2761. Prime Pairs With Target Sum

Medium Array Math Enumeration Number Theory Leetcode Link

Problem Description

This problem asks for pairs of prime numbers that sum up to a given integer `n`. A prime number pair consists of two prime numbers `x` and `y` where `1 <= x <= y <= n`, and their sum equals `n`. We need to return a 2D list sorted in ascending order by the first element of each pair (x), containing all such pairs or an empty array if no such pairs exist.

A prime number is defined as a number greater than `1` that has no positive divisors other than `1` and itself.

Intuition

The task at hand can be solved by first finding all the prime numbers up to `n` and then checking which of these can form pairs that sum to `n`.

To identify prime numbers efficiently, we can use the Sieve of Eratosthenes algorithm, which marks all non-prime numbers up to a maximum number (`n` in this case) by marking multiples of each prime number starting from 2.

After identifying all prime numbers, we only need to check for pairs where `x` is less than or equal to `n/2`. This is because if x were greater than `n/2`, then `y` would have to be less than `n/2` to sum up to `n`. But since we start checking from the smallest prime (2), once we reach numbers larger than `n/2`, we'd have already considered all possible pairs with smaller numbers, hence completing the search space for prime pairs where `x` and `y` can equal `n`.

For each potential prime `x` up to `n/2`, the complement `y` is determined by `n - x`. If both `x` and `y` are primes, we record the pair. The algorithm ensures all pairs found are unique since for each `x`, there is only one unique `y` that meets the criteria.

The pre-computed list of primes is used to quickly check if `x` and `y` are prime by referencing their values in the array with prime statuses. This results in an efficient and direct solution to the problem.

Solution Approach

The given solution employs the Sieve of Eratosthenes algorithm to pre-process all prime numbers within the range of `n`. Let's explore the steps involved:

1. Initialize an array called `primes` with `n` boolean elements set to `True`. This array will be used to mark whether a number (index) is a prime or not, with `True` representing prime.
2. Iterate over the range from 2 to `n`:
   - For each number `i` that is still marked as `True` (prime) in the `primes` array, iteratively mark its multiples as `False` (non-prime), starting from `i * 2` up to `n-1` in increments of `i`. In doing so, it skips the first multiple, which is the number itself, as that should remain marked as prime.
3. Once the prime numbers are pre-processed, we create an empty list `ans` to hold the prime pairs.
4. Now, we enumerate through values `x` from 2 up to `n // 2 + 1` to find all prime pairs. Why up to `n // 2 + 1`? Because if `x` were any larger, `y = n - x` would be less than `x`, which means we would be considering the same pair in reverse order, which is not necessary since `x <= y`.
5. For each `x`, we calculate `y` as `n - x`. We check if both `x` and `y` are marked as `True` in the `primes` array.
6. If both `x` and `y` are prime, we append the pair `[x, y]` to our answer list `ans`.
7. Finally, we return the `ans` list, which now contains all the sorted prime pairs whose elements sum up to `n`.

By using the Sieve of Eratosthenes to pre-calculate the prime numbers and then enumerating possible pairs with a range boundary of `n // 2 + 1`, the algorithm effectively reduces the problem size and avoids unnecessary comparisons.

Example Walkthrough

Let's use the integer `n = 10` as a small example to illustrate the solution approach.

1. We initialize an array `primes` with 11 elements (index 0 to 10), all set to `True`. The indices represent numbers, and the value at each index represents whether the number is prime (`True`) or not (`False`).
2. Begin the Sieve of Eratosthenes by iterating from 2 to `n`. For each prime number `i` that is still marked `True`, mark its multiples as `False`. After iterating, the `primes` array indicates that the prime numbers up to `n` are `2, 3, 5, 7` because the corresponding indices `2, 3, 5, 7` have remained `True`.
3. We create an empty list `ans` for holding our prime pairs.
4. Now, we enumerate through the values `x` from 2 to `n // 2 + 1` which gives us the range `[2, 5]`. We are looking for primes within this range that can pair with another prime number to total `n`.
5. We start with `x = 2` and calculate `y = n - x`, which gives us `y = 10 - 2 = 8`. Since 8 is not prime, we move to the next value.
6. With `x = 3`, we find `y = 10 - 3 = 7`. Both 3 and 7 are marked `True` in the `primes` array, so we add the pair `[3, 7]` to our answer list `ans`.
7. Proceeding to `x = 4`, we find `y = 10 - 4 = 6`. Since 4 is not prime, we do not consider this pair.
8. Next, with `x = 5`, we find that `y = 10 - 5 = 5`. Since both 5 and 5 are prime, we add the pair `[5, 5]` to `ans`.
9. We've now considered all values up to `n // 2 + 1`, so the enumeration is complete.
10. The final answer list `ans` contains the sorted pairs: `[[3, 7], [5, 5]]`.

Thus, for `n = 10`, the pairs of prime numbers that sum up to `n` are `[3, 7]` and `[5, 5]`.

Python Solution

```python
from typing import List

class Solution:
    def find_prime_pairs(self, n: int) -> List[List[int]]:
        # Initialize a list to mark all numbers as prime initially
        is_prime = [True] * n

        # Sieve of Eratosthenes algorithm to find primes less than n
        for i in range(2, int(n ** 0.5) + 1): # Loop only up to the square root of n
            if is_prime[i]:
                # Mark all multiples of i as non-prime
                for j in range(i * i, n, i):
                    is_prime[j] = False

        # Initialize a list to store pairs of prime numbers
        prime_pairs = []

        # Find pairs of primes where both numbers add up to n
        for x in range(2, n // 2 + 1): # Only need to check up to n // 2
            y = n - x
            if is_prime[x] and is_prime[y]:
                # If both x and y are prime, add them as a pair
                prime_pairs.append([x, y])

        # Return the list of prime pairs
        return prime_pairs

```

Java Solution

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

class Solution {
    public List<List<Integer>> findPrimePairs(int n) {
        // Initialize an array to determine the primality of each number up to n.
        boolean[] isPrime = new boolean[n];
        // Assume all numbers are prime initially, set all entries to true.
        Arrays.fill(isPrime, true);

        // Use the Sieve of Eratosthenes to find all prime numbers less than n.
        for (int i = 2; i < n; ++i) {
            if (isPrime[i]) {
                // If i is prime, then mark all of its multiples as not prime.
                for (int j = i + i; j < n; j += i) {
                    isPrime[j] = false;
                }
            }
        }

        // List to hold the prime pairs that sum up to n.
        List<List<Integer>> primePairs = new ArrayList<>();

        // Iterate over possible prime pairs where both numbers are less than n.
        for (int x = 2; x <= n / 2; ++x) {
            int y = n - x; // Calculate the complement of x that sums to n.
            // Check if both numbers are prime.
            if (isPrime[x] && isPrime[y]) {
                // Add the pair to the list of prime pairs.
                primePairs.add(Arrays.asList(x, y));
            }
        }

        // Return the list of prime pairs.
        return primePairs;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cmath>
#include <cstring>

class Solution {
public:
    // Function that returns all unique pairs of prime numbers that add up to 'n'.
    std::vector<std::vector<int>> findPrimePairs(int n) {
        // Create a boolean array 'is_prime' initialized to true for prime checking.
        std::vector<bool> is_prime(n, true);

        // Implement the Sieve of Eratosthenes algorithm to find prime numbers up to 'n'.
        for (int i = 2; i * i < n; ++i) { // Only go up to the square root of 'n'.
            if (is_prime[i]) { // If the number is still marked prime:
                // All multiples of i starting from i*i are marked as not prime.
                for (int j = i * i; j < n; j += i) {
                    is_prime[j] = false;
                }
            }
        }

        // Vector to store the prime pairs.
        std::vector<std::vector<int>> prime_pairs;

        // Iterate over the range from 2 to n/2 to find prime pairs.
        for (int x = 2; x <= n / 2; ++x) {
            int y = n - x; // The potential prime pair for x that adds up to n.
            // If both x and y are prime, add them as a pair to the answer list.
            if (is_prime[x] && is_prime[y]) {
                prime_pairs.push_back({x, y});
            }
        }

        // Return the list of prime pairs.
        return prime_pairs;
    }
};

```

Typescript Solution

```typescript
/**
 * Checks and returns all prime pairs that sum up to a given number.
 * @param n The sum target and the upper limit for the prime search.
 * @returns A two-dimensional array containing all the prime pairs.
 */
function findPrimePairs(n: number): number[][] {
    // Initialize a boolean array to track prime numbers up to n.
    const isPrime: boolean[] = new Array(n).fill(true);

    // Implement the Sieve of Eratosthenes algorithm to identify primes.
    for (let index = 2; index < n; ++index) {
        if (isPrime[index]) {
            // Mark all multiples of index as not prime.
            for (let multiple = index * 2; multiple < n; multiple += index) {
                isPrime[multiple] = false;
            }
        }
    }

    // Array to store pairs of prime numbers whose sum equals n.
    const primePairs: number[][] = [];

    // Loop through the list of potential prime numbers to find valid pairs.
    for (let primeCandidate = 2; primeCandidate <= n / 2; ++primeCandidate) {
        const pairedPrime = n - primeCandidate;
        // Check if both numbers in the potential pair are prime.
        if (isPrime[primeCandidate] && isPrime[pairedPrime]) {
            // Add the prime pair to the results array.
            primePairs.push([primeCandidate, pairedPrime]);
        }
    }

    // Return the array of prime pairs.
    return primePairs;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code can be analyzed in two parts:

1. **Sieve Creation**: The first `for` loop runs to mark non-prime numbers, which is an implementation of the Sieve of Eratosthenes algorithm. The inner loop marks off multiples of each prime found, starting from `i * i` up to `n`, in steps of `i`. The complexity of the Sieve of Eratosthenes is generally considered to be `O(n \log \log n)` as it involves multiple passes over the data within certain constraints, not purely linear passes. However, there's a minor modification needed in the given implementation because the inner loop should ideally start from `i * i` instead of `i + i` for optimization.
2. **Prime Pair Finding**: The second `for` loop finds pairs of primes that sum up to `n`. It runs halfway through the prime array (i.e., up to `n // 2`) as for any prime `x` greater than `n // 2`, `y = n - x` would be less than `x` and would have been already checked. Therefore, this part has a linear component in its complexity, which is `O(n/2)`, simplifying to `O(n)`.

Overall, when combining the `O(n \log \log n)` complexity of the Sieve with the `O(n)` linear scan for pairs, the dominating factor is `O(n \log \log n)`, as this grows faster than `O(n)` for larger `n`.

Space Complexity

The space complexity is defined by the additional space used for storing the prime number flags. This is a Boolean array of size `n`, resulting in `O(n)` space complexity.