633. Sum of Square Numbers

Medium Math Two Pointers Binary Search Leetcode Link

Problem Description

The problem is about determining if it's possible to find two non-negative integers `a` and `b`, such that when you square each of them and add them together, the result is equal to a given non-negative integer `c`. This can be expressed with the equation `a^2 + b^2 = c`. You are required to check if there exists at least one pair `(a, b)` that satisfies this equation for the given `c`.

Intuition

The intuition behind the solution is based on the properties of a right-angled triangle where the squares of the two shorter sides sum up to the square of the longest side (Pythagorean theorem). In this problem, the non-negative integers `a` and `b` are similar to the two shorter sides, and `c` is like the square of the longest side.

The solution approach is inspired by binary search, which is typically used to find an element in a sorted array. Rather than searching through all possible pairs of `a` and `b`, which would be inefficient, the algorithm starts with two potential candidates: `a` starting from 0 and `b` from the square root of `c`, since `b` can't be larger than `sqrt(c)` if `b^2` is to stay less than or equal to `c`.

The while loop then examines the sum of squares of `a` and `b`. If the sum equals `c`, the answer is `True`. If the sum is less than `c`, `a` is incremented to increase the sum, exploring the possibility of larger `a^2` contributions. If the sum is greater than `c`, `b` is decremented to decrease the sum, as `b` might be too large.

This searching process ends when `a` becomes greater than `b` because, at that point, all pairs that could potentially add up to `c` have already been tested. If no such pair has been found by then, the algorithm returns `False`, indicating no such pair exists for the given `c`.

Solution Approach

The implementation uses a two-pointer technique that starts with the two pointers at the minimal and maximal possible values of `a` and `b` that could satisfy the equation `a^2 + b^2 = c`. Pointer `a` starts from 0, as it represents the smallest possible square contributing to `c`. Pointer `b` starts from `int(sqrt(c))`, which is the largest integer that when squared does not exceed `c`.

The algorithm's core logic is a loop that continues to execute as long as `a` is less than or equal to `b`. During each iteration, it computes the sum `s` of the squares of `a` and `b`. This sum `s = a^2 + b^2` is then compared to the target sum `c`:

- If `s` is equal to `c`, then we know that such a pair `(a, b)` exists, and the function immediately returns `True`.
- If `s` is less than `c`, the sum is too small, and `a` is incremented by 1 to try a larger value for `a^2`.
- If `s` is greater than `c`, the sum is too large, and `b` is decremented by 1 to try a smaller value for `b^2`.

The loop continues this process, incrementing `a` or decrementing `b` as necessary, stopping when `a` exceeds `b`. If the loop concludes without finding a matching pair, the function returns `False`, indicating there are no two perfect square numbers that add up to `c`.

This algorithm follows an approach similar to a binary search, as the reference approach indicates, where instead of searching in a sorted array, it "searches" through a conceptualized sorted space of squared numbers, moving one of the boundaries (either `a` or `b`) closer to a potential solution at each step.

The reason why the algorithm stops once `a` exceeds `b` is because we're dealing with non-negative integers, and squares of integers grow very quickly. Once `a > b`, the possible sums of squares will no longer decrease (since `a^2` is now greater than `b^2`), which means we can be certain no subsequent iterations will result in the sum `c`.

The efficiency and elegance of this solution lie in its O(sqrt(c)) time complexity, as `b` is reduced from `sqrt(c)` to 0 in the worst case, and `a` is increased from 0 to `sqrt(c)` in the worst case. This is much more efficient than a brute-force approach that would be O(c), where we would have to check every pair `(a, b)`.

Example Walkthrough

To illustrate the solution approach, let's work through a small example with `c = 5`.

Initially, we start with two pointers for `a` and `b` such that:

- `a` starts from `0`, as it represents the smallest possible square contributing to `c`.
- `b` starts from `int(sqrt(c))`, which is `2` in our example because `2^2 = 4` is the largest square less than or equal to `c`.

Now, let's begin the process:

1. In the first iteration, the sum of squares `s` will be `a^2 + b^2 = 0^2 + 2^2 = 0 + 4 = 4`. Since `s` is less than `c`, we need to increase `a`. We increment `a` to `1`.

2. In the second iteration, `s` will be `a^2 + b^2 = 1^2 + 2^2 = 1 + 4 = 5`, which is exactly equal to `c`. At this point, we have found that the pair `(a, b) = (1, 2)` satisfies the equation `a^2 + b^2 = c`. The algorithm returns `True`.

In this example, we found that `c` can indeed be expressed as the sum of two squared integers. The algorithm works efficiently and quickly identifies the correct pair without needing to check every single possibility.

If we had a larger `c` where no squares add up to `c`, the algorithm would continue incrementing `a` and decrementing `b` until `a` surpasses `b`. If no valid pair is found by the time `a` exceeds `b`, the algorithm will return `False`, indicating no such pair exists for the given `c`.

Python Solution

```python
from math import sqrt

class Solution:
    def judgeSquareSum(self, target: int) -> bool:
        # Initialize two pointers. `left` starts at 0, and `right` starts at the square root of target,
        # truncated to an integer, which is the largest possible value for a or b if a^2 + b^2 == target.
        left, right = 0, int(sqrt(target))

        # Loop until the two pointers meet
        while left <= right:
            # Calculate the sum of squares of the two pointers
            current_sum = left ** 2 + right ** 2

            # If the sum equals the target, we found a solution
            if current_sum == target:
                return True
            # If the sum is less than the target,
            # increment the left pointer to try a larger square for `a^2`
            elif current_sum < target:
                left += 1
            # If the sum is greater than the target,
            # decrement the right pointer to try a smaller square for `b^2`
            else:
                right -= 1

        # If no pair of squares was found that sums up to the target, return False
        return False

# Example usage:
# solution = Solution()
# print(solution.judgeSquareSum(5))  # Output: True
# print(solution.judgeSquareSum(3))  # Output: False

```

Java Solution

```java
class Solution {
    public boolean judgeSquareSum(int c) {
        // Initialize two pointers. 'smallest' starts at 0 and 'largest' starts at the square root of 'c'
        long smallest = 0;
        long largest = (long) Math.sqrt(c);

        // Use a two-pointer approach to find if there exist two numbers 'a' and 'b'
        // such that a^2 + b^2 equals to 'c'
        while (smallest <= largest) {
            // Calculate the sum of squares of 'smallest' and 'largest'
            long sumOfSquares = smallest * smallest + largest * largest;

            // Check if the current sum of squares equals to 'c'
            if (sumOfSquares == c) {
                // If yes, then we found that 'c' can be expressed as a sum of squares.
                return true;
            }

            // If the sum of squares is less than 'c', we increment 'smallest' to get a larger sum
            if (sumOfSquares < c) {
                ++smallest;
            } else {
                // If the sum of squares is greater than 'c', we decrement 'largest' to get a smaller sum
                --largest;
            }
        }

        // If we exit the loop, then there are no such 'a' and 'b' that satisfy a^2 + b^2 = 'c'
        return false;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This method checks if the input 'c' can be expressed as the sum of squares of two integers.
    bool judgeSquareSum(int c) {
        // Start one pointer 'a' from the smallest possible square, 0.
        long a = 0;
        // Start another pointer 'b' from the largest possible square that is less than or equal to 'c'.
        long b = static_cast<long>(sqrt(c));

        // Continue the search as long as 'a' is less than or equal to 'b'.
        while (a <= b) {
            // Calculate the sum of the squares of 'a' and 'b'.
            long sumOfSquares = a * a + b * b;

            // If the sum equals to 'c', we found the two numbers whose squares sum up to 'c'.
            if (sumOfSquares == c) return true;

            // If the sum is less than 'c', then we need to increase 'a' to get a larger sum.
            if (sumOfSquares < c)
                ++a;
            // If the sum is greater than 'c', we need to decrease 'b' to get a smaller sum.
            else
                --b;
        }

        // If no pair of integers has been found whose squares sum up to 'c', return false.
        return false;
    }
};

```

Typescript Solution

```typescript
// This function checks if a given number c can be written as the sum
// of the squares of two integers (c = a^2 + b^2).
function judgeSquareSum(c: number): boolean {
    // Starting with a at the smallest square (0) and b at the largest
    // square not greater than the square root of c.
    let a = 0;
    let b = Math.floor(Math.sqrt(c));

    // Loop until a and b meet or cross each other.
    while (a <= b) {
        // Calculating the sum of squares of both a and b.
        let sumOfSquares = a ** 2 + b ** 2;

        // If the sum of squares is equal to c, we found a valid pair.
        if (sumOfSquares == c) {
            return true;
        }

        // If the sum of squares is less than c, we need to try a larger value of a.
        if (sumOfSquares < c) {
            ++a; // Increment a to increase the sum.
        } else {
            --b; // Decrement b to decrease the sum.
        }
    }

    // If we exit the loop, no valid pair was found that fulfills the condition.
    return false;
}

```

Time and Space Complexity

The time complexity of the given code is `O(sqrt(c))` because the while loop runs with `a` starting from `0` and `b` starting from `int(sqrt(c))`, moving closer to each other with each iteration. The loop terminates when `a` exceeds `b`, and since `b` decrements with a magnitude that is at most the square root of `c` (as that is its starting value), we can bound the number of iterations by `2*sqrt(c)` (since both `a` and `b` can move `sqrt(c)` times at most).

The space complexity of the code is `O(1)` because the space used does not scale with the value of `c`. Only a constant amount of additional memory is used for the variables `a`, `b`, and `s`.