900. RLE Iterator

Medium Design Array Counting Iterator

Leetcode Link

Problem Description

The problem presents a scenario where we are given a sequence of integers that has been encoded using the run-length encoding (RLE) technique. RLE is a simple form of data compression where sequences of the same data value are stored as a single data value and count. The encoded array, `encoding`, is an even-length array where for every even index `i`, `encoding[i]` represents the count of the following integer `encoding[i + 1]`.

The objective is to implement an iterator for this RLE encoded sequence. Two operations need to be defined for this iterator:

- `RLEIterator(int[] encoded)`- Constructor which initializes the RLEIterator with the `encoded` sequence.
- `int next(int n)`- This method should simulate the iteration over `n` elements of the encoded sequence and return the value of the last element after exhausting `n` elements. If less than `n` elements are left, the iterator should return `-1`.

An example of how RLE works: if we have a sequence `arr = [8, 8, 8, 5, 5]`, its RLE encoded form could be `encoding = [3, 8, 2, 5]`, where `[3, 8]` means that 8 appears 3 times, and `[2, 5]` means that 5 appears 2 times.

Intuition

To tackle this problem, we need to simulate the decoding process of RLE on-the-fly, without actually generating the entire decoded sequence due to potentially high space requirements.

To design the `RLEIterator` class efficiently, we keep track of our current position in the encoded sequence with an index `i`, and also track the number of elements we have already 'seen' at the current index with `curr`. During the `next(int n)` operation, we need to exhaust `n` elements. There are two cases to consider:

1. If the current count at `encoding[i]` is not enough to cover `n` (i.e., `curr + n > encoding[i]`), we know that we need to move to the next count-value pair by incrementing `i` by 2 and adjust `n` accordingly, taking into account the number of elements we have already exhausted with `curr`.

2. If the current count can cover `n`, we simply add `n` to `curr` and return the value at `encoding[i + 1]`, since `n` elements can be exhausted within the current count-value pair.

We repeat this process until we've either exhausted `n` elements and returned the last element exhausted, or we reach the end of the `encoding` array, where we return `-1` to indicate there are no more elements to iterate through.

Solution Approach

The solution makes use of a couple of important concepts: an index pointer and a count variable that together act as an iterator over the run-length encoded data. No additional data structure is required other than what's given by the `encoding`.

Here's a step-by-step breakdown of the `RLEIterator` implementation:

1. The constructor `__init__` simply initializes the `encoding` with the provided array. It also initializes two important variables: `self.i`, which represents the current index position in the `encoding` array (initially set to 0), and `self.curr`, which represents how many elements have been used up in the current run (initially set to 0).

2. The `next` function is designed to handle the iteration through the encoded sequence:

   - We initiate a `while` loop that continues as long as `self.i` is within the bounds of the `encoding` array.
   - Inside the loop, we handle two scenarios regarding the provided `n` elements that we want to exhaust:
     - If the current run (`self.encoding[self.i]`) minus the number of elements already used (`self.curr`) is less than `n`, it means we need to move to the next run. We update `n` by subtracting the remaining elements of the current run and reset `self.curr` to 0, since we will move to the next run, and increment `self.i` by 2 to jump to the next run-length pair.
     - If the current run is enough to cover `n`, we update `self.curr` to include the exhausted elements `n` and return the value `self.encoding[self.i + 1]`, which is the actual data value after using up `n` elements.
   - If we exit the loop, it means that all elements have been exhausted, and we return `-1`.

By incrementing only when necessary and by keeping track of how many elements we've 'seen' in the current run, we efficiently simulate the RLE sequence iteration.

No complex algorithms or data structures are needed, just careful indexing and counting, which keeps the space complexity to O(1) (aside from the input array) and the time complexity to O(n) in the worst case, where n is the total number of calls to `next`.

Example Walkthrough

Let's consider an encoded sequence `encoding = [5, 3, 4, 2]`. This means the number 3 appears 5 times followed by the number 2 appearing 4 times. If we translate that into its original sequence, it would look like `[3, 3, 3, 3, 3, 2, 2, 2, 2]`. We want to iterate over this sequence without actually decoding it.

Here is a step-by-step example illustrating the `RLEIterator` class functionality:

1. We first initialize our iterator with the `encoding` array by calling the constructor: `RLEIterator([5, 3, 4, 2])`.

   - Our index `i` is set to 0, meaning we are at the start of our encoded array.
   - Our current run count `curr` is set to 0, meaning we have not used up any elements from the first run.
2. We call the `next` function with `n = 2`: `iterator.next(2)`.

   - We enter the while loop since `i < len(encoding)`.
   - We check if the current run can accommodate `n`. Since `encoding[0] - curr` (5 - 0) is greater than 2, this run can accommodate it.
   - We update `curr` by adding `n`, now `curr` becomes 2.
   - We return the value 3 because it's the value associated with the current run.
3. Now, let's consider `iterator.next(5)`.

   - We check if the current run can accommodate `n` (`5` in this case). The current `curr` is `2`, so the remaining count in the current run is `3`. Since `3` isn't enough to cover `n=5`, we exhaust this run and update `n` to `n - (encoding[0] - curr)` which is `5 - 3 = 2`. Now we move to the next run by incrementing `i` by `2`, so `i` is now `2`, and reset `curr` to `0`.
   - In the next iteration, we check if the next run can cover the remaining `n=2`. Since `encoding[2]` which is `4` is greater than `2`, we can proceed.
   - We increment `curr` to `curr + n` which makes `curr = 2`, and we return `encoding[i + 1]` which is `2`.
4. If we keep calling `next`, eventually we would reach the end of the array. If `i` is no longer less than the length of `encoding`, it means we cannot return any more elements. In this case, `iterator.next()` would return `-1`.

By only moving to the next encoding pair when the current run is exhausted, and tracking the elements consumed in the `curr` variable, this implementation effectively iterates over the RLE sequence using a constant amount of extra space.

Python Solution

```python
from typing import List

class RLEIterator:
    def __init__(self, encoding: List[int]):
        self.encoding = encoding  # The run-length encoded array
        self.index = 0            # Current index in the encoding array
        self.offset = 0           # Offset to keep track of the current element count within the block

    def next(self, n: int) -> int:
        # Keep iterating until we find the n-th element or reach the end of the encoding
        while self.index < len(self.encoding):
            # If seeking past the current block
            if self.offset + n > self.encoding[self.index]:
                # Subtract the remaining elements of the current block from n
                n -= self.encoding[self.index] - self.offset
                # Reset the offset, and move to the next block (skip the value part of the block)
                self.offset = 0
                self.index += 2
            else:
                # The element is in the current block, so we update the offset
                self.offset += n
                # Return the value part of the current block
                return self.encoding[self.index + 1]
        # If we reached here, n is larger than the remaining elements
        return -1


# Example of how one would instantiate and use the RLEIterator class:
# obj = RLEIterator(encoding)
# element = obj.next(n)
```

Java Solution

```java
// RLEIterator decodes a run-length encoded sequence and supports
// retrieving the next nth element.
class RLEIterator {

    private int[] encodedSequence; // This array holds the run-length encoded data.
    private int currentIndex;      // Points to the current index of the encoded sequence.
    private int currentCount;      // Keeps track of the count of the current element

    // Constructs the RLEIterator with the given encoded sequence.
    public RLEIterator(int[] encoding) {
        this.encodedSequence = encoding;
        this.currentCount = 0;
        this.currentIndex = 0;
    }

    // Returns the element at the nth position in the decoded sequence or -1 if not present.
    public int next(int n) {
        // Iterates through the encodedSequence array.
        while (currentIndex < encodedSequence.length) {
            // If the current remainder of the sequence + n exceeds the current sequence value
            if (currentCount + n > encodedSequence[currentIndex]) {
                // Subtract the remainder of the current sequence from n
                n -= encodedSequence[currentIndex] - currentCount;
                // Move to the next sequence pair
                currentIndex += 2;
                // Reset currentCount for the new sequence
                currentCount = 0;
            } else {
                // If n is within the current sequence count, add n to currentCount
                currentCount += n;
                // Return the corresponding element
                return encodedSequence[currentIndex + 1];
            }
        }
        // If no element could be returned, return -1 indicating the end of the sequence.
        return -1;
    }
}

// Usage:
// RLEIterator iterator = new RLEIterator(new int[] {3, 8, 0, 9, 2, 5});
// int element = iterator.next(2); // Should return the 2nd element in the decoded sequence.
```

C++ Solution

```cpp
#include <vector>

// The RLEIterator class is used for Run Length Encoding (RLE) iteration.
class RLEIterator {
public:
    // Store the encoded sequence
    std::vector<int> encodedSequence;
    // The current position in the encoded sequence
    int currentCount;
    // The index of the current sequence in the encoded vector
    int currentIndex;

    // Constructor that initializes the RLEIterator with an encoded sequence
    RLEIterator(std::vector<int>& encoding) : encodedSequence(encoding), currentCount(0), currentIndex(0) {
    }

    // The next function returns the next element in the RLE sequence by advancing 'n' steps
    int next(int n) {
        // Keep iterating until we have processed all elements or until the end of the encoded sequence is reached
        while (currentIndex < encodedSequence.size()) {
            // If the steps 'n' exceed the number of occurrences of the current element
            if (currentCount + n > encodedSequence[currentIndex]) {
                // Deduct the remaining count of the current element from 'n'
                n -= encodedSequence[currentIndex] - currentCount;
                // Reset the current count as we move to the next element
                currentCount = 0;
                // Increment the index to move to the next element's occurrence count
                currentIndex += 2;
            } else {
                // If 'n' is within the current element's occurrence count
                currentCount += n;
                // Return the current element's value
                return encodedSequence[currentIndex + 1];
            }
        }
        // Return -1 if there are no more elements to iterate over
        return -1;
    }
};

/*
Exemplifying usage:
std::vector<int> encoding = {3, 8, 0, 9, 2, 5};
RLEIterator* iterator = new RLEIterator(encoding);
int element = iterator->next(2); // Outputs the current element after 2 steps
delete iterator; // Don't forget to deallocate the memory afterwards
*/
```

Typescript Solution

```typescript
// Store the encoded sequence
let encodedSequence: number[] = [];
// The current position in the encoded sequence
let currentCount: number = 0;
// The index of the current sequence in the encoded vector
let currentIndex: number = 0;

/**
 * Initializes the RLEIterator with an encoded sequence.
 * @param encoding - The initial RLE encoded sequence.
 */
function initRLEIterator(encoding: number[]): void {
    encodedSequence = encoding;
    currentCount = 0;
    currentIndex = 0;
}

/**
 * The next function returns the next element in the RLE sequence by advancing 'n' steps.
 * @param n - The number of steps to advance in the RLE sequence.
 * @returns The value at the 'n'-th position or -1 if the sequence has been exhausted.
 */
function next(n: number): number {
    // Continue iterating until all requested elements are processed or the end of the sequence is reached
    while (currentIndex < encodedSequence.length) {
        // If 'n' exceeds the occurrences of the current element
        if (currentCount + n > encodedSequence[currentIndex]) {
            // Subtract the remaining occurrences of the current element from 'n'
            n -= encodedSequence[currentIndex] - currentCount;
            // Reset the current count as we move to the next element
            currentCount = 0;
            // Move to the next element's occurrence count
            currentIndex += 2;
        } else {
            // 'n' is within the current element's occurrence count
            currentCount += n;
            // Return the current element's value
            return encodedSequence[currentIndex + 1];
        }
    }
    // Return -1 if there are no more elements
    return -1;
}

// Exemplifying usage:
initRLEIterator([3, 8, 0, 9, 2, 5]);
let element = next(2); // Outputs 8, since it's the current element after 2 steps
```

Time and Space Complexity

Time Complexity

The time complexity of the `next` method is O(K), where K is the number of calls to `next`, considering that at each call to the `next` method we process at most two elements from the encoding. In the worst case, we might traverse the entire encoding array once, processing two elements each time (the frequency and the value). The **init** method has a time complexity of O(1) since it only involves assigning the parameters to the instance variables without any iteration.

Space Complexity

The space complexity of the RLEIterator class is O(N), where N is the length of the `encoding` list. This is because we are storing the encoding in the instance variable `self.encoding`. No additional space is used that grows with the size of the input, as all other instance variables take up constant space.