1827. Minimum Operations to Make the Array Increasing

Easy Greedy Array

Problem Description

The problem provides us with an array of integers called `nums`, which uses 0-based indexing. We are tasked with finding the minimum number of operations needed to make this array strictly increasing. An operation consists of selecting any element in the array and incrementing it by 1.

A strictly increasing array is defined as one in which every element is less than the element immediately following it (`nums[i] < nums[i+1]`). A single-element array is considered strictly increasing by default since there are no adjacent elements to compare.

The ultimate goal here is to ensure that for every pair of adjacent elements `(nums[i], nums[i+1])`, the condition `nums[i] < nums[i+1]` holds true, by performing the least number of increment operations.

Intuition

The intuition behind the solution involves ensuring that for each element `nums[i]` in the array, if it's not already greater than the previous element `nums[i-1]`, we increment it enough times such that it becomes 1 more than the previous element.

To achieve this, we track the maximum value (`mx`) needed so far as we iterate through the array. For each value `v` in the array, if `v` is smaller or equal to `mx`, we know we need to increment `v` to at least `mx + 1` to maintain the strictly increasing property.

We calculate any gap that might exist between `mx + 1` (which `v` needs to be to keep the array strictly increasing) and the current value `v`. This gap represents the number of increments needed for the current element `v`. We add this gap to our running total of operations (`ans`). After considering `v`, we update `mx` to be the maximum of `mx + 1` (to ensure strict increasing order) and `v` (in case the current value is already large enough and doesn't need increments).

Here's the flow:

1. We initialize the `ans` (answer) variable to track the total number of operations performed and `mx` (maximum needed so far) with the value `0`.
2. We iterate through each value `v` in `nums`.
3. If `v` is less than or equal to `mx`, we calculate the difference `mx + 1 - v` (the number of operations needed to make the current element strictly larger than the previous one) and add it to `ans`. If `v` is already greater than `mx`, no operations are needed, so we would add `0`.
4. We update `mx` to be the greater value between `mx + 1` and `v` to ensure that we're always setting `mx` to be at least one greater than the last value (to maintain the strictly increasing order) or to account for the current value if it doesn't need to be incremented.
5. After going through all elements in `nums`, the value of `ans` will be the minimum number of operations required to make `nums` strictly increasing.

Solution Approach

The solution uses a simple, yet efficient algorithm to resolve the challenge. It does not require complex data structures or patterns. The straightforward use of a for-loop and basic arithmetic operations in combination with simple variable tracking proves to be efficient for this problem.

Here's a detailed walkthrough of the implementation based on the reference solution:

1. We start by initializing two variables, `ans` and `mx`, to `0`. `ans` will keep track of the total number of operations performed, while `mx` will hold the current maximum value needed to maintain a strictly increasing sequence.
2. The core part of our solution is a for-loop that iterates through each number `v` in the input array `nums`. This loop is where we determine if an increment operation is necessary and if so, how many:

   ```
   for v in nums:
       ans += max(0, mx + 1 - v)
       mx = max(mx + 1, v)
   ```

   Let's break down the loop operations:

   - `ans += max(0, mx + 1 - v)`: We calculate the difference between `mx + 1` and `v`, which gives us the number of operations needed to make the current number `v` comply with the strictly increasing criterion. We use `max(0, mx + 1 - v)` because if `v` is already larger than `mx`, we do not need to perform any operations, hence we add zero to `ans`.
   - `mx = max(mx + 1, v)`: We update `mx` to be the larger of `mx + 1` and `v`. This operation is crucial because it ensures that we will always compare subsequent numbers to a value that keeps the sequence strictly increasing. If `v` is already equal to or larger than `mx + 1`, we set `mx` to `v`. Otherwise, we ensure `mx` becomes `mx + 1`, which is the minimum value the next number in the sequence must exceed.
3. Once the loop completes, the variable `ans` will contain the sum of all the increments performed, which is the total number of operations needed to make the array `nums` strictly increasing. This value is then returned as the solution.

The simplicity of the algorithm comes from the realization that we can keep the problem state using only two variables and do not need to modify the original array. Essentially, it's the concept of dynamic programming without the need for memoization or auxiliary data structures, as we only care about the relationship between adjacent elements. The time complexity of the algorithm is O(n), where n is the number of elements in `nums`, because we only need to iterate through the array once. The space complexity is O(1) because we use a constant amount of extra space.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we have the following array:

```
nums = [3, 4, 2, 6, 1]
```

We want to perform the minimum number of operations to make this array strictly increasing.

Let's apply our algorithm:

1. Initialize `ans` and `mx` to `0`. These will keep track of the total number of operations and the maximum value needed respectively.
2. Start the for-loop with the first element `v = 3`. Since `mx` is 0, `mx + 1 - v` is -2. We don't need to perform any increments because `v` is already greater than `mx`. Update `ans = 0` and `mx = max(1, 3) = 3`.
3. Next, `v = 4`. No increments needed, as `v` is greater than `mx`. Update `ans = 0` and `mx = max(4, 4) = 4`.
4. Now, `v = 2`. Since 2 is not greater than `mx`, we need to increment `v` by `mx + 1 - v = 4 + 1 - 2 = 3` times. Update `ans = 0 + 3 = 3` and now `mx = max(4 + 1, 2) = 5`.
5. Then, `v = 6`. No increments needed, as `v` is greater than `mx`. Update `ans = 3` and `mx = max(5 + 1, 6) = 6`.
6. Lastly, `v = 1`. `v` is less than `mx`, we must increment `v` by `mx + 1 - v = 6 + 1 - 1 = 6` times. Update `ans = 3 + 6 = 9` and finally `mx = max(6 + 1, 1) = 7`.

After going through each element, we found the total number of operations required to make `nums` strictly increasing is `ans = 9`.

In conclusion, the output for the array `nums = [3, 4, 2, 6, 1]` would be `9`, meaning we need to perform 9 increment operations to transform it into a strictly increasing array.

Solution Implementation

Python

```python
class Solution:
    def minOperations(self, nums: List[int]) -> int:
        # Initialize the answer counter to count the minimum operations required
        operations_count = 0

        # Initialize the max_value variable to keep track of the maximum integer seen so far
        max_value = 0

        # Loop through each value in the given list
        for value in nums:
            # If the current value is less than or equal to the max value adjusted by one,
            # calculate the operations needed to make it one greater than the max_value seen so far
            operations_count += max(0, max_value + 1 - value)

            # Update max value: it should be the maximum of the previous max_value adjusted by one,
            # or the current value in the list in case it's larger
            max_value = max(max_value + 1, value)

        # Return the total number of operations counted
        return operations_count
```

Java

```java
class Solution {
    public int minOperations(int[] nums) {
        int operations = 0; // To store the minimum number of operations required
        int maxVal = 0; // To keep track of the maximum value obtained so far

        // Iterate through all elements in the array
        for (int value : nums) {
            // If the current value is less than the maxVal + 1,
            // we need to increment it, which counts as an operation
            operations += Math.max(0, maxVal + 1 - value); // Add necessary operations

            // Update the maxVal to be the maximum of the current value and maxVal + 1,
            // since we want to ensure the next number is at least maxVal + 1
            maxVal = Math.max(maxVal + 1, value); // Update the maxVal
        }

        return operations; // Return the total number of operations
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to calculate the minimum number of operations needed
    // to make the array strictly increasing
    int minOperations(std::vector<int>& nums) {
        int totalOperations = 0; // Variable to keep track of total operations performed
        int maxSoFar = 0; // Variable to keep track of the maximum value encountered so far

        // Loop through each element in the vector
        for (int& value : nums) {
            // Calculate operations needed for current element to be greater
            // than the maxSoFar. If the value is already greater than maxSoFar,
            // no operations are needed; hence, we use max with 0.
            totalOperations += std::max(0, maxSoFar + 1 - value);

            // Update maxSoFar to be either the current value or one more
            // than maxSoFar, whichever is larger, to maintain strict increasing order.
            maxSoFar = std::max(maxSoFar + 1, value);
        }

        return totalOperations; // Return the total number of operations
    }
};
```

TypeScript

```typescript
/**
 * Calculates the minimum number of operations needed to make the array strictly increasing.
 * Each operation consists of incrementing a number in the array.
 * @param {number[]} nums - The input array of numbers.
 * @returns {number} The minimum number of operations required.
 */
function minOperations(nums: number[]): number {
    // Initialize the number of operations (ans) to 0
    let operationsCount = 0;

    // Initialize the maximum number seen so far to enable comparisons
    let currentMax = 0;

    // Iterate through each number in the input array
    for (const value of nums) {
        // Calculate the number of operations needed for the current number, if any,
        // ensuring the number is at least one more than the current maximum
        operationsCount += Math.max(0, currentMax + 1 - value);

        // Update the current maximum to be either the incremented maximum or the current value,
        // whichever is larger, to maintain the strictly increasing property
        currentMax = Math.max(currentMax + 1, value);
    }

    // Return the total number of operations needed
    return operationsCount;
}
```


```python
class Solution:
    def minOperations(self, nums: List[int]) -> int:
        # Initialize the answer counter to count the minimum operations required
        operations_count = 0

        # Initialize the max_value variable to keep track of the maximum integer seen so far
        max_value = 0

        # Loop through each value in the given list
        for value in nums:
            # If the current value is less than or equal to the max value adjusted by one,
            # calculate the operations needed to make it one greater than the max_value seen so far
            operations_count += max(0, max_value + 1 - value)

            # Update max value: it should be the maximum of the previous max_value adjusted by one,
            # or the current value in the list in case it's larger
            max_value = max(max_value + 1, value)

        # Return the total number of operations counted
        return operations_count
```


Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the `nums` array. This is because there is a single for-loop that iterates over all elements of the array once, performing a constant number of operations for each element. The operations performed within the loop (calculations and comparisons) are all constant time operations.

Space Complexity

The space complexity of the given code is `O(1)` (constant space). This is because the amount of extra space used does not depend on the input size `n`. The code only uses a fixed number of variables (`ans`, `mx`, and `v`) that do not expand with the size of the input array.