Documentation for Counting Prime Numbers Less Than a Given Integer

Problem Statement:

- You are given an integer n, and the task is to return the number of prime numbers that are strictly less than n.

- A prime number is a natural number greater than 1 that is not divisible by any number other than 1 and itself. The goal is to efficiently count how many such prime numbers exist below the given integer n.

Input:

- An integer n where $0 <= n <= 5 * 10^6$.

Output:

- An integer representing the count of prime numbers that are strictly less than n.

Example 1:

- **Input:** n = 10
- **Output:** 4
- **Explanation:** There are four prime numbers less than 10, and they are: 2, 3, 5, and 7.

Example 2:

- **Input:** n = 0
- **Output:** 0
- **Explanation:** There are no prime numbers less than 0.

Example 3:

- **Input:** n = 1
- **Output:** 0
- **Explanation:** There are no prime numbers less than 1.

Constraints:

- $0 <= n <= 5 * 10^6$
  - The function needs to handle large values of n, up to 5 million efficiently.

Approach: *Sieve of Eratosthenes*

- To efficiently solve the problem, we can use the Sieve of Eratosthenes, which is a well-known algorithm for finding all prime numbers less than a given integer.

Steps in the Algorithm:

1. **Edge Case Handling:**

- If n is less than 2, there are no prime numbers to consider since prime numbers start from 2. Hence, the result for n = 0 or n = 1 should be 0.

2. Initialization:

- We create a list is_prime of length n. Each index of this list represents a number from 0 to n-1.
- Initially, all numbers are assumed to be prime, so we mark every element as True.
- However, we know that 0 and 1 are not prime numbers, so we explicitly mark them as False.

3. Mark Non-Prime Numbers:

- Starting from the smallest prime, which is 2, we mark all of its multiples as non-prime.
- For any prime number i, we mark all numbers that are multiples of i (starting from i * i) as False.
- We only need to iterate up to the square root of n (i.e., sqrt(n)) because for any composite number, there must be at least one factor that is less than or equal to its square root. This optimizes the performance of the algorithm.

4. Count Prime Numbers:

- After marking all non-prime numbers, we count how many numbers remain marked as True in the is_prime list. These True values represent the prime numbers.

Time Complexity:

- *The Sieve of Eratosthenes is an efficient algorithm with a time complexity of O(n log log n). Here's a breakdown:*
  - The outer loop runs approximately sqrt(n) times.
  - For each prime number i, marking its multiples takes n / i steps.
  - Summing these steps gives the final time complexity of O(n log log n).
  - This time complexity is efficient enough to handle the upper limit of n = 5 * 10^6 within reasonable time constraints.

Space Complexity:

- The space complexity is O(n) because we are using an additional list is_prime of size n to track which numbers are prime.

Detailed Explanation of Each Step:

1. Edge Case (n < 2):

- If n is less than 2, the function returns 0. This is because the smallest prime number is 2, so no prime number can exist below it.

2. Prime List Initialization:

- A boolean list is_prime of size n is initialized where each index represents whether the corresponding number is prime.
- Initially, all entries in the list are marked as True except for is_prime[0] and is_prime[1], which are set to False since 0 and 1 are not prime.

3. Sieve Process (Marking Non-Prime Numbers):

- For each number starting from 2, the algorithm checks if the number is marked as prime.
- If a number i is prime, its multiples are marked as non-prime.
- Instead of marking all multiples starting from 2, we start marking from i * i because any smaller multiple of i would have already been marked by a smaller prime factor.

4. Prime Counting:

- After processing the entire list, the algorithm simply counts the number of True values in the is_prime list. These True values correspond to the prime numbers less than n.

Edge Cases:

- **n = 0 or n = 1:** There are no primes less than 2, so the function should return 0.

- **n = 2:** The smallest prime number is 2, but since we are counting primes *strictly less* than n, the result should be 0.

- **Large values of n:** The algorithm is efficient enough to handle large values of n up to 5 million due to its time complexity of O(n log log n).

Summary:

- The problem of counting prime numbers less than n is efficiently solved using the Sieve of Eratosthenes algorithm. This approach ensures that we can handle large input sizes while maintaining a low time and space complexity. The algorithm works by marking non-prime numbers in a list and then counting the remaining prime numbers.