2001. Number of Pairs of Interchangeable Rectangles

**Number Theory** 

Counting

## **Problem Description**

Hash Table

<u>Math</u>

Medium <u>Array</u>

In this problem, we are given a list of n rectangles, each represented by their width and height as an element in a 2D integer array rectangles. For example, the i-th rectangle is represented as rectangles[i] = [width\_i, height\_i]. The task here is to find how many pairs of these rectangles are interchangeable. Two rectangles are considered interchangeable

if they have the equivalent width-to-height ratio. In mathematical terms, rectangle i is interchangeable with rectangle j if width\_i/height\_i equals width\_j/height\_j. Note that we will be using decimal division to compare ratios. The expected result is to return the total count of such interchangeable pairs among the rectangles provided.

Intuition

#### To find pairs of interchangeable rectangles, we focus on calculating the width-to-height ratio for each rectangle. However, directly comparing floating-point ratios might lead to precision issues. To avoid this, we normalize each width-to-height ratio by

height ratio will have the same normalized width and height, thus can be represented by the same key when stored. To efficiently track how many rectangles have the same normalized width and height, we use a Counter data structure (a type of dictionary that is optimized for counting hashable objects). When we normalize a rectangle, we check how many we've seen with that same normalized width and height so far (using the Counter), and we increment our answer by that count—this is because

dividing both width and height by their greatest common divisor (GCD). This way, two rectangles that have the same width-to-

each of those could form an interchangeable pair with the current rectangle. For each rectangle, we then increment its count in the Counter. This step ensures that we keep track of all interchangeable rectangles encountered. Combining these strategies allows us to count the pairs without comparing every rectangle to every other rectangle, which would

rectangles without floating-point arithmetic issues. **Solution Approach** 

The solution uses a combination of hashing and the greatest common divisor (GCD) algorithm to efficiently count

be inefficient. The use of the greatest common divisor (GCD) ensures that we are accurately identifying interchangeable

#### • We initialize the variable ans to store the total count of interchangeable rectangle pairs and a Counter data structure called cnt, which will be used to keep track of the occurrences of normalized width-to-height ratios.

input array.

interchangeable rectangle pairs.

• We iterate over each rectangle given in the rectangles list. For the current rectangle, w represents the width and h represents the height. • We calculate the greatest common divisor of w and h using the function gcd(w, h). This step is crucial as it allows us to normalize the

We then normalize w and h by dividing them by the calculated GCD. As a result, the tuple (w // g, h // g) represents the unique key of

width and height to their simplest ratio form.

Here is a step-by-step breakdown of the implementation:

- the current ratio in its simplest form. • The expression cnt[(w, h)] retrieves the current count of rectangles that have been seen with the same normalized ratio. This count
- directly corresponds to the number of new pairs that can be formed with the current rectangle since each of those previous rectangles could form a pair with it.
  - We increment ans by the count retrieved from cnt[(w, h)]. Finally, we increment cnt[(w, h)] by 1 because we have encountered another rectangle with the same normalized ratio.
  - By hashing the normalized ratios, the algorithm ensures constant-time lookups for previous occurrences, bypassing the need for nested loops, which would increase the time complexity significantly. The final result stored in ans is returned, which represents the total number of pairs of interchangeable rectangles within the
- gained through hashing. **Example Walkthrough**

Let's consider an example with the following list of rectangles: rectangles = [[4, 8], [3, 6], [10, 20], [15, 30]].

We want to find the total count of interchangeable rectangle pairs that can be formed from these rectangles.

(1, 2). We have not seen this ratio before, so cnt[(1, 2)] is 0. We then increment cnt[(1, 2)] by 1.

1. First, we initialize ans to 0, which will hold our answer, and cnt, a Counter to keep track of normalized width-to-height ratios.

Using this approach, the solution is both efficient and precise, utilizing the mathematical properties of ratios and the efficiencies

3. Next, rectangle [3, 6] comes down to the same normalized ratio (1, 2) after dividing by the gcd(3, 6) which is 3. We check cnt[(1, 2)] which is 1 (from the previous step), so we increase ans by 1 and cnt[(1, 2)] becomes 2. 4. Then, with rectangle [10, 20], after dividing by gcd(10, 20) which is 10, the ratio is again (1, 2). The cnt[(1, 2)] is currently 2, meaning we

can form 2 more pairs with each of the rectangles we've seen before, so we add 2 to ans, making it a total of 3 so far, and cnt[(1, 2)]

2. For the first rectangle [4, 8], we compute its GCD to normalize the ratio. gcd(4, 8) is 4, so when we divide both by 4, the normalized ratio is

## 5. Lastly, for the rectangle [15, 30], we divide by gcd(15, 30) which is 15, and we get the normalized ratio (1, 2). Since cnt[(1, 2)] is 3, we

becomes 3.

**Python** 

from math import gcd

pair\_count = 0

ratio\_counter = Counter()

# Initialize a variable to keep count of pairs

# Initialize a counter to keep track of the occurrences of each ratio

# Normalize the width and height by dividing them by the gcd to obtain the ratio

# will be the number of interchangeable rectangle pairs we can form with the current one

int n = rectangles.length + 1; // using length + 1 to ensure a unique mapping when multiplied

Map<Long, Integer> ratioCountMap = new HashMap<>(); // this map stores the counts of each unique rectangle ratio

normalized\_width, normalized\_height = width // gcd\_value, height // gcd\_value

# The number of rectangles with the same width-to-height ratio so far

pair\_count += ratio\_counter[(normalized\_width, normalized\_height)]

long countInterchangeablePairs = 0; // this will hold the final answer

add 3 to ans and increment cnt[(1, 2)]. Now, ans is 6, representing the total number of interchangeable rectangle pairs, and cnt[(1, 2)] is 4, indicating we saw the

same ratio 4 times.

and kept the computation time to a minimum. Solution Implementation

By applying our solution approach, we have efficiently found that there are 6 pairs of interchangeable rectangles in our example

list without having to compare each rectangle to every other rectangle. This process has bypassed potential floating-point issues

- from collections import Counter class Solution: def interchangeableRectangles(self, rectangles: List[List[int]]) -> int:
  - # Loop through each rectangle for width, height in rectangles: # Calculate the greatest common divisor of width and height gcd\_value = gcd(width, height)

#### # Increment the counter for the current ratio ratio\_counter[(normalized\_width, normalized\_height)] += 1 return pair\_count

// Loop through each rectangle

for (auto& rectangle : rectangles) {

answer += countMap[key];

countMap[key]++;

while (b != 0) {

b = a % b;

let pairCount: number = 0;

return pairCount;

**if** (b === 0) {

from math import gcd

class Solution:

return a;

return gcd(b, a % b);

from collections import Counter

\* @param {number} a - First number

\* @param {number} b - Second number

function qcd(a: number, b: number): number {

a = temp;

return a;

int temp = b;

width /= gcdValue: // Simplify width by dividing by the gcd

long long key = 1LL \* width \* (numRectangles + 1) + height;

// Increment the count of the current ratio by 1.

\* Function to find the number of pairs of rectangles that are interchangeable.

\* We normalize each rectangle's dimensions by their greatest common divisor (GCD)

\* Rectangles are interchangeable if one can become the other by resizing.

\* and count occurrences of each unique pair of normalized dimensions.

\* @return {number} - The number of interchangeable rectangle pairs.

function interchangeableRectangles(rectangles: number[][]): number {

const countMap: Map<number, number> = new Map();

pairCount += (countMap.get(hash) || 0);

countMap.set(hash, (countMap.get(hash) || 0) + 1);

\* @return {number} - The greatest common divisor of a and b.

# Initialize a variable to keep count of pairs

# Increment the counter for the current ratio

\* Helper function to calculate the greatest common divisor of two numbers.

def interchangeableRectangles(self, rectangles: List[List[int]]) -> int:

for (let [width, height] of rectangles) {

// Utility function to calculate greatest common divisor

// If a rectangle with the same ratio was already encountered,

height /= gcdValue; // Similarly, simplify height by dividing by the gcd

// Multiplying width by a large number (greater than the maximum height)

return answer; // Return the total count of interchangeable rectangle pairs.

// and then adding the height ensures a unique value for each width to height ratio.

// then increase the count of interchangeable pairs by the count of those rectangles.

// 'key' is the unique identifier for the ratio of width to height.

```
class Solution {
   public long interchangeableRectangles(int[][] rectangles) {
```

Java

```
for (int[] rectangle : rectangles) {
            int width = rectangle[0];
            int height = rectangle[1];
            int gcdValue = gcd(width, height); // compute the greatest common divisor of width and height
            width /= gcdValue; // normalize the width by the gcd
            height /= gcdValue: // normalize the height by the gcd
            long ratioHash = (long) width * n + height; // create a unique hash for the width/height ratio
            // Update the number of interchangeable pairs count
            countInterchangeablePairs += ratioCountMap.getOrDefault(ratioHash, 0);
            // Increase the count for this ratio in the map
            ratioCountMap.merge(ratioHash, 1, Integer::sum);
        return countInterchangeablePairs; // return the total count of interchangeable rectangle pairs
    // Helper function to compute the greatest common divisor (GCD) of two numbers
    private int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
C++
#include <vector>
#include <unordered map>
using namespace std;
class Solution {
public:
    long long interchangeableRectangles(vector<vector<int>>& rectangles) {
        long long answer = 0; // This will hold the final count of interchangeable rectangle pairs
        int numRectangles = rectangles.size(); // Total number of rectangles given
```

unordered\_map<long long, int> countMap; // Hashmap to store the count of rectangles with the same ratio

int width = rectangle[0], height = rectangle[1]; // Extract the width and height of the current rectangle

int qcdValue = qcd(width, height); // Find qreatest common divisor of width and height to get the ratio

```
int gcd(int a, int b) {
```

**}**;

\*/

**TypeScript** 

private:

```
// Calculate the greatest common divisor of width and height
const gcdValue: number = gcd(width, height);
// Normalize the width and height using gcdValue
width = Math.floor(width / gcdValue);
height = Math.floor(height / gcdValue);
// Create a unique hash (or map key) for the normalized rectangle dimensions
const hash: number = width * (rectangles.length + 1) + height;
```

// Increment the pair count by the number of occurrences already found

// Update the map with the new count for the current normalized dimensions

\* @param {number[][]} rectangles - A list of rectangle specifications, given by [width, height].

```
pair_count = 0
# Initialize a counter to keep track of the occurrences of each ratio
ratio_counter = Counter()
# Loop through each rectangle
for width, height in rectangles:
   # Calculate the greatest common divisor of width and height
   gcd_value = gcd(width, height)
    # Normalize the width and height by dividing them by the gcd to obtain the ratio
    normalized_width, normalized_height = width // gcd_value, height // gcd_value
```

# will be the number of interchangeable rectangle pairs we can form with the current one

# The number of rectangles with the same width-to-height ratio so far

pair\_count += ratio\_counter[(normalized\_width, normalized\_height)]

ratio\_counter[(normalized\_width, normalized\_height)] += 1

rectangles list. During each iteration, the following operations occur:

# The time complexity of the given code is primarily determined by the for loop that iterates over every rectangle within the

**Time Complexity** 

return pair\_count

Time and Space Complexity

#### complexity of the gcd function is generally $O(\log(\min(w, h)))$ , where w and h are the width and height of the rectangle. Reducing the width and height by dividing by their GCD. • Checking and updating the count of a particular ratio (width to height) in the hash map implemented by the Counter class.

Assuming n represents the number of rectangles, the time complexity for the gcd operation, which is the most expensive

• Calculating the greatest common divisor (GCD) for the width and height of the rectangle, which is done using the gcd function. The time

- operation inside the loop, adds up to 0(n \* log(min(w, h))). However, since this is a reduction to the simplest form, the value of log(min(w, h)) is small compared to n. Therefore, we often approximate this to O(n) when w and h are not extreme values. The lookup and update in the hash map cnt have an average-case time complexity of 0(1) for each operation since these
- Thus, the overall average time complexity of the code is O(n).

operations are constant time in a hash map (dictionary in Python) on average.

### The space complexity of the code is influenced by the space required to store the reduced width and height pairs in a hash map. In the worst case, if all rectangles have different width-to-height ratios, the space complexity would be 0(n) because each

**Space Complexity** 

rectangle would be represented in the hash map after reduction. Furthermore, the counter ans is using 0(1) space and the gcd call does not use additional space proportional to n.

Therefore, the total space complexity of the given code is O(n).