

Problem 2655: Find Maximal Uncovered Ranges

Problem Information

Difficulty: Medium

Acceptance Rate: 0.00%

Paid Only: No

Problem Description

You are given an integer

n

which is the length of a

0-indexed

array

nums

, and a

0-indexed

2D-array

ranges

, which is a list of sub-ranges of

nums

(sub-ranges may

overlap

).

Each row

`ranges[i]`

has exactly 2 cells:

`ranges[i][0]`

, which shows the start of the i

th

range (inclusive)

`ranges[i][1]`

, which shows the end of the i

th

range (inclusive)

These ranges cover some cells of

`nums`

and leave some cells uncovered. Your task is to find all of the

uncovered

ranges with

maximal

length.

Return

a 2D-array

answer

of the uncovered ranges,

sorted

by the starting point in

ascending order

.

By all of the

uncovered

ranges with

maximal

length, we mean satisfying two conditions:

Each uncovered cell should belong to

exactly

one sub-range

There should

not exist

two ranges (l

1

, r

1

) and (l

2

, r

2

) such that r

1

+ 1 = l

2

Example 1:

Input:

n = 10, ranges = [[3,5],[7,8]]

Output:

[[0,2],[6,6],[9,9]]

Explanation:

The ranges (3, 5) and (7, 8) are covered, so if we simplify the array nums to a binary array where 0 shows an uncovered cell and 1 shows a covered cell, the array becomes [0,0,0,1,1,1,0,1,1,0] in which we can observe that the ranges (0, 2), (6, 6) and (9, 9) aren't covered.

Example 2:

Input:

$n = 3$, $\text{ranges} = [[0,2]]$

Output:

`[]`

Explanation:

In this example, the whole of the array `nums` is covered and there are no uncovered cells so the output is an empty array.

Example 3:

Input:

$n = 7$, $\text{ranges} = [[2,4],[0,3]]$

Output:

`[[5,6]]`

Explanation:

The ranges $(0, 3)$ and $(2, 4)$ are covered, so if we simplify the array `nums` to a binary array where 0 shows an uncovered cell and 1 shows a covered cell, the array becomes `[1,1,1,1,1,0,0]` in which we can observe that the range $(5, 6)$ is uncovered.

Constraints:

$1 \leq n \leq 10$

9

$0 \leq \text{ranges.length} \leq 10$

6

`ranges[i].length = 2`

`0 <= ranges[i][j] <= n - 1`

`ranges[i][0] <= ranges[i][1]`

Code Snippets

C++:

```
class Solution {
public:
    vector<vector<int>> findMaximalUncoveredRanges(int n, vector<vector<int>>&
    ranges) {

    }
};
```

Java:

```
class Solution {
    public int[][] findMaximalUncoveredRanges(int n, int[][] ranges) {

    }
}
```

Python3:

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

Python:

```
class Solution(object):
    def findMaximalUncoveredRanges(self, n, ranges):
        """
        :type n: int
        :type ranges: List[List[int]]
```

```

:rtype: List[List[int]]
"""

```

JavaScript:

```

/**
 * @param {number} n
 * @param {number[][]} ranges
 * @return {number[][]}
 */
var findMaximalUncoveredRanges = function(n, ranges) {

};

```

TypeScript:

```

function findMaximalUncoveredRanges(n: number, ranges: number[][]):
number[][] {

};

```

C#:

```

public class Solution {
    public int[][] FindMaximalUncoveredRanges(int n, int[][] ranges) {

    }
}

```

C:

```

/**
 * Return an array of arrays of size *returnSize.
 * The sizes of the arrays are returned as *returnColumnSizes array.
 * Note: Both returned array and *columnSizes array must be malloced, assume
 caller calls free().
 */
int** findMaximalUncoveredRanges(int n, int** ranges, int rangesSize, int*
rangesColSize, int* returnSize, int** returnColumnSizes) {

}

```

Go:

```
func findMaximalUncoveredRanges(n int, ranges [][]int) [][]int {  
  
}
```

Kotlin:

```
class Solution {  
    fun findMaximalUncoveredRanges(n: Int, ranges: Array<IntArray>):  
        Array<IntArray> {  
  
    }  
}
```

Swift:

```
class Solution {  
    func findMaximalUncoveredRanges(_ n: Int, _ ranges: [[Int]]) -> [[Int]] {  
  
    }  
}
```

Rust:

```
impl Solution {  
    pub fn find_maximal_uncovered_ranges(n: i32, ranges: Vec<Vec<i32>>) ->  
        Vec<Vec<i32>> {  
  
    }  
}
```

Ruby:

```
# @param {Integer} n  
# @param {Integer[][]} ranges  
# @return {Integer[][]}  
def find_maximal_uncovered_ranges(n, ranges)  
  
end
```

PHP:

```

class Solution {

    /**
     * @param Integer $n
     * @param Integer[][] $ranges
     * @return Integer[][]
     */
    function findMaximalUncoveredRanges($n, $ranges) {

    }

}

```

Dart:

```

class Solution {
  List<List<int>> findMaximalUncoveredRanges(int n, List<List<int>> ranges) {

  }
}

```

Scala:

```

object Solution {
  def findMaximalUncoveredRanges(n: Int, ranges: Array[Array[Int]]):
    Array[Array[Int]] = {

  }
}

```

Elixir:

```

defmodule Solution do
  @spec find_maximal_uncovered_ranges(n :: integer, ranges :: [[integer]]) ::
    [[integer]]
  def find_maximal_uncovered_ranges(n, ranges) do

  end
end

```

Erlang:

```

-spec find_maximal_uncovered_ranges(N :: integer(), Ranges :: [[integer()]])
-> [[integer()]].

```

```
find_maximal_uncovered_ranges(N, Ranges) ->  
.
```

Racket:

```
(define/contract (find-maximal-uncovered-ranges n ranges)  
  (-> exact-integer? (listof (listof exact-integer?)) (listof (listof  
    exact-integer?))))  
)
```

Solutions

C++ Solution:

```
/*  
 * Problem: Find Maximal Uncovered Ranges  
 * Difficulty: Medium  
 * 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<vector<int>> findMaximalUncoveredRanges(int n, vector<vector<int>>&  
    ranges) {  
  
    }  
};
```

Java Solution:

```
/**  
 * Problem: Find Maximal Uncovered Ranges  
 * Difficulty: Medium  
 * 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 int[][] findMaximalUncoveredRanges(int n, int[][] ranges) {

}
}

```

Python3 Solution:

```

"""
Problem: Find Maximal Uncovered Ranges
Difficulty: Medium
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 findMaximalUncoveredRanges(self, n: int, ranges: List[List[int]]) ->
    List[List[int]]:
        # TODO: Implement optimized solution
        pass

```

Python Solution:

```

class Solution(object):
    def findMaximalUncoveredRanges(self, n, ranges):
        """
        :type n: int
        :type ranges: List[List[int]]
        :rtype: List[List[int]]
        """

```

JavaScript Solution:

```

/**
 * Problem: Find Maximal Uncovered Ranges
 * Difficulty: Medium
 * 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[][]} ranges
 * @return {number[][]}
 */
var findMaximalUncoveredRanges = function(n, ranges) {

};

```

TypeScript Solution:

```

/**
 * Problem: Find Maximal Uncovered Ranges
 * Difficulty: Medium
 * 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 findMaximalUncoveredRanges(n: number, ranges: number[][]):
number[][] {

};

```

C# Solution:

```

/*
 * Problem: Find Maximal Uncovered Ranges
 * Difficulty: Medium
 * 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 int[][] FindMaximalUncoveredRanges(int n, int[][] ranges) {

}
}

```

C Solution:

```

/*
* Problem: Find Maximal Uncovered Ranges
* Difficulty: Medium
* 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
*/

/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** findMaximalUncoveredRanges(int n, int** ranges, int rangesSize, int*
rangesColSize, int* returnSize, int** returnColumnSizes) {

}

```

Go Solution:

```

// Problem: Find Maximal Uncovered Ranges
// Difficulty: Medium
// 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 findMaximalUncoveredRanges(n int, ranges [][]int) [][]int {

}

```

Kotlin Solution:

```

class Solution {
    fun findMaximalUncoveredRanges(n: Int, ranges: Array<IntArray>):
        Array<IntArray> {

    }
}

```

Swift Solution:

```

class Solution {
    func findMaximalUncoveredRanges(_ n: Int, _ ranges: [[Int]]) -> [[Int]] {

    }
}

```

Rust Solution:

```

// Problem: Find Maximal Uncovered Ranges
// Difficulty: Medium
// 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 find_maximal_uncovered_ranges(n: i32, ranges: Vec<Vec<i32>>) ->
        Vec<Vec<i32>> {

    }
}

```

Ruby Solution:

```
# @param {Integer} n
# @param {Integer[][]} ranges
# @return {Integer[][]}

def find_maximal_uncovered_ranges(n, ranges)

end
```

PHP Solution:

```
class Solution {

    /**
     * @param Integer $n
     * @param Integer[][] $ranges
     * @return Integer[][]
     */
    function findMaximalUncoveredRanges($n, $ranges) {

    }

}
```

Dart Solution:

```
class Solution {
  List<List<int>> findMaximalUncoveredRanges(int n, List<List<int>> ranges) {

  }
}
```

Scala Solution:

```
object Solution {
  def findMaximalUncoveredRanges(n: Int, ranges: Array[Array[Int]]):
    Array[Array[Int]] = {

    }
}
```

Elixir Solution:

```

defmodule Solution do
  @spec find_maximal_uncovered_ranges(n :: integer, ranges :: [[integer]]) ::
    [[integer]]
  def find_maximal_uncovered_ranges(n, ranges) do

  end
end

```

Erlang Solution:

```

-spec find_maximal_uncovered_ranges(N :: integer(), Ranges :: [[integer()]])
-> [[integer()]].
find_maximal_uncovered_ranges(N, Ranges) ->
.

```

Racket Solution:

```

(define/contract (find-maximal-uncovered-ranges n ranges)
  (-> exact-integer? (listof (listof exact-integer?)) (listof (listof
exact-integer?)))
)

```