Problem 1025: Divisor Game

Problem Information

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

Problem Description

Alice and Bob take turns playing a game, with Alice starting first.

Initially, there is a number

n

on the chalkboard. On each player's turn, that player makes a move consisting of:

Choosing any integer

x

with

0 < x < n

and

n % x == 0

.

Replacing the number

n

on the chalkboard with

n - x

.

Also, if a player cannot make a move, they lose the game.

Return

true

if and only if Alice wins the game, assuming both players play optimally

.

Example 1:

Input:

n = 2

Output:

true

Explanation:

Alice chooses 1, and Bob has no more moves.

Example 2:

Input:

n = 3

Output:

false

Explanation:

Alice chooses 1, Bob chooses 1, and Alice has no more moves.

Constraints:

1 <= n <= 1000

Code Snippets

C++:

```
class Solution {
public:
bool divisorGame(int n) {

}
};
```

Java:

```
class Solution {
public boolean divisorGame(int n) {

}
}
```

Python3:

```
class Solution:
def divisorGame(self, n: int) -> bool:
```

Python:

```
class Solution(object):
def divisorGame(self, n):
"""
:type n: int
:rtype: bool
"""
```

JavaScript:

```
/**
* @param {number} n
* @return {boolean}
*/
var divisorGame = function(n) {

};
```

TypeScript:

```
function divisorGame(n: number): boolean {

};
```

C#:

```
public class Solution {
public bool DivisorGame(int n) {

}
}
```

C:

```
bool divisorGame(int n) {

}
```

Go:

```
func divisorGame(n int) bool {

}
```

Kotlin:

```
class Solution {
fun divisorGame(n: Int): Boolean {

}
}
```

Swift:

```
class Solution {
func divisorGame(_ n: Int) -> Bool {

}
}
```

Rust:

```
impl Solution {
pub fn divisor_game(n: i32) -> bool {

}
}
```

Ruby:

```
# @param {Integer} n
# @return {Boolean}
def divisor_game(n)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @return Boolean
*/
function divisorGame($n) {

}
}
```

Dart:

```
class Solution {
bool divisorGame(int n) {

}
```

```
}
```

Scala:

```
object Solution {
def divisorGame(n: Int): Boolean = {

}
}
```

Elixir:

```
defmodule Solution do
@spec divisor_game(n :: integer) :: boolean
def divisor_game(n) do

end
end
```

Erlang:

```
-spec divisor_game(N :: integer()) -> boolean().
divisor_game(N) ->
.
```

Racket:

```
(define/contract (divisor-game n)
(-> exact-integer? boolean?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Divisor Game
* Difficulty: Easy
* Tags: dp, math
*
```

```
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
bool divisorGame(int n) {

}
};
```

Java Solution:

```
/**
* Problem: Divisor Game
* Difficulty: Easy
* Tags: dp, math
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public boolean divisorGame(int n) {

}
}
```

Python3 Solution:

```
"""
Problem: Divisor Game
Difficulty: Easy
Tags: dp, math

Approach: Dynamic programming with memoization or tabulation
Time Complexity: O(n * m) where n and m are problem dimensions
Space Complexity: O(n) or O(n * m) for DP table
"""
```

```
class Solution:
def divisorGame(self, n: int) -> bool:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def divisorGame(self, n):
"""
:type n: int
:rtype: bool
"""
```

JavaScript Solution:

```
/**
* Problem: Divisor Game
* Difficulty: Easy
* Tags: dp, math
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number} n
* @return {boolean}
*/
var divisorGame = function(n) {

};
```

TypeScript Solution:

```
/**
* Problem: Divisor Game
* Difficulty: Easy
* Tags: dp, math
```

```
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

function divisorGame(n: number): boolean {

};
```

C# Solution:

```
/*
* Problem: Divisor Game
* Difficulty: Easy
* Tags: dp, math
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public bool DivisorGame(int n) {

}
}
```

C Solution:

```
/*
* Problem: Divisor Game
* Difficulty: Easy
* Tags: dp, math
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

bool divisorGame(int n) {
```

```
}
```

Go Solution:

```
// Problem: Divisor Game
// Difficulty: Easy
// Tags: dp, math
//
// Approach: Dynamic programming with memoization or tabulation
// Time Complexity: O(n * m) where n and m are problem dimensions
// Space Complexity: O(n) or O(n * m) for DP table

func divisorGame(n int) bool {

}
```

Kotlin Solution:

```
class Solution {
fun divisorGame(n: Int): Boolean {

}
}
```

Swift Solution:

```
class Solution {
func divisorGame(_ n: Int) -> Bool {

}
}
```

Rust Solution:

```
// Problem: Divisor Game
// Difficulty: Easy
// Tags: dp, math
//
// Approach: Dynamic programming with memoization or tabulation
// Time Complexity: O(n * m) where n and m are problem dimensions
```

```
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn divisor_game(n: i32) -> bool {

}
}
```

Ruby Solution:

```
# @param {Integer} n
# @return {Boolean}
def divisor_game(n)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @return Boolean
*/
function divisorGame($n) {

}
}
```

Dart Solution:

```
class Solution {
bool divisorGame(int n) {

}
}
```

Scala Solution:

```
object Solution {
def divisorGame(n: Int): Boolean = {
```

```
}
}
```

Elixir Solution:

```
defmodule Solution do
@spec divisor_game(n :: integer) :: boolean
def divisor_game(n) do

end
end
```

Erlang Solution:

```
-spec divisor_game(N :: integer()) -> boolean().
divisor_game(N) ->
.
```

Racket Solution:

```
(define/contract (divisor-game n)
(-> exact-integer? boolean?)
)
```