2919. Minimum Increment Operations to Make Array Beautiful

Medium Array Dynamic Programming

Problem Description

This problem involves an integer array `nums` which is 0-indexed and has length `n`. Along with the array, you're given an integer `k`. You have the ability to execute an *increment* operation on any element of the array. The increment operation involves choosing an index `i` (where `i` ranges from 0 to `n - 1`) and increasing `nums[i]` by `1`.

The main goal is to perform as few operations as possible so that the array becomes *beautiful*. An array is deemed beautiful if any subarray of size 3 or more contains at least one element that is greater than or equal to `k`. A subarray, by definition, is a contiguous non-empty sequence within the array.

You are asked to determine and return the minimum number of increment operations required to make the `nums` array beautiful.

Intuition

For this problem, we utilize a dynamic programming approach, keeping track of a set of variables `f`, `g`, and `h`. These variables represent the minimum increment operations needed to get at least one element equal to or greater than `k` in the last three elements of each subarray within the array.

We initialize `f`, `g`, and `h` to 0, as no operations have been performed yet. We then iterate over the array `nums` and for each element `x`, we perform the following:

- We update `f` to the old value of `g` because `f` stands for the minimum operations considering the subarray ending one element before the current one.
- Similarly, `g` is updated to the old value of `h` to move the window one step ahead.
- `h` is updated to be the minimum of the previous `f`, `g`, and `h` plus the number of operations required to make the current element `x` greater than or equal to `k`. If `x` is already greater than or equal to `k`, no operations are needed (`max(k - x, 0)` ensures this).

By iterating in this manner, we make sure to consider the minimum required operations for each element while taking the previous elements into account, leading us to the solution for the entire array.

In the end, since we only care about any subarray of size 3 or more being beautiful, we return the minimum of `f`, `g`, and `h`, which represents the fewest number of operations needed across all considered subarrays.

Solution Approach

In the implemented solution, the dynamic programming approach is key. The algorithm avoids the need for a traditional 2D dynamic programming table that would take into account every subarray by using three variables `f`, `g`, and `h` to store states. This is an example of space optimization within dynamic programming.

The algorithm processes the array `nums` from left to right. As described previously, `f` is the count of increment operations for the subarray ending two elements before the current, `g` is for the subarray ending one element before the current, and `h` is for the subarray ending with the current element.

Let's take a closer look at how we update these three states as we iterate through each element, `x`, in `nums`:

- We first store the old values of `g` and `h`. These represent the best solutions (minimum operation count) for subarrays ending one and two elements before `x`.
- We update `f` to be equal to the old value of `g`. This is assuming that for the next element, `f` will represent the best previous solution excluding the current element (`x`).
- The old value of `h` becomes the new `g`, adjusting the window to include `x`.
- To calculate the new value for `h`, we find the minimum among the old values of `f`, `g`, and `h`, and add the number of operations needed to make `x` at least `k`. The expression `max(k - x, 0)` yields the number of operations needed—if `x` is already equal to or larger than `k`, no operations are required, hence the `max` function ensures that the increment is zero in such cases.
- The dynamic update for `h` is `h = min(f, g, h) + max(k - x, 0)`.

By performing the above steps for each element in `nums`, we continually update the count of required operations for all relevant subarrays. This way, we achieve an `O(n)` time complexity since we pass through the array only once without the need for a nested loop.

At the code's conclusion, we return the minimum operations needed across all subarrays of size 3 or greater, which is the least of `f`, `g`, and `h` after processing all elements of `nums`. This final step delivers the minimum number of increment operations needed to make the array beautiful.

Example Walkthrough

Let's consider a small example with the array `nums = [1, 2, 3, 4]` and `k = 5`. We want to find the minimum number of increment operations to make the array beautiful according to our problem description.

Applying the solution approach described:

1. We initialize `f = g = h = 0` because no operations have been carried out yet, and we are looking at the subarrays ending before the first element.
2. We look at the first element of `nums` which is `1`. Since `k = 5`, the number of operations to make `1` at least `5` is `5 - 1 = 4`.
   - We update `h` to `min(f, g, h) + max(5 - 1, 0) = min(0, 0, 0) + 4 = 4`.
   - Now, `f` remains `0`, `g` remains `0`, and `h` becomes `4`.
3. We move to the second element, `2`. We first need to update our variables:
   - `f` becomes the old `g`, which is `0`.
   - `g` becomes the old `h`, which is `4`.
   - The number of operations to make `2` at least `5` is `5 - 2 = 3`.
   - Update `h` with `h = min(f, g, h) + max(5 - 2, 0) = min(0, 4, 4) + 3 = 3`.
   - Now, `f` is `0`, `g` is `4`, and `h` is `3`.
4. Next, we look at the third element, `3`. Again, update the variables:
   - `f` becomes the old `g`, which is `4`.
   - `g` becomes the old `h`, which is `3`.
   - The number of operations for `3` is `5 - 3 = 2`.
   - Update `h` with `h = min(f, g, h) + max(5 - 3, 0) = min(4, 3, 3) + 2 = 3 + 2 = 5`.
   - After this, `f` is `4`, `g` is `3`, and `h` is `5`.
5. Lastly, we consider the fourth element, `4`:
   - Update `f` to the old value of `g`, which is `3`.
   - Update `g` to the old value of `h`, which is `5`.
   - The number of operations to make `4` at least `5` is `5 - 4 = 1`.
   - Update `h` with `h = min(f, g, h) + max(5 - 4, 0) = min(3, 5, 5) + 1 = 3 + 1 = 4`.
   - The final values are `f` is `3`, `g` is `5`, and `h` is `4`.

Given our variables, the minimum of `f`, `g`, and `h` will give us the least number of operations needed. In this case:

- The minimum number of operations required is `min(f, g, h) = min(3, 5, 4) = 3`.

Therefore, according to our solution approach and this example, we would need a minimum of 3 operations to make the array `nums` beautiful.

Solution Implementation

Python

```python
class Solution:
    def minIncrementOperations(self, nums: List[int], k: int) -> int:
        # Initialize the dp variables for the minimum operations
        # needed to make the previous, current, and next numbers at least 'k'.
        prev_op_count = cur_op_count = next_op_count = 0

        # Loop through each number in the array to calculate the
        # minimum operations required.
        for num in nums:
            # Update the minimum operation counts for the three states
            # State transitions:
            # prev_op_count -> the previous minimum operation count
            # cur_op_count -> the current minimum operation count
            # next_op_count -> the estimated operation count for next number
            prev_op_count, cur_op_count, next_op_count = \
                cur_op_count, next_op_count, min(prev_op_count, cur_op_count, next_op_count) + max(k - num, 0)

        # Return the minimum of the three states' operation counts as
        # that would be the minimum operations needed for the entire array.
        return min(prev_op_count, cur_op_count, next_op_count)
```

Java

```java
class Solution {
    public long minIncrementOperations(int[] nums, int k) {
        long firstPrevOperationCount = 0; // Initialize previous counts for recursive state
        long secondPrevOperationCount = 0;
        long currentOperationCount = 0;

        // Iterate over the array of numbers
        for (int number : nums) {
            // Calculate the minimum count of operations required for the current number.
            // This involves taking the minimum count from the previous two iterations
            // and adding the required increments to reach at least k.
            long optimalCurrentOperationCount = Math.min(
                Math.min(firstPrevOperationCount, secondPrevOperationCount),
                currentOperationCount
            ) + Math.max(k - number, 0);

            // Shift the operation counts for the next iteration.
            // The second previous becomes the first previous,
            // the current becomes the second previous,
            // and the new current count is calculated above.
            firstPrevOperationCount = secondPrevOperationCount;
            secondPrevOperationCount = currentOperationCount;
            currentOperationCount = optimalCurrentOperationCount;
        }

        // Return the minimum count of operations amongst the last three counts.
        return Math.min(
            Math.min(firstPrevOperationCount, secondPrevOperationCount),
            currentOperationCount
        );
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // for std::min and std::max

class Solution {
public:
    long long minIncrementOperations(std::vector<int>& nums, int k) {
        // Initialize the variables to store the minimal operations count for the last 3 increments
        long long prevPrevCount = 0; // f represents the count 2 steps before
        long long prevCount = 0;     // g represents the count 1 step before
        long long currCount = 0;     // h represents the current count

        // Iterate over the values in the 'nums' array
        for (int x : nums) {
            // Calculate the minimal operations needed for the current element
            // It's the minimum out of the last three results plus any increments needed to reach 'k'
            long long newCurrCount = std::min({prevPrevCount, prevCount, currCount}) + std::max(k - x, 0);

            // Shift the counts for the next iteration
            prevPrevCount = prevCount;
            prevCount = currCount;
            currCount = newCurrCount;
        }

        // The final answer is the minimal operations count out of the last three calculations
        return std::min({prevPrevCount, prevCount, currCount});
    }
};
```

TypeScript

```typescript
function minIncrementOperations(nums: number[], k: number): number {
    // Initialize three variables to store intermediate results.
    let [minOp1, minOp2, minOp3] = [0, 0, 0];

    // Loop through the array of numbers to determine the minimal operations needed.
    for (const number of nums) {
        // Update the variables to store the minimal operations.
        // minOp3 holds the result of the current computation,
        // which is the minimal of the previous computations (minOp1, minOp2, minOp3) plus
        // the max between 0 and (k - current number), ensuring we never subtract.
        [minOp1, minOp2, minOp3] = [
            minOp2,
            minOp3,
            Math.min(minOp1, minOp2, minOp3) + Math.max(k - number, 0)
        ];
    }

    // Return the minimum of the three variables, which represents the minimum operations required.
    return Math.min(minOp1, minOp2, minOp3);
}
```

```python
class Solution:
    def minIncrementOperations(self, nums: List[int], k: int) -> int:
        # Initialize the dp variables for the minimum operations
        # needed to make the previous, current, and next numbers at least 'k'.
        prev_op_count = cur_op_count = next_op_count = 0

        # Loop through each number in the array to calculate the
        # minimum operations required.
        for num in nums:
            # Update the minimum operation counts for the three states
            # State transitions:
            # prev_op_count -> the previous minimum operation count
            # cur_op_count -> the current minimum operation count
            # next_op_count -> the estimated operation count for next number
            prev_op_count, cur_op_count, next_op_count = \
                cur_op_count, next_op_count, min(prev_op_count, cur_op_count, next_op_count) + max(k - num, 0)

        # Return the minimum of the three states' operation counts as
        # that would be the minimum operations needed for the entire array.
        return min(prev_op_count, cur_op_count, next_op_count)
```

Time and Space Complexity

The time complexity of the provided code is `O(n)` where `n` is the length of the input list `nums`. This is because the code iterates over the list `nums` exactly once, and within each iteration, it performs a constant number of operations that include comparisons and basic arithmetic, which do not depend on the size of the list.

The space complexity of the code is `O(1)`. Despite the size of the input, the code uses a fixed amount of space, represented by the variables `f`, `g`, and `h`. No matter how large the input list is, these variables do not require any additional space that scales with the size of the input. Hence the space required by the algorithm remains constant.