370. Range Addition

Medium | Array | Prefix Sum | Leetcode Link

Problem Description

In this problem, we are given an integer `length` which refers to the length of an array initially filled with zeros. We are also given an array of update operations called `updates`. Each update operation is described as a tuple or list with three integers: `[startIdx, endIdx, inc]`. For each update operation, we are supposed to add the value `inc` to each element of the array starting at index `startIdx` up to and including the index `endIdx`.

The goal is to apply all the update operations to the array and then return the modified array.

For instance, if `length = 5` and an update action specifies `[1, 3, 2]`, then after this update, the array will have `2` added to its 1st, 2nd, and 3rd positions (keeping zero-based indexing in mind), resulting in the array `[0, 2, 2, 2, 0]` after this single operation.

After applying all updates, we need to return the final state of the array.

Intuition

The intuitive brute force approach would be to go through each update and add `inc` to all elements ranging from `startIdx` to `endIdx` for every update in `updates`. However, this would be time-consuming, especially for a large number of updates or a large range within the updates.

This is where the prefix sum technique comes into play. It is a very efficient way for handling operations that involve adding some value to a range of elements in an array. The main intuition behind prefix sums is that we can record changes at the borders – at the start index, we begin to add the increment, and just after the end index, we cancel it out.

In more detail, for each update `[startIdx, endIdx, inc]`, we add `inc` to the position `startIdx` and subtract `inc` from `endIdx + 1`. This marks the range where the increment is valid. When we compute the prefix sum of this array, it will apply the `inc` increment to all elements since `startIdx`, and the subtraction at `endIdx + 1` will counterbalance it, returning the array to its original state beyond `endIdx`.

The accumulation step goes through the array and adds each element to the sum of all previous elements, effectively applying the increments and decrements outlined in the updates.

In the presented solution, the Python `accumulate` function from the `itertools` module takes care of the accumulation step for us, summing up the differences and giving us the array after all updates have been applied.

Solution Approach

The implementation of this solution is straightforward once we understand the intuition behind using prefix sums. Here's a step-by-step rundown of the algorithm:

1. We initialize an array `d` with a length of `length` filled with zeros. This array will serve as our difference array which records the difference of each position compared to the previous one.
2. We then iterate over each update in the `updates` array. Each update is in the format `[startIdx, endIdx, inc]`.
3. For each update, we add `inc` to `d[startIdx]`. This signifies that from `startIdx` onwards, we have an increment of `inc` to be applied.
4. We then check if `endIdx + 1 < length`, which is to ensure we do not go out of bounds of the array. If we are still within bounds, we subtract `inc` from `d[endIdx + 1]`. This effectively cancels out the previous increment beyond the `endIdx`.
5. After processing all updates, `d` now contains all the changes that are needed to be applied in the form of a difference array.
6. Finally, we use the `accumulate` function of Python to calculate the prefix sum array from the difference array `d`. This step goes through the array adding each element to the sum of all the previous elements and thus applies the increments tracked in `d` at their respective starting indices and cancels them after their respective ending indices.
7. The returned value from the `accumulate` function gives us the modified array `arr` after all updates have been applied, and this is returned as the final result.

This approach effectively reduces the time complexity of the problem, as we only need to make a constant-time update for each range increment, rather than incrementing all elements within the range for each update which could lead to a much higher time complexity.

Here's the mentioned code for the described solution approach that uses these steps:

```python
from itertools import accumulate

class Solution:
    def getModifiedArray(self, length: int, updates: List[List[int]]) -> List[int]:
        d = [0] * length
        for l, r, c in updates:
            d[l] += c
            if r + 1 < length:
                d[r + 1] -= c
        return list(accumulate(d))
```

In this code example, `accumulate` is an in-built Python function from the `itertools` module that computes the cumulative sum of the elements. This effectively does the last step of our prefix sum implementation for us.

Example Walkthrough

Let's illustrate the solution approach with a small example.

Consider an array of `length = 5` which is initially `[0, 0, 0, 0, 0]`. Suppose we have the following `updates = [[1, 3, 2], [2, 4, 3]]` which includes two update operations.

1. Initialize the difference array `d` which is the same size as our initial array: `d = [0, 0, 0, 0, 0]`
2. Apply the first update `[1, 3, 2]`:
   - Add the increment `2` to `d[startIdx]` which is `d[1]`, so now `d = [0, 2, 0, 0, 0]`.
   - Subtract the increment `2` from `d[endIdx + 1]` which is `d[4]`, but `d[4]` is out of bounds for the first update's end index, so `d` remains unchanged here.
3. Apply the second update `[2, 4, 3]`:
   - Add the increment `3` to `d[startIdx]` which is `d[2]`, now `d = [0, 2, 3, 0, 0]`.
   - Subtract the increment `3` from `d[endIdx + 1]` which is not applicable here as `endIdx + 1` equals `5` which is out of bounds, hence no subtraction is done.
4. With all updates applied, the difference array `d` is: `d = [0, 2, 3, 0, 0]`
5. Now use the `accumulate` function to calculate the prefix sum array from the difference array `d`: `Final array = list(accumulate(d)) = [0, 2, 5, 5, 5]`

The final array after applying all updates will be `[0, 2, 5, 5, 5]`.

This example confirms that the prefix sum technique updates the initial zero-filled array efficiently with the given range update operations.

Python Solution

```python
from itertools import accumulate  # Import the accumulate function from itertools

class Solution:
    def getModifiedArray(self, length: int, updates: List[List[int]]) -> List[int]:
        # Initialize the result array with zeros of given length
        result = [0] * length

        # Iterate through each update operation described by [start, end, increment]
        for start, end, increment in updates:
            # Apply the increment to the start index
            result[start] += increment

            # If the end index + 1 is within bounds, apply the negative increment
            # This is done to cancel the previous addition beyond the end index
            if end + 1 < length:
                result[end + 1] -= increment

        # Use accumulate to compute the running total, which applies the updates
        return list(accumulate(result))

# Example usage:
# sol = Solution()
# print(sol.getModifiedArray(5, [[1,3,2],[2,4,3],[0,2,-2]]))
```

Java Solution

```java
class Solution {
    // Method to compute the modified array after a sequence of updates
    public int[] getModifiedArray(int length, int[][] updates) {
        // Create an array 'difference' initialized to zero, with the given length
        int[] difference = new int[length];

        // Apply each update in the updates array
        for (int[] update : updates) {
            int startIndex = update[0]; // Start index for the update
            int endIndex = update[1];   // End index for the update
            int increment = update[2];  // Value to add to the subarray

            // Apply increment to the start index
            difference[startIndex] += increment;

            // If the end index is not the last element,
            // apply the negation of increment to the element after the end index
            if (endIndex + 1 < length) {
                difference[endIndex + 1] -= increment;
            }
        }

        // Convert the 'difference' array into the actual array 'result'
        // where each element is the cumulative sum from start to that index
        for (int i = 1; i < length; i++) {
            difference[i] += difference[i - 1];
        }

        // Return the resultant modified array
        return difference;
    }
}
```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to calculate the modified array based on intervals of updates
    std::vector<int> getModifiedArray(int length, std::vector<std::vector<int>>& updates) {
        // Initialize the difference array with zeros
        std::vector<int> diff_array(length, 0);

        // Iterate through each update operation represented by a triplet [startIdx, endIdx, inc]
        for (auto& update : updates) {
            int start_idx = update[0]; // Starting index for the update
            int end_idx = update[1];   // Ending index for the update
            int increment = update[2]; // Increment value to be added

            // Apply the increment to the start index in the difference array
            diff_array[start_idx] += increment;

            // Apply the negative increment to the position after the end index if in bounds
            // This marks the end of the increment segment
            if (end_idx + 1 < length) {
                diff_array[end_idx + 1] -= increment;
            }
        }

        // Iterate through the difference array to compute the final values
        // by adding the current value to the cumulative sum
        for (int i = 1; i < length; ++i) {
            diff_array[i] += diff_array[i - 1];
        }

        // Return the result - the final array after all updates have been applied
        return diff_array;
    }
};
```

Typescript Solution

```typescript
// Define TypeScript Function with specified input types and return type.
/**
 * Get the modified array after applying a series of updates.
 * @param {number} arrayLength - The length of the array to be modified.
 * @param {number[][]} updates - Array containing the updates to be applied.
 * @returns {number[]} - The modified array after all updates.
 */
function getModifiedArray(arrayLength: number, updates: number[][]): number[] {
    // Create an array filled with zeros of the specified length.
    const differenceArray = new Array<number>(arrayLength).fill(0);

    // Iterate over each update operation provided.
    for (const [startIdx, endIdx, increment] of updates) {
        // Apply the increment to the start index of the difference array.
        differenceArray[startIdx] += increment;

        // If the end index + 1 is within the bounds of the array, decrement the value.
        if (endIdx + 1 < arrayLength) {
            differenceArray[endIdx + 1] -= increment;
        }
    }

    // Iterate over the array, adding the previous element's value to each current element,
    // effectively applying the range updates.
    for (let i = 1; i < arrayLength; ++i) {
        differenceArray[i] += differenceArray[i - 1];
    }

    // Return the modified array with all updates applied.
    return differenceArray;
}
```

Time and Space Complexity

The given Python code utilizes the prefix sum (accumulation) strategy to compute the results of multiple range updates on an array. The analysis of the time and space complexity is as follows:

- **Time Complexity:**
  - The algorithm iterates over the `updates` list once. If there are `k` updates given, this part of the algorithm has a time complexity of `O(k)`.
  - Each update operation itself is constant time (i.e., `O(1)`), since it only involves updating two elements in the `d` array: the start index and the end index (or one past the end index).
  - After applying all updates, the algorithm uses `accumulate` from the `itertools` module to compute the prefix sums over the entire `d` array. Computing the prefix sum of an array of length `n` is an `O(n)` operation.

  Combining the two parts, the overall time complexity of the algorithm is `O(k + n)`.
- **Space Complexity:**
  - The space complexity of the algorithm is primarily determined by the `d` array, which holds the prefix sum and has a length equal to the input `length`. Therefore, it is `O(n)`, where `n` is the length of the result array.
  - The `updates` list does not count towards the extra space since it is part of the input.
  - All other operations use constant space, meaning they do not depend on the size of the input, hence do not significantly contribute to the space complexity.

  Therefore, the space complexity is `O(n)`.

In conclusion, the given algorithm has a time complexity of `O(k + n)` and a space complexity of `O(n)`.