Problem 2614: Prime In Diagonal

Problem Information

Difficulty: Easy
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

You are given a 0-indexed two-dimensional integer array

nums

.

Return

the largest

prime

number that lies on at least one of the

diagonals

of

nums

. In case, no prime is present on any of the diagonals, return

0.

Note that:

An integer is

prime

if it is greater than

1

and has no positive integer divisors other than

1

and itself.

An integer

val

is on one of the

diagonals

of

nums

if there exists an integer

i

for which

nums[i][i] = val

or an

i

for which

nums[i][nums.length - i - 1] = val

.

1	2	3
4	5	6
7	8	9

In the above diagram, one diagonal is

[1,5,9]

and another diagonal is

[3,5,7]

.

Example 1:

Input:

nums = [[1,2,3],[5,6,7],[9,10,11]]

Output:

11

Explanation:

The numbers 1, 3, 6, 9, and 11 are the only numbers present on at least one of the diagonals. Since 11 is the largest prime, we return 11.

Example 2:

Input:

nums = [[1,2,3],[5,17,7],[9,11,10]]

Output:

17

Explanation:

The numbers 1, 3, 9, 10, and 17 are all present on at least one of the diagonals. 17 is the largest prime, so we return 17.

Constraints:

1 <= nums.length <= 300

nums.length == nums

i

.length

1 <= nums

[i][j]

<= 4*10

6

Code Snippets

C++:

```
class Solution {
public:
int diagonalPrime(vector<vector<int>>& nums) {

}
};
```

Java:

```
class Solution {
public int diagonalPrime(int[][] nums) {

}
}
```

Python3:

```
class Solution:
def diagonalPrime(self, nums: List[List[int]]) -> int:
```

Python:

```
class Solution(object):
def diagonalPrime(self, nums):
"""
:type nums: List[List[int]]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[][]} nums
* @return {number}
*/
var diagonalPrime = function(nums) {

};
```

TypeScript:

```
function diagonalPrime(nums: number[][]): number {

};
```

C#:

```
public class Solution {
public int DiagonalPrime(int[][] nums) {

}
}
```

C:

```
int diagonalPrime(int** nums, int numsSize, int* numsColSize) {

}
```

Go:

```
func diagonalPrime(nums [][]int) int {

}
```

Kotlin:

```
class Solution {
fun diagonalPrime(nums: Array<IntArray>): Int {

}
}
```

Swift:

```
class Solution {
func diagonalPrime(_ nums: [[Int]]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn diagonal_prime(nums: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[][]} nums
# @return {Integer}
def diagonal_prime(nums)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $nums
* @return Integer
*/
function diagonalPrime($nums) {

}
}
```

Dart:

```
class Solution {
int diagonalPrime(List<List<int>> nums) {

}
}
```

Scala:

```
object Solution {
def diagonalPrime(nums: Array[Array[Int]]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec diagonal_prime(nums :: [[integer]]) :: integer
def diagonal_prime(nums) do

end
end
```

Erlang:

```
-spec diagonal_prime(Nums :: [[integer()]]) -> integer().
diagonal_prime(Nums) ->
.
```

Racket:

```
(define/contract (diagonal-prime nums)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Prime In Diagonal
* Difficulty: Easy
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
int diagonalPrime(vector<vector<int>>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Prime In Diagonal
* Difficulty: Easy
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public int diagonalPrime(int[][] nums) {

}
}
```

Python3 Solution:

```
"""
Problem: Prime In Diagonal
Difficulty: Easy
Tags: array, math

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def diagonalPrime(self, nums: List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def diagonalPrime(self, nums):
"""
:type nums: List[List[int]]
:rtype: int
```

```
"""
```

JavaScript Solution:

```
/**
* Problem: Prime In Diagonal
* Difficulty: Easy
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number[][]} nums
* @return {number}
*/
var diagonalPrime = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Prime In Diagonal
* Difficulty: Easy
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function diagonalPrime(nums: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Prime In Diagonal
* Difficulty: Easy
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public int DiagonalPrime(int[][] nums) {

}
}
```

C Solution:

```
/*
* Problem: Prime In Diagonal
* Difficulty: Easy
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

int diagonalPrime(int** nums, int numsSize, int* numsColSize) {

}
```

Go Solution:

```
// Problem: Prime In Diagonal
// Difficulty: Easy
// Tags: array, math
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach
```

```
func diagonalPrime(nums [][]int) int {

}
```

Kotlin Solution:

```
class Solution {
fun diagonalPrime(nums: Array<IntArray>): Int {

}
}
```

Swift Solution:

```
class Solution {
func diagonalPrime(_ nums: [[Int]]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Prime In Diagonal
// Difficulty: Easy
// Tags: array, math
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn diagonal_prime(nums: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} nums
# @return {Integer}
def diagonal_prime(nums)
```

```
end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $nums
* @return Integer
*/
function diagonalPrime($nums) {

}
}
```

Dart Solution:

```
class Solution {
int diagonalPrime(List<List<int>> nums) {

}
}
```

Scala Solution:

```
object Solution {
def diagonalPrime(nums: Array[Array[Int]]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec diagonal_prime(nums :: [[integer]]) :: integer
def diagonal_prime(nums) do

end
end
```

Erlang Solution:

```
-spec diagonal_prime(Nums :: [[integer()]]) -> integer().
diagonal_prime(Nums) ->
  .
```

Racket Solution:

```
(define/contract (diagonal-prime nums)
  (-> (listof (listof exact-integer?)) exact-integer?)
  )
```