2655. Find Maximal Uncovered Ranges

Medium <u>Array</u> Sorting

## **Problem Description**

In this problem, you are given two inputs: an integer n, specifying the length of a 0-indexed array nums, and a 2D-array ranges. Each entry in ranges represents a sub-range of the array nums with a start and end index (inclusive). These ranges can overlap

each other. The task is to identify and return all the sub-ranges of nums that are maximally uncovered by any of the ranges

provided in ranges. An uncovered range is considered maximal if: 1. Each cell within the uncovered range belongs to just one uncovered sub-range. This implies that uncovered sub-ranges cannot overlap. 2. There are no such consecutive uncovered ranges that the end of one is immediately followed by the start of another. Essentially, no two uncovered ranges should be adjacent.

The expected output is a 2D-array of the uncovered ranges, sorted by their start index in ascending order.

Intuition

an uncovered sub-range exists from last + 1 to n - 1, and we append this range to our answer.

to easily compare the current range with the last uncovered range found.

The intuition behind the solution approach can be broken down into the following steps:

**Sorting the Ranges:** First, we sort the given ranges by their start indices. Sorting helps us to process the ranges sequentially, thus making it simpler to find any gaps between successive ranges.

- Finding Uncovered Ranges: We initialize a variable last that keeps track of the end index of the last covered sub-range (initialized to -1 since the array is 0-indexed). We then iterate over the sorted ranges, and for each range (1, r): a. We check if there is an uncovered range that starts after the end of the last covered range (last + 1) and ends before the
- start of the current range (1 1). If such an uncovered range exists, we add it to the answer. b. We update last to be the maximum of the current end index r and the previously stored last, as this keeps last pointing
- to the end of the current furthest covered range. Checking for Rightmost Uncovered Range: After processing all the given ranges, there might still be an uncovered range right at the end of the nums array. We check if the index one more than the last covered index (last + 1) is less than n. If it is,

Returning the Result: Finally, the 2D-array ans contains all the maximally uncovered ranges and is returned as the solution.

- **Solution Approach** The implementation of the solution uses a straightforward approach to solve the problem. Here's a step-by-step explanation:
- Sort Ranges: The first step is to sort the ranges array. This is done with the sort() method, which sorts the list in place. Sorting is based on the first element of each sub-list, which represents the start of the range. Sorting is critical as it allows us

### Initialize Variables: We initialize a list ans to collect the answers and a variable last to keep track of the end of the last

ranges.sort()

covered range. The last variable starts at -1 since the array is 0-indexed and we want to find if there's an uncovered range

starting from the first index (0). last = -1ans = [] Iterate and Find Uncovered Ranges: We loop through each range (1, r) in the sorted ranges list. For each range, two main

a. Uncovered Range Check: If the start of the current range (1) is greater than last + 1, there is an uncovered range

```
between last + 1 and l - 1. This range is added to the ans list.
`python
```

```python

last = max(last, r)

**if** last + 1 < n:

checks occur:

if last + 1 < l: ans.append([last + 1, l - 1])

b. Update Last Covered: We then update last to be the maximum of last and the current range's end r. This step effectively

```
Check for Tail Uncovered Range: After the loop, we check if there's an uncovered range from the end of the last covered
```

skips over any overlap and ensures that we extend our covered area to include the current range.

range to the end of the array nums. If such an uncovered range exists, it is added to ans.

the final answer. return ans

The simple but effective approach works due to the properties of sorting and the greedy method of extending the covered range

Suppose we are given an integer n = 10, representing an array nums of length 10, and a 2D-array ranges = [[1, 3], [6, 9], [2,

**Sort Ranges**: Sort the ranges array to process in order. After sorting based on start index, we get [[1, 3], [2, 5], [6, 9]].

to the furthest end reached by any range. The implementation is efficient and straightforward, not requiring any complex

Return Answer: The ans list now contains all the maximal uncovered ranges, sorted by starting point, and can be returned as

```
Let's consider a small example to illustrate the solution approach.
```

Following the solution approach:

0]]. Update last to 3.

is not larger than last).

5]]. We want to find all the maximally uncovered sub-ranges in nums.

algorithms or data structures.

**Example Walkthrough** 

ans.append([last + 1, n - 1])

**Initialize Variables:** We initialize last = -1 and last = []. Iterate and Find Uncovered Ranges: • For the first range [1, 3], since last + 1 (0) is less than 1, there's an uncovered range [0, 0]. We add it to ans, resulting in ans = [[0,

• For the second range [2, 5], last + 1 (4) is not less than 2, so no new uncovered range is added. Update last to 5 (the current end index

covered range (last = 9) to the length of nums (n = 10). Since last + 1 (10) is less than n, there's an uncovered range [10,

This example helps us understand how the solution approach works step by step to identify uncovered ranges in the array nums.

#### • For the third range [6, 9], last + 1 (6) is equal to the start index, so again, no uncovered range is added. Update last to 9. Check for Tail Uncovered Range: After processing all ranges, we check for any uncovered range from the end of the last

**Python** 

- 9], which is invalid because the start is greater than the end. Thus, no range is added to ans. **Return Answer**: The final answer is ans = [[0, 0]], which is the list of all maximally uncovered ranges in nums.
- Solution Implementation
- from typing import List # Import List from typing for type annotations class Solution: def findMaximalUncoveredRanges(self, n: int, ranges: List[List[int]]) -> List[List[int]]: # Sort the ranges on the basis of start of each range
- # Check if there's a gap between this range and the last covered index if last covered + 1 < left:</pre> # Append the uncovered range to the list uncovered\_ranges.append([last\_covered + 1, left - 1])

# Update the last covered index to be the maximum of current right or the previous last\_covered

# Initialize the last seen covered index to -1 (outside the range of indices)

# After iterating ranges, check if there's still an uncovered range up to n - 1

# Initialize a list to store the maximal uncovered ranges

last\_covered = max(last\_covered, right)

# Append the final uncovered range if any

uncovered\_ranges.append([last\_covered + 1, n - 1])

```
# Return the list of uncovered ranges
       return uncovered_ranges
Java
```

import java.util.Arrays;

import java.util.List;

class Solution {

import java.util.ArrayList;

ranges.sort()

last covered = -1

uncovered\_ranges = []

# Iterate over the sorted ranges

for left, right in ranges:

if last covered + 1 < n:</pre>

```
/**
* Finds the maximal uncovered ranges given a set of ranges and a limit.
* @param n The upper limit (exclusive) for the range of numbers considered.
* @param ranges An array of integer arrays, each representing a covered range [start, end].
* @return A 2D integer array with the maximal uncovered ranges.
*/
public int[][] findMaximalUncoveredRanges(int n, int[][] ranges) {
   // Sort the input ranges by the start of each range.
   Arrays.sort(ranges, (a, b) -> Integer.compare(a[0], b[0]));
   // Initialize the last covered point to -1.
   int lastCovered = -1;
   // Prepare a list to hold the uncovered ranges.
   List<int[]> uncoveredRanges = new ArrayList<>();
   // Loop over each range.
    for (int[] range : ranges) {
        int start = range[0];
        int end = range[1];
       // If there is a gap between the last covered point and current range's start,
       // then it is an uncovered range.
       if (lastCovered + 1 < start) {</pre>
            uncoveredRanges.add(new int[] {lastCovered + 1, start - 1});
       // Update the last covered point to the maximum of the current last and range's end.
        lastCovered = Math.max(lastCovered, end);
   // After processing all ranges, if there are still uncovered numbers until n,
   // add the final uncovered range.
   if (lastCovered + 1 < n) {</pre>
       uncoveredRanges.add(new int[] {lastCovered + 1, n - 1});
   // Convert the list of uncovered ranges to a 2D array and return.
   return uncoveredRanges.toArray(new int[uncoveredRanges.size()][]);
```

```
TypeScript
```

});

**}**;

C++

public:

#include <vector>

class Solution {

#include <algorithm>

});

int lastCovered = -1;

```
// Check for an uncovered range before the start of the current range.
    if (lastCovered + 1 < start) {</pre>
        // Add the uncovered range to the uncoveredRanges array.
        uncoveredRanges.push([lastCovered + 1, start - 1]);
    // Update the lastCovered marker with the maximum of lastCovered and current end point.
    lastCovered = Math.max(lastCovered, end);
});
```

// Finds maximal uncovered ranges given an upper bound 'n' and a list of ranges

// Sort the ranges based on their start points

return firstRange[0] < secondRange[0];</pre>

int start = range[0], end = range[1];

lastCovered = max(lastCovered, end);

// Add the uncovered range to the answer list

uncoveredRanges.push\_back({lastCovered + 1, start - 1});

// Add the uncovered range up to 'n - 1' to the answer list

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

// After processing all ranges, check for an uncovered range at the end.

uncoveredRanges.push([lastCovered + 1, n - 1]);

// Return the array containing all the uncovered ranges.

// Add the final uncovered range up to 'n - 1' to the uncoveredRanges array.

// Sort the given ranges based on their starting points.

ranges.sort((firstRange, secondRange) => {

return firstRange[0] - secondRange[0];

uncoveredRanges.push\_back({lastCovered + 1, n - 1});

vector<vector<int>> uncoveredRanges;

if (lastCovered + 1 < start) {</pre>

// Iterate through each range

for (auto& range : ranges) {

if (lastCovered + 1 < n) {</pre>

return uncoveredRanges;

let lastCovered: number = -1;

ranges.forEach((range) => {

if (lastCovered + 1 < n) {</pre>

return uncoveredRanges;

let uncoveredRanges: number[][] = [];

// Iterate through the sorted ranges.

const [start, end] = range;

vector<vector<int>> findMaximalUncoveredRanges(int n, vector<vector<int>>& ranges) {

// Check if there is an uncovered range before the start of the current range

// Update the last covered position with the furthest point covered so far

// After all ranges are processed, check if there's an uncovered range at the end

sort(ranges.begin(), ranges.end(), [](const vector<int>& firstRange, const vector<int>& secondRange) {

```
from typing import List # Import List from typing for type annotations
class Solution:
   def findMaximalUncoveredRanges(self, n: int, ranges: List[List[int]]) -> List[List[int]]:
       # Sort the ranges on the basis of start of each range
        ranges.sort()
       # Initialize the last seen covered index to -1 (outside the range of indices)
        last\_covered = -1
       # Initialize a list to store the maximal uncovered ranges
       uncovered_ranges = []
       # Iterate over the sorted ranges
        for left, right in ranges:
           # Check if there's a gap between this range and the last covered index
            if last_covered + 1 < left:</pre>
                # Append the uncovered range to the list
                uncovered_ranges.append([last_covered + 1, left - 1])
```

# After iterating ranges, check if there's still an uncovered range up to n - 1

# Update the last covered index to be the maximum of current right or the previous last\_covered

# **Sorting the ranges:** The sort operation for the ranges at the beginning of the function has a time complexity of O(m log m),

last\_covered = max(last\_covered, right)

# Append the final uncovered range if any

# Return the list of uncovered ranges

uncovered\_ranges.append([last\_covered + 1, n - 1])

if last\_covered + 1 < n:</pre>

return uncovered\_ranges

Time and Space Complexity

the sorted ranges list.

where m is the number of intervals in the ranges list. Iterating through the sorted ranges: The iteration through ranges involves comparing and possibly appending to the ans list. This iteration occurs once for each element in the ranges list, giving it a time complexity of O(m).

The time complexity of the provided code primarily depends on the sorting of the ranges and the subsequent iteration through

- Combining the above steps: Since the sort operation dominates the overall time complexity, we can state that the overall 3. time complexity of the function is  $O(m \log m)$ .
- The space complexity of the code comes from the additional memory used to store the ans list. 1. Space for ans list: In the worst case, the ans list could potentially store every single uncovered range between adjacent ranges in the sorted list

as well as the uncovered range before the first range and after the last range if applicable. This worst-case scenario happens when none of the

ranges overlap at all, which theoretically could create up to m+1 entries in the ans list in the case where each interval has a gap with adjacent

- intervals. Thus, the space complexity is O(m). In summary:
  - The **Space Complexity** of the code is O(m).

• The **Time Complexity** of the code is  $0 (m \log m)$ .