2507. Smallest Value After Replacing With Sum of Prime Factors

Medium | Math | Number Theory | Leetcode Link

Problem Description

The problem involves manipulating a given positive integer `n` by continuously replacing it with the sum of its prime factors. A key detail is that if a prime factor is repeated in the factorization (that is, `n` is divisible by a prime number multiple times), that prime factor must be counted multiple times in the sum.

For example, if our given number `n` is 18, its prime factors are 2 and 3, but 3 is counted twice because 18 is divisible by 3 twice (18 = 2 * 3 * 3). So the sum of its prime factors is 2 + 3 + 3 = 8.

The goal is to repeat this process: take the sum of the prime factors of `n` and then replace `n` with this sum, and continue this until `n` no longer changes—it reaches its smallest possible value.

The challenge is to write an algorithm that performs this computation efficiently and find when the number `n` stops changing, which is the result to be returned.

Intuition

The solution makes use of a loop that continuously replaces the number `n` with the sum of its prime factors until `n` cannot be reduced any further.

We start with a loop that runs indefinitely, checking if we can find prime factors of `n` and their sums. In each iteration, we initialize three variables - `t` to keep track of the original value of `n` at the start of the iteration, `s` to calculate the sum of prime factors, and `i` to iterate over potential prime factors starting from 2.

The inner `while` loop checks if `i` is a divisor of `n`, and if so, it keeps dividing `n` by `i` and adds `i` to the sum `s` until `n` is no longer divisible by `i`. The variable `i` is incremented to check for the next possible prime factor.

Once we have tried all possible divisors up to `n // i`, we check if `n` itself is greater than 1 (which would mean `n` is a prime number and should be included in the sum).

At this point, if the calculated sum of prime factors `s` is equal to the temporary variable `t`, it means `n` can no longer be reduced any further, and we return `t`.

If `s` is not equal to `t`, we replace `n` with `s` to continue the process with a new reduced value of `n`.

The key intuition behind the algorithm is that the prime factorization of a number can help find a smaller representation of the number itself and by repeating this process and summing these factors repeatedly, we can converge to the smallest possible value of `n`.

Solution Approach

The solution makes use of a basic factorization algorithm and control flow to reduce the number `n` to its smallest possible value by continuously summing its prime factors.

Here's a step-by-step breakdown of the approach used in the solution:

1. The solution uses a `while` loop that will run until `n` no longer changes.
2. Inside the loop, a temporary variable `t` is assigned the current value of `n` to keep track of its value through each iteration. This is important to identify when there is no further change possible.
3. Another variable `s` is initialized to 0; this will be used to calculate the sum of the prime factors of `n`.
4. An index `i` is set to 2, which is the first prime number. This variable is used to test potential factors of `n`.
5. The first inner `while` loop runs as long as `i` is less than or equal to `n // i` (since a factor larger than the square root of `n` would have already been identified by its corresponding smaller factor except when `n` is prime).
6. A nested inner `while` loop checks if `i` divides `n` perfectly. If it does, `n` is divided by `i`, and `i` is added to the sum `s`. This loop continues until `n` is no longer divisible by `i`, accommodating for all instances of `i` as a factor.
7. The index `i` is then incremented to check the next potential factor.
8. After all possible factors up to `n // i` have been tested, a final check outside the inner loops evaluates if `n` itself is greater than 1, implying `n` is a prime number. If it is prime, it is added to the sum `s`.
9. At the end of the iteration, the algorithm checks if the sum of the prime factors `s` is equal to the starting value `t`. If so, `n` has reached its smallest value as no primes other than itself can be extracted and summed, and the algorithm returns the value of `t`.
10. If `s` is not equal to `t`, the algorithm replaces the value of `n` with `s` to repeat the factorization process on this new reduced number.

The algorithm essentially terminates when a number only comprises its prime self or when it's reduced to 1, both of which are indivisible and signaling the end of the reduction process.

This algorithm does not employ any complex data structures and follows a straightforward but effective pattern to reduce the number to its smallest form based on prime factor accumulation.

Example Walkthrough

Let's walk through the solution approach with a small example where our given number `n` is 12. The goal is to continuously replace `n` with the sum of its prime factors and repeat this process until `n` is no longer reduced.

Step 1: Start with `n = 12` and enter the `while` loop.

Step 2: Assign the value of `n` to a temporary variable `t`, so `t = 12`.

Step 3: Initialize a variable `s` to 0. This will hold the sum of the prime factors of `n`.

Step 4: Start with `i = 2`, which is the smallest prime factor.

Step 5: As `i <= n // i` (since `2 <= 12 // 2`), we proceed with factorization.

Step 6: The inner loop checks if `12` is divisible by `2`. It is, so we divide `n` by `2` to get `6` and add `2` to `s`. Now, `s = 2` and `n = 6`.

We continue with the inner loop since `6` is still divisible by `2`. We divide `6` by `2` to get `3` and add `2` to `s`. Now, `s = 4` and `n = 3`.

Step 7: Increment `i` to the next integer, which is `3`.

Step 8: Since `3` is a prime number larger than `n // i` but still divides `n` perfectly, we add `3` to `s`. Now, `s = 4 + 3` giving us `s = 7` and `n` becomes `1`.

Step 9: Since `n` is now `1`, we exit the inner loop. We check if `s` is equal to the temporary variable `t`. As `s = 7` and `t = 12`, they are not equal.

Step 10: Because `s` is not equal to `t`, we set `n` to `s`; hence `n` is now `7`.

Now `n` has changed from `12` to `7`. We repeat the entire process again with `n = 7`.

Step 1: `n` is `7`, so we enter the `while` loop.

Step 2: Set `t` to `7`.

Step 3: Initialize `s` to `0`.

Step 4: Start with `i = 2`.

Step 5: Begin inner loop. Because no `i` exists such that `i <= n // i` and `i` divides `7`, skip to step 8.

Step 8: Since `n` is still greater than `1`, and `7` is a prime number, we add `7` to `s`. Now `s = 7`.

Step 9: Compare the sum of the prime factors `s` with `t`. They are equal (`s = t = 7`), indicating that we cannot reduce `n` any further.

The algorithm would then terminate and return the value `7` as the result, as `n` can no longer be reduced by the process described.

Python Solution

```python
class Solution:
    def smallestValue(self, num: int) -> int:
        # Continue the loop until we find the smallest value
        while True:
            # Initialize temp variable to store original number, sum_of_factors, and start divisor from 2
            temp, sum_of_factors, divisor = num, 0, 2
            # Check for factors of the number starting with the smallest prime factor
            while divisor <= num // divisor:
                # If the divisor is a factor, divide num by the divisor and add to the sum_of_factors
                while num % divisor == 0:
                    num //= divisor
                    sum_of_factors += divisor
                # Move to the next potential factor
                divisor += 1
            # If there's a remaining number greater than one, it's a prime factor; add it to the sum_of_factors
            if num > 1:
                sum_of_factors += num
            # If the sum of factors is equal to the original number, we found the smallest value
            if sum_of_factors == temp:
                return temp
            # Set num to sum_of_factors for the next iteration to check the new number
            num = sum_of_factors

```

Java Solution

```java
class Solution {
    // Method to find the smallest value according to specified conditions
    public int smallestValue(int n) {
        // Loop indefinitely until we find the smallest value
        while (true) {
            // Store the original value of n
            int originalValue = n;
            // Initialize the sum of the factors
            int sumOfFactors = 0;
            // Start dividing the number from 2 onwards to find its factors
            for (int i = 2; i <= originalValue / i; ++i) {
                // Divide by i as long as it is a factor of n
                while (n % i == 0) {
                    sumOfFactors += i; // Add factor to the sum
                    n /= i; // Divide n by the factor
                }
            }
            // If there is a remaining factor greater than 1, add it to the sum
            if (n > 1) {
                sumOfFactors += n;
            }
            // Check if the sum of the factors equals the original number
            if (sumOfFactors == originalValue) {
                // If it matches, return the sum (as it is the smallest value)
                return sumOfFactors;
            }
            // If it does not match, set n to the sumOfFactors for another iteration
            n = sumOfFactors;
        }
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the smallest integer value with the same
    // sum of factors (including 1 and the number itself) as 'n'
    int smallestValue(int n) {
        // Loop indefinitely until we find the smallest value
        while (true) {
            int originalValue = n; // Preserve the original value of 'n'
            int sumOfFactors = 0;  // Initialize the sum of factors to 0

            // Factorize and sum up the factors
            for (int i = 2; i <= n / i; ++i) { // Only need to check up to sqrt(n)
                // While 'n' is divisible by 'i'
                while (n % i == 0) {
                    sumOfFactors += i; // Add the factor to the sum
                    n /= i; // Divide 'n' by the factor for further factorization
                }
            }

            // If there is a remaining prime factor greater than sqrt(n), add it to the sum
            if (n > 1) sumOfFactors += n;

            // If the sum of factors is the same as the original value, return it
            if (sumOfFactors == originalValue) {
                return sumOfFactors;
            }

            // Otherwise, set 'n' to the calculated sum of factors for the next iteration
            n = sumOfFactors;
        }
    }
};

```

Typescript Solution

```typescript
// Function to find the smallest integer value with the same
// sum of factors (including 1 and the number itself) as 'n'
function smallestValue(n: number): number {
    // Loop indefinitely until we find the smallest value
    while (true) {
        const originalValue: number = n; // Preserve the original value of 'n'
        let sumOfFactors: number = 1;  // Initialize the sum of factors to 1, since 1 is a factor of all numbers

        // Factorize and sum up the factors
        for (let i = 2; i <= Math.sqrt(n); ++i) { // Only need to check up to sqrt(n)
            // While 'n' is divisible by 'i'
            while (n % i === 0) {
                sumOfFactors += i; // Add the factor to the sum
                if (i != n / i) {
                    sumOfFactors += n / i; // Add the complement factor to the sum if it's different
                }
                n /= i; // Divide 'n' by the factor for further factorization
            }
        }

        // Check for a remaining prime factor greater than sqrt(n)
        if (n > 1 && n < originalValue) {
            sumOfFactors += n; // Add it to the sum only if it's different from the current value
        }

        // If the sum of factors is the same as the original value, return the smallest integer
        if (sumOfFactors == originalValue) {
            return originalValue;
        }

        n = sumOfFactors; // Set 'n' to the calculated sum of factors for the next iteration
    }
}

```

Time and Space Complexity

Time Complexity

The time complexity of this code is governed by two nested loops. The outer loop runs indefinitely until a number is found where the sum of its prime factors is equal to itself. The inner loop, on the other hand, is used for factorization and runs at most `sqrt(n)` times because it checks for factors from `2` to at most `sqrt(n)`.

The factorization loop:

- In the worst-case scenario for a given `n`, we may have to factorize `n`, then `s`, and so on if `n` is initially not a prime or semiprime (a product of exactly two primes). Each new `s` will be smaller than `n` as we are adding up the factors. Thus, the inner factorization process will take `O(sqrt(m))` time for each number `m` we factorize, where `m` starts from `n` and gets smaller.

The outer while loop:

- The code does not have a clear stopping condition within predictable bounds, due to the uncertain nature of the sum `s` approaching the target condition `s == t`. However, for every iteration, the sum of the prime factors of `n` (`s`) gets closer to being a prime number itself. Once `s` equals a prime number, it will become equal to `t` in the next iteration, and the function will return `t`. The number of iterations can be considered proportional to the number of distinct prime factors of `n` in a sense but is not easy to express in standard `O()` notation.

Considering the worst-case scenario, where `n` is a large composite number with many small prime factors, the time complexity can be high, but the exact upper bound is intricate to determine without more constraints on `n`.

Space Complexity

The space complexity is `O(1)`. There are only a few integer variables (`t`, `s`, `i`, and `n`) being used, which do not depend on the input size `n`, unless considering the size `n` itself needs. The space used by these variables is constant and does not grow with `n`.