2521. Distinct Prime Factors of Product of Array

Medium Array Hash Table Math Number Theory

Leetcode Link

Problem Description

Given an array `nums` composed of positive integers, we are asked to determine the count of distinct prime factors present in the product formed by multiplying all the elements in `nums`. It's important to remember that a prime number is a number greater than 1 that has no divisors other than 1 and itself. Additionally, a prime factor of a number is a prime number that divides it without leaving a remainder. The task is to ensure we find each unique prime factor once, regardless of how many times it might divide different numbers in the array.

Intuition

The intuition behind the solution is to identify all prime factors for each number in the `nums` array and then collect these factors without duplicates to determine their total count.

Here's how we can approach the problem:

1. Initialize an empty set that will be used to store unique prime factors.
2. Loop through each number in the `nums` array.
3. For each number, find all prime factors by checking for divisibility from 2 onwards:
    - Start from 2 (the smallest prime) and try to divide the number.
    - If divisible, it's a prime factor, and we add it to the set.
    - Divide the number by this factor repeatedly until it's no longer divisible by this factor.
    - Increment the factor and repeat the process until the square of the factor is larger than the number.
4. If the remaining number is greater than 1, it means the number itself is a prime and should be included in the set of prime factors.
5. Finally, the distinct count of prime factors equals the size of the set holding these factors.

The brilliance of this approach lies in the fact that once a number in `nums` is divisible by a prime factor and we divide it out completely, the reduced number cannot be divisible by any previously checked factors. This means we do not have to worry about non-prime numbers during the factorization since they would've already been checked as multiples of the primes. Thus, the remaining number after this process is either 1 or a prime number.

Solution Approach

The implementation of the solution follows these steps:

1. Initialize a set named `s` which will be used to store distinct prime factors encountered across all numbers in `nums`. A set is chosen because it inherently prevents duplicate entries, ensuring that each prime factor is only counted once, regardless of how many times it divides numbers in the array.

2. Iterate through each number `n` in `nums`.

3. For the current number `n`, we initiate a loop to find its prime factors starting from `i = 2` (the smallest prime). The use of `while i <= n // i` as the loop condition is a key optimization that reduces the range we need to check. Since if `i` is greater than the square root of `n`, `i` cannot be a factor of `n`.

4. Inside the loop, we check if `n` is divisible by `i`:

    - If it is, `i` is a prime factor of `n`. We add `i` to the set `s`.
    - Then, we use a nested `while` loop to divide `n` by `i` repeatedly until `n` is no longer divisible by `i`. This step ensures that we eliminate all powers of `i` in `n`.

5. After exiting the inner `while` loop, we increment `i` by 1 to check the next potential prime factor.

6. Once the outer `while` loop completes, there may be a case where `n` (the reduced number) is greater than 1. This remainder `n` must be a prime number itself since all its other prime factors have already been divided out. So, we add `n` to the set `s`.

7. After the for-loop completes, all prime factors of all numbers in `nums` have been collected in the set `s`. We return the size of the set `s`, which gives us the count of distinct prime factors.

This solution approach leans heavily on the fact that every composite number has at least one prime factor less than or equal to its square root, allowing us to search only until `n // i` instead of `n`. This is a fundamental property of prime numbers and plays a crucial role in the efficiency of the algorithm. Additionally, the solution enforces the uniqueness of prime factors by using a set, and only adds factors to this set, ensuring each factor is only counted once no matter how many numbers it divides in the input array `nums`.

Example Walkthrough

Let's illustrate the solution approach using a simple example. Assume our `nums` array is `[12, 15]`.

1. We initialize the set `s` to store distinct prime factors encountered across all numbers in `nums`. Initially, `s = {}`.

2. We begin iterating through each number in `nums`. The first number `n` is `12`.

3. We start finding prime factors of `n = 12` beginning with `i = 2`.

    - We find that `12` is divisible by `2`, so we add `2` to set `s`. Now, `s = {2}`.
    - We keep dividing `12` by `2` to eliminate all powers of `2` in `n`. After this process, `n` becomes `3` (12 divided by 2 twice).
    - We increment `i` to `3` and find that `3` is divisible. So we add `3` to set `s`. Now, `s = {2, 3}`.
    - `n` is now `1`, so we are done with the prime factorization of `12`.

4. We move to the next number in `nums` which is `15`.

5. We start finding prime factors of `n = 15` starting with `i = 2`.

    - `15` is not divisible by `2`, so we increment `i` to `3`.
    - We find `15` is divisible by `3`, and since `3` is already in set `s`, we do not need to add it again. We divide `15` by `3` to remove all factors of `3`, and `n` becomes `5`.
    - We increment `i` to `4`, but since `4` is not prime, we go to `5`.
    - `5` divides `n` (now `5`), and we add `5` to the set `s` since it is not already there. Now, `s = {2, 3, 5}`.
    - `n` is now `1`, so we have completed the prime factorization for `15`.

6. All numbers in `nums` have been processed. The set `s` now contains all the distinct prime factors `{2, 3, 5}`.

7. The size of the set `s` represents the count of distinct prime factors. There are `3` distinct prime factors in the product of all numbers in `nums`.

Therefore, the answer is `3`—there are three distinct prime factors in the product of all numbers in the given `nums` array `[12, 15]`.

Python Solution

```python
from typing import List

class Solution:
    def distinctPrimeFactors(self, nums: List[int]) -> int:
        # Initialize an empty set to hold unique prime factors
        prime_factors_set = set()

        # Iterate over all the numbers in the list
        for number in nums:
            factor = 2  # Start checking for prime factors from the smallest prime

            # Use trial division to find prime factors of the number
            while factor <= number // factor:
                # If the factor divides the number, it is a prime factor
                if number % factor == 0:
                    prime_factors_set.add(factor)
                    # Divide the number by the prime factor until it is no longer divisible
                    while number % factor == 0:
                        number //= factor
                # Increment the factor by 1 to check the next possible prime factor
                factor += 1

            # If the remaining number is greater than 1, it is a prime factor itself
            if number > 1:
                prime_factors_set.add(number)

        # The result is the count of unique prime factors found in all numbers
        return len(prime_factors_set)

```

Java Solution

```java
class Solution {

    // Function to count the number of distinct prime factors among all numbers in the array
    public int distinctPrimeFactors(int[] nums) {
        // Initialize a set to store unique prime factors
        Set<Integer> uniquePrimeFactors = new HashSet<>();

        // Iterate over each number in the array
        for (int number : nums) {
            // Check for factors starting from 2 up to the square root of the number
            for (int factor = 2; factor <= number / factor; ++factor) {
                // If 'factor' is a divisor of 'number'
                if (number % factor == 0) {
                    // Add 'factor' to the set of unique prime factors
                    uniquePrimeFactors.add(factor);
                    // Remove all occurrences of this prime factor from 'number'
                    while (number % factor == 0) {
                        number /= factor;
                    }
                }
            }
            // If the remaining 'number' is a prime number greater than 1, add it to the set
            if (number > 1) {
                uniquePrimeFactors.add(number);
            }
        }

        // Return the size of the set, i.e., the number of distinct prime factors
        return uniquePrimeFactors.size();
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_set>
using namespace std;

class Solution {
public:
    // Function to calculate the number of distinct prime factors from an array of numbers.
    int distinctPrimeFactors(vector<int>& nums) {
        unordered_set<int> uniquePrimes; // Set to store unique prime factors.

        // Iterate through each number in the array.
        for (int& num : nums) {
            // Check for factors starting from 2 to the square root of the number.
            for (int i = 2; i <= num / i; ++i) {
                // If 'i' is a divisor of 'num', it could be a prime factor.
                if (num % i == 0) {
                    uniquePrimes.insert(i); // Insert the prime factor into the set.
                    // Divide 'num' by 'i' completely.
                    while (num % i == 0) {
                        num /= i;
                    }
                }
            }
            // If 'num' is still greater than 1, then it is a prime number itself.
            if (num > 1) {
                uniquePrimes.insert(num); // Insert the remaining prime factor into the set.
            }
        }

        // Return the number of distinct prime factors found.
        return uniquePrimes.size();
    }
};

```

Typescript Solution

```typescript
// Importing necessary types for Set.
import { Set } from 'typescript-collections';

// Function to calculate the number of distinct prime factors from an array of numbers.
function distinctPrimeFactors(nums: Array<number>): number {
    let uniquePrimes: Set<number> = new Set<number>(); // Set to store unique prime factors.

    // Iterate through each number in the array.
    for (let num of nums) {
        // Check for factors starting from 2 to the square root of the number.
        for (let i = 2; i <= num / i; i++) {
            // If 'i' is a divisor of 'num', it could be a prime factor.
            if (num % i === 0) {
                uniquePrimes.add(i); // Insert the prime factor into the set.
                // Divide 'num' by 'i' completely.
                while (num % i === 0) {
                    num /= i;
                }
            }
        }
        // If 'num' is still greater than 1, then it is a prime number itself.
        if (num > 1) {
            uniquePrimes.add(num); // Insert the remaining prime factor into the set.
        }
    }

    // Return the number of distinct prime factors found.
    return uniquePrimes.size();
}

// Note: TypeScript does not have a Set type with a size() method in the standard library.
// The `typescript-collections` package is a third-party library that could be used here.
// If you want to stay with the standard Set from TypeScript, modify the return line to:
// return uniquePrimes.size;

```

Time and Space Complexity

The time complexity of the given code is determined by the nested loop where we are finding the distinct prime factors for each number in the input list `nums`. The outer loop iterates over each number in the list. The inner while loop checks for factors starting from 2 up to `sqrt(n)` (since `i <= n // i` is equivalent to `i * i <= n`). Factoring a number takes at most `O(sqrt(n))` where `n` is the number being factored. Hence, if the largest number in the list is N, the time complexity of factoring all numbers is $O(M * \sqrt{N})$, where M is the length of the list `nums`.

The space complexity of the code is determined by the set `s` which stores the distinct prime factors of all the numbers in the list. In the worst-case scenario, this could store all primes less than or equal to N (the largest number in the list). The number of primes less than N is approximately $N / \log(N)$ by the prime number theorem. Consequently, the space complexity is $O(N / \log(N))$.