1492. The kth Factor of n

Medium Math Number Theory [Leetcode Link]

Problem Description

You are given two integers `n` and `k`. The task is to find the `k`th smallest factor of `n`. A factor of `n` is defined as a number `i` such that when `n` is divided by `i`, there is no remainder, i.e., `n % i == 0`. The factors are considered in ascending order. If `n` does not have `k` factors, the function should return `-1`.

Intuition

The intuition behind the solution is to find all the factors of the given integer `n`, sort them in increasing order and then return the `k`th one in the sequence. Since the factors of a number are symmetric around the square root of the number, the solution starts by iterating through all numbers from 1 to the square root of `n`. For each number `i` that is a factor of `n` (meaning `n % i == 0`), it decrements `k` because it is one step closer to the `k`th smallest factor.

If `k` reaches 0 during this process, it returns the current factor, `i`. However, if `k` is still not 0 after this loop ends, the solution then uses a slightly different approach. It checks the symmetric factors by dividing `n` by `i` and continues decreasing `k` until it finds the `k`th factor or until there are no more factors to check. Again, if `k` reaches 0 during this process, it returns the current factor, `n // i`.

One important case to handle is when `n` is a perfect square. In such a case, the factor that is exactly the square root of `n` is only counted once, hence the second loop begins by decrementing `i` by 1, to avoid double-counting.

The solution efficiently finds the `k`th smallest factor by avoiding the generation of all factors, instead, all the while counting the position of each factor it finds until it reaches the `k`th one.

Solution Approach

The implementation of the solution involves a straightforward iteration through possible factors and can be broken down into two main stages:

1. **Finding Factors Less Than or Equal to Square Root:**
   - Initialize `i` to 1.
   - Loop until `i` squared is less than `n`:
     - Check if `i` is a factor using the modulus operation `n % i == 0`. If it is, decrement `k`.
     - If `k` reaches 0 within this loop, return `i` because you've found the `k`th factor.
     - Increment `i` to check the next potential factor.
2. **Finding Factors Greater Than Square Root:**
   - Before starting this loop, check if `n` is a perfect square by verifying if the square of `i-1` (last `i` from previous loop) is not equal to `n`; if it isn't, we reduce `i` by 1 to ensure we do not repeat the factor at the square root of `n` in case `n` is a perfect square.
   - Loop downwards from `i`:
     - Check for the other factor by dividing `n` by `i` and see if the modulo with `n // i` is 0 (`(n % (n // i)) == 0`).
     - As before, with each successful factor, decrement `k`.
     - If `k` reaches 0 within this loop, return the corresponding factor `n // i` because you've found the `k`th factor.
     - Decrement `i` and continue until `i` reaches 0.
3. **Returning -1:**
   - In case the total number of factors is less than `k`, return `-1`.

This approach uses no additional data structures, relying only on integer variables to track the current candidate factor and the countdown of `k` to reach the desired factor. It is efficient because it minimizes the number of iterations to potentially a little more than twice the square root of `n`. The algorithm capitalizes on the symmetry of factors and avoids a full enumeration of factors beyond the 'middle' factor (the square root), thus optimizing the search for the `k`th factor.

Example Walkthrough

Let's illustrate the solution approach with an example. Say we are given the integers `n = 12` and `k = 3`. Our goal is to find the 3rd smallest factor of 12.

According to the problem, the factors of 12 are 1, 2, 3, 4, 6, and 12. The 3rd smallest factor in this list is 3.

Step 1 - Finding Factors Less Than or Equal to Square Root:

- We start by initializing `i` to 1. Since 12 is not a perfect square, we will iterate up to its square root. The square root of 12 is approximately 3.46, so we'll consider all numbers up to 3.
- For `i = 1`, we check if it's a factor of 12 (`12 % 1 == 0`). It is, so we decrement `k` to 2.
- For `i = 2`, we check if it's a factor (`12 % 2 == 0`). It is, so we decrement `k` to 1.
- For `i = 3`, we check if it's a factor (`12 % 3 == 0`). It is, and since `k` is now 1, decrementing it will bring it to 0. We have found the 3rd smallest factor, so we can return `i` which is 3.

Since we found the `k`th smallest factor before exhausting all factors up to the square root of `n`, there is no need to proceed to Step 2.

If `k` had been greater, such as 5, the steps would continue as follows:

Step 2 - Finding Factors Greater Than Square Root:

- As `n = 12` is not a perfect square, we continue to look for factors greater than the square root. We had last checked `i = 3`, so we start with `i = 4`.
- We decrement `i` to 3 and loop downwards:
  - Check `i = 3`: Since we've already counted this factor in the first loop, we don't count it again.
  - Check `i = 2`: We find 12 // 2 = 6 is a factor (`12 % 6 == 0`). We decrement `k` to 0. We've found the 5th smallest factor, so we return `n // i` which is 6.

Step 3 - Returning -1:

- This step is not necessary for our example as we've found our `k`th factor. If `k` were larger than the number of factors 12 has, this step would ensure that we return `-1` to indicate that there is no `k`th factor. For instance, if `k = 8`, after checking all possible factors, `k` would not be 0, so we would return `-1`.

The solution approach has successfully found the 3rd smallest factor of 12 in a few simple steps. It’s efficient and effective for any pair of `n` and `k` provided.

Python Solution

```python
class Solution:
    def kthFactor(self, n: int, k: int) -> int:
        # Initialize a variable to count factors
        factor_count = 0
        # Iterate over potential factors from 1 up to the square root of n
        for potential_factor in range(1, int(n**0.5) + 1):
            # If the potential_factor is a factor of n
            if n % potential_factor == 0:
                # Increment the count of factors found
                factor_count += 1
                # If this is the k-th factor, return it
                if factor_count == k:
                    return potential_factor

        # If the loop completes, there are two possibilities:
        # Either k is too large (more than the number of factors), or
        # the k-th factor is the complement factor of a factor before the square root of n
        # To handle this, start decrementing potential_factor and check if its complement is a factor
        # If n is a perfect square, we need to avoid counting the square root twice
        if int(n**0.5)**2 == n:
            potential_factor = int(n**0.5) - 1
        else:
            potential_factor = int(n**0.5)

        # Iterate over remaining potential factors from the square root of n to 1
        for i in range(potential_factor, 0, -1):
            # Check if i is a factor by seeing if the division of n by i has no remainder
            if n % i == 0:
                # Decrease k as we are counting down from the total number of factors
                factor_count += 1
                # If this is the k-th factor, return its complement factor
                if factor_count == k:
                    return n // i

        # If no k-th factor was found, return -1
        return -1

```

Java Solution

```java
class Solution {
    // Method to find the k-th factor of a number n
    public int kthFactor(int n, int k) {
        // Starting from 1, trying to find factors in increasing order
        int factor = 1;
        for (; factor <= n / factor; ++factor) {
            // If 'factor' is a factor of 'n' and it's the k-th one found
            if (n % factor == 0 && (--k == 0)) {
                // Return 'factor' as the k-th factor of 'n'
                return factor;
            }
        }
        // Adjust 'factor' if we've surpassed the square root of 'n'
        // because we will look for factors in the opposite direction now
        if (factor * factor != n) {
            factor--;
        }
        // Starting from the last found factor, searching in decreasing order
        for (; factor > 0; --factor) {
            // Calculate the corresponding factor pair
            if (n % (n / factor) == 0) {
                // Decrease k for each factor found
                k--;
                // If we found the k-th factor from the largest end
                if (k == 0) {
                    // Return the factor as it's the k-th factor of 'n'
                    return n / factor;
                }
            }
        }
        // If no k-th factor is found, return -1
        return -1;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int kthFactor(int n, int k) {
        // Initialize the factor candidate
        int factor = 1;

        // Loop through potential factors starting from 1
        for (; factor < n / factor; ++factor) {
            // Check if current factor divides n without remainder
            if (n % factor == 0) {
                // Decrease k for each factor found
                --k;
                // If k reaches 0, the current factor is the kth factor
                if (k == 0) {
                    return factor;
                }
            }
        }

        // If the loop exited normally, check for a perfect square
        if (factor * factor == n) {
            // If n is a perfect square, we do not double count it
            --factor;
        }

        // Iterate backwards from the potential largest factor
        for (; factor > 0; --factor) {
            // Find the corresponding factor pair
            int correspondingFactor = n / factor;
            // Check if it divides n without remainder
            if (n % correspondingFactor == 0) {
                // Decrease k for each factor found
                --k;
                // If k reaches 0, this is the kth factor
                if (k == 0) {
                    return correspondingFactor;
                }
            }
        }

        // If kth factor does not exist, return -1
        return -1;
    }
};

```

Typescript Solution

```typescript
// Function to find the kth factor of n
function kthFactor(n: number, k: number): number {
    // Initialize the factor candidate
    let factor = 1;

    // Loop through potential factors starting from 1
    while (factor < n / factor) {
        // Check if the current factor divides n without remainder
        if (n % factor === 0) {
            // Decrease k for each factor found
            k--;

            // If k reaches 0, the current factor is the kth factor
            if (k === 0) {
                return factor;
            }
        }
        factor++;
    }

    // If the loop exited normally, check for a perfect square
    if (factor * factor === n) {
        // If n is a perfect square, we do not double count it
        factor--;
    }

    // Iterate backwards from the potential largest factor
    while (factor > 0) {
        // Find the corresponding factor pair
        let correspondingFactor = n / factor;

        // Check if it divides n without remainder
        if (n % correspondingFactor === 0) {
            // Decrease k for each factor found
            k--;

            // If k reaches 0, this is the kth factor
            if (k === 0) {
                return correspondingFactor;
            }
        }
        factor--;
    }

    // If the kth factor does not exist, return -1
    return -1;
}

```

Time and Space Complexity

The time complexity of the given code can be assessed by examining the two while loops that are run in sequence to find the `k`-th factor of the integer `n`.

The first loop runs while `i * i < n`, which means it will run approximately `sqrt(n)` times, because it stops when `i` is just less than the square root of `n`. Within this loop, the operation performed is a modulo operation to check if `i` is a factor of `n`, which is an `O(1)` operation. Therefore, the time complexity contributed by the first loop is `O(sqrt(n))`.

The second loop starts with `i` set to a value slightly less than `sqrt(n)` (assuming `n` is not a perfect square) and counts down to 1. For each iteration, it performs a modulo operation, which is `O(1)`. However, not every `i` will lead to an iteration because the counter is reduced only when `(n % (n // i)) == 0`, which corresponds to the outer factors of `n`. Since there are as many factors less than `sqrt(n)` as there are greater than `sqrt(n)`, we can expect the second loop also to contribute a time complexity of `O(sqrt(n))`.

Combining both loops, the overall time complexity remains `O(sqrt(n))`, as they do not compound on each other but are sequential.

The space complexity of the code is `O(1)` as there are only a finite number of variables used (`i`, `k`, `n`), and no additional space is allocated that would grow with the input size.