507. Perfect Number

Easy Math

Leetcode Link

Problem Description

The problem requires us to determine whether a given integer `n` is a **perfect number** or not. A perfect number is defined as a positive integer which is equal to the sum of its positive divisors, excluding the number itself. This means we need to find all the integers that can divide `n` without leaving a remainder, sum them up, and compare this sum to `n` to confirm if `n` is a perfect number. Examples of perfect numbers are 6, 28, and 496, where each of them equals the sum of their respective divisors (1, 2, 3 for 6; 1, 2, 4, 7, 14 for 28; etc.).

Intuition

When thinking about how to determine if a number is perfect, the straightforward approach is to iterate through all numbers less than `n` to find its divisors and calculate the sum. However, this is not efficient for large values of `n`. To optimize this process, we can take advantage of two key insights:

1. Divisors come in pairs. For a divisor `i`, there's often a corresponding divisor `n/i` (except when `i` is the square root of `n`, in which case it should not be counted twice).

2. We only need to iterate up to the square root of `n`. This is because if `n` has a divisor larger than its square root, the corresponding smaller divisor has already been accounted for in the pair.

The provided solution uses these insights to iterate only up to the square root of `n`, adding both `i` and `n/i` to the sum `s` anytime `i` is a divisor of `n`. Special care is taken for the square root case so it is not added twice. After the loop, if the sum `s` equals `n`, it is indeed a perfect number, and the function returns `true`. Otherwise, it returns `false`.

It's also important to note that `1` is not considered a perfect number as per the problem definition and is immediately returned as `false`.

Solution Approach

The solution is implemented with a simple function that checks for positive divisors using a while loop. Here are the steps and algorithms used in the provided solution:

1. **Special Case Check**: Since `1` is not a perfect number (it does not meet the definition's requirement), the function immediately returns `false` if the number is `1`.

2. **Initialization**: Two variables are initialized - `s` is set to `1` (as `1` is always a divisor for any number) and `i` is set to `2`. Here, `s` will hold the sum of the divisors, and `i` is used to find potential divisors starting from `2`.

3. **Loop Through Potential Divisors**: The solution employs a `while` loop that runs as long as `i` squared is less than or equal to `num`. This effectively checks all potential divisors up to the square root of the number.

4. **Check for Divisibility**: Inside the loop, `num % i == 0` checks if `i` is a divisor. If it is, `i` is added to the sum `s`.

5. **Find and Add the Pair Divisor**: When a divisor is found, its corresponding pair (`num // i`) is also considered. If the pair divisor is not equal to `i` themselves (to avoid adding square roots twice), it is also added to the sum `s`.

6. **Increment to Next Potential Divisor**: The variable `i` is incremented by `1` to check the next potential divisor.

7. **Final Comparison**: After the loop concludes, the sum of the divisors `s` is compared to the original number `num`. If they are equal, it means that the number is a perfect number, and `true` is returned. Else, `false` is returned.

This approach is efficient since it reduces the number of iterations significantly compared to the brute-force method of checking all numbers less than `n`. Furthermore, no additional data structures are used, keeping the space complexity at O(1), and the time complexity is O(√n) since the loop runs up to the square root of `num`.

Example Walkthrough

Let's walk through a small example using the solution approach. Suppose we are given `n = 28` and we want to determine if it is a perfect number.

1. **Special Case Check**: We check if `n` is `1`. In this case, `n` is `28`, so we do not return `false` immediately.

2. **Initialization**: We initialize `s` to `1`, since `1` is a divisor of every number. We also set `i` to `2` because we'll start checking for divisors from this number.

3. **Loop Through Potential Divisors**: We start the `while` loop, where `i` will go from `2` up to the square root of `28`. The square root of `28` is approximately `5.29`, so our loop will run until `i` is less than or equal to `5`.

4. **Check for Divisibility and Add Divisors**: We check each `i` (from `2` to `5`) to see if `28 % i == 0`. This will be true for `i = 2`, `i = 4`, and `i = 7`. When it's true, we add `i` to `s`, and if `i` is not the square root of `28`, we also add the pair divisor (`n / i`):

   - For `i = 2`, `n % i == 28 % 2 == 0`; we add `2` to `s`. The pair divisor is `28 / 2 = 14`, which we also add to `s`. Now, `s = 1 + 2 + 14 = 17`.
   - For `i = 3`, `n % i == 28 % 3 != 0`; we do not add anything to `s`.
   - For `i = 4`, `n % i == 28 % 4 == 0`; we add `4` to `s`. The pair divisor is `28 / 4 = 7`, which we also add to `s`. Now, `s = 17 + 4 + 7 = 28`.
   - We do not need to check `i = 5` because `28 % 5 != 0`.
5. **Increment to Next Potential Divisor**: The loop continues, and `i` is incremented till `i > 5`, at which point we stop the loop.

6. **Final Comparison**: The sum `s` is now `28`, which is equal to the original number `n`. Thus, we conclude that `28` is a perfect number and return `true`.

Throughout this process, we've checked far fewer divisors than if we had run a loop from `2` to `27`, which illustrates the efficiency of the solution approach. The approach brings down the complexity significantly and works well for checking if a number is a perfect number.

Python Solution

```python
class Solution:
    def checkPerfectNumber(self, num: int) -> bool:
        # Perfect numbers are positive integers that are equal to the sum of their proper divisors.
        # The smallest perfect number is 6, which is the sum of its divisors 1, 2, and 3.

        # A perfect number must be at least 2, as 1 cannot be perfect.
        if num == 1:
            return False

        # Initialize the sum of divisors with the first proper divisor, which is always 1.
        sum_of_divisors = 1
        # Start checking for other divisors from 2 onwards.
        divisor = 2

        # Only iterate up to the square root of num to avoid redundant calculations.
        # Any divisor greater than the square root would have already been accounted for.
        while divisor * divisor <= num:
            # If divisor is a proper divisor of num,
            # then num is divisible by divisor without a remainder.
            if num % divisor == 0:
                # Add the divisor to the sum.
                sum_of_divisors += divisor
                # If divisor is not the square root of num,
                # add the corresponding co-divisor (num / divisor).
                if divisor != num // divisor:
                    sum_of_divisors += num // divisor
            # Proceed to check the next potential divisor.
            divisor += 1

        # A number is perfect if the sum of its proper divisors,
        # including 1 but not the number itself, equals the number.
        return sum_of_divisors == num

```

Java Solution

```java
class Solution {
    public boolean checkPerfectNumber(int num) {
        // If the number is 1, it's not a perfect number by definition
        if (num == 1) {
            return false;
        }

        int sumOfDivisors = 1; // Start with 1 since it's a divisor of every number

        // Loop through possible divisors from 2 to sqrt(num)
        for (int i = 2; i * i <= num; ++i) {
            // If i is a divisor of num
            if (num % i == 0) {
                sumOfDivisors += i; // Add the divisor
                // Add the complement divisor only if it's different from i
                if (i != num / i) {
                    sumOfDivisors += num / i;
                }
            }
        }

        // A perfect number equals the sum of its divisors (excluding itself)
        return sumOfDivisors == num;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to check if a number is a perfect number
    bool checkPerfectNumber(int num) {
        // A perfect number must be greater than 1 because the sum starts from 1
        if (num == 1) return false;

        // Initialize sum of divisors as 1, considering 1 is always a divisor
        int sumOfDivisors = 1;

        // Iterate over possible divisors from 2 up to the square root of num
        for (int divisor = 2; divisor * divisor <= num; ++divisor) {
            // Check if divisor is a factor of num
            if (num % divisor == 0) {
                // Add the divisor to the sum
                sumOfDivisors += divisor;

                // If the divisor is not the square root of num, add its counterpart
                if (divisor != num / divisor) sumOfDivisors += num / divisor;
            }
        }

        // Check if the sum of divisors equals the original number
        return sumOfDivisors == num;
    }
};

```

Typescript Solution

```typescript
// Function to check if a number is a perfect number
function checkPerfectNumber(num: number): boolean {
    // A perfect number must be greater than 1 because the sum starts from 1
    if (num === 1) return false;

    // Initialize sum of divisors as 1, considering 1 is always a divisor
    let sumOfDivisors: number = 1;

    // Iterate over possible divisors from 2 up to the square root of num
    for (let divisor = 2; divisor * divisor <= num; divisor++) {
        // Check if divisor is a factor of num
        if (num % divisor === 0) {
            // Add the divisor to the sum
            sumOfDivisors += divisor;

            // If the divisor is not the square root of num, add its counterpart
            if (divisor !== num / divisor) sumOfDivisors += num / divisor;
        }
    }

    // Check if the sum of divisors equals the original number
    return sumOfDivisors === num;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(sqrt(n))`. This is because the loop runs from 2 up to the square root of `num`. The check for each divisor only includes numbers less than or equal to the square root of `num` since any larger divisor would have been encountered as a smaller pair divisor before reaching the square root.

Space Complexity

The space complexity of the code is `O(1)`, which means it's constant. This is due to the fact that the code only uses a finite number of variables (`s`, `i`, `num`) that do not depend on the size of the input `num`.