Problem 1900: The Earliest and Latest Rounds Where Players Compete

Problem Information

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

Problem Description

There is a tournament where

n

players are participating. The players are standing in a single row and are numbered from

1

to

n

based on their

initial

standing position (player

1

is the first player in the row, player

2

is the second player in the row, etc.).

The tournament consists of multiple rounds (starting from round number

1

). In each round, the

i

th

player from the front of the row competes against the

i

th

player from the end of the row, and the winner advances to the next round. When the number of players is odd for the current round, the player in the middle automatically advances to the next round.

For example, if the row consists of players

1, 2, 4, 6, 7

Player

1

competes against player

7

.

Player

2

competes against player

6

.

Player

4

automatically advances to the next round.

After each round is over, the winners are lined back up in the row based on the

original ordering

assigned to them initially (ascending order).

The players numbered

firstPlayer

and

secondPlayer

are the best in the tournament. They can win against any other player before they compete against each other. If any two other players compete against each other, either of them might win, and thus you may

choose

the outcome of this round.

Given the integers

n

,

firstPlayer

, and

secondPlayer

, return

an integer array containing two values, the

earliest

possible round number and the

latest

possible round number in which these two players will compete against each other, respectively

.

Example 1:

Input:

n = 11, firstPlayer = 2, secondPlayer = 4

Output:

[3,4]

Explanation:

One possible scenario which leads to the earliest round number: First round: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 Second round: 2, 3, 4, 5, 6, 11 Third round: 2, 3, 4 One possible scenario which leads to the latest round number: First round: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 Second round: 1, 2, 3, 4, 5, 6 Third round: 1, 2, 4 Fourth round: 2, 4

Example 2:

Input:

n = 5, firstPlayer = 1, secondPlayer = 5

Output:

[1,1]

Explanation:

The players numbered 1 and 5 compete in the first round. There is no way to make them compete in any other round.

Constraints:

2 <= n <= 28

1 <= firstPlayer < secondPlayer <= n

Code Snippets

C++:

```
class Solution {
public:
vector<int> earliestAndLatest(int n, int firstPlayer, int secondPlayer) {

}
};
```

Java:

```
class Solution {
public int[] earliestAndLatest(int n, int firstPlayer, int secondPlayer) {

}
}
```

Python3:

```
class Solution:
def earliestAndLatest(self, n: int, firstPlayer: int, secondPlayer: int) ->
List[int]:
```

Python:

```
class Solution(object):
def earliestAndLatest(self, n, firstPlayer, secondPlayer):
"""
:type n: int
:type firstPlayer: int
:type secondPlayer: int
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number} n
* @param {number} firstPlayer
* @param {number} secondPlayer
* @return {number[]}
*/
var earliestAndLatest = function(n, firstPlayer, secondPlayer) {

};
```

TypeScript:

```
function earliestAndLatest(n: number, firstPlayer: number, secondPlayer:
number): number[] {

};
```

C#:

```
public class Solution {
public int[] EarliestAndLatest(int n, int firstPlayer, int secondPlayer) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* earliestAndLatest(int n, int firstPlayer, int secondPlayer, int*
returnSize) {

}
```

Go:

```
func earliestAndLatest(n int, firstPlayer int, secondPlayer int) []int {

}
```

Kotlin:

```
class Solution {
fun earliestAndLatest(n: Int, firstPlayer: Int, secondPlayer: Int): IntArray
{

}
}
```

Swift:

```
class Solution {
func earliestAndLatest(_ n: Int, _ firstPlayer: Int, _ secondPlayer: Int) ->
[Int] {

}
}
```

Rust:

```
impl Solution {
pub fn earliest_and_latest(n: i32, first_player: i32, second_player: i32) ->
Vec<i32> {

}
}
```

Ruby:

```
# @param {Integer} n
# @param {Integer} first_player
# @param {Integer} second_player
# @return {Integer[]}
def earliest_and_latest(n, first_player, second_player)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @param Integer $firstPlayer
* @param Integer $secondPlayer
* @return Integer[]
*/
function earliestAndLatest($n, $firstPlayer, $secondPlayer) {

}
}
```

Dart:

```
class Solution {
List<int> earliestAndLatest(int n, int firstPlayer, int secondPlayer) {

}
}
```

Scala:

```
object Solution {
def earliestAndLatest(n: Int, firstPlayer: Int, secondPlayer: Int):
Array[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec earliest_and_latest(n :: integer, first_player :: integer,
second_player :: integer) :: [integer]
def earliest_and_latest(n, first_player, second_player) do

end
end
```

Erlang:

```
-spec earliest_and_latest(N :: integer(), FirstPlayer :: integer(),
SecondPlayer :: integer()) -> [integer()].
earliest_and_latest(N, FirstPlayer, SecondPlayer) ->
.
```

Racket:

```
(define/contract (earliest-and-latest n firstPlayer secondPlayer)
(-> exact-integer? exact-integer? exact-integer? (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: The Earliest and Latest Rounds Where Players Compete
* Difficulty: Hard
* Tags: array, dp
*
* 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:
vector<int> earliestAndLatest(int n, int firstPlayer, int secondPlayer) {

}
};
```

Java Solution:

```
/**
* Problem: The Earliest and Latest Rounds Where Players Compete
* Difficulty: Hard
* Tags: array, dp
*
* 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 int[] earliestAndLatest(int n, int firstPlayer, int secondPlayer) {

}
}
```

Python3 Solution:

```
"""
Problem: The Earliest and Latest Rounds Where Players Compete
Difficulty: Hard
Tags: array, dp

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 earliestAndLatest(self, n: int, firstPlayer: int, secondPlayer: int) ->
List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def earliestAndLatest(self, n, firstPlayer, secondPlayer):
"""
:type n: int
```

```
:type firstPlayer: int
:type secondPlayer: int
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: The Earliest and Latest Rounds Where Players Compete
* Difficulty: Hard
* Tags: array, dp
*
* 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} n
* @param {number} firstPlayer
* @param {number} secondPlayer
* @return {number[]}
*/
var earliestAndLatest = function(n, firstPlayer, secondPlayer) {

};
```

TypeScript Solution:

```
/**
* Problem: The Earliest and Latest Rounds Where Players Compete
* Difficulty: Hard
* Tags: array, dp
*
* 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 earliestAndLatest(n: number, firstPlayer: number, secondPlayer:
number): number[] {
```

```
};
```

C# Solution:

```
/*
* Problem: The Earliest and Latest Rounds Where Players Compete
* Difficulty: Hard
* Tags: array, dp
*
* 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 int[] EarliestAndLatest(int n, int firstPlayer, int secondPlayer) {

}
}
```

C Solution:

```
/*
* Problem: The Earliest and Latest Rounds Where Players Compete
* Difficulty: Hard
* Tags: array, dp
*
* 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
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* earliestAndLatest(int n, int firstPlayer, int secondPlayer, int*
returnSize) {

}
```

Go Solution:

```
// Problem: The Earliest and Latest Rounds Where Players Compete
// Difficulty: Hard
// Tags: array, dp
//
// 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 earliestAndLatest(n int, firstPlayer int, secondPlayer int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun earliestAndLatest(n: Int, firstPlayer: Int, secondPlayer: Int): IntArray
{

}
}
```

Swift Solution:

```
class Solution {
func earliestAndLatest(_ n: Int, _ firstPlayer: Int, _ secondPlayer: Int) ->
[Int] {

}
}
```

Rust Solution:

```
// Problem: The Earliest and Latest Rounds Where Players Compete
// Difficulty: Hard
// Tags: array, dp
//
// 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 earliest_and_latest(n: i32, first_player: i32, second_player: i32) ->
Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer} n
# @param {Integer} first_player
# @param {Integer} second_player
# @return {Integer[]}
def earliest_and_latest(n, first_player, second_player)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @param Integer $firstPlayer
* @param Integer $secondPlayer
* @return Integer[]
*/
function earliestAndLatest($n, $firstPlayer, $secondPlayer) {

}
}
```

Dart Solution:

```
class Solution {
List<int> earliestAndLatest(int n, int firstPlayer, int secondPlayer) {

}
}
```

Scala Solution:

```
object Solution {
def earliestAndLatest(n: Int, firstPlayer: Int, secondPlayer: Int):
Array[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec earliest_and_latest(n :: integer, first_player :: integer,
second_player :: integer) :: [integer]
def earliest_and_latest(n, first_player, second_player) do

end
end
```

Erlang Solution:

```
-spec earliest_and_latest(N :: integer(), FirstPlayer :: integer(),
SecondPlayer :: integer()) -> [integer()].
earliest_and_latest(N, FirstPlayer, SecondPlayer) ->
.
```

Racket Solution:

```
(define/contract (earliest-and-latest n firstPlayer secondPlayer)
(-> exact-integer? exact-integer? exact-integer? (listof exact-integer?))
)
```