2263. Make Array Non-decreasing or Non-increasing

Hard · Greedy · Dynamic Programming · Leetcode Link

Problem Description

In this LeetCode problem, we are given an array `nums` of integers with a 0-based index. The goal is to determine the minimum number of operations needed to transform `nums` into a non-decreasing (every element is greater than or equal to the previous one) or non-increasing (every element is less than or equal to the previous one) sequence.

An operation consists of selecting any element in the array and either incrementing it by 1 or decrementing it by 1.

When looking for the minimum number of operations, it's important to keep in mind that we can use an operation on the same element multiple times if required.

Intuition

The intuition behind the solution is to consider separately the scenarios where we convert the array to non-decreasing and non-increasing orders. We will calculate the minimum operations required for both scenarios and then return the smaller result.

For both cases, the approach is to use dynamic programming (DP) to efficiently calculate the minimum number of operations. The basic idea of the DP solution is to create a 2D array `f` where each entry `f[i][j]` represents the minimum number of operations to convert the first `i` elements of the array into a sequence that ends with the number `j`.

To do this, the solution iterates over each element `x` of the input array and updates the DP table to reflect the minimum number of operations considering both the possibility of incrementing and decrementing.

Specifically, during the update process for each number `x`, it considers the possible outcomes for having each value from 0 to 1000 at position `i`. For each of these values, it finds the minimum operations from the previous step and the current cost of making `x` equal to the considered value. The cost is simply the absolute difference between `x` and the considered value.

Finally, the solution iterates over the values of the last element (the last row of the DP table), collecting the minimum number of operations required to achieve both non-decreasing and non-increasing order, respectively. The result is the minimum of these two values.

In summary, the intuition is to explore all possible target values for each element separately for both non-decreasing and non-increasing cases, combine them smartly using dynamic programming to keep track of the minimum cost at each step, and finally take the minimum of the costs from each of these two scenarios.

Solution Approach

The solution uses dynamic programming as its core approach, which is a method usually used to solve problems by breaking them down into smaller and simpler sub-problems, solving each sub-problem just once, and saving their solutions – typically using a memory-based data structure (array, map, etc.).

The `solve` function in the code defines the process for transforming the array to either non-decreasing or non-increasing order by considering one direction at a time. The function sets up a 2D array `f` to store the minimum operations needed to achieve the desired order with an array ending in value `j` at index `i`. The size of `f` is `(n+1) x 1001`, where `n` is the length of `nums` and `1001` accommodates all possible values an element can take after the operations (assuming the limit as 1000).

Let's walk through the main steps of the `solve` function:

- Iterate through each element `x` in `nums`, starting at index `1` for the array `f` since index `0` is used for the base case where no operations have been applied (`f[0]` is initialized with zeros).
- For each element `x`, we iterate from `0` to `1000` to consider all possible values `j` that the current element could take. `mi` stores the minimum number of operations seen so far for transforming the sequence up to the previous element.
- For every possible value `j`, it calculates the number of operations required to adjust `x` to `j` (`abs(x - j)`) and adds it to the minimum operations needed to get to the previous element's state, stored in `mi`. This dynamic programming step ensures that for each element, `f` stores the accumulated minimum operations to adjust the sequence to non-decreasing/non-increasing up to that element.
- After updating `f[i][j]` for all values from `0` to `1000`, it retains the minimum value reached for the entire sequence by taking the minimum of the last row of `f`, essentially, `min(f[n])`.

Post the `solve` function, the main part of the `convertArray` method applies this process twice: once for `nums` as is, and once for the reversed `nums`. Reversing `nums` effectively solves for a non-increasing sequence, as the reversed-non-decreasing sequence is the original non-increasing sequence.

The final answer is the minimum number of operations between the non-decreasing and non-increasing transformations.

Algorithm Used:

- Dynamic Programming.

Data Structures:

- A 2D list `f` to serve as the DP table.

Patterns:

- Separating the problem into two scenarios (non-decreasing and non-increasing) and taking the minimum of the two results.
- Iterative approach for dynamic programming with memory optimization properties.

The clever part of this algorithm is its ability to handle all possible end values for each subarray in an efficient manner, allowing for the determination of the minimum number of operations needed dynamically as the array is processed.

Example Walkthrough

Let's take a small example to illustrate the solution approach with the given array `nums`:

```
nums = [3, 2, 5, 1, 6]
```

We want to transform this array into either a non-decreasing or non-increasing sequence using the minimum number of operations.

Converting to Non-Decreasing:

Initialize 2D array `f` with size `(n+1) x 1001`, where `n` is the length of `nums`. In our case, `n` is 5, so we have `f` of size `6 x 1001`. We're considering all possible values that each element can take post operations (0 to 1000).

We loop over each element in `nums` starting from index 1 in `f` as index 0 represents the base case where the array is empty (no operations needed).

1. Element `3`: The initial minimum operations matrix `f` at index 1 will be set considering the cost of making `3` all possible values from 0 to 1000. For example, to make 3 a 0, we would need `f[1][0] = 3` operations.

2. Element `2`: Now, for each target value `j` from 0 to 1000, we calculate the minimum of `f[1][0..1000]` and add the cost to convert 2 to `j`, updating `f[2][j]` accordingly.

3. Continue filling `f` using the same approach for elements `5`, `1`, and `6`.

At the end of this process, `f[5][j]` contains the minimum operations needed to convert the array into a non-decreasing sequence ending with value `j`. The answer will be the minimum value in `f[5][0..1000]`.

Converting to Non-Increasing:

We repeat the same process, but before starting, we reverse the array:

```
Reversed nums = [6, 1, 5, 2, 3]
```

1. Element `6`: Similar to the non-decreasing process, `f[1][0..1000]` is set according to the cost to change `6` into all possible ending values.

2. Element `1`: Just as in the non-decreasing case, we iterate over all possible target values `j`, and for each, add the minimum of `f[1][0..1000]` and the cost to convert `1` to `j` into `f[2][j]`.

3. Proceed with each element (5, 2, 3) in the reversed array filling `f` accordingly.

After this, the `f[5][j]` row reflects the minimum operations required to make the original array non-increasing. The final step is to take the minimum value from `f[5][0..1000]`.

Summary:

Finally, the solution will return the smaller value between the two minimums obtained for both the non-decreasing and non-increasing scenarios. This represents the minimum number of operations required to convert the initial array into either a non-decreasing or a non-increasing sequence.

Python Solution

```python
from typing import List

class Solution:
    def convertArray(self, nums: List[int]) -> int:
        # Helper function to solve the problem for both original and reversed arrays
        def solve(arr):
            n = len(arr)  # Length of the array
            # Initialize a 2D array to store the minimum number of operations
            # needed to make all elements equal to any value between 0 and 1000
            min_ops = [[0] * 1001 for _ in range(n + 1)]

            for i, num in enumerate(arr, 1):  # Enumerate from 1 for 1-based indexing
                min_value = float('inf')  # Initialize the minimum value to be infinity
                for j in range(1001):
                    # Update the minimum value for the previous row
                    if min_value > min_ops[i - 1][j]:
                        min_value = min_ops[i - 1][j]
                    # Compute the minimum operations for current element to be j
                    min_ops[i][j] = min_value + abs(num - j)

            # Find and return the minimum value from the last row
            return min(min_ops[n])

        # Call the solve function on both the original and reversed array, and return the minimum result
        return min(solve(nums), solve(nums[::-1]))

# Usage
# sol = Solution()
# print(sol.convertArray([1, 5, 3, 3]))
```

Java Solution

```java
class Solution {
    // Main method to convert the array.
    public int convertArray(int[] nums) {
        // Find the minimum of the original array and the reversed array
        return Math.min(computeMinimumOperations(nums), computeMinimumOperations(reverseArray(nums)));
    }

    // Method to compute the minimum operations to satisfy the condition.
    private int computeMinimumOperations(int[] nums) {
        int n = nums.length; // Length of the array
        int[][] dp = new int[n + 1][1001]; // DP table to store minimal operations

        // Populate the dp table
        for (int i = 1; i <= n; ++i) {
            int minPrevious = Integer.MAX_VALUE; // Initialize to max value
            for (int j = 0; j <= 1000; ++j) {
                // Update minimum from the previous row
                minPrevious = Math.min(minPrevious, dp[i - 1][j]);
                // Calculate min operations for getting value j at index i
                dp[i][j] = minPrevious + Math.abs(j - nums[i - 1]);
            }
        }

        // Find the minimal operations among all possibilities for the last element
        int answer = Integer.MAX_VALUE; // Initialize to max value
        for (int cost : dp[n]) {
            answer = Math.min(answer, cost); // Update the answer with the minimal cost
        }

        return answer; // Return the final minimal cost
    }

    // Helper method to reverse the given array.
    private int[] reverseArray(int[] nums) {
        int[] copiedNums = nums.clone(); // Clone the array to avoid modifying the original one

        // Reverse the array in place
        for (int start = 0, end = copiedNums.length - 1; start < end; ++start, --end) {
            // Swap elements
            int temp = copiedNums[start];
            copiedNums[start] = copiedNums[end];
            copiedNums[end] = temp;
        }

        return copiedNums; // Return the reversed array
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the minimum cost to make all elements of the array equal
    int convertArray(vector<int>& nums) {
        int cost1 = calculateMinimumCost(nums);

        // Reversing the nums vector to calculate cost from both ends
        reverse(nums.begin(), nums.end());

        // Cost calculated after reversing array
        int cost2 = calculateMinimumCost(nums);

        // Return the minimum cost of the two possible scenarios
        return min(cost1, cost2);
    }

    // Helper function to calculate the minimum cost of making all
    // elements equal by incrementing or decrementing
    int calculateMinimumCost(vector<int>& nums) {
        int n = nums.size();

        // Create a 2D array to store subproblem solutions, where f[i][j] will
        // hold minimum cost to make the first i elements equal to j
        int f[n + 1][1001];

        // Initialize the 2D array with zero using memset
        memset(f, 0, sizeof(f));

        for (int i = 1; i <= n; ++i) {
            int minCostPrevious = INT_MAX; // Initialize the minimum cost to a large value
            for (int j = 0; j <= 1000; ++j) {
                // Update minCostPrevious to the minimum cost found in previous row
                minCostPrevious = min(minCostPrevious, f[i - 1][j]);

                // Calculate cost of making nums[i-1] equal to j and update f[i][j]
                // with the sum of minCostPrevious and current cost
                f[i][j] = minCostPrevious + abs(nums[i - 1] - j);
            }
        }

        // Return the minimum cost of making all elements of the array equal until the last element
        return *min_element(f[n], f[n] + 1001);
    }
};
```

Typescript Solution

```typescript
// Global variable to define the maximum value for calculation
const MAX_VALUE: number = 1000;

// Function to find the minimum cost to make all elements of the array equal
function convertArray(nums: number[]): number {
    let cost1: number = calculateMinimumCost(nums);

    // Reversing the nums array to calculate cost from both ends
    nums.reverse();

    // Cost calculated after reversing array
    let cost2: number = calculateMinimumCost(nums);

    // Return the minimum cost of the two possible scenarios
    return Math.min(cost1, cost2);
}

// Helper function to calculate the minimum cost of making all elements equal by incrementing or decrementing
function calculateMinimumCost(nums: number[]): number {
    let n: number = nums.length;

    // Create a 2D array to store subproblem solutions, where dp[i][j] will
    // hold minimum cost to make the first i elements equal to j
    let dp: number[][] = Array.from({length: n + 1}, () => Array(MAX_VALUE + 1).fill(0));

    // Initialize the subproblem solutions for base cases here if necessary

    for (let i: number = 1; i <= n; ++i) {
        let minCostPrevious: number = Number.MAX_SAFE_INTEGER; // Initialize the minimum cost to a large value
        for (let j: number = 0; j <= MAX_VALUE; ++j) {
            // Update minCostPrevious to the minimum cost found in previous row
            if (i > 1) {
                minCostPrevious = Math.min(minCostPrevious, dp[i - 1][j]);
            }

            // Calculate cost of making nums[i-1] equal to j and update dp[i][j]
            // with the sum of minCostPrevious and current cost
            dp[i][j] = minCostPrevious + Math.abs(nums[i - 1] - j);
        }
    }

    // Return the minimum cost of making all elements of the array equal until the last element
    return Math.min(...dp[n]);
}

// Example usage:
// let nums = [1, 2, 3]
// let cost = convertArray(nums)
// console.log(cost);
```

Time and Space Complexity

Time Complexity

The given function `convertArray` includes a nested function `solve` which computes the minimum cost to make all elements of the array equal by only increments or only decrements.

- The outer function `convertArray` calls `solve` twice, once for the original array and once for the reversed array.

- Within `solve`, there is a nested loop. The outer loop runs `n` times, where `n` is the length of `nums`.

- The inner loop runs for a constant 1001 times since it iterates over a preset range from 0 to 1000.

- Inside the inner loop, the computation is done in constant time, namely the minimum comparison `mi > f[i - 1][j]`, the minimum update `mi = f[i - 1][j]`, and the cost calculation `f[i][j] = mi + abs(x - j)`.

Putting it together, the time complexity of the `solve` function is `O(n * 1001)`, and since it is called twice, the overall time complexity of `convertArray` is `O(2 * n * 1001)` which simplifies to `O(n)` because constants are dropped in Big O notation and `1001` is a constant.

Space Complexity

The space complexity is determined by the amount of memory used by the function relative to the input size:

- The function `solve` uses a 2D array `f` of size `(n + 1) * 1001`, where `n` is the length of the input array `nums`.

- No other significant data structures are used that scale with the input size.

The space complexity of the `solve` function is `O((n + 1) * 1001)` which simplifies to `O(n)`, again dropping the constant `1001`.

Thus, the overall space complexity of `convertArray` is `O(n)` as only one instance of the 2D array `f` is maintained throughout the calls to `solve`.