Problem 348: Design Tic-Tac-Toe

Problem Information

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

Problem Description

Assume the following rules are for the tic-tac-toe game on an

n x n

board between two players:

A move is guaranteed to be valid and is placed on an empty block.

Once a winning condition is reached, no more moves are allowed.

A player who succeeds in placing

n

of their marks in a horizontal, vertical, or diagonal row wins the game.

Implement the

TicTacToe

class:

TicTacToe(int n)

Initializes the object the size of the board

n

.

int move(int row, int col, int player)

Indicates that the player with id

player

plays at the cell

(row, col)

of the board. The move is guaranteed to be a valid move, and the two players alternate in making moves. Return

0

if there is

no winner

after the move,

1

if

player 1

is the winner after the move, or

2

if

player 2

is the winner after the move.

Example 1:

Input

["TicTacToe", "move", "move", "move", "move", "move", "move", "move"] [[3], [0, 0, 1], [0, 2, 2], [2, 2, 1], [1, 1, 2], [2, 0, 1], [1, 0, 2], [2, 1, 1]]

Output

[null, 0, 0, 0, 0, 0, 0, 1]

Explanation

TicTacToe ticTacToe = new TicTacToe(3); Assume that player 1 is "X" and player 2 is "O" in the board. ticTacToe.move(0, 0, 1); // return 0 (no one wins) |X| | | | | | // Player 1 makes a move at (0, 0). | | | |

ticTacToe.move(0, 2, 2); // return 0 (no one wins) |X| |O| | | | | // Player 2 makes a move at (0, 2). | | | |

ticTacToe.move(2, 2, 1); // return 0 (no one wins) |X| |O| | | | | // Player 1 makes a move at (2, 2). | | |X|

ticTacToe.move(1, 1, 2); // return 0 (no one wins) |X| |O| | |O| | // Player 2 makes a move at (1, 1). | | |X|

ticTacToe.move(2, 0, 1); // return 0 (no one wins) |X| |O| | |O| | // Player 1 makes a move at (2, 0). |X| |X|

ticTacToe.move(1, 0, 2); // return 0 (no one wins) |X| |O| |O|O| | // Player 2 makes a move at (1, 0). |X| |X|

ticTacToe.move(2, 1, 1); // return 1 (player 1 wins) |X| |O| |O|O| | // Player 1 makes a move at (2, 1). |X|X|X|

Constraints:

2 <= n <= 100

player is

1

or

2

.

0 <= row, col < n

(row, col)

are

unique

for each different call to

move

.

At most

n

2

calls will be made to

move

.

Follow-up:

Could you do better than

O(n

2

)

per

move()

operation?

Code Snippets

C++:

```
class TicTacToe {
public:
TicTacToe(int n) {

}

int move(int row, int col, int player) {

}
};

/**
* Your TicTacToe object will be instantiated and called as such:
* TicTacToe* obj = new TicTacToe(n);
* int param_1 = obj->move(row,col,player);
*/
```

Java:

```
class TicTacToe {

public TicTacToe(int n) {
```

```
}

public int move(int row, int col, int player) {

}
}

/**
* Your TicTacToe object will be instantiated and called as such:
* TicTacToe obj = new TicTacToe(n);
* int param_1 = obj.move(row,col,player);
*/
```

Python3:

```
class TicTacToe:

def __init__(self, n: int):


def move(self, row: int, col: int, player: int) -> int:




# Your TicTacToe object will be instantiated and called as such:
# obj = TicTacToe(n)
# param_1 = obj.move(row,col,player)
```

Python:

```
class TicTacToe(object):

def __init__(self, n):
"""
:type n: int
"""



def move(self, row, col, player):
"""
:type row: int
```

```
:type col: int
:type player: int
:rtype: int
"""



# Your TicTacToe object will be instantiated and called as such:
# obj = TicTacToe(n)
# param_1 = obj.move(row,col,player)
```

JavaScript:

```
/**
* @param {number} n
*/
var TicTacToe = function(n) {

};

/**
* @param {number} row
* @param {number} col
* @param {number} player
* @return {number}
*/
TicTacToe.prototype.move = function(row, col, player) {

};

/**
* Your TicTacToe object will be instantiated and called as such:
* var obj = new TicTacToe(n)
* var param_1 = obj.move(row,col,player)
*/
```

TypeScript:

```
class TicTacToe {
constructor(n: number) {

}
```

```
move(row: number, col: number, player: number): number {

}
}

/**
* Your TicTacToe object will be instantiated and called as such:
* var obj = new TicTacToe(n)
* var param_1 = obj.move(row,col,player)
*/
```

C#:

```
public class TicTacToe {

public TicTacToe(int n) {

}

public int Move(int row, int col, int player) {

}
}

/**
* Your TicTacToe object will be instantiated and called as such:
* TicTacToe obj = new TicTacToe(n);
* int param_1 = obj.Move(row,col,player);
*/
```

C:

```
typedef struct {

} TicTacToe;


TicTacToe* ticTacToeCreate(int n) {
```

```
}

int ticTacToeMove(TicTacToe* obj, int row, int col, int player) {

}

void ticTacToeFree(TicTacToe* obj) {

}

/**
* Your TicTacToe struct will be instantiated and called as such:
* TicTacToe* obj = ticTacToeCreate(n);
* int param_1 = ticTacToeMove(obj, row, col, player);

* ticTacToeFree(obj);
*/
```

Go:

```
type TicTacToe struct {

}


func Constructor(n int) TicTacToe {

}


func (this *TicTacToe) Move(row int, col int, player int) int {

}


/**
* Your TicTacToe object will be instantiated and called as such:
* obj := Constructor(n);
* param_1 := obj.Move(row,col,player);
*/
```

Kotlin:

```
class TicTacToe(n: Int) {

fun move(row: Int, col: Int, player: Int): Int {

}

}

/**
* Your TicTacToe object will be instantiated and called as such:
* var obj = TicTacToe(n)
* var param_1 = obj.move(row,col,player)
*/
```

Swift:

```
class TicTacToe {

init(_ n: Int) {

}

func move(_ row: Int, _ col: Int, _ player: Int) -> Int {

}
}

/**
* Your TicTacToe object will be instantiated and called as such:
* let obj = TicTacToe(n)
* let ret_1: Int = obj.move(row, col, player)
*/
```

Rust:

```
struct TicTacToe {

}
```

```
/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl TicTacToe {

fn new(n: i32) -> Self {

}

fn make_a_move(&self, row: i32, col: i32, player: i32) -> i32 {

}
}

/**
* Your TicTacToe object will be instantiated and called as such:
* let obj = TicTacToe::new(n);
* let ret_1: i32 = obj.move(row, col, player);
*/
```

Ruby:

```
class TicTacToe

=begin
:type n: Integer
=end
def initialize(n)

end


=begin
:type row: Integer
:type col: Integer
:type player: Integer
:rtype: Integer
=end
def move(row, col, player)

end
```

```
end

# Your TicTacToe object will be instantiated and called as such:
# obj = TicTacToe.new(n)
# param_1 = obj.move(row, col, player)
```

PHP:

```
class TicTacToe {
/**
* @param Integer $n
*/
function __construct($n) {

}

/**
* @param Integer $row
* @param Integer $col
* @param Integer $player
* @return Integer
*/
function move($row, $col, $player) {

}
}

/**
* Your TicTacToe object will be instantiated and called as such:
* $obj = TicTacToe($n);
* $ret_1 = $obj->move($row, $col, $player);
*/
```

Scala:

```
class TicTacToe(_n: Int) {

def move(row: Int, col: Int, player: Int): Int = {

}
```

```
}

/**
* Your TicTacToe object will be instantiated and called as such:
* var obj = new TicTacToe(n)
* var param_1 = obj.move(row,col,player)
*/
```

Elixir:

```
defmodule TicTacToe do
@spec init_(n :: integer) :: any
def init_(n) do

end

@spec move(row :: integer, col :: integer, player :: integer) :: integer
def move(row, col, player) do

end
end

# Your functions will be called as such:
# TicTacToe.init_(n)
# param_1 = TicTacToe.move(row, col, player)

# TicTacToe.init_ will be called before every test case, in which you can do
some necessary initializations.
```

Erlang:

```
-spec tic_tac_toe_init_(N :: integer()) -> any().
tic_tac_toe_init_(N) ->
.

-spec tic_tac_toe_move(Row :: integer(), Col :: integer(), Player ::
integer()) -> integer().
tic_tac_toe_move(Row, Col, Player) ->
.
```

```
%% Your functions will be called as such:
%% tic_tac_toe_init_(N),
%% Param_1 = tic_tac_toe_move(Row, Col, Player),

%% tic_tac_toe_init_ will be called before every test case, in which you can
do some necessary initializations.
```

Racket:

```
(define tic-tac-toe%
(class object%
(super-new)

; n : exact-integer?
(init-field
n)

; move : exact-integer? exact-integer? exact-integer? -> exact-integer?
(define/public (move row col player)

)))

;; Your tic-tac-toe% object will be instantiated and called as such:
;; (define obj (new tic-tac-toe% [n n]))
;; (define param_1 (send obj move row col player))
```

Solutions

C++ Solution:

```
/*
* Problem: Design Tic-Tac-Toe
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/
```

```
class TicTacToe {
public:
TicTacToe(int n) {

}

int move(int row, int col, int player) {

}
};

/**
* Your TicTacToe object will be instantiated and called as such:
* TicTacToe* obj = new TicTacToe(n);
* int param_1 = obj->move(row,col,player);
*/
```

Java Solution:

```
/**
* Problem: Design Tic-Tac-Toe
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class TicTacToe {

public TicTacToe(int n) {

}

public int move(int row, int col, int player) {

}
}

/**
```

```
* Your TicTacToe object will be instantiated and called as such:
* TicTacToe obj = new TicTacToe(n);
* int param_1 = obj.move(row,col,player);
*/
```

Python3 Solution:

```
"""
Problem: Design Tic-Tac-Toe
Difficulty: Medium
Tags: array, hash

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

class TicTacToe:

def __init__(self, n: int):


def move(self, row: int, col: int, player: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class TicTacToe(object):

def __init__(self, n):
"""
:type n: int
"""


def move(self, row, col, player):
"""
:type row: int
:type col: int
:type player: int
```

```
:rtype: int
"""



# Your TicTacToe object will be instantiated and called as such:
# obj = TicTacToe(n)
# param_1 = obj.move(row,col,player)
```

JavaScript Solution:

```
/**
* Problem: Design Tic-Tac-Toe
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* @param {number} n
*/
var TicTacToe = function(n) {

};

/**
* @param {number} row
* @param {number} col
* @param {number} player
* @return {number}
*/
TicTacToe.prototype.move = function(row, col, player) {

};

/**
* Your TicTacToe object will be instantiated and called as such:
* var obj = new TicTacToe(n)
```

```
* var param_1 = obj.move(row,col,player)
*/
```

TypeScript Solution:

```
/**
* Problem: Design Tic-Tac-Toe
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class TicTacToe {
constructor(n: number) {

}

move(row: number, col: number, player: number): number {

}
}

/**
* Your TicTacToe object will be instantiated and called as such:
* var obj = new TicTacToe(n)
* var param_1 = obj.move(row,col,player)
*/
```

C# Solution:

```
/*
* Problem: Design Tic-Tac-Toe
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
```

```
*/

public class TicTacToe {

public TicTacToe(int n) {

}

public int Move(int row, int col, int player) {

}
}

/**
* Your TicTacToe object will be instantiated and called as such:
* TicTacToe obj = new TicTacToe(n);
* int param_1 = obj.Move(row,col,player);
*/
```

C Solution:

```
/*
* Problem: Design Tic-Tac-Toe
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/




typedef struct {

} TicTacToe;


TicTacToe* ticTacToeCreate(int n) {
```

```
}

int ticTacToeMove(TicTacToe* obj, int row, int col, int player) {

}

void ticTacToeFree(TicTacToe* obj) {

}

/**
* Your TicTacToe struct will be instantiated and called as such:
* TicTacToe* obj = ticTacToeCreate(n);
* int param_1 = ticTacToeMove(obj, row, col, player);

* ticTacToeFree(obj);
*/
```

Go Solution:

```
// Problem: Design Tic-Tac-Toe
// Difficulty: Medium
// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

type TicTacToe struct {

}


func Constructor(n int) TicTacToe {

}


func (this *TicTacToe) Move(row int, col int, player int) int {

}
```

```
/**
* Your TicTacToe object will be instantiated and called as such:
* obj := Constructor(n);
* param_1 := obj.Move(row,col,player);
*/
```

Kotlin Solution:

```
class TicTacToe(n: Int) {

fun move(row: Int, col: Int, player: Int): Int {

}

}

/**
* Your TicTacToe object will be instantiated and called as such:
* var obj = TicTacToe(n)
* var param_1 = obj.move(row,col,player)
*/
```

Swift Solution:

```
class TicTacToe {

init(_ n: Int) {

}

func move(_ row: Int, _ col: Int, _ player: Int) -> Int {

}
}

/**
* Your TicTacToe object will be instantiated and called as such:
* let obj = TicTacToe(n)
```

```
* let ret_1: Int = obj.move(row, col, player)
*/
```

Rust Solution:

```
// Problem: Design Tic-Tac-Toe
// Difficulty: Medium
// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

struct TicTacToe {

}


/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl TicTacToe {

fn new(n: i32) -> Self {

}

fn make_a_move(&self, row: i32, col: i32, player: i32) -> i32 {

}
}

/**
* Your TicTacToe object will be instantiated and called as such:
* let obj = TicTacToe::new(n);
* let ret_1: i32 = obj.move(row, col, player);
*/
```

Ruby Solution:

```
class TicTacToe

=begin
:type n: Integer
=end
def initialize(n)

end


=begin
:type row: Integer
:type col: Integer
:type player: Integer
:rtype: Integer
=end
def move(row, col, player)

end



end

# Your TicTacToe object will be instantiated and called as such:
# obj = TicTacToe.new(n)
# param_1 = obj.move(row, col, player)
```

PHP Solution:

```
class TicTacToe {
/**
* @param Integer $n
*/
function __construct($n) {

}

/**
* @param Integer $row
* @param Integer $col
* @param Integer $player
* @return Integer
```

```
*/
function move($row, $col, $player) {

}
}

/**
* Your TicTacToe object will be instantiated and called as such:
* $obj = TicTacToe($n);
* $ret_1 = $obj->move($row, $col, $player);
*/
```

Scala Solution:

```
class TicTacToe(_n: Int) {

def move(row: Int, col: Int, player: Int): Int = {

}

}

/**
* Your TicTacToe object will be instantiated and called as such:
* var obj = new TicTacToe(n)
* var param_1 = obj.move(row,col,player)
*/
```

Elixir Solution:

```
defmodule TicTacToe do
@spec init_(n :: integer) :: any
def init_(n) do

end

@spec move(row :: integer, col :: integer, player :: integer) :: integer
def move(row, col, player) do

end
end
```

```
# Your functions will be called as such:
# TicTacToe.init_(n)
# param_1 = TicTacToe.move(row, col, player)

# TicTacToe.init_ will be called before every test case, in which you can do some necessary initializations.
```

Erlang Solution:

```
-spec tic_tac_toe_init_(N :: integer()) -> any().
tic_tac_toe_init_(N) ->
.

-spec tic_tac_toe_move(Row :: integer(), Col :: integer(), Player :: integer()) -> integer().
tic_tac_toe_move(Row, Col, Player) ->
.


%% Your functions will be called as such:
%% tic_tac_toe_init_(N),
%% Param_1 = tic_tac_toe_move(Row, Col, Player),

%% tic_tac_toe_init_ will be called before every test case, in which you can do some necessary initializations.
```

Racket Solution:

```
(define tic-tac-toe%
(class object%
(super-new)

; n : exact-integer?
(init-field
n)

; move : exact-integer? exact-integer? exact-integer? -> exact-integer?
(define/public (move row col player)

)))
```

```
;; Your tic-tac-toe% object will be instantiated and called as such:
;; (define obj (new tic-tac-toe% [n n]))
;; (define param_1 (send obj move row col player))
```