2289. Steps to Make Array Non-decreasing

Medium Stack Array Linked List Monotonic Stack

Problem Description

In this problem, we are given an array `nums` which is 0-indexed, meaning the first element is at position 0. We perform a certain process repeatedly on this array. In a single step of this process, we remove every element `nums[i]` if it is smaller than its predecessor `nums[i - 1]`, for all positions `i` that are greater than 0. We continue performing this step until the array becomes non-decreasing, meaning each element is greater than or equal to the one before it. The task is to calculate the total number of steps required to make the array non-decreasing.

Intuition

The intuition behind the solution can stem from realizing that elements in the array only need to be considered for removal if they are less than some element that previously appeared. One efficient way to keep track of which elements could possibly be removed is to use a stack. The stack is a data structure allowing us to store elements and remove the tops if conditions are met (in this case, if an element is greater than the elements on top of it).

We iterate the array from right to left because we want to know which elements on the right (upcoming elements in this backward traversal) can be removed. Each step of the iteration is an opportunity to either push a new element onto the stack or to remove one or more elements that are smaller than the current element being considered. To avoid removing elements multiple times, we use a `dp` (dynamic programming) array to keep track of the number of steps taken to remove the next smaller elements.

When we hit an element that could cause the removal of other elements, we need to calculate how many removal steps it took for the next-smallest elements and update it to the maximum of the current steps recorded plus one or the steps taken for that element already. This ensures we always have the maximum number of steps it would take for each element to be removed, considering all its future smaller elements.

The solution completes when we have considered every element and the `dp` array contains the steps required to make each individual element non-decreasing in the context of the elements to its right. The answer to the problem is the maximum value in the `dp` array because that will be the maximum number of steps required to make any element satisfy the non-decreasing property in the context of the entire array.

Solution Approach

The implementation of the solution is based on a stack-based approach coupled with dynamic programming. Here are the key parts of the implementation:

- We initialize a stack named `stk` to keep track of the indices of elements that potentially need to be removed. This stack helps us to maintain the current sequence of elements under consideration for removal in descending order.
- Additionally, we create an array `dp` of the same size as the input array `nums`. Each entry `dp[i]` represents the number of steps required to remove the element at `nums[i]` due to a larger preceding element.
- We then iterate through the array `nums` in reverse (from right to left). At each index `i`, we consider whether `nums[i]` can lead to the removal of elements that are currently on the stack.
- Inside the loop, we perform the following actions:
  - While the stack is not empty and the top element of the stack (the element at the index `stk[-1]`) is less than `nums[i]`, it means that `nums[i]` can lead to the removal of `nums[stk[-1]]`. Therefore, we calculate the `dp` value for `nums[i]` which involves incrementing `dp[i]` by 1 and then comparing it to the `dp` value of `stk[-1]`, and taking the maximum between them.
  - After processing potential removals, `i` is appended to the stack.
- Once we finish iterating through `nums`, the `dp` array contains the number of steps each element requires to be removed. The result of the entire operation, which is the number of steps required for `nums` to become non-decreasing, is the maximum value in the `dp` array.
- In summary, the algorithm effectively processes elements in such a way that it counts how many removal rounds each can survive based on the larger elements before them. This is accomplished by keeping track of the previous elements' survival rounds and updating the current element's count accordingly.
- By using a stack, we efficiently manage the sequence of comparisons, only performing removal operations when necessary, and by leveraging dynamic programming, we store intermediate results to avoid redundant calculations.

The concept of dynamic programming, particularly memoization of steps needed for removal, is crucial in avoiding revisiting the same subproblems multiple times, which significantly reduces the overall time complexity of the solution.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Consider the array `nums` = [5, 3, 7, 6, 2].

1. We initialize an empty stack `stk` and an array `dp` of the same size as `nums` initialized to 0s, representing the number of steps needed to remove each element. So in this case, `dp` = [0, 0, 0, 0, 0].

2. We start iterating through the array `nums` from right to left:

   i. For `i` = 4 (`nums[4]` = 2), there are no previous elements, so we push `i` onto `stk` which now becomes [4].

   ii. Move to `i` = 3 (`nums[3]` = 6), `stk` is not empty and `nums[stk[-1]]` = 2 < 6, so we pop 4 from `stk`, update `dp[3]` = max(`dp[3]+1`, `dp[4]`) ⇒ `dp[3]` = 1. `stk` becomes empty and we push 3 onto `stk` - stack is [3].

   iii. Move to `i` = 2 (`nums[2]` = 7), `stk` is not empty and `nums[stk[-1]]` = 6 < 7, so we pop 3 from `stk`, update `dp[2]` = max(`dp[2]+1`, `dp[3]`) ⇒ `dp[2]` = 2. `stk` becomes empty and we push 2 onto `stk` - stack is [2].

   iv. Move to `i` = 1 (`nums[1]` = 3), `stk` is not empty but `nums[stk[-1]]` = 7 > 3, no need to pop 2 from `stk`, just push 1 onto `stk` - stack is now [2, 1].

   v. Move to `i` = 0 (`nums[0]` = 5), `stk` is not empty and `nums[stk[-1]]` = 3 < 5, so we pop 1 from the stack, update `dp[0]` = max(`dp[0]+1`, `dp[1]`) ⇒ `dp[0]` = 1. Since `nums[stk[-1]]` = 7 > 5, we stop popping from the stack and push 0 onto the stack - `stk` becomes [2, 0].

3. After iterating, our `dp` array has the values [1, 0, 2, 1, 0]. Each entry in `dp` represents the number of rounds an element can survive (0 indicates it doesn't need to be removed).

4. We look for the maximum value in the `dp` array, which is `dp[2]` = 2, meaning the total number of steps required to make the array non-decreasing is 2. These steps translate to:

   - Step 1: Remove `nums[3]` (6) because it is less than `nums[2]` (7). Array becomes [5, 3, 7, 2].
   - Step 2: Remove `nums[1]` (3) and `nums[3]` (2) because they are less than `nums[0]` (5) and `nums[2]` (7), respectively. Array becomes [5, 7].

The result indicates that the original array `[5, 3, 7, 6, 2]` can be made non-decreasing in 2 steps, resulting in the array `[5, 7]`.

Solution Implementation

Python

```python
class Solution:
    def totalSteps(self, nums: List[int]) -> int:
        # Initialize an empty stack to keep track of indices
        stack = []

        # Use this list to keep track of the number of steps
        # required to remove each number
        steps_to_remove = [0] * len(nums)

        # Initialize the max steps as 0, to store the maximum
        # number of steps needed to remove any element
        max_steps = 0

        # Iterate the nums list in reverse order
        for i in range(len(nums) - 1, -1, -1):
            # While the stack is not empty and the current number is greater than
            # the top element of the stack
            while stack and nums[i] > nums[stack[-1]]:
                # Calculate the steps to remove the current number
                # It's either 1 step more than its last or the steps needed to remove
                # the number at the top of the stack, whichever is greater
                steps_to_remove[i] = max(steps_to_remove[i] + 1, steps_to_remove[stack.pop()])

            # Add the current index to the stack
            stack.append(i)

        # Return the maximum steps needed to remove an element from nums
        max_steps = max(steps_to_remove)
        return max_steps
```

Java

```java
class Solution {

    /**
     * Calculate the total steps needed to remove all possible elements according to the game rules.
     *
     * @param nums The array of integers representing the initial sequence.
     * @return The total number of steps to remove all possible elements.
     */
    public int totalSteps(int[] nums) {
        // Creating a stack to keep track of indices of elements in 'nums'
        Deque<Integer> stack = new ArrayDeque<>();

        // Initialize the answer (total steps) to 0
        int totalSteps = 0;

        // The length of the input array
        int length = nums.length;

        // Creating an array to store the number of rounds each element can survive before being removed
        int[] rounds = new int[length];

        // Iterate over the elements in reverse
        for (int i = length - 1; i >= 0; --i) {
            // While stack is not empty and the current element is greater than the next element in the stack
            while (!stack.isEmpty() && nums[i] > nums[stack.peek()]) {
                // Calculate the number of rounds needed for the current element
                // by comparing and taking the maximum between the rounds for the current element
                // and the rounds for the element that will be popped from the stack plus one.
                rounds[i] = Math.max(rounds[i] + 1, rounds[stack.pop()]);

                // Update the total steps with the maximum number of rounds we've seen
                totalSteps = Math.max(totalSteps, rounds[i]);
            }

            // Push the current index onto the stack
            stack.push(i);
        }

        // Return the total number of rounds (total steps)
        return totalSteps;
    }
}
```

C++

```cpp
class Solution {
public:
    int totalSteps(vector<int>& nums) {
        stack<int> indicesStack; // Stack to maintain the indices of nums
        int maxSteps = 0; // Variable to store the maximum number of steps needed
        int numsSize = nums.size(); // Size of the nums vector
        vector<int> steps(numsSize); // Vector to store the number of steps for each element

        // Iterate over the elements from the end to the beginning
        for (int i = numsSize - 1; i >= 0; --i) {
            // Pop elements from the stack while the current element is greater
            // than the element at the top of the stack
            while (!indicesStack.empty() && nums[i] > nums[indicesStack.top()]) {
                // Calculate the steps for the current element based on the existing steps
                // for the top element and itself
                steps[i] = max(steps[i] + 1, steps[indicesStack.top()]);
                // Update maxSteps with the maximum steps needed so far
                maxSteps = max(maxSteps, steps[i]);
                // Remove the top element from the stack
                indicesStack.pop();
            }
            // Push the current index onto the stack
            indicesStack.push(i);
        }
        // Return the maximum number of steps required
        return maxSteps;
    }
};
```

TypeScript

```typescript
/**
 * Calculates the maximum number of steps to make the array non-increasing by
 * removing the minimum element from each non-increasing subarray.
 * @param nums The array of numbers to be processed.
 * @return The total number of steps required.
 */
function totalSteps(nums: number[]): number {
    // Initialize the answer to zero.
    let answer = 0;

    // Use a stack to keep track of elements and their steps to be removed.
    let stack: [number, number][] = [];

    // Iterate over the numbers in the provided array.
    for (let num of nums) {
        // Track the maximum number of steps needed so far.
        let maxSteps = 0;

        // Remove elements from the stack while the top is less than or equal to the current number.
        while (stack.length && stack[0][0] <= num) {
            // Update the maximum steps using the steps value from the element being removed.
            maxSteps = Math.max(stack[0][1], maxSteps);
            stack.shift(); // Remove the element from the stack's front.
        }

        // If the stack is not empty, increment the maximum steps.
        if (stack.length) maxSteps++;

        // Update the global answer with the maximum steps if necessary.
        answer = Math.max(maxSteps, answer);

        // Push the current number and its steps onto the stack.
        stack.unshift([num, maxSteps]);
    }

    // Return the calculated answer.
    return answer;
}
```

```python
class Solution:
    def totalSteps(self, nums: List[int]) -> int:
        # Initialize an empty stack to keep track of indices
        stack = []

        # Use this list to keep track of the number of steps
        # required to remove each number
        steps_to_remove = [0] * len(nums)

        # Initialize the max steps as 0, to store the maximum
        # number of steps needed to remove any element
        max_steps = 0

        # Iterate the nums list in reverse order
        for i in range(len(nums) - 1, -1, -1):
            # While the stack is not empty and the current number is greater than
            # the top element of the stack
            while stack and nums[i] > nums[stack[-1]]:
                # Calculate the steps to remove the current number
                # It's either 1 step more than its last or the steps needed to remove
                # the number at the top of the stack, whichever is greater
                steps_to_remove[i] = max(steps_to_remove[i] + 1, steps_to_remove[stack.pop()])

            # Add the current index to the stack
            stack.append(i)

        # Return the maximum steps needed to remove an element from nums
        max_steps = max(steps_to_remove)
        return max_steps
```

Time and Space Complexity

The given code implements an algorithm that computes the total number of removal steps required for an array of numbers, given certain conditions for removal. The algorithm utilizes a stack and a dynamic programming array to keep track of the maximum steps needed.

Time Complexity:

The time complexity of the algorithm is O(n). Here's why:

- We iterate through the list `nums` in reverse order, which is a linear operation with complexity O(n).
- For each element, we perform a while loop that pops elements from the stack until a condition is met. Although it seems like it could lead to a higher complexity, each element is pushed to and popped from the stack at most once. Therefore, the total number of operations for the while loop across all iterations is at most O(n).
- The `max()` function inside the while loop operates in constant time since it's just comparing two integers.

Combining these factors, the complexity remains linear with respect to the length of the input array.

Space Complexity:

The space complexity of the algorithm is O(n) due to the following:

- We use a dynamic programming array `dp` of the same length as the input array `nums`, which takes O(n) space.
- A stack `stk` is used, which in the worst case could store all elements if they are in increasing order. This also takes O(n) space.

Hence, the overall space complexity is O(n), dominated by the `dp` array and the stack `stk`.