2614. Prime In Diagonal

Easy Array Math **Matrix** **Number Theory** Leetcode Link

Problem Description

In this problem, we are given a square two-dimensional integer array `nums` which is 0-indexed. The task is to find the largest prime number that is present on either or both of the two main diagonals of this array.

The first main diagonal runs from the top-left corner to the bottom-right corner and includes elements where the row and column indices are equal, i.e., `nums[i][i]`. The second main diagonal goes from the top-right corner to the bottom-left corner, consisting of elements `nums[i][nums.length - i - 1]`.

A number is considered prime if it's greater than 1 and does not have any divisors other than 1 and itself. If no prime numbers are found on the diagonals, the function should return 0.

An example illustration provided in the problem statement shows a grid where the first diagonal is `[1,5,9]` and the second diagonal is `[3,5,7]`.

Intuition

The intuition behind the solution is straightforward. Since the problem requires us to only inspect elements along the diagonals of a square matrix, we don't need to consider every element in the array.

We define a utility function, `is_prime`, that determines if a given number is prime. The function iterates from `2` to `sqrt(x)` where `x` is the number we're testing for primality, and checks if `x` is divisible by any number in this range. If `x` is divisible, then it is not a prime number. We use the square root of `x` as an optimization because a larger factor of `x` would have a corresponding smaller factor that would have been identified by the time we reach `sqrt(x)`.

The solution approach involves iterating through each row of the array once, inspecting two elements in each row - one from each diagonal. For each of these elements, we use the `is_prime` function to check if it's prime. If it is, we update our answer with this value if it happens to be greater than the current answer.

This way, by the end of the iteration, we'll have checked all numbers on both diagonals and kept track of the largest prime number encountered. The answer is the maximum prime number found; or `0` if there were no prime numbers on the diagonals.

Solution Approach

The algorithm follows the steps outlined in the intuition section. Specifically, we make use of a simple linear scan through the diagonals of the matrix and a prime-checking function.

Here is a step-by-step breakdown of the implementation:

1. We define a helper function `is_prime` which takes an integer `x` as an argument. This function checks if `x` is a prime number by attempting to find a divisor from `2` to `sqrt(x)`.

   ```
   def is_prime(x: int) -> bool:
       if x < 2:
           return False
       return all(x % i for i in range(2, int(sqrt(x)) + 1))
   ```

   - If `x` is less than 2, it is not prime by definition, and the function returns `False`.
   - Otherwise, it uses the `all` function along with a generator expression to check for any divisors. If any divisor is found (the remainder is zero), `all` will return `False`, and the function will also return `False`, indicating that `x` is not prime.
   - If no divisors are found, the number is prime, and `True` is returned.
2. The main `diagonalPrime` function scans the matrix's diagonals. It initializes `ans` to `0`, which will store the largest prime found.

```
n = len(nums)
ans = 0
```

3. It then uses a loop to iterate through each element of the array, but only checks the values at the diagonal positions - `nums[i][i]` and `nums[i][n - i - 1]`:

   ```
   for i, row in enumerate(nums):
       if is_prime(row[i]):
           ans = max(ans, row[i])
       if is_prime(row[n - i - 1]):
           ans = max(ans, row[n - i - 1])
   ```

   - In this loop, `enumerate(nums)` is used to get both the index `i` and the `row` of the matrix.
   - We check both elements on the diagonals using our `is_prime` function.
   - If the number on the diagonal is prime, we update `ans` with the larger of `ans` and the prime number found.
   - The use of the `max` function ensures that we always have the largest prime number encountered.
4. After the for loop completes, we return `ans`, which by then contains the largest prime number found on the diagonals or remains at its initial value, `0`, if there were no prime numbers.

Returning `ans` completes the function:

```
return ans
```

No additional data structures are needed for this approach, and the patterns used are direct access of items in an array and linear iteration, which keeps the space complexity to O(1) and time complexity to O(n * sqrt(m)) (where `n` is the length of a side of the square matrix and `m` is the value of the largest number to be checked for primality).

Example Walkthrough

Let's illustrate the solution approach using a small example. Consider the following 3×3 grid for the array `nums`:

```
nums = [
  [11,  2, 5],
  [ 9,  7, 4],
  [ 1,  6, 3]
]
```

1. The first main diagonal elements are `nums[0][0]`, `nums[1][1]`, `nums[2][2]`, which are `11`, `7`, `3`, respectively. The second main diagonal elements are `nums[0][2]`, `nums[1][1]`, `nums[2][0]`, which are `5`, `7`, `1`.

2. Now, we define our `is_prime` function and use it to check each number along the diagonals for primality:

   - `11` is prime.
   - `7` is prime (noted twice as it appears in both diagonals).
   - `3` is prime.
   - `5` is prime.
   - `1` is not prime since it is less than 2.
3. As we iterate through the diagonals, we use the `is_prime` function and keep track of the largest prime number:

   - When we check `nums[0][0]`, which is `11`, it's prime, and since `ans` is initially `0`, we now update `ans` to `11`.
   - Checking `nums[0][2]` gives us `5`, which is prime but less than `11`, so `ans` remains `11`.
   - Next, both `nums[1][1]` and `nums[2][0]` yield a `7`, which is prime, but still less than `11`, so `ans` remains unchanged.
   - Finally, `nums[2][2]` is `3`, also prime, but less than the current `ans` of `11`.
4. After inspecting all numbers on both diagonals, the largest prime number we found is `11`, so our function returns `11`. If no prime numbers had been encountered, `ans` would have remained `0`, and the function would return `0`.

This example captures all steps in the algorithm, iterates over the diagonals while checking for primality, and concludes by returning the largest prime found or `0`. It demonstrates how the `is_prime` function works in conjunction with the linear iteration over array elements, meeting the objective of the problem with efficiency in terms of time and space complexity.

Python Solution

```
from typing import List
from math import isqrt

class Solution:
    def diagonal_prime(self, nums: List[List[int]]) -> int:
        # Helper function to check if a number is prime
        def is_prime(x: int) -> bool:
            if x < 2:
                return False  # Numbers less than 2 are not prime
            for i in range(2, isqrt(x) + 1):  # Iterating only up to the square root of x
                if x % i == 0:
                    return False  # Number is divisible by i, hence not prime
            return True  # Number is prime as it wasn't divisible by any i

        # Get the length of the square matrix
        n = len(nums)
        # Initialize the maximum prime number found on the diagonals as 0
        max_prime = 0
        # Loop through each row of the matrix
        for i, row in enumerate(nums):
            # Check if the element in the primary diagonal is prime
            if is_prime(row[i]):
                max_prime = max(max_prime, row[i])
            # Check if the element in the secondary diagonal is prime
            if is_prime(row[n - i - 1]):
                max_prime = max(max_prime, row[n - i - 1])
        # Return the largest prime number found on the diagonals
        return max_prime

```

Java Solution

```
class Solution {
    // Method to find the maximum prime number on either of the diagonals of a square matrix
    public int diagonalPrime(int[][] matrix) {
        int size = matrix.length; // The length of the matrix (number of rows)
        int maxPrime = 0; // Variable to store the maximum prime number found

        // Loop through the matrix from the first row to the last row
        for (int i = 0; i < size; ++i) {
            // Check if the element on the primary diagonal is prime
            if (isPrime(matrix[i][i])) {
                maxPrime = Math.max(maxPrime, matrix[i][i]); // Update max if it's a new maximum
            }
            // Check if the element on the secondary diagonal is prime
            if (isPrime(matrix[i][size - i - 1])) {
                maxPrime = Math.max(maxPrime, matrix[i][size - i - 1]); // Update max if it's a new maximum
            }
        }
        return maxPrime; // Return the largest prime number found on the diagonals
    }

    // Helper method to check if an integer is prime
    private boolean isPrime(int number) {
        // Numbers less than 2 are not prime
        if (number < 2) {
            return false;
        }
        // Check for factors from 2 to the square root of number
        for (int i = 2; i <= number / i; ++i) {
            // If a divisor is found, number is not prime
            if (number % i == 0) {
                return false;
            }
        }
        // No divisors were found; number is prime
        return true;
    }
}

```

C++ Solution

```
class Solution {
public:
    // Function to find the maximum prime number on both diagonals of a square matrix.
    int diagonalPrime(vector<vector<int>>& matrix) {
        int size = matrix.size();  // The size of the NxN matrix.
        int maxPrime = 0;  // Variable to keep track of the maximum prime found.

        // Loop through each element on the diagonals of the matrix.
        for (int i = 0; i < size; ++i) {
            // Check the primary diagonal element for primality.
            if (isPrime(matrix[i][i])) {
                maxPrime = max(maxPrime, matrix[i][i]);
            }
            // Check the secondary diagonal element for primality.
            if (isPrime(matrix[i][size - i - 1])) {
                maxPrime = max(maxPrime, matrix[i][size - i - 1]);
            }
        }
        return maxPrime;  // Return the maximum prime number found on the diagonals.
    }

    // Helper function to check if a number is prime.
    bool isPrime(int num) {
        if (num < 2) {
            return false;  // Numbers less than 2 are not prime.
        }
        // Check divisibility of num from 2 to sqrt(num).
        for (int i = 2; i <= num / i; ++i) {
            if (num % i == 0) {
                return false;  // num is divisible by i, hence not prime.
            }
        }
        return true;  // num is prime as it wasn't divisible by any number from 2 to sqrt(num).
    }
};

```

Typescript Solution

```
// Function to check if a number is prime.
const isPrime = (num: number): boolean => {
  if (num < 2) {
    return false; // Numbers less than 2 are not prime.
  }
  // Check the divisibility of num from 2 to the square root of num.
  for (let i = 2; i * i <= num; i++) {
    if (num % i === 0) {
      return false; // num is divisible by i, hence not prime.
    }
  }
  return true; // num is prime as it wasn't divisible by any number from 2 to sqrt(num).
};

// Function to find the maximum prime number on both diagonals of a square matrix.
const diagonalPrime = (matrix: number[][]): number => {
  const size: number = matrix.length; // The size of the NxN matrix.
  let maxPrime: number = 0; // Variable to keep track of the maximum prime found.

  // Loop through each element on the diagonals of the matrix.
  for (let i = 0; i < size; i++) {
    // Check the primary diagonal element for primality.
    if (isPrime(matrix[i][i])) {
      maxPrime = Math.max(maxPrime, matrix[i][i]);
    }
    // Check the secondary diagonal element for primality.
    if (isPrime(matrix[i][size - i - 1])) {
      maxPrime = Math.max(maxPrime, matrix[i][size - i - 1]);
    }
  }
  return maxPrime; // Return the maximum prime number found on the diagonals.
};

// Example usage:
// const myMatrix: number[][] = [[3, 2, 3], [2, 7, 11], [17, 14, 5]];
// const maxDiagonalPrime: number = diagonalPrime(myMatrix);
// console.log(maxDiagonalPrime); // Outputs the maximum prime number found on the diagonals.

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is primarily influenced by two operations: the iteration over the rows of the matrix and the check for prime numbers.

1. There is a loop over the `n` rows of the input matrix `nums`. For each row, the code checks if the elements at the primary diagonal (`row[i]`) and the secondary diagonal (`row[n - i - 1]`) are prime.
2. The function `is_prime` is used to determine if a number is prime, which operates in `O(sqrt(x))`, where `x` is the number being checked for primality.

Given that the method `is_prime` is called at most twice for each of the `n` rows of the matrix, the overall time complexity is `O(n * sqrt(M))`. Here, `M` represents the value of the largest number within the matrix, which is being checked for primality.

Thus, the resulting formula for time complexity is: `O(n * sqrt(M))`.

Space Complexity

The space complexity is determined by the amount of extra memory space required by the algorithm.

- No extra space is needed apart from a few variables (`n`, `ans`, `i`, and `row`), which are not dependent on the size of the input.
- The `is_prime` function uses a range within a loop, but since Python’s range function generates values on the fly, it does not add to space complexity.

As a result, the space complexity of the algorithm is `O(1)`, indicating constant space usage regardless of input size.