1868. Product of Two Run-Length Encoded Arrays

Medium Array Two Pointers Leetcode Link

Problem Description

The problem provides us with a compression technique called "run-length encoding". This technique compresses an array of integers by representing segments of consecutive repeated numbers as a 2D array. Each entry in the encoded array has two elements: the value being repeated and the frequency of that value.

For example, `nums = [1,1,1,2,2,2,2,2]` is encoded as `encoded = [[1,3],[2,5]]`, meaning that the number 1 is repeated 3 times followed by number 2 which is repeated 5 times.

The task is to calculate the product of two run-length encoded arrays, `encoded1` and `encoded2`. To find the product, we need to:

1. Expand `encoded1` and `encoded2` back into the original full arrays (`nums1` and `nums2`).
2. Multiply the corresponding elements of `nums1` and `nums2` to form a new array `prodNums`.
3. Compress `prodNums` back into run-length encoded format.

We need to ensure that the final encoded product array is as short as possible.

Intuition

The intuition behind the solution is to simulate the product of the two arrays without fully expanding them. This is important for efficiency, especially when the encoded arrays represent very long sequences.

By iterating through the segments of the first array, we:

- Keep track of how many times we have used the value in the current segment.
- Determine the product with the corresponding part of the second encoded array.
- Update the result by either adding a new segment or extending the last segment if the product value is the same.
- Keep track of the portion of the segment of the second array that has been consumed and move to the next segment once we have used it up.

We use the minimum frequency from the current segments of `encoded1` and `encoded2` to decide how to extend or create the new segments in the resulting array. This way, we simulate the expansion and multiplication steps by directly compressing the product. We avoid unnecessary computation and achieve the efficient calculation of the run-length encoded product.

Solution Approach

The implementation of the solution is straightforward in logic but requires careful handling of the indices and frequencies from the encoded arrays. Here's a step-by-step breakdown of how the solution works:

1. Initialize an empty list `ans`, which will store our result in run-length encoded format.
2. Initialize a variable `j` to keep track of our position in `encoded2` while we iterate through `encoded1`.
3. Loop through each segment `[val_i, freq_i]` of `encoded1`. For each segment, we do the following:
   - While `freq_i` is greater than zero:
     - We take the minimum of `freq_i` and the frequency of the current segment in `encoded2` (i.e., `encoded2[j][1]`). This decides how much of the segment we can use in this step of the product.
     - Calculate the product `v` of `val_i` and the value of the current segment in `encoded2` (i.e., `encoded2[j][0]`).
     - Check if the last segment in `ans` has the same value as `v`. If so, increase the frequency of that segment by the frequency we are currently using `f`. This step ensures we are compressing the result as we go.
     - If the last segment in `ans` has a different value, append a new segment `[v, f]` to `ans`.
     - Decrease `freq_i` by `f` to keep track of the remaining frequency that needs to be accounted for from the current segment in `encoded1`.
     - Decrease the frequency of the current segment in `encoded2` by `f` as we have used up `f` frequency of it.
     - If the current segment in `encoded2` is fully used (frequency becomes zero), move to the next segment in `encoded2` by incrementing `j`.

By following these steps, the `while` loop effectively takes care of creating the product without ever needing to fully expand the encoded arrays. Once we've finished processing all segments in `encoded1`, we're left with `ans` which contains our compressed product.

This approach uses a single pass and keeps space complexity low, as we never create the fully expanded arrays. It combines elements of two-pointers technique—`i` iterating over `encoded1` and `j` keeping track of our place in `encoded2`—with a merge-like operation where we merge contributions from corresponding segments.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have two run-length encoded arrays:

`encoded1 = [[2,3],[3,2]]` and `encoded2 = [[6,1],[3,3],[9,1]]`

The first array `encoded1` can be expanded to `[2,2,2,3,3]` and the second array `encoded2` to `[6,3,3,3,9]`. We are to find the product of these two arrays and then compress it back to run-length encoding format. Let's follow the steps in the solution approach.

1. Initialize `ans = []`, an empty list to hold the result.

2. Initialize `j = 0`, a variable to keep track of our position in `encoded2`.

3. Begin looping over `encoded1`. Our first segment is `[2,3]` indicating three 2s.

   - While `freq_i` (3 for the first segment of `encoded1`) is greater than zero:
     - The first segment of `encoded2` has only one element left, so we take the minimum of `freq_i` and `encoded2[j][1]`, which is `min(3, 1) = 1`.
     - The product `v` of values `2` (`val_i`) and `6` (`encoded2[j][0]`) is `12`.
     - `ans = []` is currently empty, so we append `[12, 1]` to `ans`.
     - We decrease `freq_i` by 1, now `freq_i` is 2.
     - We also decrease `encoded2[j][1]` by 1, and since it would now be 0, we move to the next segment by incrementing `j` (now `j=1` and segment `[3,3]` of `encoded2` is the current).
4. We continue the while loop because `freq_i` is still greater than zero:

   - Now we have `freq_i = 2` and the frequency of the current segment in `encoded2[j][1]` is 3.
   - We take the minimum, which is 2.
   - The next product `v` of values `2` and `3` is `6`.
   - `ans = [[12, 1]]`, and as 6 is different from 12, we append `[6, 2]` to `ans`.
   - Both `freq_i` and `encoded2[j][1]` are decreased by 2, resulting in `freq_i = 0` and `encoded2[j][1] = 1`.
5. Since `freq_i` is zero, we move to the next segment of `encoded1`.

6. The next segment is `[3,2]`. We repeat a similar process here:

   - We have `freq_i = 2` and `encoded2[j][1]` is 1 (from the remaining part of the previous segment `[3,1]`).
   - We take the minimum, which is 1.
   - The product of values 3 and 3 is 9.
   - Since the last value in `ans` is 6, we add a new segment `[9, 1]` to `ans`.
   - Decrease `freq_i` to 1 and since `encoded2[j][1]` is now 0, we increase `j` to 2 (next segment `[9,1]` of `encoded2`).
7. We continue in the loop with the remaining `freq_i = 1` for value 3 in `encoded1`:

   - We have `freq_i = 1` and `encoded2[j][1]` is also 1.
   - The product of values 3 and 9 is 27.
   - As 27 is different from the last value in `ans`, we add another segment `[27, 1]` to `ans`.
   - Both `freq_i` and `encoded2[j][1]` are decreased to 0.

Finally, we're finished processing all segments in `encoded1` and `encoded2`, resulting in `ans = [[12, 1], [6, 2], [9, 1], [27, 1]]` which is the run-length encoded product of `nums1` and `nums2`.

Python Solution

```python
from typing import List

class Solution:
    def findRLEArray(
        self, encoded1: List[List[int]], encoded2: List[List[int]]
    ) -> List[List[int]]:
        # Initialize an empty list to store the resulting encoded RLE array
        result = []

        # Initialize an index to track the position in encoded2
        index_encoded2 = 0

        # Iterate through each pair (value, frequency) in the first encoded array
        for value1, frequency1 in encoded1:

            # Process the current range until its frequency becomes zero
            while frequency1:
                # Determine the minimum frequency to merge from both arrays
                min_frequency = min(frequency1, encoded2[index_encoded2][1])

                # Compute the product of the values
                product_value = value1 * encoded2[index_encoded2][0]

                # If the last pair in the result has the same value, merge them by adding the frequencies
                if result and result[-1][0] == product_value:
                    result[-1][1] += min_frequency
                else:
                    # Otherwise, append a new pair of product value and frequency
                    result.append([product_value, min_frequency])

                # Update the frequencies in both arrays
                frequency1 -= min_frequency
                encoded2[index_encoded2][1] -= min_frequency

                # If the current range in encoded2 is exhausted, move to the next one
                if encoded2[index_encoded2][1] == 0:
                    index_encoded2 += 1

        # Once done processing, return the resulting array
        return result

```

Java Solution

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

class Solution {

    public List<List<Integer>> findRLEArray(int[][] encoded1, int[][] encoded2) {
        // Initialize the answer list to hold the product RLE
        List<List<Integer>> result = new ArrayList<>();
        // Index for tracking the current position in encoded2
        int currentIndex = 0;

        // Iterate over each pair in encoded1
        for (int[] pairEncoded1 : encoded1) {
            // Grab the value and frequency from the current pair in encoded1
            int value1 = pairEncoded1[0];
            int frequency1 = pairEncoded1[1];

            // Continue until we have processed all of this value
            while (frequency1 > 0) {
                // Find the frequency to be processed which is the minimum of the
                // remaining frequency from encoded1 and the current frequency from encoded2
                int minFrequency = Math.min(frequency1, encoded2[currentIndex][1]);
                // Multiply the values from encoded1 and encoded2
                int product = value1 * encoded2[currentIndex][0];

                // Check if the last element in the result list has the same value
                int resultSize = result.size();
                if (resultSize > 0 && result.get(resultSize - 1).get(0) == product) {
                    // If yes, add the minFrequency to the frequency of the last element
                    int currentFreq = result.get(resultSize - 1).get(1);
                    result.get(resultSize - 1).set(1, currentFreq + minFrequency);
                } else {
                    // If not, add a new pair with the product and minFrequency
                    result.add(new ArrayList<>(Arrays.asList(product, minFrequency)));
                }
                // Decrease the respective frequencies
                frequency1 -= minFrequency;
                encoded2[currentIndex][1] -= minFrequency;

                // If we have processed all frequencies of the current pair in encoded2,
                // move to the next pair
                if (encoded2[currentIndex][1] == 0) {
                    currentIndex++;
                }
            }
        }
        return result;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<vector<int>> findRLEArray(vector<vector<int>>& encoded1, vector<vector<int>>& encoded2) {
        vector<vector<int>> result; // This will store the final Run Length Encoded product of encoded1 and encoded2

        int indexInEncoded2 = 0; // Index for tracking position in encoded2

        // We iterate through each element-pair in encoded1
        for (const auto& pairInEncoded1 : encoded1) {
            int valueInEncoded1 = pairInEncoded1[0], // Current value from encoded1
                frequencyInEncoded1 = pairInEncoded1[1]; // Current frequency of this value in encoded1

            // While we have not accounted for all instances of this particular value
            while (frequencyInEncoded1 > 0) {

                // Compute minimum frequency to consider from both encoded1 and encoded2
                int frequencyToProcess = min(frequencyInEncoded1, encoded2[indexInEncoded2][1]);

                // Compute the product of the values from encoded1 and encoded2
                int productValue = valueInEncoded1 * encoded2[indexInEncoded2][0];

                // If this value was already the last value we processed, we just update its frequency
                if (!result.empty() && result.back()[0] == productValue) {
                    result.back()[1] += frequencyToProcess;
                } else {
                    // Otherwise, we add a new pair {product value, frequency} to our result
                    result.push_back({productValue, frequencyToProcess});
                }

                // Decrease the frequencies in encoded1 and encoded2 by the frequency we have just processed
                frequencyInEncoded1 -= frequencyToProcess;
                encoded2[indexInEncoded2][1] -= frequencyToProcess;

                // If we've used up all instances of the current value in encoded2, move to the next pair
                if (encoded2[indexInEncoded2][1] == 0) {
                    ++indexInEncoded2;
                }
            }
        }

        return result; // Return the final product after Run Length Encoding
    }
};

```

Typescript Solution

```typescript
// Define the type for the encoded arrays to increase code readability.
type EncodedArray = Array<[number, number]>;

// Given two run-length encoded arrays encoded1 and encoded2, this function will
// calculate the run-length encoded product of the two arrays.
function findRLEArray(encoded1: EncodedArray, encoded2: EncodedArray): EncodedArray {
    let result: EncodedArray = []; // This will store the final RLE product of encoded1 and encoded2.

    let indexInEncoded2 = 0; // Index for tracking position in encoded2.

    // Iterate through each value-frequency pair in encoded1.
    for (const [valueInEncoded1, originalFrequencyInEncoded1] of encoded1) {
        let frequencyInEncoded1 = originalFrequencyInEncoded1; // Current frequency of this value in encoded1.

        // While we have not processed all instances of this particular value
        while (frequencyInEncoded1 > 0) {

            // Compute minimum frequency to consider from both encoded1 and encoded2.
            let frequencyToProcess = Math.min(frequencyInEncoded1, encoded2[indexInEncoded2][1]);

            // Compute the product of the values from encoded1 and encoded2.
            let productValue = valueInEncoded1 * encoded2[indexInEncoded2][0];

            // If this value was already the last value we processed, we just update its frequency.
            if (result.length > 0 && result[result.length - 1][0] === productValue) {
                result[result.length - 1][1] += frequencyToProcess;
            } else {
                // Otherwise, we add a new pair [product value, frequency] to our result.
                result.push([productValue, frequencyToProcess]);
            }

            // Decrease the frequencies in encoded1 and encoded2 by the frequency we have just processed.
            frequencyInEncoded1 -= frequencyToProcess;
            encoded2[indexInEncoded2][1] -= frequencyToProcess;

            // If we've used up all instances of the current value in encoded2, move to the next pair.
            if (encoded2[indexInEncoded2][1] === 0) {
                indexInEncoded2++;
            }
        }
    }

    return result; // Return the final RLE product.
}

// You can then use the findRLEArray function with two run-length encoded arrays.
// For example: findRLEArray([[1,4], [2,3]], [[3,4], [5,2]]);

```

Time and Space Complexity

Time Complexity

The time complexity of the given code primarily depends on these factors:

1. Iterating through `encoded1`, which has `n` elements: `O(n)`.
2. A nested loop for processing elements in `encoded2`, which can be iterated up to `m` times in the worst case (in case the sizes in `encoded1` are much larger): `O(m)`.

However, since the second loop decreases the frequency from `encoded2` without resetting, each element of `encoded2` will be processed at most once throughout the entire iteration of `encoded1`. Therefore, the total time complexity will be `O(n + m)` where `n` is the total number of elements in `encoded1` and `m` is the total number of elements in `encoded2`.

Space Complexity

The space complexity is determined by the size of the output, which in the worst case might contain an individual element for each multiplication of pairs from `encoded1` and `encoded2`. In the worst case, every pair multiplication might result in a distinct value not matching the last element of the `ans` list, thus:

1. The `ans` list: Up to `O(n + m)` in the worst case.
2. Constant space for the pointers and temporary variables like `vi`, `fi`, `f`, `v`, `j`.

Therefore, the total space complexity of the given code would be `O(n + m)`.