1175. Prime Arrangements

Easy Math Leetcode Link

Problem Description

The problem is to find out how many different permutations of the numbers 1 to `n` can be constructed in such a way that all prime numbers are at prime indices, with indices considered to be 1-indexed (meaning they start from 1, not 0). Note that prime numbers are integers greater than 1 that have no divisors other than 1 and themselves.

It is important to understand that indices in a list and the actual numbers placed at those indices can be prime. Therefore, we need to consider two separate conditions for the permutation to meet the prime placement criteria: (1) The numbers themselves must be prime, and (2) Their positions in the permutation must also be prime index locations.

Given a number `n`, there will be a specific count of prime numbers within the range from 1 to `n`, and those prime numbers must be arranged in the prime indices. All the non-prime numbers, therefore, will go into the remaining positions.

Since the answer could potentially be a very large number, the problem asks us to return the result modulo `10^9 + 7`, a common technique in algorithms to avoid integer overflow and to keep the numbers within a manageable range.

Intuition

The solution is based on combinatorics. Specifically, we can break down the problem into two separate tasks:

1. Counting the prime numbers from 1 to `n`. This tells us how many numbers need to be placed in prime index positions.
2. Calculating permutations of the prime numbers in the prime indices and the permutations of the non-prime numbers in the non-prime indices.

The Sieve of Eratosthenes algorithm, which is an efficient way to find all primes smaller than a given number, is a good approach to get the count of prime numbers. With this algorithm, we iterate over each number from 2 to `n` and mark multiples of each number (which cannot be prime) as non-prime. The numbers that remain unmarked at the end of this process are the prime numbers.

Once we count the number of primes within the range of 1 to `n` (let's say there are `cnt` primes), we then must calculate the total number of permutations of these `cnt` prime numbers, which is simply the factorial of `cnt` (`factorial(cnt)`).

Simultaneously, we must also calculate the permutations for the remaining `n - cnt` non-prime numbers, which can be arranged in any order in the remaining positions. The total permutations for these non-prime numbers are `factorial(n - cnt)`.

The total number of prime arrangements is then the product of these two permutations: `factorial(cnt)` for the prime numbers and `factorial(n - cnt)` for the non-prime numbers.

The final step is to return this product modulo `10^9 + 7` to ensure the output fits within the expected range of values as required by the problem.

The solution code encloses the prime-counting logic within a function `count`, then calculates the factorials, multiplies them, and applies the modulo operation to return the answer.

Solution Approach

The implementation of the solution can be broken down into two main parts: counting prime numbers and computing factorials for the permutations.

1. **Counting Prime Numbers using Sieve of Eratosthenes**:
   - The `count` function initiates a list `primes` of boolean values, initially set to `True`, representing the numbers from 0 to `n`. The boolean values will represent if a number is prime (`True`) or not (`False`).
   - It then iterates over the list starting from the first prime number 2. For each number `i`, if `i` is marked as `True` (indicating it is still considered prime), we:
     - Increment the `cnt` which counts the number of primes.
     - Proceed to mark all multiples of `i` as `False` since they are not primes.
   - After the loop completes, `cnt` holds the number of prime numbers between 1 and `n`.
2. **Calculating Factorials**:
   - The number of permissible arrangements of prime numbers is calculated by taking the factorial of the count of prime numbers (`factorial(cnt)`).
   - Similarly, for non-prime numbers, we use the factorial of the difference between the total count `n` and the count of prime numbers (`factorial(n - cnt)`).
3. **Calculating the Answer**:
   - The answer to how many unique permutations of numbers are available such that primes are at prime indices is given by the product of these two factorials: `factorial(cnt) * factorial(n - cnt)`.
   - Since the product of these factorials can be very large, the final answer is taken modulo `10^9 + 7` to fit within the bounds specified by the problem and to avoid integer overflow.

Throughout the implementation, we see the use of basic data structures like lists (`primes` list for the sieve algorithm) and the use of the `range` function to traverse numbers and multiples. Due to Python's built-in modulo operation and factorial computation, no additional custom data structures or algorithms are required for computing factorials and their modulo.

In programming terms, this implementation leverages memoization implicitly by pre-computing the factorials of the numbers from 1 to `n` only once, thereby avoiding redundant calculations. This approach, combined with the efficiency of the Sieve of Eratosthenes, ensures that the algorithm runs in a relatively fast time frame appropriate for the size of input `n`.

Example Walkthrough

Let us illustrate the solution approach by walking through a small example. Consider the number `n = 6`. We want to find out how many permutations of the numbers 1 to 6 can be formed where prime numbers occupy prime indices (1-indexed).

Firstly, we need to identify the prime numbers from 1 to 6 and also the prime indices. The prime numbers will be `2, 3, and 5`, and the prime indices are `2, 3, and 5` since 1 is not considered prime.

Now, apply **Step 1: Counting Prime Numbers using the Sieve of Eratosthenes**:

- Initialize the `primes` list to track prime numbers up to `n` (ignoring 0 and 1). This looks like `[True, True, True, True, True, True]`.
- Starting from `2`, mark non-prime multiples as `False`. The updated `primes` list becomes `[True, True, True, True, False, True]`.
- Count the number of `True` values excluding the first position which corresponds to `1`. We have three primes `2, 3, 5` (which is `cnt` = 3).

Next, for **Step 2: Calculating Factorials**:

- Calculate the factorial of the count of prime numbers (which is 3): `factorial(3) = 3! = 6` to account for permutations among prime numbers.
- Also calculate the factorial of non-prime numbers (`n - cnt`): `factorial(6 - 3) = factorial(3) = 3! = 6`.

Finally, in **Step 3: Calculating the Answer**:

- Multiply the results of these factorials to find the total permutations: `6 * 6 = 36`.
- Take the result modulo `10^9 + 7` gives `36 mod (10^9 + 7) = 36`.

So, for `n = 6`, there are `36` permutations where prime numbers occupy the prime indices.

Python Solution

```python
class Solution:
    def numPrimeArrangements(self, n: int) -> int:
        from math import factorial

        def count_primes(n: int) -> int:
            """
            Count the number of prime numbers less than or equal to n using the Sieve of Eratosthenes algorithm.
            """
            count = 0
            is_prime = [True] * (n + 1)  # Initialize a list to track prime numbers

            for i in range(2, n + 1):
                if is_prime[i]:  # If i is a prime number
                    count += 1
                    for j in range(i * 2, n + 1, i):
                        is_prime[j] = False  # Mark multiples of i as not prime
            return count

        prime_count = count_primes(n)

        # Calculate the number of arrangements as the factorial of the prime count,
        # multiplied by the factorial of the count of non-prime numbers
        arrangements = factorial(prime_count) * factorial(n - prime_count)

        # Return the number of arrangements modulo (10**9 + 7)
        return arrangements % (10**9 + 7)

```

Java Solution

```java
class Solution {
    private static final int MOD = (int) 1e9 + 7; // Constant for the modulo operation

    // Counts the number of prime arrangements possible up to n
    public int numPrimeArrangements(int n) {
        int primeCount = countPrimes(n); // Counts the number of prime numbers up to n
        long arrangements = factorial(primeCount) * factorial(n - primeCount);
        // Computes the arrangements as prime! * (n - prime)!
        return (int) (arrangements % MOD); // Returns the result modulo MOD
    }

    // Calculates the factorial of a number using the modulo operation
    private long factorial(int n) {
        long result = 1;
        for (int i = 2; i <= n; ++i) {
            result = (result * i) % MOD; // Calculates factorial with modulo at each step
        }
        return result;
    }

    // Counts the number of prime numbers up to n
    private int countPrimes(int n) {
        int count = 0; // Initialize count of primes to 0
        boolean[] isPrime = new boolean[n + 1]; // Create an array to mark non-prime numbers
        Arrays.fill(isPrime, true); // Assume all numbers are prime initially
        for (int i = 2; i <= n; ++i) {
            if (isPrime[i]) { // Check if the number is marked as a prime
                ++count; // Increment the count of prime numbers
                // Mark all multiples of i as non-prime
                for (int j = i + i; j <= n; j += i) {
                    isPrime[j] = false;
                }
            }
        }
        return count; // Return the count of prime numbers
    }
}

```

C++ Solution

```cpp
using ll = long long; // Alias for long long type
const int MOD = 1e9 + 7; // Constants should be in uppercase

// Solution class containing methods for prime arrangements calculation
class Solution {
public:
    // Calculates the number of prime arrangements for a given number n
    int numPrimeArrangements(int n) {
        int primeCount = countPrimes(n); // Count the number of primes up to n
        // Calculate the factorial of prime count and non-prime count respectively,
        // then multiply them together modulo MOD
        ll arrangements = factorial(primeCount) * factorial(n - primeCount) % MOD;
        return static_cast<int>(arrangements);
    }

    // Function to calculate factorial of a number n modulo MOD
    ll factorial(int n) {
        ll result = 1;
        for (int i = 2; i <= n; ++i) {
            result = (result * i) % MOD;
        }
        return result;
    }

    // Function to count the number of prime numbers up to n
    int countPrimes(int n) {
        vector<bool> isPrime(n + 1, true); // Create a sieve initialized to true
        int primeCount = 0;
        for (int i = 2; i <= n; ++i) {
            if (isPrime[i]) {
                ++primeCount; // Increment count if i is a prime
                // Mark all multiples of i as non-prime
                for (int j = i * 2; j <= n; j += i) {
                    isPrime[j] = false;
                }
            }
        }
        return primeCount;
    }
};

```

Typescript Solution

```typescript
// Define type alias for bigint
type BigIntAlias = bigint;

// Constant for modular arithmetic
const MOD: BigIntAlias = BigInt(1e9 + 7);

/**
 * Calculates the factorial of a number `n` modulo `MOD`.
 * @param n The number to calculate the factorial of.
 * @returns The factorial of `n` modulo `MOD`.
 */
function factorial(n: number): BigIntAlias {
    let result: BigIntAlias = BigInt(1);
    for (let i = 2; i <= n; ++i) {
        result = (result * BigInt(i)) % MOD;
    }
    return result;
}

/**
 * Counts the number of prime numbers up to `n`.
 * @param n The number up to which to count primes.
 * @returns The count of prime numbers up to `n`.
 */
function countPrimes(n: number): number {
    const isPrime: boolean[] = new Array<boolean>(n + 1).fill(true);
    let primeCount: number = 0;
    for (let i = 2; i <= n; ++i) {
        if (isPrime[i]) {
            primeCount++;
            for (let j = i * 2; j <= n; j += i) {
                isPrime[j] = false;
            }
        }
    }
    return primeCount;
}

/**
 * Calculates the number of prime arrangements for a given number `n`.
 * @param n The number to calculate prime arrangements for.
 * @returns The number of prime arrangements for `n`.
 */
function numPrimeArrangements(n: number): number {
    const primeCount: number = countPrimes(n);
    const arrangements: BigIntAlias = (factorial(primeCount) * factorial(n - primeCount)) % MOD;
    return Number(arrangements);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the `count` function is O(n * log(log(n))). This is because the sieve of Eratosthenes, which is used to find all prime numbers up to `n`, has a time complexity of O(n * log(log(n))).

The `factorial` function (which is not shown here, but its complexity can be derived from typical implementations) typically has a time complexity of O(n). Given that the maximum value for factorial calculation is `n`, two such calculations are performed - one for the prime count `cnt` and one for the non-prime count `n - cnt`.

Therefore, the overall time complexity of the code is dominated by the sieve in the `count` function, which gives us the total time complexity: O(n * log(log(n)) + n + n), simplifying to O(n * log(log(n))).

Space Complexity

For space complexity, the `count` function allocates an array `primes` of size `n + 1`, which leads to O(n) space complexity. The space requirement for the calculation of the factorial depends on the implementation, but typically, it can be calculated in O(1) space.

Therefore, the overall space complexity of the code is O(n) for the sieve of Eratosthenes array storage.