y3` is `3 <= 2`, which is `false`.
   - `y4 <= y1` is `4 <= 1`, which is `false`.

Since none of these conditions are `true`, it means none of the non-overlapping conditions are met. Therefore, according to our function:

```
return not (y3 >= y2 or y4 <= y1 or x3 >= x2 or x4 <= x1)
```

The result of the function is `not (false or false or false or false)`, which simplifies to `not (false)`, thus returning `true`. Hence, using our approach, we determine that rectangle 1 and rectangle 2 do indeed overlap.

## Python Solution

```python
class Solution:
    def isRectangleOverlap(self, rec1: List[int], rec2: List[int]) -> bool:
        """
        Determine if two rectangles overlap.

        The rectangles are defined by their bottom-left and top-right corners:
        rec1 corresponds to (x1, y1, x2, y2)
        rec2 corresponds to (x3, y3, x4, y4)

        We return True if the rectangles overlap, False otherwise.
        """

        # Unpack coordinates for the first rectangle
        x1, y1, x2, y2 = rec1

        # Unpack coordinates for the second rectangle
        x3, y3, x4, y4 = rec2

        # Check for overlap:
        # There is no overlap if:
        # - The top edge of rec2 is below or on the bottom edge of rec1 (y3 >= y2)
        # - The bottom edge of rec2 is above or on the top edge of rec1 (y4 <= y1)
        # - The right edge of rec2 is to the left or on the left edge of rec1 (x3 >= x2)
        # - The left edge of rec2 is to the right or on the right edge of rec1 (x4 <= x1)
        # If none of these conditions are met, the rectangles overlap.

        return not (y3 >= y2 or y4 <= y1 or x3 >= x2 or x4 <= x1)

```

## Java Solution

```java
class Solution {
    public boolean isRectangleOverlap(int[] rec1, int[] rec2) {
        // Extract the coordinates for the first rectangle.
        int rect1X1 = rec1[0], rect1Y1 = rec1[1], rect1X2 = rec1[2], rect1Y2 = rec1[3];
        // Extract the coordinates for the second rectangle.
        int rect2X1 = rec2[0], rect2Y1 = rec2[1], rect2X2 = rec2[2], rect2Y2 = rec2[3];

        // Check if the rectangles do not overlap; return the negation of this statement.
        // If one rectangle is above the top edge of the other or one rectangle is to the left of the other's right edge,
        // the rectangles are not overlapping.
        return !(rect2Y1 >= rect1Y2 || // Rectangle 2 is below Rectangle 1
                 rect2Y2 <= rect1Y1 || // Rectangle 2 is above Rectangle 1
                 rect2X1 >= rect1X2 || // Rectangle 2 is to the right of Rectangle 1
                 rect2X2 <= rect1X1); // Rectangle 2 is to the left of Rectangle 1
    }
}

```

## C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    bool isRectangleOverlap(vector<int>& rec1, vector<int>& rec2) {
        // Extracting coordinates for the first rectangle
        int leftX1 = rec1[0], bottomY1 = rec1[1], rightX1 = rec1[2], topY1 = rec1[3];
        // Extracting coordinates for the second rectangle
        int leftX2 = rec2[0], bottomY2 = rec2[1], rightX2 = rec2[2], topY2 = rec2[3];

        // Check for no overlap conditions
        // If one rectangle is to the left of the other
        bool isSeparateHorizontally = leftX2 >= rightX1 || rightX2 <= leftX1;
        // If one rectangle is above the other
        bool isSeparateVertically = bottomY2 >= topY1 || topY2 <= bottomY1;

        // Two rectangles overlap if neither separation condition is true
        return !(isSeparateHorizontally || isSeparateVertically);
    }
};

```

## Typescript Solution

```typescript
// Define the type for a rectangle as an array of four numbers
type Rectangle = [number, number, number, number];

// Helper function to check if two rectangles overlap
function isRectangleOverlap(rectangle1: Rectangle, rectangle2: Rectangle): boolean {
    // Extracting coordinates for the first rectangle
    const [leftX1, bottomY1, rightX1, topY1] = rectangle1;
    // Extracting coordinates for the second rectangle
    const [leftX2, bottomY2, rightX2, topY2] = rectangle2;

    // Check for no overlap conditions
    // If one rectangle is to the left of the other
    const isSeparateHorizontally = leftX2 >= rightX1 || rightX2 <= leftX1;
    // If one rectangle is above the other
    const isSeparateVertically = bottomY2 >= topY1 || topY2 <= bottomY1;

    // Two rectangles overlap if neither separation condition is true
    return !(isSeparateHorizontally || isSeparateVertically);
}

```

## Time and Space Complexity

The given Python function `isRectangleOverlap` checks if two rectangles overlap. The function takes the coordinates of the bottom-left and top-right corners of each rectangle (rec1 and rec2) as inputs. The rectangles are defined by their coordinates on the x and y axes: `(x1, y1)` for the bottom-left corner and `(x2, y2)` for the top-right corner of the first rectangle, and similarly `(x3, y3)` and `(x4, y4)` for the second.

### Time Complexity:

The function consists of a single return statement with a few comparisons between the input variables. Since each comparison is an `O(1)` operation and there are a fixed number of comparisons (specifically four), the overall time complexity of the function is `O(1)`. This constant time complexity indicates that the function's running time does not depend on the size of the input, but rather it executes in a fixed amount of time regardless of the input.

### Space Complexity:

The function does not use any additional data structures that grow with the size of the input. The space used is only for the input parameters and a fixed amount of variables for unpacking these parameters, all of which do not depend on the input size. Therefore, the space complexity of the function is also `O(1)`. This constant space complexity denotes that the amount of memory required by the function does not change with the size of the input.