Problem 228: Summary Ranges

Problem Information

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

Problem Description

You are given a

sorted unique

integer array

nums

.

A

range

[a,b]

is the set of all integers from

a

to

b

(inclusive).

Return

the

smallest sorted

list of ranges that

cover all the numbers in the array exactly

. That is, each element of

nums

is covered by exactly one of the ranges, and there is no integer

x

such that

x

is in one of the ranges but not in

nums

.

Each range

[a,b]

in the list should be output as:

"a->b"

if

a != b

"a"

if

a == b

Example 1:

Input:

nums = [0,1,2,4,5,7]

Output:

["0->2","4->5","7"]

Explanation:

The ranges are: [0,2] --> "0->2" [4,5] --> "4->5" [7,7] --> "7"

Example 2:

Input:

nums = [0,2,3,4,6,8,9]

Output:

["0","2->4","6","8->9"]

Explanation:

The ranges are: [0,0] --> "0" [2,4] --> "2->4" [6,6] --> "6" [8,9] --> "8->9"

Constraints:

0 <= nums.length <= 20

-2

31

<= nums[i] <= 2

31

- 1

All the values of

nums

are

unique

.

nums

is sorted in ascending order.

Code Snippets

C++:

```
class Solution {
public:
vector<string> summaryRanges(vector<int>& nums) {

}
};
```

Java:

```
class Solution {
public List<String> summaryRanges(int[] nums) {
```

```
}
}
```

Python3:

```
class Solution:
def summaryRanges(self, nums: List[int]) -> List[str]:
```

Python:

```
class Solution(object):
def summaryRanges(self, nums):
"""
:type nums: List[int]
:rtype: List[str]
"""
```

JavaScript:

```
/**
* @param {number[]} nums
* @return {string[]}
*/
var summaryRanges = function(nums) {

};
```

TypeScript:

```
function summaryRanges(nums: number[]): string[] {

};
```

C#:

```
public class Solution {
public IList<string> SummaryRanges(int[] nums) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
char** summaryRanges(int* nums, int numsSize, int* returnSize) {

}
```

Go:

```
func summaryRanges(nums []int) []string {

}
```

Kotlin:

```
class Solution {
fun summaryRanges(nums: IntArray): List<String> {

}
}
```

Swift:

```
class Solution {
func summaryRanges(_ nums: [Int]) -> [String] {

}
}
```

Rust:

```
impl Solution {
pub fn summary_ranges(nums: Vec<i32>) -> Vec<String> {

}
}
```

Ruby:

```
# @param {Integer[]} nums
# @return {String[]}
```

```
def summary_ranges(nums)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums
* @return String[]
*/
function summaryRanges($nums) {

}
}
```

Dart:

```
class Solution {
List<String> summaryRanges(List<int> nums) {

}
}
```

Scala:

```
object Solution {
def summaryRanges(nums: Array[Int]): List[String] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec summary_ranges(nums :: [integer]) :: [String.t]
def summary_ranges(nums) do

end
end
```

Erlang:

```
-spec summary_ranges(Nums :: [integer()]) -> [unicode:unicode_binary()].
summary_ranges(Nums) ->
.
```

Racket:

```
(define/contract (summary-ranges nums)
(-> (listof exact-integer?) (listof string?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Summary 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<string> summaryRanges(vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Summary 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<String> summaryRanges(int[] nums) {

}
}
```

Python3 Solution:

```
"""
Problem: Summary 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 summaryRanges(self, nums: List[int]) -> List[str]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def summaryRanges(self, nums):
"""
:type nums: List[int]
:rtype: List[str]
"""
```

JavaScript Solution:

```
/**
* Problem: Summary 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
* @return {string[]}
*/
var summaryRanges = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Summary 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 summaryRanges(nums: number[]): string[] {

};
```

C# Solution:

```
/*
* Problem: Summary 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<string> SummaryRanges(int[] nums) {

}
}
```

C Solution:

```
/*
* Problem: Summary 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
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
char** summaryRanges(int* nums, int numsSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Summary 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 summaryRanges(nums []int) []string {

}
```

Kotlin Solution:

```
class Solution {
fun summaryRanges(nums: IntArray): List<String> {

}
}
```

Swift Solution:

```
class Solution {
func summaryRanges(_ nums: [Int]) -> [String] {

}
}
```

Rust Solution:

```
// Problem: Summary 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 summary_ranges(nums: Vec<i32>) -> Vec<String> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums
# @return {String[]}
def summary_ranges(nums)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums
* @return String[]
*/
function summaryRanges($nums) {

}
}
```

Dart Solution:

```
class Solution {
List<String> summaryRanges(List<int> nums) {

}
}
```

Scala Solution:

```
object Solution {
def summaryRanges(nums: Array[Int]): List[String] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec summary_ranges(nums :: [integer]) :: [String.t]
def summary_ranges(nums) do

end
end
```

Erlang Solution:

```
-spec summary_ranges(Nums :: [integer()]) -> [unicode:unicode_binary()].
summary_ranges(Nums) ->
.
```

Racket Solution:

```
(define/contract (summary-ranges nums)
(-> (listof exact-integer?) (listof string?))
)
```