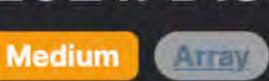
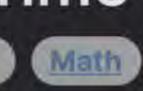
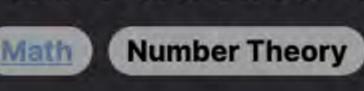
# 2521. Distinct Prime Factors of Product of Array











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.

Initialize an empty set that will be used to store unique prime factors.

Here's how we can approach the problem:

- 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

## 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.

The implementation of the solution follows these steps:

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

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

square root of n, i cannot be a factor of n.

- 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.
- 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.

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

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.

6. Once the outer while loop completes, there may be a case where n (the reduced number) is greater than 1. This remainder n

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

This solution approach leans heavily on the fact that every composite number has at least one prime factor less than or equal to its

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

adds factors to this set, ensuring each factor is only counted once no matter how many numbers it divides in the input array nums.

#### 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}.

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

 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.

• We find 15 is divisible by 3, and since 3 is already in set 5, we do not need to add it again. We divide 15 by 3 to remove all

- 15 is not divisible by 2, so we increment i to 3.
- factors of 3, and n becomes 5.

# Iterate over all the numbers in the list

while factor <= number // factor:</pre>

number //= factor

n is now 1, so we have completed the prime factorization for 15.

- We increment i to 4, but since 4 is not prime, we go to 5.  $\circ$  5 divides n (now 5), and we add 5 to the set s since it is not already there. Now,  $s = \{2, 3, 5\}$ .
- 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

factor = 2 # Start checking for prime factors from the smallest prime

# Increment the factor by 1 to check the next possible prime factor

# If the factor divides the number, it is a prime factor

# Use trial division to find prime factors of the number

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

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

```
if number % factor == 0:
15
                        prime_factors_set.add(factor)
16
                       # Divide the number by the prime factor until it is no longer divisible
                        while number % factor == 0:
18
```

numbers in nums.

Python Solution

class Solution:

9

11

12

13

14

19

20

21

22

21

24

25

26

27

28

29

30

32

31 }

from typing import List

for number in nums:

factor += 1

**if** (number > 1) {

return uniquePrimeFactors.size();

uniquePrimeFactors.add(number);

```
23
               # If the remaining number is greater than 1, it is a prime factor itself
24
               if number > 1:
25
                    prime_factors_set.add(number)
26
27
           # The result is the count of unique prime factors found in all numbers
28
           return len(prime_factors_set)
29
Java Solution
   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) {
 9
               // Check for factors starting from 2 up to the square root of the number
                for (int factor = 2; factor <= number / factor; ++factor) {</pre>
                   // If 'factor' is a divisor of 'number'
12
                   if (number % factor == 0) {
13
14
                       // Add 'factor' to the set of unique prime factors
                        uniquePrimeFactors.add(factor);
15
                       // Remove all occurrences of this prime factor from 'number'
16
                       while (number % factor == 0) {
17
                            number /= factor;
18
19
20
```

```
C++ Solution
 1 #include <vector>
 2 #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.
10
           // Iterate through each number in the array.
11
           for (int& num : nums) {
               // Check for factors starting from 2 to the square root of the number.
13
                for (int i = 2; i <= num / i; ++i) {
14
                    // If 'i' is a divisor of 'num', it could be a prime factor.
15
                   if (num % i == 0) {
16
17
                        uniquePrimes.insert(i); // Insert the prime factor into the set.
18
                       // Divide 'num' by 'i' completely.
                        while (num % i == 0) {
19
20
                            num /= i;
21
22
23
24
               // If 'num' is still greater than 1, then it is a prime number itself.
25
               if (num > 1) {
26
                    uniquePrimes.insert(num); // Insert the remaining prime factor into the set.
27
28
29
           // Return the number of distinct prime factors found.
30
31
           return uniquePrimes.size();
32
33 };
34
```

// If the remaining 'number' is a prime number greater than 1, add it to the set

// Return the size of the set, i.e., the number of distinct prime factors

```
// Iterate through each number in the array.
       for (let num of nums) {
 9
           // Check for factors starting from 2 to the square root of the number.
10
           for (let i = 2; i * i <= num; i++) {
11
12
               // If 'i' is a divisor of 'num', it could be a prime factor.
               if (num % i === 0) {
13
14
                   uniquePrimes.add(i); // Insert the prime factor into the set.
15
                   // Divide 'num' by 'i' completely.
                   while (num % i === 0) {
16
17
                       num /= i;
18
19
20
21
           // If 'num' is still greater than 1, then it is a prime number itself.
22
           if (num > 1) {
23
               uniquePrimes.add(num); // Insert the remaining prime factor into the set.
24
25
26
27
       // Return the number of distinct prime factors found.
28
       return uniquePrimes.size();
29 }
30
  // 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;
35
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
```

// Function to calculate the number of distinct prime factors from an array of numbers.

let uniquePrimes: Set<number> = new Set<number>(); // Set to store unique prime factors.

Typescript Solution

1 // Importing necessary types for Set.

import { Set } from 'typescript-collections';

function distinctPrimeFactors(nums: Array<number>): number {

31 // Note: TypeScript does not have a Set type with a size() method in the standard library.

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)).

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  $i \ll n$  // i is equivalent to  $i * i \ll n$ ). Factoring a number takes at most 0(sqrt(n)) where n is the