163. Missing Ranges

Easy Array

Problem Description

In this problem, you have two integers `lower` and `upper` representing the inclusive range of numbers. Also, you have an array `nums` which is sorted and contains unique integers that are within the given range. Some numbers within the range `[lower, upper]` might not be present in the `nums` array, and these numbers are considered "missing." The goal is to find the smallest sorted list of ranges, where each range covers the missing numbers without overlapping with any of the numbers in `nums`. The resulting ranges should together cover exactly all the missing numbers without any gaps or duplicates.

For example, if the `lower` is 0, `upper` is 99, and `nums` is `[7, 28, 63]`, then the missing numbers are from 0 to 6, 8 to 27, and 29 to 62, and 64 to 99. You need to return these missing ranges in a sorted manner which could look like `[[0, 6], [8, 27], [29, 62], [64, 99]]`.

Intuition

The solution involves a direct simulation of the problem by iterating through the elements in the array `nums` and identifying the gaps between the consecutive numbers as well as the gaps between the `lower`, `upper` limits, and the first and last number in `nums`.

Firstly, if the `nums` array is empty, the missing range is simply from `lower` to `upper`. However, if the array is not empty, we check for the following:

1. Is there a gap between `lower` and the first element of the `nums`? If so, this forms a range that needs to be added to the answer.
2. We then iterate over the given array, checking the difference between each pair of consecutive numbers. If the difference is more than 1, we've found a gap (missing numbers) and thus a range. These gaps are then made into ranges and added to the answer.
3. Lastly, we check for a potential gap between the last element of the `nums` and `upper`.

By handling both ends and the gaps in between, we're able to construct a list of ranges that covers all the missing numbers within the `[lower, upper]` range without including any number from `nums`.

Solution Approach

The implementation of the solution involves iterating over the elements in `nums` alongside handling the edge cases around lower and upper bounds. Here's the detailed walkthrough:

1. **Handling Empty Array**: If `nums` is empty, there are no numbers to compare against, hence the entire range from `lower` to `upper` is missing. We return this as a single range.

2. **Check Lower Bound**: If `nums` is not empty, check if there's a gap between `lower` and the first element of `nums`. If a gap exists, this forms our first missing range.

3. **Iterate Through `nums`**: Use Python's `pairwise` generator to loop through pairs of consecutive elements in `nums`. The `pairwise` utility yields tuples containing pairs of consecutive elements, which allows us to compare each pair without manual index management.

4. **Find Gaps Between Numbers**: For each consecutive pair `(a, b)`, if `b - a > 1`, we have found a gap which means there are missing numbers between `a` and `b`. Each of these gaps forms a range that goes from `a + 1` to `b - 1`; these are appended to our answer list.

5. **Check Upper Bound**: Finally, check if there's a gap between the last element of `nums` and `upper`. If there is, the gap from the last number plus one to `upper` is our final missing range.

The algorithm utilizes simple iteration and comparison with a focus on edge cases before and after the main loop. It takes advantage of Python's `pairwise` which is part of the `itertools` module (in Python versions 3.10 and later). The `pairwise` function makes these comparisons easier and the code more readable. However, if you are working with a Python version older than 3.10, you can manually compare elements using a loop with index `i` and `i+1`.

By applying these steps, we can ensure that all missing ranges are found and returned in the format of a list of lists, where each nested list contains two elements indicating the start and end of the missing range.

Example Walkthrough

Let's apply the solution approach to a simple example where `lower = 1`, `upper = 10`, and `nums = [2, 3, 7, 9]`. We expect to find the missing ranges that are not covered by numbers in `nums`.

1. **Handling Empty Array**: First, we check if `nums` is empty, which it is not. So, we continue with further steps.

2. **Check Lower Bound**: We look for gaps starting from the lower bound. The first element of `nums` is 2, and since `lower = 1`, we have a missing range `[1, 1]` which is essentially just the number 1.

3. **Iterate Through `nums`**: Now we iterate through the array and compare consecutive elements. We'll use `pairwise` on `nums` which would give us the pairs `(2, 3)`, `(3, 7)`, and `(7, 9)`.

4. **Find Gaps Between Numbers**:
   - For the pair `(2, 3)`, there is no gap since 3 - 2 is 1, indicating they are consecutive.
   - For `(3, 7)`, we have a gap because 7 - 3 is greater than 1. The missing range is `[4, 6]`.
   - For `(7, 9)`, there is no gap since 9 - 7 is 2, indicating only one number (8) between them, and it's not considered a range.
5. **Check Upper Bound**: Lastly, we check after the last element in `nums`, which is 9. The `upper` bound is 10, so we have a missing number `[10, 10]`, indicating just the number 10.

After applying the solution approach, we can conclude the missing ranges are `[[1, 1], [4, 6], [10, 10]]`. Each range encompasses the missing numbers without overlapping with any of the numbers in `nums`, satisfying the problem requirement.

Python Solution

```python
from typing import List

class Solution:
    def findMissingRanges(self, nums: List[int], lower: int, upper: int) -> List[List[int]]:
        # Initialize the size of the nums array
        num_elements = len(nums)

        # If nums is empty, return the entire range from lower to upper
        if num_elements == 0:
            return [[lower, upper]]

        # List to store the missing ranges
        missing_ranges = []

        # Check if there is a missing range before the start of the array
        if nums[0] > lower:
            missing_ranges.append([lower, nums[0] - 1])

        # Use zip to create pairs of sequential elements (a, b) and loop through
        for a, b in zip(nums, nums[1:]):
            # If there is a gap greater than one between the two numbers, a missing range is found
            if b - a > 1:
                missing_ranges.append([a + 1, b - 1])

        # Check if there is a missing range after the end of the array
        if nums[-1] < upper:
            missing_ranges.append([nums[-1] + 1, upper])

        # Return the list of missing ranges
        return missing_ranges
```

Java Solution

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

class Solution {

    /**
     * Finds the missing ranges between the given array elements and the specified lower and upper bounds.
     *
     * @param nums  The array of integers where missing ranges are to be found.
     * @param lower The lower bound of the range to find missing elements for.
     * @param upper The upper bound of the range to find missing elements for.
     * @return A list of lists containing the start and end of each missing range.
     */
    public List<List<Integer>> findMissingRanges(int[] nums, int lower, int upper) {
        // Initialize the length of the nums array.
        int n = nums.length;

        // Handle the edge case where the input array is empty.
        if (n == 0) {
            // The entire range from lower to upper is missing.
            return List.of(List.of(lower, upper));
        }

        // This list will store the missing ranges.
        List<List<Integer>> missingRanges = new ArrayList<>();

        // Check if there is a missing range before the first element.
        if (nums[0] > lower) {
            // Add the range from lower to the element before the first number in the array.
            missingRanges.add(List.of(lower, nums[0] - 1));
        }

        // Loop over the array to find missing ranges between the numbers.
        for (int i = 1; i < n; ++i) {
            // Check if the current element and the previous element are not consecutive.
            if (nums[i] - nums[i - 1] > 1) {
                // Add the range from the element after the previous number to the element before the current number.
                missingRanges.add(List.of(nums[i - 1] + 1, nums[i] - 1));
            }
        }

        // Check if there is a missing range after the last element.
        if (nums[n - 1] < upper) {
            // Add the range from the element after the last number in the array to the upper bound.
            missingRanges.add(List.of(nums[n - 1] + 1, upper));
        }

        // Return the list of missing ranges.
        return missingRanges;
    }
}
```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to find missing ranges between lower and upper bounds
    std::vector<std::vector<int>> findMissingRanges(std::vector<int>& nums, int lower, int upper) {
        // Get the size of the input vector
        int size = nums.size();

        // If the input vector is empty, return a vector with the complete range
        if (size == 0) {
            return {{lower, upper}};
        }

        // Initialize the answer vector
        std::vector<std::vector<int>> missingRanges;

        // Handle the case where the first element is greater than the lower bound
        if (nums[0] > lower) {
            missingRanges.push_back({lower, nums[0] - 1});
        }

        // Iterate over the input vector to find the missing ranges
        for (int i = 1; i < size; ++i) {
            // Check if there is a missing range between consecutive numbers
            if (nums[i] - nums[i - 1] > 1) {
                // Add the missing range to the answer vector
                missingRanges.push_back({nums[i - 1] + 1, nums[i] - 1});
            }
        }

        // Handle the case where the last element is less than the upper bound
        if (nums[size - 1] < upper) {
            missingRanges.push_back({nums[size - 1] + 1, upper});
        }

        // Return the final vector of missing ranges
        return missingRanges;
    }
};
```

Typescript Solution

```typescript
// Function to find missing ranges in a sorted array of numbers.
// It returns a list of missing ranges between the lower and upper bounds, inclusive.
function findMissingRanges(nums: number[], lower: number, upper: number): number[][] {
    const length = nums.length; // Get the length of the input array
    const missingRanges: number[][] = []; // Initialize an array to hold missing ranges

    // Handle the case where the input array is empty.
    // If so, the entire range between lower and upper is missing.
    if (length === 0) {
        return [[lower, upper]];
    }

    // Check if there is a missing range before the first element.
    if (nums[0] > lower) {
        missingRanges.push([lower, nums[0] - 1]);
    }

    // Iterate through the array to find missing ranges between consecutive elements.
    for (let i = 1; i < length; ++i) {
        // If there is a gap greater than 1 between two consecutive numbers,
        // then there is a missing range between them.
        if (nums[i] - nums[i - 1] > 1) {
            missingRanges.push([nums[i - 1] + 1, nums[i] - 1]);
        }
    }

    // Check if there is a missing range after the last element.
    if (nums[length - 1] < upper) {
        missingRanges.push([nums[length - 1] + 1, upper]);
    }

    // Return the list of missing ranges.
    return missingRanges;
}
```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the array `nums`. This is because the script iterates over the array once with a single loop through `pairwise(nums)` and two additional checks at the beginning and end.

The space complexity, disregarding the output list, is `O(1)`. This constant space complexity is due to only using a fixed amount of extra space (variables such as `n` and `ans`, which are negligible as their sizes do not scale with the input).