Problem 1560: Most Visited Sector in a Circular Track

Problem Information

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

Problem Description

Given an integer

n

and an integer array

rounds

. We have a circular track which consists of

n

sectors labeled from

1

to

n

. A marathon will be held on this track, the marathon consists of

m

rounds. The

i

th

round starts at sector

rounds[i - 1]

and ends at sector

rounds[i]

. For example, round 1 starts at sector

rounds[0]

and ends at sector

rounds[1]

Return

an array of the most visited sectors

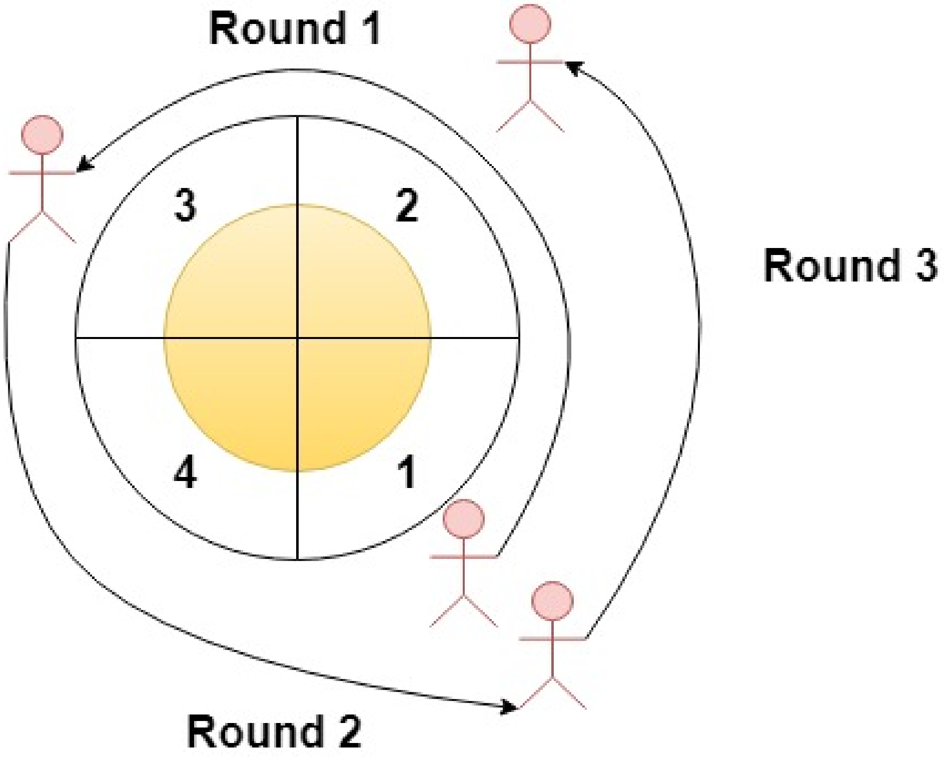
sorted in

ascending

order.

Notice that you circulate the track in ascending order of sector numbers in the counter-clockwise direction (See the first example).

Example 1:



Input:

n = 4, rounds = [1,3,1,2]

Output:

[1,2]

Explanation:

The marathon starts at sector 1. The order of the visited sectors is as follows: 1 --> 2 --> 3 (end of round 1) --> 4 --> 1 (end of round 2) --> 2 (end of round 3 and the marathon) We can see that both sectors 1 and 2 are visited twice and they are the most visited sectors. Sectors 3 and 4 are visited only once.

Example 2:

Input:

n = 2, rounds = [2,1,2,1,2,1,2,1,2]

Output:

[2]

Example 3:

Input:

n = 7, rounds = [1,3,5,7]

Output:

[1,2,3,4,5,6,7]

Constraints:

2 <= n <= 100

1 <= m <= 100

rounds.length == m + 1

1 <= rounds[i] <= n

rounds[i] != rounds[i + 1]

for

0 <= i < m

Code Snippets

C++:

```
class Solution {
public:
vector<int> mostVisited(int n, vector<int>& rounds) {

}
};
```

Java:

```
class Solution {
public List<Integer> mostVisited(int n, int[] rounds) {

}
}
```

Python3:

```
class Solution:
def mostVisited(self, n: int, rounds: List[int]) -> List[int]:
```

Python:

```
class Solution(object):
def mostVisited(self, n, rounds):
"""
:type n: int
:type rounds: List[int]
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number} n
* @param {number[]} rounds
* @return {number[]}
*/
var mostVisited = function(n, rounds) {

};
```

TypeScript:

```
function mostVisited(n: number, rounds: number[]): number[] {

};
```

C#:

```
public class Solution {
public IList<int> MostVisited(int n, int[] rounds) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* mostVisited(int n, int* rounds, int roundsSize, int* returnSize) {

}
```

Go:

```
func mostVisited(n int, rounds []int) []int {

}
```

Kotlin:

```
class Solution {
fun mostVisited(n: Int, rounds: IntArray): List<Int> {

}
}
```

Swift:

```
class Solution {
func mostVisited(_ n: Int, _ rounds: [Int]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn most_visited(n: i32, rounds: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby:

```
# @param {Integer} n
# @param {Integer[]} rounds
# @return {Integer[]}
def most_visited(n, rounds)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @param Integer[] $rounds
* @return Integer[]
*/
function mostVisited($n, $rounds) {

}
}
```

Dart:

```
class Solution {
List<int> mostVisited(int n, List<int> rounds) {

}
}
```

Scala:

```
object Solution {
def mostVisited(n: Int, rounds: Array[Int]): List[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec most_visited(n :: integer, rounds :: [integer]) :: [integer]
def most_visited(n, rounds) do

end
end
```

Erlang:

```
-spec most_visited(N :: integer(), Rounds :: [integer()]) -> [integer()].
most_visited(N, Rounds) ->
.
```

Racket:

```
(define/contract (most-visited n rounds)
(-> exact-integer? (listof exact-integer?) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Most Visited Sector in a Circular Track
* Difficulty: Easy
* Tags: array, sort
*
* 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:
vector<int> mostVisited(int n, vector<int>& rounds) {

}
};
```

Java Solution:

```
/**
* Problem: Most Visited Sector in a Circular Track
* Difficulty: Easy
* Tags: array, sort
*
* 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 List<Integer> mostVisited(int n, int[] rounds) {

}
}
```

Python3 Solution:

```
"""
Problem: Most Visited Sector in a Circular Track
Difficulty: Easy
Tags: array, sort

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 mostVisited(self, n: int, rounds: List[int]) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def mostVisited(self, n, rounds):
"""
:type n: int
:type rounds: List[int]
:rtype: List[int]
"""
```

JavaScript Solution:

```javascript
/**
* Problem: Most Visited Sector in a Circular Track
* Difficulty: Easy
* Tags: array, sort
*
* 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} n
* @param {number[]} rounds
* @return {number[]}
*/
var mostVisited = function(n, rounds) {

};
```

TypeScript Solution:

```typescript
/**
* Problem: Most Visited Sector in a Circular Track
* Difficulty: Easy
* Tags: array, sort
*
* 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 mostVisited(n: number, rounds: number[]): number[] {

};
```

C# Solution:

```csharp
/*
* Problem: Most Visited Sector in a Circular Track
* Difficulty: Easy
* Tags: array, sort
*
* 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 IList<int> MostVisited(int n, int[] rounds) {

}
}
```

C Solution:

```
/*
* Problem: Most Visited Sector in a Circular Track
* Difficulty: Easy
* Tags: array, sort
*
* 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
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* mostVisited(int n, int* rounds, int roundsSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Most Visited Sector in a Circular Track
// Difficulty: Easy
// Tags: array, sort
//
// 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 mostVisited(n int, rounds []int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun mostVisited(n: Int, rounds: IntArray): List<Int> {

}
}
```

Swift Solution:

```
class Solution {
func mostVisited(_ n: Int, _ rounds: [Int]) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Most Visited Sector in a Circular Track
// Difficulty: Easy
// Tags: array, sort
//
// 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 most_visited(n: i32, rounds: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer} n
# @param {Integer[]} rounds
# @return {Integer[]}
def most_visited(n, rounds)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @param Integer[] $rounds
* @return Integer[]
*/
function mostVisited($n, $rounds) {

}
}
```

Dart Solution:

```
class Solution {
List<int> mostVisited(int n, List<int> rounds) {

}
}
```

Scala Solution:

```
object Solution {
def mostVisited(n: Int, rounds: Array[Int]): List[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec most_visited(n :: integer, rounds :: [integer]) :: [integer]
def most_visited(n, rounds) do
```

```
end
end
```

Erlang Solution:

```
-spec most_visited(N :: integer(), Rounds :: [integer()]) -> [integer()].
most_visited(N, Rounds) ->
.
```

Racket Solution:

```
(define/contract (most-visited n rounds)
(-> exact-integer? (listof exact-integer?) (listof exact-integer?))
)
```