1313. Decompress Run-Length Encoded List

Easy Array

Leetcode Link

Problem Description

The given problem presents a run-length encoded list, which is a common method for compressing data in which consecutive elements are replaced by just one value and the count of that value. The `nums` list contains pairs where the first element of each pair represents how many times the next element should appear in the decompressed list.

For instance, if we have a compressed list `[3, 4, 1, 2]`, it means we have 3 of `4`s and 1 of `2` in the decompressed list: `[4, 4, 4, 2]`.

The challenge is to convert this run-length encoded list into a decompressed list where each pair `[freq, val]` is expanded into `freq` instances of `val`.

Intuition

To solve this problem, we iterate over the given list `nums` to process each `[freq, val]` pair. Because `freq` and `val` are always adjacent, starting with `freq`, we skip every other element by using a loop that begins at index 1 and moves in steps of 2. This allows us to directly access each `val` while its corresponding `freq` is just the previous element.

For each `freq` and `val` pair, we replicate `val`, `freq` times. The replication can be efficiently accomplished in Python with list multiplication (`[val] * freq`). This creates a list of `val` repeated `freq` times. We then extend our result list with this new list.

The solution is achieved in a straightforward manner through this looping and extending, transforming the encoded list into the desired decompressed version in line with the problem requirements.

Solution Approach

The provided solution employs a simple yet efficient method that directly corresponds to the idea of run-length decoding. We use a basic `for` loop to iterate over the input list `nums`. The list data structure is the only structure used, which is apt for this task, as Python lists provide an efficient way to manage sequence data and support operations required for the solution.

Here's a step-by-step breakdown of the solution:

1. We initialize an empty list `res` which will hold our decompressed elements.
2. The `for` loop starts at index 1 and iterates till the end of the `nums` list with a step of 2. This step is important because our `freq` is always the element just before `val` (i.e., at index `i - 1`), and `val` is at every second index starting from 1 (which is `i` in this context).
3. In each iteration, we create a list with `val` repeated `freq` times. This is done by multiplying a single-element list containing `val` by `freq` (`[nums[i]] * nums[i - 1]`).
4. We then extend the `res` list with this repeated list. The `.extend()` method is used because it adds elements of the list argument to the end of the current list, effectively concatenating them to `res`.
5. Once the loop has processed all the `[freq, val]` pairs, the `res` list contains the fully decompressed list, which is then returned.

Through the use of list indexing, list multiplication, and the `.extend()` method, the provided Python solution efficiently decompresses the run-length encoded list with minimal complexity and optimal use of Python's list capabilities.

Here is the code snippet embedded in markdown for clarity:

```python
class Solution:
    def decompressRLElist(self, nums: List[int]) -> List[int]:
        res = []
        for i in range(1, len(nums), 2):
            res.extend([nums[i]] * nums[i - 1])
        return res
```

This approach is linear in time complexity (O(n)), where n is the number of elements in the input list `nums`, as it requires a single pass through the list. The space complexity is also linear (O(n)), as it directly depends on the total number of elements that will be in the decompressed list.

Example Walkthrough

Let's take a small example to illustrate the solution approach. Suppose we are given the following compressed list:

```
[2, 5, 3, 6]
```

This implies we should have 2 of '5's and 3 of '6's in our decompressed list.

Following the steps of the solution:

1. Initialize an empty list `res`. This will store the final decompressed list.
2. Begin a loop starting at index 1 (`i = 1`), since it's the position of the first value to be repeated. We will increment the index by 2 every iteration to jump to the next value to be repeated.
3. At index `i = 1`, the element is `5`, and the frequency (`freq`) is the previous element (`nums[i - 1]`), which is `2`. We create a list with `5` repeated `2` times: `[5, 5]`.
4. Extend the `res` list with this new list: `res` now becomes `[5, 5]`.
5. Move to the next value in the list that needs to be repeated, which is at index `i = 3` (the element is `6`). The frequency for `6` is `3` (`nums[i - 1]`).
6. Repeat the value `6`, `3` times to get `[6, 6, 6]`.
7. Extend the `res` list with this new list: `res` becomes `[5, 5, 6, 6, 6]`.

After the loop has finished, the `res` list contains the decompressed list that we need, and we return it:

```
[5, 5, 6, 6, 6]
```

This approach directly transforms the encoded list `[2, 5, 3, 6]` into the desired decompressed version `[5, 5, 6, 6, 6]`.

The given Python code, when executed with our example `nums`, would return `[5, 5, 6, 6, 6]` as expected:

```python
class Solution:
    def decompressRLElist(self, nums: List[int]) -> List[int]:
        res = []
        for i in range(1, len(nums), 2):
            res.extend([nums[i]] * nums[i - 1])
        return res
```

The provided example demonstrates how the solution method applies to a specific instance, reflecting the simplicity and effectiveness of the approach for decompressing a run-length encoded list.

Python Solution

```python
from typing import List

class Solution:
    def decompressRLElist(self, nums: List[int]) -> List[int]:
        # Initialize the resulting decompressed list
        decompressed_list = []

        # Iterate over the list 'nums' step by 2, starting from the second item
        for i in range(1, len(nums), 2):
            # For each pair, extend the decompressed list with 'freq' times of 'val'
            # where 'freq' is the frequency (previous element) and 'val' is the value (current element)
            freq = nums[i - 1]  # Frequency of the current element
            val = nums[i]       # The value to be repeated
            decompressed_list.extend([val] * freq)

        # Return the decompressed list after processing all pairs
        return decompressed_list

```

Java Solution

```java
class Solution {
    // The decompressRLElist method takes in an array of integers as input where consecutive pair elements
    // represent (frequency, value) and decompresses the list based on these.
    public int[] decompressRLElist(int[] nums) {
        // Initialize variable to track the size of the decompressed list.
        int decompressedLength = 0;

        // Calculate the length of the decompressed list.
        for (int i = 0; i < nums.length; i += 2) {
            decompressedLength += nums[i];
        }

        // Initialize the result array with the calculated length.
        int[] decompressedArray = new int[decompressedLength];

        // Index for inserting elements into the decompressedArray.
        int insertPosition = 0;

        // Loop through the nums array in steps of 2 to decompress.
        for (int i = 1; i < nums.length; i += 2) {
            // Decompress the current pair.
            // nums[i - 1] is the frequency and nums[i] is the value to be repeated.
            for (int j = 0; j < nums[i - 1]; ++j) {
                // Insert the value into decompressedArray and increment the insert position.
                decompressedArray[insertPosition++] = nums[i];
            }
        }

        // Return the decompressed array.
        return decompressedArray;
    }
}

```

C++ Solution

```cpp
#include <vector> // Include the vector header for std::vector

// Class name 'Solution' with a single public member function.
class Solution {
public:
    // Function to decompress a run-length encoded list.
    // Takes a reference to a vector of integers as an argument,
    // and returns a vector of integers.
    vector<int> decompressRLElist(vector<int>& nums) {
        vector<int> decompressedList; // Create a vector to store decompressed elements

        // Iterate over the input vector, stepping by 2, since the list is in (freq, val) pairs
        for (int i = 1; i < nums.size(); i += 2) {
            // For each pair, replicate the value 'nums[i]' according to the frequency 'nums[i - 1]'
            for (int freq = 0; freq < nums[i - 1]; ++freq) {
                decompressedList.push_back(nums[i]); // Add the value to the decompressed list
            }
        }

        return decompressedList; // Finally, return the decompressed vector
    }
};

```

Typescript Solution

```typescript
function decompressRLElist(nums: number[]): number[] {
    // 'halfLength' calculates half the length of the input array.
    // It is used because each pair (freq, val) takes two places in the array.
    let halfLength = nums.length >> 1;
    let decompressedList = [];

    // Loop through the input array in steps of two to process the pairs
    for (let i = 0; i < halfLength; i++) {
        // Get the frequency and value out of the 'nums' array
        let frequency = nums[2 * i],
            value = nums[2 * i + 1];

        // Create an array with 'frequency' number of 'value' elements and concatenate it to 'decompressedList'
        decompressedList.push(...new Array(frequency).fill(value));
    }

    // Return the decompressed result as an array
    return decompressedList;
}

```

Time and Space Complexity

Time Complexity

The time complexity is `O(n)`, where `n` is the total number of elements in the input list `nums`. This is because the loop iterates through every other element of `nums`, performing a constant amount of work for each pair of freq-value (since the `.extend()` method is called once per pair), and the number of pairs is `n/2`.

Space Complexity

The space complexity is also `O(n)`, but here `n` represents the total number of elements in the decompressed list `res`. Although the input list size is halved due to the pairs, the decompression may lead to a larger output. In the worst case, all the freq-values are large, causing the output list size to be significantly larger than the size of the input list.