2427. Number of Common Factors

Easy Math Enumeration Number Theory

Problem Description

In this problem, you are given two positive integers, `a` and `b`. Your task is to find out how many common factors these two numbers share. A common factor is defined as an integer `x` that can divide both `a` and `b` without leaving a remainder. This is straightforward in the sense that all you need to do is count all numbers from 1 to the smallest of `a` and `b`, checking if they are factors of both.

Intuition

To efficiently solve this problem, we leverage the fact that the common factors of `a` and `b` must also be factors of the greatest common divisor (gcd) of `a` and `b`. The greatest common divisor of two numbers is the largest number that divides both of them without leaving a remainder. Once we find the gcd of `a` and `b`, we just need to count the number of factors this gcd has because they will naturally be common factors of both `a` and `b`.

The steps to arrive at this solution are:

1. Calculate the gcd of `a` and `b`, using the `gcd` function (which is presumably imported from the `[math](/problems/math-basics)` module, although it is not shown in the code). The gcd represents the largest common factor of both numbers.
2. Initialize two variables; `ans` to keep the count of common factors found, and `x` to iterate through possible factors, starting at `1`.
3. Use a while loop to check every integer smaller or equal to the square root of `gcd` (`x * x <= g`). We perform this check only until the square root because if `x` is a factor of `g`, there will be a corresponding factor `g / x` which is either equal to `x` (when `x*x == g`) or greater than `x` (when `x` is less than the square root of `g`).
4. Within the loop, if `x` divides `g` without a remainder (`g % x == 0`), it means `x` is a factor. Thus, we increase `ans` by 1. We also check if `x * x` is strictly less than `g`, which suggests there is another factor at the other side of the square root of `g`. If so, we increase `ans` by another 1.
5. Increment `x` by 1 to check the next integer.
6. After finishing the loop, we return `ans`, which is the total count of the common factors.

This method is significantly faster than checking each number from `1` to `min(a, b)`. Instead, it only checks up to the square root of the gcd, which is an optimization based on the property that if `x` is a factor of `y`, then `y` must have another factor that is either `y / x` or `x` itself.

Solution Approach

In solving this problem, the algorithm follows a series of logical steps based on number theory. There's reliance on a core mathematical concept—every number has a unique factorization and can be represented as the product of their prime factors raised to various powers. Extending this, the common factors of two numbers will be associated with the common prime factors of those numbers. Therefore, calculating the greatest common divisor (gcd) of the two numbers gives us a number that has all and only the prime factors shared between the two.

The code structure:

1. First, we find the gcd of the two given numbers, `a` and `b`, using a gcd function. The `gcd(a, b)` in the code represents this highest common factor and is stored in variable `g`.
2. We then initialize two variables: `ans` to `0`, which will serve as a counter for the common factors, and `x` to `1`, to iterate through potential factors starting from 1 up to the square root of `g`.
3. The code uses a `while` loop to iterate through all the numbers from `1` up until the square root of the gcd. The condition `x * x <= g` guarantees that we do not check past the square root, maximizing efficiency.
4. Inside the loop, the modulo operation `g % x` checks if `x` divides `g` with no remainder—if it does, `x` is a factor of `g`. Since `g` is the gcd of `a` and `b`, `x` is also a common factor of `a` and `b`. So, we increment `ans` by `1`.
5. We also look for the factor pair of `x`, which is `g / x`. If `x * x` is less than `g`, then `g / x` is another factor of `g` that is distinct from `x`. Therefore, we have another factor to add to our `ans` count, so we increment `ans` by `1` again.
6. After evaluating a potential factor `x`, we increment `x` by `1` to check the next number. This continues until `x` exceeds the square root of `g`.
7. Lastly, we return `ans`, which now contains the count of all the unique common factors of `a` and `b`.

No complex data structures are needed for this approach; simple integer variables suffice for the implementation. The pattern applied here is the efficient detection of factors via examining up to the square root, which is a widely-adopted practice in algorithms involving factorization or prime numbers, owing to its computational efficiency.

Example Walkthrough

Let's say you are given `a = 8` and `b = 12` and you want to find the common factors. According to the aforementioned approach, here is a step-by-step process to compute the answer:

1. **Calculate the gcd**: We start by calculating the greatest common divisor of `8` and `12`. The gcd of 8 and 12 is 4, as 4 is the largest number that can divide both 8 and 12 without leaving a remainder.
2. **Initialize variables**: We initialize `ans` to 0 to count the number of common factors. We also initialize `x` to 1 to start checking possible factors from 1.
3. **Loop through potential factors**: We use a while loop to check for factors starting from x = 1. Since the gcd is 4, we only need to check for factors up to the square root of 4, which is 2.
4. **Check divisibility**: During the first iteration with `x = 1`, we find that `4 % 1 == 0`. Therefore, `1` is a factor of `4`, and we increment `ans` to 1.
5. **Find factor pairs**: We then check if 1 has a factor pair distinct from itself by confirming that `1 * 1 < 4`. Since this is not the case (1*1 == 1), we do not increment `ans`.
6. **Increment and check next potential factor**: We increment `x` to 2 and perform the check again. We find that `4 % 2 == 0`, so `2` is a factor and `ans` becomes `2`.
7. **No other factors to check**: There aren't any numbers between 2 and the square root of 4 (which is 2), so we stop the loop here. There is no need to increment `ans` further, as `2 * 2 == 4` which does not suggest that there is another correspondent factor besides `2`.
8. **Return the count**: The variable `ans` contains the count of common factors, which in this case is `2`. Hence, `8` and `12` have 2 common factors: `1` and `2`.

Both `1` and `2` are common factors because both numbers can be divided by these factors without a remainder. So the result for the input `a = 8` and `b = 12` would be `2`.

This example fits into the solution approach as a straightforward instance of the general method, illustrating the reasoning and calculations behind finding common factors of two numbers by means of their greatest common divisor.

Solution Implementation

Python

```python
from math import gcd

class Solution:
    def commonFactors(self, a: int, b: int) -> int:
        # Calculate the greatest common divisor (GCD) of a and b
        greatest_common_divisor = gcd(a, b)
        # Initialize answer as 0 (will count common factors)
        number_of_common_factors = 0
        # Start with potential factor 1
        potential_factor = 1

        # Check all numbers up to and including the square root of the GCD
        while potential_factor * potential_factor <= greatest_common_divisor:
            # If potential_factor is a factor of the GCD
            if greatest_common_divisor % potential_factor == 0:
                # Increment count due to one factor found
                number_of_common_factors += 1
                # If it is not a perfect square, count the corresponding
                # larger factor (greatest_common_divisor // potential_factor)
                if potential_factor * potential_factor < greatest_common_divisor:
                    number_of_common_factors += 1
            # Move to the next potential factor
            potential_factor += 1

        # Return the total count of common factors
        return number_of_common_factors
```

Java

```java
class Solution {
    // Function to calculate the number of common factors of numbers a and b
    public int commonFactors(int a, int b) {
        // Calculate the Greatest Common Divisor (GCD) of a and b
        int gcdValue = gcd(a, b);
        int count = 0; // Initialize a counter for the number of common factors

        // Iterate through all numbers from 1 to the square root of the gcdValue
        for (int x = 1; x * x <= gcdValue; ++x) {
            // Check if x is a divisor of gcdValue
            if (gcdValue % x == 0) {
                ++count; // Increment the count for the divisor x

                // If x is not the square root of gcdValue, increment the count for the quotient as well
                if (x * x < gcdValue) {
                    ++count; // Increment the count for the divisor gcdValue / x
                }
            }
        }
        return count; // Return the total count of common factors
    }

    // Helper function to calculate the GCD of a and b using Euclid's algorithm
    private int gcd(int a, int b) {
        // If b is 0, a is the GCD
        return b == 0 ? a : gcd(b, a % b);
    }
}
```

C++

```cpp
#include <algorithm> // include the algorithm header for std::gcd

class Solution {
public:
    // Function to find the number of common factors of two numbers
    int commonFactors(int a, int b) {
        // std::gcd is a library function to find the greatest common divisor of a and b
        int greatestCommonDivisor = std::gcd(a, b);
        int commonFactorCount = 0; // Initialize the count of common factors

        // Loop through all numbers from 1 to greatestCommonDivisor
        for (int divisor = 1; divisor <= greatestCommonDivisor; ++divisor) {
            // Increase the count if divisor is a factor of greatestCommonDivisor
            if (greatestCommonDivisor % divisor == 0) {
                commonFactorCount++;
            }
        }

        // Return the total count of common factors
        return commonFactorCount;
    }
};
```

TypeScript

```typescript
// Function to count all common factors of two numbers
function commonFactors(a: number, b: number): number {
    // Calculate the greatest common divisor (GCD) of a and b
    const greatestCommonDivisor = gcd(a, b);
    let commonFactorCount = 0;

    // Loop through each number from 1 up to the square root of the GCD
    for (let divisor = 1; divisor * divisor <= greatestCommonDivisor; ++divisor) {
        // If the divisor is a factor of GCD
        if (greatestCommonDivisor % divisor === 0) {
            // Count the divisor as a common factor
            ++commonFactorCount;
            // If the divisor is not the square root of GCD, count the complementary factor
            if (divisor * divisor < greatestCommonDivisor) {
                ++commonFactorCount;
            }
        }
    }
    // Return the total count of common factors
    return commonFactorCount;
}

// Function to calculate the greatest common divisor (GCD) of two numbers using Euclid's algorithm
function gcd(a: number, b: number): number {
    // If second number is zero, the GCD is the first number
    return b === 0 ? a : gcd(b, a % b);
}
```

```python
from math import gcd

class Solution:
    def commonFactors(self, a: int, b: int) -> int:
        # Calculate the greatest common divisor (GCD) of a and b
        greatest_common_divisor = gcd(a, b)
        # Initialize answer as 0 (will count common factors)
        number_of_common_factors = 0
        # Start with potential factor 1
        potential_factor = 1

        # Check all numbers up to and including the square root of the GCD
        while potential_factor * potential_factor <= greatest_common_divisor:
            # If potential_factor is a factor of the GCD
            if greatest_common_divisor % potential_factor == 0:
                # Increment count due to one factor found
                number_of_common_factors += 1
                # If it is not a perfect square, count the corresponding
                # larger factor (greatest_common_divisor // potential_factor)
                if potential_factor * potential_factor < greatest_common_divisor:
                    number_of_common_factors += 1
            # Move to the next potential factor
            potential_factor += 1

        # Return the total count of common factors
        return number_of_common_factors
```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be broken down into two parts: computing the greatest common divisor (GCD) of `a` and `b`, and finding the common factors based on the GCD.

1. The first line in the function makes a call to `gcd(a, b)`. The GCD of two numbers `a` and `b` is typically computed using the Euclidean algorithm, which has a time complexity of $O(\log(\min(a, b)))$.
2. The `while` loop runs as long as `x * x <= g`, which means it runs approximately `sqrt(g)` times where `g` is the GCD of `a` and `b`. In each iteration, the loop performs a constant amount of work, so the loop contributes $O(\sqrt{g})$ to the time complexity.

Combining these parts, the total time complexity of the given code is $O(\log(\min(a, b)) + \sqrt{g})$, where `g` is the GCD of `a` and `b`.

Space Complexity

The space complexity of the code is determined by the amount of extra space used aside from input storage. Since no additional data structures are utilized that grow with the size of the input, and only a fixed number of integer variables are used (`g`, `ans`, `x`), the space complexity of the code is $O(1)$, which refers to constant space complexity.