1925. Count Square Sum Triples

Easy Math Enumeration

Problem Description

In this LeetCode problem, we are asked to find the count of **square triples** within a given range. A square triple is a set of three integers `(a, b, c)` where the sum of the squares of `a` and `b` equals the square of `c`. Formally, a square triple meets the condition `a^2 + b^2 = c^2`. The challenge is to count all such triples where each member of the triple `(a, b, c)` is an integer within the range from 1 to `n`, inclusive.

Intuition

To approach this problem, we can consider two loops `a` and `b` that iterate through every integer pair within the specified range. For each pair `(a, b)`, we calculate the sum of their squares and take the square root of this sum to find `c`. Once we have `c`, the primary check is to ensure that `c` is an integer and it also falls within the range 1 to `n`. If both conditions are met, it indicates that `(a, b, c)` is a valid square triple, and thus, we increment our result counter.

The key reason we only check for `c` by taking the square root of `a^2 + b^2` is because we are looking for a `c` that will satisfy the Pythagorean theorem, indicating a right-angled triangle with sides of integer lengths. This is a direct application of finding Pythagorean triples. The check `c <= n and c**2 == t` ensures both that `c` is not larger than `n` and that `c` is an exact square root of the sum `a^2 + b^2`, hence `c` must be an integer. The counting of each valid triple helps us to return the exact number of square triples within the given range.

Solution Approach

The solution to this problem follows a straightforward brute-force approach. Since we aren't given any constraints that allow a more efficient solution, we simply iterate through all possible combinations of `a` and `b` and calculate `c`. Here's a step-by-step breakdown of the implementation:

1. Initialize a counter `res` set to 0. This will keep track of the number of valid square triples that we find.
2. Loop through all integer values of `a` from 1 to `n`. This loop will consider each potential first member of the triple.
3. Inside the loop for `a`, nest another loop for `b`, also ranging from 1 to `n`. This inner loop goes through every possible second member of the triple, for each value of `a`.
4. For each pair of `(a, b)`, calculate the sum of the squares: `t = a**2 + b**2`. This follows from the definition of a square triple, which requires the square of `c` to equal the sum of the squares of `a` and `b`.
5. Determine if there exists a `c` such that `c**2 = t`. We do this by calculating the square root of `t` and storing it in `c`. Since square roots can be floating-point numbers, we cast it to `int`.
6. Verify two conditions for the calculated `c`:
   - Check if `c` is less than or equal to `n`, as `c` must be within the given range.
   - Confirm that `c` when squared, equals the original sum `t` to ensure that `c` is a whole number.
7. If both conditions are satisfied, it means that we have found a valid square triple, and hence we increase our counter `res` by 1.
8. After iterating through all possible pairs `(a, b)`, return the value of `res`, which now contains the count of all valid square triples within the given range.

This solution does not use any specific complex algorithms, advanced data structures, or patterns. It's a simple double iteration based on the properties of square numbers and the well-known Pythagorean theorem. The key to implementing this algorithm is correctly checking each potential triple and only counting those that satisfy the conditions of being a square triple and having all members less than or equal to `n`. The time complexity of the solution is O(n^2) since it involves two nested loops each running up to n times.

Example Walkthrough

Let's illustrate the solution approach with a small example by considering `n = 5`. We want to find all the square triples `(a, b, c)` such that each element is within the range from 1 to 5 inclusive.

1. We initialize our counter `res` to 0.
2. Starting with `a = 1`, we loop through all values up to 5.
3. For each `a`, we loop `b` starting from 1 to 5 as well.
4. Take `a = 1, b = 1`, calculate `t = a^2 + b^2 = 1^2 + 1^2 = 2`. The square root of `t` is around 1.414, which is not an integer, so we don't increment `res`.
5. Continue the iterations, let's say now we are at `a = 3, b = 4`, compute `t = 3^2 + 4^2 = 9 + 16 = 25`. The square root of `t` is 5, which is an integer and less than or equal to our range limit of 5, and `(5)^2 = 25 = t`. Therefore, `(3, 4, 5)` forms a square triple and we increment `res` by 1.
6. We continue this process looking for other valid squares. When `a = 4, b = 3` we again encounter a valid square triple `(4, 3, 5)`. However, since these values (`a = 4, b = 3` and `a = 3, b = 4`) represent the same right-angled triangle, we consider this a single unique square triple.
7. After considering all possible pairs `(a, b)`, we find that there is only one unique square triple within the given range `1 - 5`, which is `(3, 4, 5)`.
8. Finally, we finish iterating and return the value of `res`, which is now equal to 1, the count of all valid unique square triples within the given range.

This walkthrough illustrates how the brute-force iterative solution finds valid square triples by examining every possible combination of `a` and `b` within the specified range and verifying that `c` is an integer within that range as well. This method ensures that all potential solutions are checked, and the conditions for being a square triple are met before incrementing the result counter.

Solution Implementation

Python

```python
from math import sqrt

class Solution:
    def countTriples(self, n: int) -> int:
        # Initialize the result counter
        count = 0

        # Iterate through all possible pairs of numbers (a, b) up to n
        for a in range(1, n + 1):
            for b in range(1, n + 1):
                # Calculate the sum of squares of 'a' and 'b'
                sum_of_squares = a ** 2 + b ** 2
                # Take the square root of the sum to find 'c'
                c = int(sqrt(sum_of_squares))
                # Check if 'c' is an integer less than or equal to 'n' and if 'c' squared is equal to the original sum
                if c <= n and c ** 2 == sum_of_squares:
                    # If conditions are met, increment the result counter
                    count += 1

        # Return the total count of Pythagorean triples
        return count
```

Java

```java
class Solution {
    // This method counts the number of ways we can find (a, b, c)
    // such that a^2 + b^2 = c^2 with a, b, c <= n
    public int countTriples(int n) {
        int result = 0; // Initialize result count to 0

        // Iterate over all pairs of numbers (a, b)
        for (int a = 1; a <= n; ++a) {
            for (int b = 1; b <= n; ++b) {
                // Calculate the sum of squares of a and b
                int sumOfSquares = a * a + b * b;
                // Find the square root of the sum which should be integer if a^2 + b^2 = c^2
                int c = (int) Math.sqrt(sumOfSquares);
                // Check if c is within the limit and whether the square of c equals the sum
                if (c <= n && c * c == sumOfSquares) {
                    // If both conditions hold, increment result
                    ++result;
                }
            }
        }
        // Return the total count of triples found
        return result;
    }
}
```

C++

```cpp
class Solution {
public:
    int countTriples(int n) {
        int result = 0; // Initialize counter for the number of triples
        // Iterate over all possible values for 'a'
        for (int a = 1; a <= n; ++a) {
            // Nested iteration for all possible values for 'b'
            for (int b = 1; b <= n; ++b) {
                // Calculate the sum of squares of 'a' and 'b'
                int sumOfSquares = a * a + b * b;
                // Find the square root of the sumOfSquares, truncate towards zero
                int c = static_cast<int>(sqrt(sumOfSquares));
                // If 'c' is within the bound and its square is equal to the sumOfSquares
                if (c <= n && c * c == sumOfSquares) {
                    ++result; // Increment the counter
                }
            }
        }
        return result; // Return the final count of triples
    }
};
```

TypeScript

```typescript
// Function to count the number of Pythagorean triples up to a given limit 'n'
function countTriples(n: number): number {
    let result: number = 0; // Initialize counter for the number of triples

    // Iterate over all possible values for 'a'
    for (let a: number = 1; a <= n; ++a) {
        // Nested iteration for all possible values for 'b'
        for (let b: number = 1; b <= n; ++b) {
            // Calculate the sum of squares of 'a' and 'b'
            let sumOfSquares: number = a * a + b * b;
            // Find the square root of sumOfSquares, using Math.floor to trim the decimal part
            let c: number = Math.floor(Math.sqrt(sumOfSquares));
            // If 'c' is within the bound and its square is equal to the sumOfSquares
            if (c <= n && c * c === sumOfSquares) {
                result++; // Increment the counter
            }
        }
    }

    return result; // Return the final count of triples
}
```

```python
from math import sqrt

class Solution:
    def countTriples(self, n: int) -> int:
        # Initialize the result counter
        count = 0

        # Iterate through all possible pairs of numbers (a, b) up to n
        for a in range(1, n + 1):
            for b in range(1, n + 1):
                # Calculate the sum of squares of 'a' and 'b'
                sum_of_squares = a ** 2 + b ** 2
                # Take the square root of the sum to find 'c'
                c = int(sqrt(sum_of_squares))
                # Check if 'c' is an integer less than or equal to 'n' and if 'c' squared is equal to the original sum
                if c <= n and c ** 2 == sum_of_squares:
                    # If conditions are met, increment the result counter
                    count += 1

        # Return the total count of Pythagorean triples
        return count
```

Time and Space Complexity

The time complexity of the given code is `O(n^2)` since there are two nested for-loops iterating from 1 to `n`. Each iteration performs a constant amount of work, checking if `c` is less than or equal to `n` and if `c**2` is equal to `t`.

The space complexity is `O(1)` because the space used does not grow with the input size `n`. Only a constant amount of extra space is used for variables `res`, `a`, `b`, `t`, and `c`. There are no data structures that grow with the size of the input.