367. Valid Perfect Square

Easy Math Binary Search

Problem Description

The problem is straightforward: given a positive integer `num`, our task is to determine whether it is a perfect square without using any built-in library functions, such as `sqrt`. A perfect square is a number that can be expressed as the product of an integer with itself. For example, the number 16 is a perfect square because it can be expressed as 4 × 4.

Intuition

To solve this problem, we can use two different methods - Binary Search and the Math Trick.

Method 1: Binary Search

The Binary Search approach involves setting two pointers `left` and `right`, where `left` starts at 1 (the smallest perfect square) and `right` starts at `num` (as the largest possible perfect square our input could be). We iteratively narrow down the search range by finding the midpoint of `left` and `right` and squaring it. If the square of this midpoint is larger than or equal to `num`, we know our perfect square root, if it exists, is at or before this midpoint, so we move the `right` pointer to the midpoint. Otherwise, we move `left` up to `mid + 1`. The moment `left` and `right` converge, we check if the square of `left` is equal to `num` to conclude whether `num` is a perfect square.

Method 2: Math Trick

This method uses the observation that every perfect square is the sum of a sequence of odd numbers starting from 1. We keep adding sequentially larger odd numbers to a sum. This sum starts at 0, and we increase the odd number to add by 2 each time. Whenever the sum equals the number `num`, we confirm that `num` is a perfect square. The underlying math of this trick is that the sum of the first `n` odd numbers is `n^2`, which is exactly the definition of a perfect square.

Both methods have an upper time complexity of O(log n), however, the math trick can sometimes conclude that a number isn't a perfect square more quickly since not all numbers are perfect squares and the sum can exceed `num` before we've added `n` terms.

The solution implements two approaches to determine if a number is a perfect square without using the built-in `sqrt` function.

Method 1: Binary Search Approach

In the binary search approach, we use the concept of a binary search algorithm to efficiently find the target perfect square, if it exists. We start by initializing two pointers: `left` at 1 (since 1 is the smallest square) and `right` at `num` (since a number cannot be a perfect square of any number larger than itself).

Here's the step-by-step binary search algorithm applied to this problem:

- While `left` is less than `right`, perform the following steps to narrow down the search space:
  - Calculate the midpoint `mid` by averaging `left` and `right` (using bitwise shifting `>> 1` to divide by 2 for efficiency).
  - Multiply `mid` by itself to check if it gives `num`.
  - If `mid * mid` is greater than or equal to `num`, we set `right` to `mid`. This is because if `mid` squared is larger than or equal to `num`, the number we're looking for, if it exists, cannot be greater than `mid`.
  - If `mid * mid` is less than `num`, we set `left` to `mid + 1`. This is because the number we're looking for must be larger than `mid`.
- After the loop, we check if `left * left` equals `num` to determine if `num` is indeed a perfect square.

The binary search approach ensures that we can quickly zone in on the potential candidate for the square root of the number and confirm if it's a perfect square in O(log(n)) time complexity.

Method 2: Math Trick

The math trick approach makes use of a mathematical pattern where every perfect square can be represented as a sum of odd numbers sequentially. For instance:

- `1 = 1`
- `4 = 1 + 3`
- `9 = 1 + 3 + 5`
- ...

This pattern can be continued indefinitely, and each time the resulting sum will be a perfect square.

The algorithm for this approach is as follows:

- Initialize a variable `sum` as 0 to keep track of the sum of odd numbers, and another variable `i` representing the current odd number to add to the sum, starting at 1.
- While `sum` is less than `num`, add `i` to `sum` and check if `sum` equals `num`:
  - If `sum` becomes equal to `num`, then `num` is a perfect square and we return `true`.
  - If `sum` is still less than `num`, increment `i` by 2 to get to the next odd number.
- If the loop ends without returning `true`, then `num` isn't a perfect square, return `false`.

This method runs in O(sqrt(n)) time complexity since in the worst case, it adds up the sequence of odd numbers up to the square root of the number.

Both of these approaches efficiently determine whether a number is a perfect square without using any built-in square root function, providing a reliable way to solve the problem with a guaranteed logarithmic or square root time complexity.

Example Walkthrough

Let's illustrate the solution using the number `num = 16` to determine whether it is a perfect square through both methods.

Binary Search Approach Example with `num = 16`:

- Set `left` to 1 and `right` to `num` (16).
- While `left < right`:
  - Calculate `mid` which is `(left + right) / 2`, so initially `(1 + 16) / 2 = 8.5`, we take the integer part and consider `mid = 8`.
  - Now, check `mid * mid` which equals `8 * 8 = 64`. This is greater than `num` (16), so set `right = mid` to 8.
- Update the while loop, and calculate the new `mid` as `(1 + 8) / 2 = 4.5`, consider `mid = 4`.
  - Check `mid * mid` which is `4 * 4 = 16`. This equals `num` (16), break the loop and confirm that `num` is a perfect square.

In a real run, the loop would continue until `left` and `right` converge to the point where `left == right`. However, in this case, we've found that 16 is a perfect square of 4 during the process. The other steps are not performed because we already found a match.

Math Trick Approach Example with `num = 16`:

- Initially, `sum = 0` and `i = 1` (the first odd number).
- Add `i` to `sum`, `sum` becomes `1`. Then increment `i` by 2 to get `3`.
- Now, `sum` is `1` and `i` is `3`. Add `i` to `sum`, `sum` becomes `1 + 3 = 4`. Increment `i` by 2 to get `5`.
- Continuing, add the new `i` to `sum` to get `4 + 5 = 9`. Increment `i` by 2 to get `7`.
- Add `7` to `sum` to get `9 + 7 = 16`, which equals `num`.
- Since the sum now equals `num`, we can assert that `num` is indeed a perfect square.

Through both methods, we have confirmed that 16 is a perfect square. The binary search approached the conclusion more directly by halving the possible range, while the math trick added sequential odd numbers until the sum matched the input (`num`).

Solution Implementation

Python

```python
class Solution:
    def isPerfectSquare(self, num: int) -> bool:
        # Initialize the binary search boundaries.
        left, right = 1, num

        # Use binary search to find the potential square root of the number.
        while left < right:
            # Calculate the middle point of the current search boundary.
            mid = (left + right) // 2

            # If the square of mid is greater than or equal to num, we narrow the search space
            # to the left half including mid.
            if mid * mid >= num:
                right = mid
            # Otherwise, we narrow the search space to the right half excluding mid.
            else:
                left = mid + 1

        # After the loop, left will be equal to right and should be the smallest number
        # whose square is greater than or equal to num.
        # Check if it's a perfect square of num.
        return left * left == num
```

Java

```java
class Solution {
    // Method to check if a given number is a perfect square
    public boolean isPerfectSquare(int num) {
        long left = 1;              // Set the lower bound of the search range
        long right = num;           // Set the upper bound of the search range

        // Binary search to find the square root of num
        while (left < right) {
            // Calculate the midpoint to avoid overflow
            long mid = (left + right) >>> 1;

            // If mid squared is greater than or equal to num, it could be the root
            if (mid * mid >= num) {
                right = mid;        // Adjust the upper bound for the next iteration
            } else {
                left = mid + 1;     // Adjust the lower bound if mid squared is less than num
            }
        }

        // Check if the final left value squared equals the original number to confirm if it's a perfect square
        return left * left == num;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to check if a given number is a perfect square
    bool isPerfectSquare(int num) {
        long left = 1;           // Initializing the lower boundary of the search space
        long right = num;        // Initializing the upper boundary of the search space

        // Using binary search to find the square root of the number
        while (left < right) {
            long mid = left + (right - left) / 2; // Calculating the mid-value to prevent overflow
            // If mid squared is greater than or equal to num, we narrow down the upper boundary
            if (mid * mid >= num) {
                right = mid;
            } else {
                // If mid squared is less than num, we narrow down the lower boundary
                left = mid + 1;
            }
        }

        // Once left and right converge, we verify if the number is indeed a perfect square
        return left * left == num;
    }
};
```

TypeScript

```typescript
/**
 * Checks whether a given number is a perfect square or not.
 *
 * @param {number} num - The number to check.
 * @returns {boolean} - True if num is a perfect square, false otherwise.
 */
function isPerfectSquare(num: number): boolean {
    // Initialize the search range
    let left: number = 1;
    let right: number = num >> 1; // Equivalent to Math.floor(num / 2)

    // Perform binary search to find the square root of num
    while (left < right) {
        // Calculate the midpoint of the current search range, using bitwise shift for division by 2
        const mid: number = (left + right) >>> 1;

        // Compare the square of the mid value with num
        if (mid * mid < num) {
            left = mid + 1; // If mid^2 is less than num, narrow the range to the upper half
        } else {
            right = mid; // If mid^2 is greater or equal to num, narrow the range to the lower half, including mid
        }
    }

    // After the loop, left should be the integer part of the square root if it exists.
    // Check if the square of 'left' is exactly num to conclude if num is a perfect square.
    return left * left === num;
}
```

```python
class Solution:
    def isPerfectSquare(self, num: int) -> bool:
        # Initialize the binary search boundaries.
        left, right = 1, num

        # Use binary search to find the potential square root of the number.
        while left < right:
            # Calculate the middle point of the current search boundary.
            mid = (left + right) // 2

            # If the square of mid is greater than or equal to num, we narrow the search space
            # to the left half including mid.
            if mid * mid >= num:
                right = mid
            # Otherwise, we narrow the search space to the right half excluding mid.
            else:
                left = mid + 1

        # After the loop, left will be equal to right and should be the smallest number
        # whose square is greater than or equal to num.
        # Check if it's a perfect square of num.
        return left * left == num
```

Time and Space Complexity

The time complexity of the given binary search algorithm is `O(log n)`, where `n` is the value of the input `num`. This is because the algorithm effectively halves the search space with each iteration by updating either the `left` or `right` variable to the `mid` value.

The space complexity of the algorithm is `O(1)` since it uses a fixed amount of extra space - variables `left`, `right`, `mid`, and the space needed for a few calculations do not depend on the size of the input `num`.