204. Count Primes

Medium | Array | Math | Enumeration | Number Theory

Problem Description

The problem requires us to find the count of prime numbers less than a given integer `n`. Remember, a prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself.

Intuition

The solution is based on an ancient algorithm known as the "Sieve of Eratosthenes". The algorithm works by iteratively marking the multiples of each number starting from 2. Once all multiples of a particular number are marked, we move on to the next unmarked number and repeat the process.

Here's the step-by-step intuition:

1. We start with a boolean array `primes` where each entry is set to `True`, meaning we assume all numbers are primes initially.
2. Starting from the first prime number 2, we mark all of its multiples as `False`, since multiples of 2 cannot be primes.
3. We continue this process for the next unmarked number (which would be the next prime) and mark all of its multiples as `False`.
4. We repeat the process until we have processed all numbers less than `n`.
5. During the process, every time we encounter a number that's not marked as `False`, it means this number is a prime number, and we increment the counter `ans`.

This solution is efficient because once a number is marked as `False`, it will not be checked again, which greatly reduces the number of operations needed compared to checking every number individually for primality.

Solution Approach

The implementation of the `countPrimes` function follows the Sieve of Eratosthenes algorithm:

1. We initialize a list called `primes` filled with `True`, representing all numbers from 0 to `n-1`. Here, `True` signifies that the number is assumed to be a prime number.

2. We iterate over each number starting from 2 (the smallest prime number) using a for loop `for i in range(2, n):`. If the number is marked as `True` in our `primes` list, it is a prime, as it hasn't been marked as not prime by an earlier iteration (via its multiples).

3. When we find such a prime number `i`, we increment our answer counter `ans` by 1, as we've just found a prime.

4. To mark the multiples of `i` as not prime, we loop through a range starting from `i*2` up to `n` (exclusive), in steps of `i`, using the inner loop `for j in range(i + i, n, i):`. The step of `i` makes sure we only hit the multiples of `i`.

5. For each multiple `j` of the prime number `i`, we set `primes[j]` to `False` to denote that `j` is not a prime.

6. Continue the process until all numbers in our list have been processed.

7. Finally, we return `ans`, which now holds the count of prime numbers strictly less than `n`.

Throughout the process, the use of the array `primes` and the marking of non-prime numbers optimizes the approach and avoids unnecessary checks, making this a classic and time-efficient solution for counting prime numbers.

Here is the final code that implements this approach:

```python
class Solution:
    def countPrimes(self, n: int) -> int:
        if n < 2:
            return 0
        primes = [True] * n
        ans = 0
        for i in range(2, n):
            if primes[i]:
                ans += 1
                for j in range(i * i, n, i):
                    primes[j] = False
        return ans
```

In this implementation, notice how we optimized the inner loop's starting point from `i + i` to `i * i`. Since any multiple `k * i` (where `k < i`) would already have been marked False by a prime less than `i`, it suffices to start marking from `i * i`.

Example Walkthrough

Let's illustrate the solution approach with a small example where `n = 10`. We want to find the count of prime numbers less than 10.

Following the steps outlined in the solution approach:

1. We start by initializing a list `primes` that represents the numbers from 0 to 9. All the values are set to `True`, indicating we assume they are prime until proven otherwise:

   ```
   primes = [True, True, True, True, True, True, True, True, True, True]
   ```

2. We start checking numbers from 2 (the smallest prime number). Since `primes[2]` is `True`, 2 is a prime number, so we increment our prime count `ans`.

3. Now, we mark all multiples of 2 as not prime by setting their respective positions in the `primes` array to `False`. This will mark 4, 6 and 8 as not prime:

   ```
   primes = [True, True, True, True, False, True, False, True, False, True]
   ```

   The prime count `ans` is now 1.

4. The next number is 3, which is also `True` in the `primes` list, so we increment `ans` again. We then mark all multiples of 3 as `False`, affecting 6 and 9:

   ```
   primes = [True, True, True, True, False, True, False, True, False, False]
   ```

   The prime count `ans` is now 2.

5. The next number is 4, which is `False` in the `primes` list, so we skip it.

6. Then we check 5, which is `True`. Therefore, we increment `ans` and mark its multiples (none within our range, as the first would be 10, which is outside our range).

   The prime count `ans` is now 3.

7. Continuing this process, we check 6 (marked as not prime), 7 (prime), and 8 (not prime). When we reach 7, we mark it as prime and increment `ans`.

   The prime count `ans` is now 4.

8. Finally, we process 9 (marked as not prime) and the `primes` list won't change anymore as there's no need to mark further multiples.

9. Our final prime count `ans` is 4. Therefore, there are 4 prime numbers less than 10.

Here's a visualization of the `primes` list after processing primes:

```
primes = [True, True, True, True, False, True, False, True, False, False]
                ^      ^      ^                    ^
Indices:        2      3      4     5      6      7     8     9
```

At the indices where `primes` list is `True` (excluding the indices 0 and 1 since we start counting primes from 2), those numbers are the primes less than 10, and we count them up to get our answer, which is 4. This is how the Sieve of Eratosthenes algorithm works and the code from the solution approach implements this efficiently to count the number of prime numbers less than any given integer `n`.

Python Solution

```python
class Solution:
    def countPrimes(self, n: int) -> int:
        if n < 3:  # There are no prime numbers less than 2
            return 0

        # Initialize a list to track prime numbers.
        # True means the number is initially assumed to be prime
        is_prime = [True] * n
        # Count the number of primes
        prime_count = 0

        # Start from the first prime number, which is 2
        for current_number in range(2, n):
            if is_prime[current_number]:
                prime_count += 1  # Increment count if current number is prime

                # Mark multiples of the current number as not prime
                for multiple in range(current_number * 2, n, current_number):
                    is_prime[multiple] = False

        # Return the total count of prime numbers found
        return prime_count

```

Java Solution

```java
class Solution {
    // Method to count the number of prime numbers less than a non-negative number, n.
    public int countPrimes(int n) {
        // Initialize an array to mark non-prime numbers (sieve of Eratosthenes).
        boolean[] isPrime = new boolean[n];
        // Assume all numbers are prime initially (except index 0 and 1).
        Arrays.fill(isPrime, true);

        // Counter for the number of primes found.
        int primeCount = 0;

        // Iterate through the array to find prime numbers.
        for (int i = 2; i < n; i++) {
            // Check if the number at current index is marked as prime.
            if (isPrime[i]) {
                // Increment the count as we found a prime.
                primeCount++;

                // Mark the multiples of the current number as non-prime.
                for (int j = i * 2; j < n; j += i) {
                    isPrime[j] = false;
                }
            }
        }

        // Return the total count of prime numbers found.
        return primeCount;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to count the number of prime numbers less than a non-negative number, n
    int countPrimes(int n) {
        vector<bool> isPrime(n, true); // Create a vector of boolean values, filled with 'true', representing prime status
        int primeCount = 0; // Initialize a count of prime numbers

        // Use the Sieve of Eratosthenes algorithm to find all primes less than n
        for (int i = 2; i < n; ++i) { // Start at the first prime, 2, and check up to n
            if (isPrime[i]) { // If the number is marked as prime
                ++primeCount; // Increment the count of primes
                // Mark all multiples of i as not prime starting from i^2 to avoid redundant work (i * i can be optimized to skip nor
                for (long long j = (long long)i * i; j < n; j += i) {
                    isPrime[j] = false; // Mark the multiple as not prime
                }
            }
        }
        return primeCount; // Return the total count of primes found
    }
};

```

Typescript Solution

```typescript
/**
 * Counts the number of prime numbers less than a non-negative number, n.
 * Implements the Sieve of Eratosthenes algorithm for finding all prime numbers in a given range.
 * @param {number} n - The upper limit (exclusive) up to which to count prime numbers.
 * @return {number} The count of prime numbers less than n.
 */
const countPrimes = (n: number): number => {
    // Initialize an array of boolean values representing the primality of each number.
    // Initially, all numbers are assumed to be prime (true), except for indices 0 and 1.
    let isPrime: boolean[] = new Array(n).fill(true);
    let primeCount: number = 0;

    // Loop through the array starting from the first prime number, 2.
    for (let i: number = 2; i < n; ++i) {
        if (isPrime[i]) {
            // Increment the prime counter when a prime number is encountered.
            ++primeCount;

            // Mark all multiples of i as non-prime (false).
            for (let multiple: number = i + i; multiple < n; multiple += i) {
                isPrime[multiple] = false;
            }
        }
    }

    // Return the count of prime numbers found.
    return primeCount;
};

```

Time and Space Complexity

Time Complexity

The time complexity for this Sieve of Eratosthenes algorithm is `O(n log(log n))`. This is because the inner loop for marking the multiples as non-primes runs `n/i` times for each prime number `i`, and the sum of these series approximates `n` multiplied by the harmonic series, which tends to `log(log n)` as `n` approaches infinity.

Space Complexity

The space complexity of the algorithm is `O(n)` due to the `primes` list which stores a boolean for each number up to `n` to indicate whether it's a prime number or not.