2655. Find Maximal Uncovered Ranges

Medium Array Sorting Leetcode Link

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

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

1. **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.

2. **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 `(l, 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 (`l - 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.

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

4. **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:

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

   ```
   ranges.sort()
   ```

2. **Initialize Variables**: We initialize a list `ans` to collect the answers and a variable `last` to keep track of the end of the last 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 = -1
   ans = []
   ```

3. **Iterate and Find Uncovered Ranges**: We loop through each range `(l, r)` in the sorted `ranges` list. For each range, two main checks occur:

   a. **Uncovered Range Check**: If the start of the current range (`l`) 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
   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 skips over any overlap and ensures that we extend our covered area to include the current range.

   ~~~
   ```python
   last = max(last, r)
   ```
   ~~~

4. **Check for Tail Uncovered Range**: After the loop, we check if there's an uncovered range from the end of the last covered range to the end of the array `nums`. If such an uncovered range exists, it is added to `ans`.

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

5. **Return Answer**: The `ans` list now contains all the maximal uncovered ranges, sorted by starting point, and can be returned as 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 to the furthest end reached by any range. The implementation is efficient and straightforward, not requiring any complex algorithms or data structures.

Example Walkthrough

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

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

Following the solution approach:

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

2. **Initialize Variables**: We initialize `last = -1` and `ans = []`.

3. **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, 0]]`. Update `last` to `3`.
   - 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 is not larger than `last`).
   - 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`.
4. **Check for Tail Uncovered Range**: After processing all ranges, we check for any uncovered range from the end of the last 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, 9]`, which is invalid because the start is greater than the end. Thus, no range is added to `ans`.

5. **Return Answer**: The final answer is `ans = [[0, 0]]`, which is the list of all maximally uncovered ranges in `nums`.

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

Python Solution

```python
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:
                # 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
            last_covered = max(last_covered, right)

        # After iterating ranges, check if there's still an uncovered range up to n - 1
        if last_covered + 1 < n:
            # 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 Solution

```java
import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;

class Solution {

    /**
     * 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) {
                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) {
            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()][]);
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Finds maximal uncovered ranges given an upper bound 'n' and a list of ranges
    vector<vector<int>> findMaximalUncoveredRanges(int n, vector<vector<int>>& ranges) {
        // Sort the ranges based on their start points
        sort(ranges.begin(), ranges.end(), [](const vector<int>& firstRange, const vector<int>& secondRange) {
            return firstRange[0] < secondRange[0];
        });

        int lastCovered = -1;
        vector<vector<int>> uncoveredRanges;

        // Iterate through each range
        for (auto& range : ranges) {
            int start = range[0], end = range[1];

            // Check if there is an uncovered range before the start of the current range
            if (lastCovered + 1 < start) {
                // Add the uncovered range to the answer list
                uncoveredRanges.push_back({lastCovered + 1, start - 1});
            }
            // Update the last covered position with the furthest point covered so far
            lastCovered = max(lastCovered, end);
        }

        // After all ranges are processed, check if there's an uncovered range at the end
        if (lastCovered + 1 < n) {
            // Add the uncovered range up to 'n - 1' to the answer list
            uncoveredRanges.push_back({lastCovered + 1, n - 1});
        }

        return uncoveredRanges;
    }
};

```

Typescript Solution

```typescript
function findMaximalUncoveredRanges(n: number, ranges: number[][]): number[][] {
    // Sort the given ranges based on their starting points.
    ranges.sort((firstRange, secondRange) => {
        return firstRange[0] - secondRange[0];
    });

    let lastCovered: number = -1;
    let uncoveredRanges: number[][] = [];

    // Iterate through the sorted ranges.
    ranges.forEach((range) => {
        const [start, end] = range;

        // Check for an uncovered range before the start of the current range.
        if (lastCovered + 1 < start) {
            // 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);
    });

    // After processing all ranges, check for an uncovered range at the end.
    if (lastCovered + 1 < n) {
        // Add the final uncovered range up to 'n - 1' to the uncoveredRanges array.
        uncoveredRanges.push([lastCovered + 1, n - 1]);
    }

    // Return the array containing all the uncovered ranges.
    return uncoveredRanges;
}

```

Time and Space Complexity

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

1. **Sorting the ranges**: The sort operation for the ranges at the beginning of the function has a time complexity of `O(m log m)`, where `m` is the number of intervals in the ranges list.

2. **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)`.

3. **Combining the above steps**: Since the sort operation dominates the overall time complexity, we can state that the overall 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 **Time Complexity** of the code is `O(m log m)`.
- The **Space Complexity** of the code is `O(m)`.