Problem 163: Missing Ranges

Problem Information

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

Problem Description

You are given an inclusive range

[lower, upper]

and a

sorted unique

integer array

nums

, where all elements are within the inclusive range.

A number

x

is considered

missing

if

x

is in the range

[lower, upper]

and

x

is not in

nums

.

Return

the

shortest sorted

list of ranges that

exactly covers all the missing numbers

. That is, no element of

nums

is included in any of the ranges, and each missing number is covered by one of the ranges.

Example 1:

Input:

nums = [0,1,3,50,75], lower = 0, upper = 99

Output:

[[2,2],[4,49],[51,74],[76,99]]

Explanation:

The ranges are: [2,2] [4,49] [51,74] [76,99]

Example 2:

Input:

nums = [-1], lower = -1, upper = -1

Output:

[]

Explanation:

There are no missing ranges since there are no missing numbers.

Constraints:

-10

9

<= lower <= upper <= 10

9

0 <= nums.length <= 100

lower <= nums[i] <= upper

All the values of

nums

are

unique

.

Code Snippets

C++:

```
class Solution {
public:
vector<vector<int>> findMissingRanges(vector<int>& nums, int lower, int
upper) {

}
};
```

Java:

```
class Solution {
public List<List<Integer>> findMissingRanges(int[] nums, int lower, int
upper) {

}
}
```

Python3:

```
class Solution:
def findMissingRanges(self, nums: List[int], lower: int, upper: int) ->
List[List[int]]:
```

Python:

```
class Solution(object):
def findMissingRanges(self, nums, lower, upper):
"""
:type nums: List[int]
:type lower: int
:type upper: int
:rtype: List[List[int]]
"""
```

JavaScript:

```
/**
* @param {number[]} nums
* @param {number} lower
* @param {number} upper
* @return {number[][]}
*/
var findMissingRanges = function(nums, lower, upper) {

};
```

TypeScript:

```
function findMissingRanges(nums: number[], lower: number, upper: number):
number[][] {

};
```

C#:

```
public class Solution {
public IList<IList<int>> FindMissingRanges(int[] nums, int lower, int upper)
{

}
}
```

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** findMissingRanges(int* nums, int numsSize, int lower, int upper, int*
returnSize, int** returnColumnSizes) {

}
```

Go:

```
func findMissingRanges(nums []int, lower int, upper int) [][]int {

}
```

Kotlin:

```
class Solution {
fun findMissingRanges(nums: IntArray, lower: Int, upper: Int):
List<List<Int>> {

}
}
```

Swift:

```
class Solution {
func findMissingRanges(_ nums: [Int], _ lower: Int, _ upper: Int) -> [[Int]]
{

}
}
```

Rust:

```
impl Solution {
pub fn find_missing_ranges(nums: Vec<i32>, lower: i32, upper: i32) ->
Vec<Vec<i32>> {

}
}
```

Ruby:

```
# @param {Integer[]} nums
# @param {Integer} lower
# @param {Integer} upper
# @return {Integer[][]}
def find_missing_ranges(nums, lower, upper)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums
* @param Integer $lower
* @param Integer $upper
* @return Integer[][]
*/
function findMissingRanges($nums, $lower, $upper) {

}
}
```

Dart:

```
class Solution {
List<List<int>> findMissingRanges(List<int> nums, int lower, int upper) {

}
}
```

Scala:

```
object Solution {
def findMissingRanges(nums: Array[Int], lower: Int, upper: Int):
List[List[Int]] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec find_missing_ranges(nums :: [integer], lower :: integer, upper ::
integer) :: [[integer]]
def find_missing_ranges(nums, lower, upper) do

end
end
```

Erlang:

```
-spec find_missing_ranges(Nums :: [integer()], Lower :: integer(), Upper ::
integer()) -> [[integer()]].
find_missing_ranges(Nums, Lower, Upper) ->
.
```

Racket:

```
(define/contract (find-missing-ranges nums lower upper)
(-> (listof exact-integer?) exact-integer? exact-integer? (listof (listof
exact-integer?)))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Missing Ranges
* 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<vector<int>> findMissingRanges(vector<int>& nums, int lower, int
upper) {

}
};
```

Java Solution:

```
/**
* Problem: Missing Ranges
* 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<List<Integer>> findMissingRanges(int[] nums, int lower, int
upper) {

}
}
```

Python3 Solution:

```
"""
Problem: Missing Ranges
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 findMissingRanges(self, nums: List[int], lower: int, upper: int) ->
List[List[int]]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def findMissingRanges(self, nums, lower, upper):
"""
:type nums: List[int]
:type lower: int
:type upper: int
:rtype: List[List[int]]
"""
```

JavaScript Solution:

```
/**
* Problem: Missing Ranges
* 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[]} nums
* @param {number} lower
* @param {number} upper
* @return {number[][]}
*/
var findMissingRanges = function(nums, lower, upper) {

};
```

TypeScript Solution:

```
/**
* Problem: Missing Ranges
* 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 findMissingRanges(nums: number[], lower: number, upper: number):
number[][] {

};
```

C# Solution:

```
/*
* Problem: Missing Ranges
```

```
* 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<IList<int>> FindMissingRanges(int[] nums, int lower, int upper)
{

}
}
```

C Solution:

```
/*
* Problem: Missing Ranges
* 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
*/

/**
* 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** findMissingRanges(int* nums, int numsSize, int lower, int upper, int*
returnSize, int** returnColumnSizes) {

}
```

Go Solution:

```
// Problem: Missing Ranges
// 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 findMissingRanges(nums []int, lower int, upper int) [][]int {

}
```

Kotlin Solution:

```
class Solution {
fun findMissingRanges(nums: IntArray, lower: Int, upper: Int):
List<List<Int>> {

}
}
```

Swift Solution:

```
class Solution {
func findMissingRanges(_ nums: [Int], _ lower: Int, _ upper: Int) -> [[Int]]
{

}
}
```

Rust Solution:

```
// Problem: Missing Ranges
// 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 find_missing_ranges(nums: Vec<i32>, lower: i32, upper: i32) ->
```

```
Vec<Vec<i32>> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums
# @param {Integer} lower
# @param {Integer} upper
# @return {Integer[][]}
def find_missing_ranges(nums, lower, upper)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums
* @param Integer $lower
* @param Integer $upper
* @return Integer[][]
*/
function findMissingRanges($nums, $lower, $upper) {

}
}
```

Dart Solution:

```
class Solution {
List<List<int>> findMissingRanges(List<int> nums, int lower, int upper) {

}
}
```

Scala Solution:

```
object Solution {
def findMissingRanges(nums: Array[Int], lower: Int, upper: Int):
```

```
List[List[Int]] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec find_missing_ranges(nums :: [integer], lower :: integer, upper ::
integer) :: [[integer]]
def find_missing_ranges(nums, lower, upper) do

end
end
```

Erlang Solution:

```
-spec find_missing_ranges(Nums :: [integer()], Lower :: integer(), Upper ::
integer()) -> [[integer()]].
find_missing_ranges(Nums, Lower, Upper) ->
.
```

Racket Solution:

```
(define/contract (find-missing-ranges nums lower upper)
(-> (listof exact-integer?) exact-integer? exact-integer? (listof (listof
exact-integer?)))
)
```