2826. Sorting Three Groups

Medium Array Dynamic Programming Leetcode Link

Problem Description

The challenge presents us with a 0-indexed integer array `nums` of length `n`. In this scenario, there is an intriguing way numbers are grouped: each number from `0` to `n - 1` is assigned to one of three possible groups (1, 2, or 3) based on the value of `nums[i]`, meaning the index of the number indicates which number we're referring to, and the value at that index tells us the group to which it belongs. It's important to note that some groups might not have any numbers assigned to them—they can be empty.

The task is to transform `nums` into a "beautiful array". For an array to be considered beautiful, the numbers must be sorted in non-decreasing order across all groups when reconvened into a single array. This can be achieved through a series of operations where we may choose any number `x` (index) and change its group by setting `nums[x]` to 1, 2, or 3.

The processes used to make a "beautiful array" are as such:

- First, the numbers in each individual group are sorted.
- Then, the numbers from groups 1, 2, and 3 are appended to form a new array `res` in that specific group order.

The challenge posed to us is to determine the minimum number of such operations necessary to change the original `nums` array into a beautiful array.

Intuition

Creating a beautiful array is essentially similar to sorting, but with an unique constraint: we have to sort numbers into three buckets or groups before concatenating them. However, we can only move numbers between these groups, not arbitrarily reorder them within a group.

Given this constraint, we recognize the problem as one of dynamic programming (DP), because the optimal solution to make a portion of the array beautiful can help inform the next step. What we're effectively doing in the DP approach is keeping track of the cost of sorting each prefix of the array `nums` up to index `i`, while considering that each element `nums[i]` can belong to any of the three groups. The state, therefore, involves the minimum cost of operations (the number of group changes) required to achieve a sorted array up to that point. This cost depends on the last group in which we've placed `nums[i]`, as that determines where we can put `nums[i + 1]` without increasing the number of moves.

In the solution code provided, `f` represents an array that maintains this cost for each of the three groups. For each number in `nums`, we create a new array `g` that temporarily stores the updated costs after considering the current number's possible group placements. If the current number is 1, we must place it in group 1 without any operation, hence `g[0]` just carries over the existing cost in `f[0]`. For the number to be in group 2, it would require an operation if the last number was in group 1, but not if it was already in group 2, thereby determining the cost as `min(f[:2]) + 1`. Similar logic is applied for when the number is 2 and when it is 3. Each possible number placement is examined, and the costs are updated accordingly.

After considering all the numbers in `nums`, the minimum value in `f` reflects the smallest number of operations required to achieve a beautiful array.

Solution Approach

The provided Python solution implements a dynamic programming (DP) strategy to minimize the number of operations required for creating a beautiful array. Here's a step-by-step breakdown of the implementation:

1. **Initialization**: A list `f` is used to keep track of the minimum number of operations necessary to reach a state where the last group (either 1, 2, or 3) has been populated. Initially, `f` is set to `[0, 0, 0]`, assuming no operations are performed yet.

2. **Iterating Over the Array**: The solution iterates over each element `x` of `nums`. A temporary list `g` is created to store the new costs calculated based on the current number and prior costs.

3. **Evaluating Placement Options**: Depending on the value of `x` (which corresponds to the current group the number is in), there are different implications for the number of operations needed.

   - If `x` is 1, it means the current number is already in group 1, which is the correct placement (no extra operation needed for this group). So, we copy the previous cost for group 1 to `g[0]`.
   - To keep the array sorted (beautiful), the number can be placed in group 2 only if it follows a number that was in group 1 or already in group 2 (hence `g[1]` is the minimum of the previous costs for groups 1 and 2 plus one possible operation).
   - To place a number in group 3, it may follow any of the three previous group placements (so `g[2]` is assigned as the minimum of all previous costs incremented by one for the operation).
4. **Handling Different Values of `x`**: The process slightly varies when `x` is 2 or 3, as the cost of 'leaving the number where it is' applies only to the group `x` is currently in and needs an increment for the other groups.

5. **Updating State**: After considering all placement options for the current number, we update `f` to the new costs stored in `g`. This continues for the entire array.

6. **Extracting Result**: Once we have processed all numbers in `nums`, the minimum number in `f` gives us the minimum operations required to make the entire array beautiful. At this point, each element of `f` represents the cost of the operations performed up to the last number in `nums`, assuming the last number is in group 1, 2, or 3, respectively.

This solution emphasizes the importance of maintaining a historical context for costs (DP) and understanding that only the 'state transitions' (change of groups) incur a cost, not the internal ordering within the groups.

Example Walkthrough

Let's illustrate the solution approach with a small example where we have an integer array `nums` of length `n = 5`, and the content of `nums` is `[2, 1, 2, 3, 1]`. We need to determine the minimum number of operations to transform `nums` into a beautiful array.

1. **Initialization**: We start with an array `f = [0, 0, 0]` to keep track of the minimum operations needed for the three possible group placements of the last number.

2. **Iterating Over the Array**:

- For the first element, `x = nums[0] = 2`, we need `f` to reflect the cost after considering this element. The temporary array `g` is initialized.

3. **Evaluating Placement Options**:

- Since `x` is 2, we can consider placing it in group 2 with no cost (`g[1] = f[1]`). But if we want to place it in group 1 or 3, since it's not naturally there, it requires one operation (`g[0] = f[0] + 1` for group 1, and `g[2] = f[2] + 1` for group 3). So, `g` becomes `[1, 0, 1]`.

4. **Updating State**:

- We update `f` to be the same as `g`, so `f` now becomes `[1, 0, 1]`.

5. **Continuing with Array**:

- The next element, `x = nums[1] = 1`, naturally belongs to group 1, so `g[0] = f[0]`. For group 2 it could either stay as it is or follow an element in group 1, so `g[1] = min(f[0], f[1]) + 1`. And for group 3, it can follow any of the previous groups, so `g[2] = min(f) + 1`. The updated `g` becomes `[1, 1, 1]`.
- Repeat this process for each element in the array, updating the state `f` with `g` after each iteration:

Element = 2: `g` recalculated to `[2, 1, 1]` Element = 3: `g` recalculated to `[2, 2, 1]` Element = 1: `g` recalculated to `[2, 2, 2]`

6. **Extracting Result**:

- After processing all elements in `nums`, the final values of `f` is `[2, 2, 2]`. The minimum number of operations required to make `nums` a beautiful array is the minimum of these values, which is 2.

In conclusion, given the array `[2, 1, 2, 3, 1]`, we find that at least 2 operations are required to transform it into a beautiful array where each group is sorted, and then concatenated together in group order.

Python Solution

```python
from typing import List

class Solution:
    def minimumOperations(self, nums: List[int]) -> int:
        # f will hold the minimum operations required to adjust the list
        # to the constraints of having 1s, 2s and no elements in position i
        min_operations = [0] * 3

        # Loop through each element in the nums list
        for number in nums:
            # g will temporarily hold the new calculated minimum operations
            new_min_operations = [0] * 3

            # If the current number is 1
            if number == 1:
                new_min_operations[0] = min_operations[0]  # No change needed
                new_min_operations[1] = min(min_operations[:2]) + 1  # Increment operations for 1s or 2s
                new_min_operations[2] = min(min_operations) + 1  # Increment as 1 is not allowed here
            # If the current number is 2
            elif number == 2:
                new_min_operations[0] = min_operations[0] + 1  # Increment as 2 is not allowed here
                new_min_operations[1] = min(min_operations[:2])  # No change needed
                new_min_operations[2] = min(min_operations) + 1  # Increment operations for no number
            # If the current number is neither 1 nor 2
            else:
                new_min_operations[0] = min_operations[0] + 1  # Increment as number is not allowed here
                new_min_operations[1] = min(min_operations[:2]) + 1  # Increment operations for 1s or 2s
                new_min_operations[2] = min(min_operations)  # No change needed

            # Update min_operations with the new calculated minimum operations
            min_operations = new_min_operations

        # Return the minimum of the calculated operations
        return min(min_operations)

```

Java Solution

```java
class Solution {
    public int minimumOperations(List<Integer> nums) {
        // Initialize an array to keep track of the minimum operations
        int[] minOps = new int[3];

        // Loop through each number in the input list
        for (int num : nums) {
            // Create a temporary array to store the current state of minimum operations
            int[] currentOps = new int[3];

            // Check the current number and update the temporary state array accordingly
            if (num == 1) {
                // If the current number is 1, we don't change the first state
                currentOps[0] = minOps[0];
                // We can reach the second state from the first or second state with one operation
                currentOps[1] = Math.min(minOps[0], minOps[1]) + 1;
                // We can reach the third state from any state with one operation
                currentOps[2] = Math.min(Math.min(minOps[0], minOps[1]), minOps[2]) + 1;
            } else if (num == 2) {
                // If the current number is 2, we can reach the first state with one operation
                currentOps[0] = minOps[0] + 1;
                // We can reach the second state from the first or second state with no operation
                currentOps[1] = Math.min(minOps[0], minOps[1]);
                // We can reach the third state from any state with one operation
                currentOps[2] = Math.min(Math.min(minOps[0], minOps[1]), minOps[2]) + 1;
            } else {
                // If the current number is neither 1 nor 2, we can reach the first state with one operation
                currentOps[0] = minOps[0] + 1;
                // We can reach the second state from the first or second state with one operation
                currentOps[1] = Math.min(minOps[0], minOps[1]) + 1;
                // We can reach the third state from any state with no additional operation
                currentOps[2] = Math.min(Math.min(minOps[0], minOps[1]), minOps[2]);
            }

            // Update our tracking array to the current state
            minOps = currentOps;
        }

        // Calculate and return the minimum number of operations among all three states
        return Math.min(Math.min(minOps[0], minOps[1]), minOps[2]);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int minimumOperations(vector<int>& nums) {
        // The vector 'minimumOps' stores the minimum operations needed to
        // convert the sequence up to the current point, based on the last number in the sequence.
        // minimumOps[0]: no operations performed, still at first number
        // minimumOps[1]: one operation performed, at second number
        // minimumOps[2]: two operations performed, at third number
        vector<int> minimumOps(3, 0);

        for (int num : nums) {
            // 'nextOps' stores the number of operations needed for the current state.
            vector<int> nextOps(3, 0);

            if (num == 1) {
                // If current number is 1, keep the same state or increment operations.
                nextOps[0] = minimumOps[0]; // Previous number was also 1, no change in operations.
                nextOps[1] = min(minimumOps[0], minimumOps[1]) + 1; // Move from 1st number or stay at 2nd number, with additional
                nextOps[2] = min({minimumOps[0], minimumOps[1], minimumOps[2]}) + 1; // Move to 3rd number from any state.
            } else if (num == 2) {
                // If current number is 2, we need to perform one more operation if at first number, or we can stay at second numbe
                nextOps[0] = minimumOps[0] + 1; // Add one operation to move from 1st number.
                nextOps[1] = min(minimumOps[0], minimumOps[1]); // Keep the minimum operations from 1st or 2nd number.
                nextOps[2] = min(minimumOps[0], min(minimumOps[1], minimumOps[2])) + 1; // Move to 3rd number from any state with a
            } else {
                // If current number is neither 1 nor 2, just increment operations needed as one operation is needed to change this
                nextOps[0] = minimumOps[0] + 1; // Add one operation to move away from 1st number.
                nextOps[1] = min(minimumOps[0], minimumOps[1]) + 1; // Add operation to keep it second number or move from first to
                nextOps[2] = min(minimumOps[0], min(minimumOps[1], minimumOps[2])); // Stay at third number as it is the most diffe
            }

            // Update 'minimumOps' with the new calculated operations 'nextOps'.
            minimumOps = move(nextOps);
        }

        // Return the minimum of all the calculated operations to make the sequence of either 1s or 2s.
        return min({minimumOps[0], minimumOps[1], minimumOps[2]});
    }
};

```

Typescript Solution

```typescript
function minimumOperations(nums: number[]): number {
    // Initialize an array to store the frequencies of operations required to reach states 0, 1, 2.
    let operationsFrequencies: number[] = new Array(3).fill(0);

    // Iterate over each number in the input array.
    for (const num of nums) {
        // Initialize an array to store the new frequencies of operations after considering the current number.
        const newFrequencies: number[] = new Array(3).fill(0);

        // Check the value of the current number to decide the operations needed.
        if (num === 1) {
            // If the number is 1, the operations required to reach each state are updated.
            newFrequencies[0] = operationsFrequencies[0];
            newFrequencies[1] = Math.min(operationsFrequencies[0], operationsFrequencies[1]) + 1;
            newFrequencies[2] = Math.min(operationsFrequencies[0], Math.min(operationsFrequencies[1], operationsFrequencies[2])) +
        } else if (num === 2) {
            // If the number is 2, similar calculations are done for each state.
            newFrequencies[0] = operationsFrequencies[0] + 1;
            newFrequencies[1] = Math.min(operationsFrequencies[0], operationsFrequencies[1]);
            newFrequencies[2] = Math.min(operationsFrequencies[0], Math.min(operationsFrequencies[1], operationsFrequencies[2])) +
        } else {
            // For a number that's neither 1 nor 2, again the states are updated accordingly.
            newFrequencies[0] = operationsFrequencies[0] + 1;
            newFrequencies[1] = Math.min(operationsFrequencies[0], operationsFrequencies[1]) + 1;
            newFrequencies[2] = Math.min(operationsFrequencies[0], Math.min(operationsFrequencies[1], operationsFrequencies[2]));
        }
        // The current frequencies are updated to be the new frequencies calculated for this iteration.
        operationsFrequencies = newFrequencies;
    }
    // The function returns the minimum number of operations to be in any of the states for the entire array.
    return Math.min(...operationsFrequencies);
}

```

Time and Space Complexity

The code snippet provided is a dynamic programming solution that aims to find the minimum number of operations required to reduce an array of numbers to a certain condition. The array `f` represents the state at the previous step, and array `g` represents the state at the current step.

Time Complexity

The time complexity of the code is determined primarily by the for-loop that iterates over each element in the `nums` array. Inside this loop, a fixed number of operations are performed. Specifically, the loop executes once for each of the `n` elements of `nums`, and within the loop, a constant number of assignments and minimum function calls are performed. Each minimum function call operates over an array of fixed size 3 (or 2 in the case of `min(f[:2])`), which is a constant time operation.

Thus, the time complexity is `O(n)`, where `n` is the length of the `nums` array.

Space Complexity

The space complexity of the algorithm is determined by the space required to store the `f` and `g` arrays, and any additional variables used for iteration and temporary storage. Since `f` and `g` are arrays of constant size 3, they do not scale with the input size `n`.

Hence, the space complexity is `O(1)`, as the space used does not increase with the size of the input.