1526. Minimum Number of Increments on Subarrays to Form a Target Array

Hard · Stack · Greedy · Array · Dynamic Programming · Monotonic Stack

Problem Description

The problem presents an integer array called `target`. We also have an array of the same size as `target`, named `initial`, which is filled with zeros. The key objective is to transform the `initial` array into the `target` array. The only operation allowed to achieve this is to select a subarray from `initial` and increment each of its elements by one. The task is to determine the minimum number of such operations required to reach the `target` array configuration. It's also specified that the final answer will fit within a 32-bit integer, which means it won't be too huge for an int data type to store.

To solve this problem, one needs to intuitively understand that the minimum number of operations needed is closely related to the differences between the heights (values) of adjacent elements in the target array.

Intuition

Intuition behind the solution arises from looking at the problem in terms of “height differences”. Imagine a graph where the value of each element in the target array represents the height at that particular position. Incrementing a subarray by 1 is like raising the ground level of a terrain from one point to another by one unit. To minimize operations, we should always try to raise the terrain in the largest possible steps rather than incrementing little by little.

The first insight is that we'll have to increment the first element of `initial` to the first element of `target` for sure since there are no previous elements to consider, which accounts for `target[0]` operations.

Following that, we move through the target array and look at the difference between each pair of adjacent elements (`b - a` for elements `a` and `b`). Each time we encounter a rise in height (where `b > a`), we need additional operations equivalent to the height difference to "build up" our terrain to that new height. If the height decreases or remains equal (`b - a` is less than or equal to 0), we do not need extra operations since we can imagine that the previous increments have already covered that height.

By summing up all necessary height differences, we can get the total number of increments required. Therefore, the number of operations can be given by the first element of the `target` array plus the sum of all positive differences between successive elements in the `target` array. The given Python function `pairwise` from the itertools library can be used here to generate the adjacent pairs of elements and calculate the sum of the positive differences.

Solution Approach

The solution to this problem utilizes a straightforward Greedy algorithm approach and Python’s built-in functionality to generate adjacent pairs easily.

Here's a brief walkthrough of the implementation:

1. **Initialization**:
   - We start by initializing the number of operations to the value of the first element in `target`, because the initial array is filled with zeros, and we must at least increment the first element 'target[0]' times to match the first element of the `target` array.
2. **Iterating through the array**:
   - We then iterate over the `target` array, comparing each element to its predecessor to find the difference between them. The iteration of pairs is done using the `pairwise` function, which neatly returns a tuple containing pairs of adjacent elements from the iterable `target`.
3. **Calculating the Needed Operations**:
   - For each pair `(a, b)` obtained from the `pairwise`, we add to our operations count the difference `(b - a)` if `b` is greater than `a`. The expression `max(0, b - a)` ensures that we only consider positive differences, which represent an increase in value from one element in `target` to the next.
   - The rationale behind this is that whenever `b` is greater than `a`, we need additional operations to raise `initial[a]` to `initial[b]`. If `b` is equal to or less than `a`, no extra operations are needed because we assume `initial[a]` is already at least at height `a`, so we can effectively "carry over" this height to `initial[b]`.
4. **Summation**:
   - We sum up all the extra operations needed for each increase in the array, adding it to our initial count of `target[0]` operations. The sum is computed using the built-in `sum` function, which takes an iterable.
5. **Result**:
   - The final result is returned as the sum of the initial operations count and the accumulated extra operations needed. This sum represents the minimum number of operations required to form the `target` array from the `initial` array.

The data structures and patterns used in this approach are quite simple and consist of:

- A **List** (`target`) to store the integer values we're working with,
- The **Greedy algorithm** pattern, where we take the optimal decision step-by-step based on the current and next element,
- Python's built-in **`sum`** function to calculate the aggregate of operations needed,
- **`pairwise`** utility from the itertools library to iterate through the array in adjacent element pairs.

The beauty of this solution lies in its simplicity and efficiency. There are no complex data structures required, and the algorithm runs in linear time, making it a very practical solution for this problem.

Example Walkthrough

Let's walk through a small example to clearly understand how the solution approach works.

Suppose we have a `target` array given as `[1, 3, 2, 4]`. Our goal is to transform an `initial` array from `[0, 0, 0, 0]` to match the `target` array using the fewest number of operations.

1. **Initialization**:
   - Start with operations count equal to the first element of `target`: `operations = target[0] = 1`.
2. **Iterating through the array**:
   - Generate pairs using `pairwise`: (1, 3), (3, 2), (2, 4).
3. **Calculating the Needed Operations**:
   - Pair (1, 3): Increase is 2, so add 2 to operations: `operations += max(0, 3 - 1) = 3`.
   - Pair (3, 2): No increase (it's actually a decrease), so add 0: `operations += max(0, 2 - 3) = 3`.
   - Pair (2, 4): Increase is 2, so add 2 to operations: `operations += max(0, 4 - 2) = 5`.
4. **Summation**:
   - We combine the counted operations to get the total.
5. **Result**:
   - The result is `operations = 1 + 2 + 0 + 2 = 5`. So, it takes a minimum of 5 operations to transform `initial` into `target`.

In this example, we increment the first element (index 0) of the initial array 1 time to match the first element of the target array, then we perform another operation on the subarray from index 0 to index 1 to increment both elements (making the array `[1, 1, 0, 0]`). Next, we perform a third operation on the same subarray, resulting in `[2, 2, 0, 0]`. No operation is needed for the transition from 3 to 2, as the previous operations have already covered this. Finally, we perform two more operations from index 2 to index 3 to increment the last two elements by 1 each time, resulting in `[2, 3, 1, 1]` and then `[2, 3, 2, 2]`, and finally our target `[1, 3, 2, 4]`. Thus, a total of 5 operations were needed.

Solution Implementation

Python

```python
from itertools import pairwise  # Python 3 function from the itertools module

class Solution:
    def minNumberOperations(self, target: List[int]) -> int:
        # The function calculates the minimum number of operations needed to form the target array
        # from a starting array of zeros. Each operation increments a contiguous subarray by 1.

        # The first operation will increment the contiguous subarray from the start to set the first element
        # Thus, the number of operations for the first element is equal to its value
        operations = target[0]

        # sum up the differences between consecutive elements if they are positive
        # as these represent the additional operations needed for increasing the array to match the target
        for previous, current in pairwise(target):
            operations += max(0, current - previous)  # Only positive differences are added

        # The total number of operations to match the target array is returned
        return operations
```

Java

```java
class Solution {

    public int minNumberOperations(int[] target) {
        // Initialize minimum operations with the first element
        // since that's the minimum number of operations needed
        // to increase from 0 to the first element's value.
        int minOperations = target[0];

        // Iterate through the array starting from the second element
        for (int i = 1; i < target.length; ++i) {
            // If the current element is greater than the previous one,
            // it means we need additional operations to increase from
            // the previous element's value to the current element's value.
            if (target[i] > target[i - 1]) {
                minOperations += target[i] - target[i - 1];
            }
            // If not, no additional operations are needed because the
            // operations required to reach the previous element are adequate.
        }

        // Return the total minimum number of operations required.
        return minOperations;
    }
}
```

C++

```cpp
#include <vector>

class Solution {
public:
    /**
     * Calculate the minimum number of operations to form a target array from a zero array.
     * An operation is incrementing a subarray by 1.
     *
     * @param target Vector of integers representing the target array.
     * @return The minimum number of operations required.
     */
    int minNumberOperations(std::vector<int>& target) {
        int totalOperations = target[0]; // Start with the first value as the initial number of operations.
        // Iterate through the target array starting from the second element.
        for (int i = 1; i < target.size(); ++i) {
            // If the current target value is greater than the previous,
            // this signifies a need for additional operations to increment the subarray.
            if (target[i] > target[i - 1]) {
                totalOperations += target[i] - target[i - 1];
            }
        }
        // Return the total number of operations needed to form the target array.
        return totalOperations;
    }
};
```

TypeScript

```typescript
// Function to calculate the minimum number of operations needed to
// raise a series of values to match the heights specified in the 'target' array.
// Each operation increments a subarray's elements by 1.
function minNumberOperations(target: number[]): number {
    // Initialize the number of operations with the first element of the target array,
    // since we start with an array of zeros, the first element itself will be
    // the number of operations needed to reach its value.
    let operations = target[0];

    // Loop through the target array starting from the second element.
    for (let index = 1; index < target.length; ++index) {
        // If the current element is greater than the previous one,
        // we need additional operations for the difference between
        // the current and the previous elements.
        if (target[index] > target[index - 1]) {
            operations += target[index] - target[index - 1];
        }
    }

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


```python
from itertools import pairwise  # Python 3 function from the itertools module

class Solution:
    def minNumberOperations(self, target: List[int]) -> int:
        # The function calculates the minimum number of operations needed to form the target array
        # from a starting array of zeros. Each operation increments a contiguous subarray by 1.

        # The first operation will increment the contiguous subarray from the start to set the first element
        # Thus, the number of operations for the first element is equal to its value
        operations = target[0]

        # sum up the differences between consecutive elements if they are positive
        # as these represent the additional operations needed for increasing the array to match the target
        for previous, current in pairwise(target):
            operations += max(0, current - previous)  # Only positive differences are added

        # The total number of operations to match the target array is returned
        return operations
```


Time and Space Complexity

The code provided calculates the minimum number of operations to form a target array where each operation increments a subarray by 1. The function `minNumberOperations` does this by summing the differences between consecutive elements where the second element is larger than the first, plus the value of the first element in the target array. Let’s analyze both the time complexity and space complexity:

Time Complexity:

The function iterates once over the list of `target` elements using the `pairwise` utility to consider pairs of consecutive elements. The time complexity for this operation is O(n), where n is the length of the `target` array, because it goes through the list once.

The `pairwise` utility itself, which is presumably a wrapper around a simple loop-like construct that yields successive pairs of elements, does not add any significant time overhead as it just creates a tuple of the current and next elements in the array.

Therefore, the overall time complexity of the code is `O(n)`.

Space Complexity:

The function is a one-liner without the use of additional data structures that depend on the size of the input. It relies on generator expressions which yield one item at a time. The `max` function is applied in a generator expression, so it does not require additional space proportional to the input size.

The space complexity is not affected by the size of the input array, as the `sum` function processes the generator expression iteratively. The `pairwise` utility is expected to yield one pair at a time and doesn't store the entire list of pairs in memory.

As a result, the overall space complexity of the code is `O(1)`, reflecting constant space usage.