Problem 1908: Game of Nim

Problem Information

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

Problem Description

Alice and Bob take turns playing a game with

Alice starting first

.

In this game, there are

n

piles of stones. On each player's turn, the player should remove any

positive

number of stones from a non-empty pile

of his or her choice

. The first player who cannot make a move loses, and the other player wins.

Given an integer array

piles

, where

piles[i]

is the number of stones in the

i

th

pile, return

true

if Alice wins, or

false

if Bob wins

.

Both Alice and Bob play

optimally

.

Example 1:

Input:

piles = [1]

Output:

true

Explanation:

There is only one possible scenario: - On the first turn, Alice removes one stone from the first pile. piles = [0]. - On the second turn, there are no stones left for Bob to remove. Alice wins.

Example 2:

Input:

piles = [1,1]

Output:

false

Explanation:

It can be proven that Bob will always win. One possible scenario is: - On the first turn, Alice removes one stone from the first pile. piles = [0,1]. - On the second turn, Bob removes one stone from the second pile. piles = [0,0]. - On the third turn, there are no stones left for Alice to remove. Bob wins.

Example 3:

Input:

piles = [1,2,3]

Output:

false

Explanation:

It can be proven that Bob will always win. One possible scenario is: - On the first turn, Alice removes three stones from the third pile. piles = [1,2,0]. - On the second turn, Bob removes one stone from the second pile. piles = [1,1,0]. - On the third turn, Alice removes one stone from the first pile. piles = [0,1,0]. - On the fourth turn, Bob removes one stone from the second pile. piles = [0,0,0]. - On the fifth turn, there are no stones left for Alice to remove. Bob wins.

Constraints:

n == piles.length

1 <= n <= 7

1 <= piles[i] <= 7

Follow-up:

Could you find a linear time solution? Although the linear time solution may be beyond the scope of an interview, it could be interesting to know.

Code Snippets

C++:

```
class Solution {
public:
bool nimGame(vector<int>& piles) {

}
};
```

Java:

```
class Solution {
public boolean nimGame(int[] piles) {

}
}
```

Python3:

```
class Solution:
def nimGame(self, piles: List[int]) -> bool:
```

Python:

```
class Solution(object):
def nimGame(self, piles):
"""
```

```
:type piles: List[int]
:rtype: bool
"""
```

JavaScript:

```
/**
* @param {number[]} piles
* @return {boolean}
*/
var nimGame = function(piles) {

};
```

TypeScript:

```
function nimGame(piles: number[]): boolean {

};
```

C#:

```
public class Solution {
public bool NimGame(int[] piles) {

}
}
```

C:

```
bool nimGame(int* piles, int pilesSize) {

}
```

Go:

```
func nimGame(piles []int) bool {

}
```

Kotlin:

```
class Solution {
fun nimGame(piles: IntArray): Boolean {

}
}
```

Swift:

```
class Solution {
func nimGame(_ piles: [Int]) -> Bool {

}
}
```

Rust:

```
impl Solution {
pub fn nim_game(piles: Vec<i32>) -> bool {

}
}
```

Ruby:

```
# @param {Integer[]} piles
# @return {Boolean}
def nim_game(piles)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $piles
* @return Boolean
*/
function nimGame($piles) {

}
}
```

Dart:

```
class Solution {
bool nimGame(List<int> piles) {

}
}
```

Scala:

```
object Solution {
def nimGame(piles: Array[Int]): Boolean = {

}
}
```

Elixir:

```
defmodule Solution do
@spec nim_game(piles :: [integer]) :: boolean
def nim_game(piles) do

end
end
```

Erlang:

```
-spec nim_game(Piles :: [integer()]) -> boolean().
nim_game(Piles) ->
.
```

Racket:

```
(define/contract (nim-game piles)
(-> (listof exact-integer?) boolean?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Game of Nim
* Difficulty: Medium
* Tags: array, dp, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
bool nimGame(vector<int>& piles) {

}
};
```

Java Solution:

```
/**
* Problem: Game of Nim
* Difficulty: Medium
* Tags: array, dp, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public boolean nimGame(int[] piles) {

}
}
```

Python3 Solution:

```
"""
Problem: Game of Nim
Difficulty: Medium
Tags: array, dp, math
```

```
Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

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

Python Solution:

```
class Solution(object):
def nimGame(self, piles):
"""
:type piles: List[int]
:rtype: bool
"""
```

JavaScript Solution:

```
/**
* Problem: Game of Nim
* Difficulty: Medium
* Tags: array, dp, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number[]} piles
* @return {boolean}
*/
var nimGame = function(piles) {

};
```

TypeScript Solution:

```
/**
* Problem: Game of Nim
* Difficulty: Medium
* Tags: array, dp, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function nimGame(piles: number[]): boolean {

};
```

C# Solution:

```
/*
* Problem: Game of Nim
* Difficulty: Medium
* Tags: array, dp, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public bool NimGame(int[] piles) {

}
}
```

C Solution:

```
/*
* Problem: Game of Nim
* Difficulty: Medium
* Tags: array, dp, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
```

```
*/

bool nimGame(int* piles, int pilesSize) {

}
```

Go Solution:

```
// Problem: Game of Nim
// Difficulty: Medium
// Tags: array, dp, math
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func nimGame(piles []int) bool {

}
```

Kotlin Solution:

```
class Solution {
fun nimGame(piles: IntArray): Boolean {

}
}
```

Swift Solution:

```
class Solution {
func nimGame(_ piles: [Int]) -> Bool {

}
}
```

Rust Solution:

```
// Problem: Game of Nim
// Difficulty: Medium
// Tags: array, dp, math
```

```
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn nim_game(piles: Vec<i32>) -> bool {

}
}
```

Ruby Solution:

```
# @param {Integer[]} piles
# @return {Boolean}
def nim_game(piles)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $piles
* @return Boolean
*/
function nimGame($piles) {

}
}
```

Dart Solution:

```
class Solution {
bool nimGame(List<int> piles) {

}
}
```

Scala Solution:

```
object Solution {
def nimGame(piles: Array[Int]): Boolean = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec nim_game(piles :: [integer]) :: boolean
def nim_game(piles) do

end
end
```

Erlang Solution:

```
-spec nim_game(Piles :: [integer()]) -> boolean().
nim_game(Piles) ->
.
```

Racket Solution:

```
(define/contract (nim-game piles)
(-> (listof exact-integer?) boolean?)
)
```