228. Summary Ranges

Easy Array

Leetcode Link

Problem Description

You are provided with an array of integers, `nums`, which is sorted and contains no duplicates. Your task is to transform this array into a more compact representation using a list of ranges. A range `[a, b]` includes all integers that lie between `a` and `b`, inclusive. For example, the range `[2, 5]` represents the numbers `2, 3, 4, 5`.

You need to come up with the smallest list of such ranges that together cover exactly the same set of numbers as in the original array `nums`. For each range in that list:

- If the range contains a single number, say `a`, it should be represented by just that number as a string, e.g., `"a"`.
- If the range contains more than one number, `a` through `b`, it should be represented as the string `"a->b"`.

The output should be a list of these string representations, arranged in ascending order, showcasing the continuous and discrete ranges found in `nums`.

Intuition

To generate the list of ranges efficiently, we can employ a two-pointer approach, which is a common technique used in array traversals. Here's the thought process that leads to the solution:

- Since the array is sorted and contains unique integers, any consecutive elements that differ by 1 can be considered part of the same range.
- We can keep track of the start of the current range with pointer `i` and use another pointer `j` to explore the extent of the range.
- Starting with the left pointer `i` fixed at the beginning of the array, we move the right pointer `j` forward as long as consecutive elements are part of the same range (i.e., `nums[j + 1]` is exactly 1 greater than `nums[j]`).
- Once we find that `nums[j + 1]` is not a direct successor of `nums[j]`, we've identified a complete range. This range may be just a single number if `i` and `j` are at the same position, or it could be a sequence of numbers if `j` has moved from its starting position.
- For each range, we use a helper function `f(i, j)` to format the range's representation as a string, based on the start and end indices `i`, `j` of the range in the array.
- Once the range is added to the result list, we reset the left pointer `i` to be one position ahead of `j`, since `j + 1` is where our last range ended, and continue the search for the next range.
- We repeat this process until the end of the array is reached.

This method allows us to break down the problem into smaller parts, making the task of finding and representing ranges in the array both straightforward and efficient.

Solution Approach

To implement the solution, we apply the two-pointer technique as previously discussed. The code structure follows a clear pattern that helps us to identify and construct the ranges:

1. **Initialization:** We create an empty list `ans` to store the range strings, and initiate our pointers `i` and `j` to `0`. The pointer `i` serves as the start of a new range, and `j` is used to find the end of this range.
2. **Iterating through the array:** We proceed with a `while` loop that runs as long as `i` is less than the length of the array.
3. **Exploration within the range:** Inside the while loop:
    - We assign `j` to the same value as `i` to start exploring the range.
    - While `j + 1` is within the bounds of the array and the element at `j + 1` is one greater than the element at `j`, we increment `j`. This step essentially "gathers" all the consecutive numbers into one range.
4. **Range formation:** Once `j` stops moving, we reached the end of a range. We then call our helper function `f(i, j)` which takes the indices `i` and `j` and returns the string representation:
    - If `i == j`, this means the range consists of a single number, so we simply convert `nums[i]` to a string.
    - Otherwise, we format the string to be `"nums[i]->nums[j]"` to represent a range spanning multiple numbers.
5. **Storing the range and moving on:** After calling the helper function `f(i, j)`, we add the resulting string to our `ans` list, indicating that this range is complete.
6. **Advance the `i` pointer:** We then set `i` to `j + 1` to begin a new range and repeat the process.
7. **Return the result:** Once the entire array has been traversed, all ranges have been found, formatted, and added to `ans`. Finally, we return this list as our output.

This algorithm ensures that we only pass through the array once, therefore, the time complexity is `O(n)` where `n` is the length of the array, making the solution time-efficient. Space complexity is `O(1)` since we don't use any additional data structures that grow with the size of the input; the space used is just for the output list and the pointers.

The solution is a neat application of the two-pointer pattern, which is valuable for problems involving consecutive elements or pairs within a single array.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider the following sorted integer array with no duplicates:

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

We want to convert this array into a compact list of ranges and represent each range as a string. Here is how the provided method works:

1. **Initialization:**
    - The output list `ans` is initialized to `[]`.
    - Set pointer `i = 0` and pointer `j = 0`.
2. **Iterating through the array (`i < len(nums)`):**
    - `i` starts at `0`. The first number is `nums[i] = 0`.
3. **Exploration within the range:**
    - We set `j` to the same value as `i`, so `j = 0`.
    - The condition `j + 1 < len(nums)` and `nums[j + 1] == nums[j] + 1` allows `j` to move forward. Since `nums[j+1] = 1` is one greater than `nums[j] = 0`, we increment `j` to `1`.
    - Continue moving `j` to `2` since `nums[j+1] = 2` is one greater than `nums[j] = 1`.
    - `j` cannot move past `2` because `nums[j+1] = 4` is not one greater than `nums[j] = 2`.
4. **Range formation:**
    - With `j` stopped at `2`, we call the helper function `f(i, j)` to get the range string.
    - Since `i = 0` and `j = 2`, the range is `0, 1, 2`, so `ans` receives the string `"0->2"`.
5. **Storing the range and moving on:**
    - The string `"0->2"` is added to the `ans` list.
    - `i` is then set to `j + 1`, which is `3`.

(Repeat steps 2-4 for the next set of ranges)

6. **The same process repeats with the new value of `i`:**
    - Set `i = j + 1 = 3` and so `nums[i] = 4`. Since the next number `5` is a direct successor, `j` is incremented to `4`.
    - The range `nums[i]` to `nums[j]` is `4->5`, so `"4->5"` is added to `ans`.
    - Increment `i` to `j + 1 = 5` where `nums[i] = 7`. Since `7` is not followed by an immediate successor, this is a single-value range.
7. **Final range:**
    - The final number `7` forms a range by itself.
    - `ans` receives the string `"7"`.
8. **Return the result:**
    - The process concludes since `j` has reached the end of the array.
    - The final output list is `ans = ["0->2", "4->5", "7"]`.

By this approach, the original array `[0, 1, 2, 4, 5, 7]` is succinctly represented as `["0->2", "4->5", "7"]`, where each string in the list represents a compact range from the array.

Python Solution

```python
from typing import List

class Solution:
    def summaryRanges(self, nums: List[int]) -> List[str]:
        # Helper function to format the range as a string.
        def format_range(start: int, end: int) -> str:
            # If the range has only one element, return that element.
            # Otherwise, return the range in "start->end" format.
            return str(nums[start]) if start == end else f'{nums[start]}->{nums[end]}'

        # Initialize the start index of each potential range.
        start_index = 0
        # Get the length of the input list.
        n = len(nums)
        # Initialize the list that will hold the resulting ranges.
        ranges = []

        # Iterate through the list of numbers.
        while start_index < n:
            # Set the current end of range to the current start.
            end_index = start_index

            # Keep incrementing the end index as long as the next number is consecutive.
            while end_index + 1 < n and nums[end_index + 1] == nums[end_index] + 1:
                end_index += 1

            # Append the current range (start to end) to the result list.
            ranges.append(format_range(start_index, end_index))

            # Move the start index to the next potential start after the current end.
            start_index = end_index + 1

        # Return the list containing all the ranges in string format.
        return ranges
```

Java Solution

```java
class Solution {
    // Method to generate a summary of ranges from the array.
    public List<String> summaryRanges(int[] nums) {
        // Initialize a list to store the resulting summary of ranges.
        List<String> result = new ArrayList<>();

        // Iterate through the elements of the array to find consecutive ranges.
        for (int startIndex = 0, endIndex, n = nums.length; startIndex < n; startIndex = endIndex + 1) {
            // Initialize the end index of the range to the current start index.
            endIndex = startIndex;

            // Expand the range while the next number is consecutive.
            while (endIndex + 1 < n && nums[endIndex + 1] == nums[endIndex] + 1) {
                endIndex++;
            }

            // Add the current range to the result list.
            result.add(createRangeString(nums, startIndex, endIndex));
        }
        // Return the list containing all the summary ranges.
        return result;
    }

    // Helper method to format the range into a string.
    private String createRangeString(int[] nums, int start, int end) {
        // If the start index and end index are the same, return just one number.
        // Otherwise, return the formatted string representing the range.
        return start == end ? Integer.toString(nums[start]) : String.format("%d->%d", nums[start], nums[end]);
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    vector<string> summaryRanges(vector<int>& nums) {
        vector<string> ranges; // This vector will contain the final list of ranges in string format

        // A lambda function that formats a range string based on the start and end indices
        auto formatRange = [&](int start, int end) {
            return start == end ? to_string(nums[start]) : to_string(nums[start]) + "->" + to_string(nums[end]);
        };

        int n = nums.size(); // The total number of elements in the input vector

        // Iterate through all the numbers in the vector
        for (int startIdx = 0, endIdx; startIdx < n; startIdx = endIdx + 1) {
            endIdx = startIdx; // Initialize endIdx to be the same as startIdx
            // Continue to increment endIdx as long as consecutive numbers are sequential
            while (endIdx + 1 < n && nums[endIdx + 1] == nums[endIdx] + 1) {
                ++endIdx;
            }
            // Append the formatted range to the ranges result vector
            ranges.emplace_back(formatRange(startIdx, endIdx));
        }
        return ranges; // Return the accumulated list of formatted range strings
    }
};
```

Typescript Solution

```typescript
function summaryRanges(nums: number[]): string[] {
    // Helper function to format the range as a string based on the start and end indices.
    const formatRange = (start: number, end: number): string => {
        return start === end ? `${nums[start]}` : `${nums[start]}->${nums[end]}`;
    };

    // Variable `n` is the length of the `nums` array.
    const n = nums.length;

    // `ranges` will store the ranges in string format.
    const ranges: string[] = [];

    // Iterate over the array to find consecutive ranges.
    for (let i = 0, j = 0; i < n; i = j + 1) {
        // Initialize `j` to the current index `i`.
        j = i;

        // Increment `j` until the end of the array or the consecutive sequence breaks.
        while (j + 1 < n && nums[j + 1] === nums[j] + 1) {
            ++j;
        }

        // Add the current range to the `ranges` array using the `formatRange` helper.
        ranges.push(formatRange(i, j));
    }

    // Return the array of range summaries.
    return ranges;
}
```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the number of elements in the input list `nums`. This is because the code iterates through the list only once, with a while loop that progresses from one range-starting element to the next. Within this loop, there is another while loop that finds the end of the current range, effectively advancing the outer loop's index `i`. Even though there is this nested loop, each element is considered only once for the range formation, which guarantees linear time complexity.

The space complexity of the given code is also `O(n)`. In the worst case scenario, when there are no consecutive sequences, the `ans` list would contain the same number of elements as `nums`, with each element converted into a string. There is no additional space used that scales with the input size, except for the `ans` list which stores the resulting ranges.