2640. Find the Score of All Prefixes of an Array

Medium Array Prefix Sum

Leetcode Link

Problem Description

The problem provides us with an array `arr`. We are asked to calculate a "conversion" array based on this original array. The conversion array `conver` is defined such that each of its elements `conver[i]` is the sum of `arr[i]` and the maximum value in the subarray `arr[0..i]`, which includes all elements from the start of the array up to the current index `i`.

In addition to this, we need to find the "score" of each prefix of the original array `arr`. A prefix `arr[0..i]` includes all elements in the array from index `0` to index `i`. The score of a prefix is the sum of all the values in its conversion array. So for each prefix, we calculate its score and store it in an output array `ans`, where `ans[i]` equals the score of the prefix `arr[0..i]`.

To summarize, the problem is asking us to:

1. For each index `i` in the input array `nums`, calculate the conversion array `conver` as `conver[i] = nums[i] + max(nums[0..i])`.
2. Calculate the score for each prefix `nums[0..i]` as the sum of `conver[0..i]`.
3. Return an array `ans` where each `ans[i]` is the score of the prefix `nums[0..i]`.

Intuition

To approach this problem, we need to iterate through the input array `nums` to build both the conversion array `conver` and keep track of the sum for calculating the score simultaneously.

1. **Iterating Through the Array**: As we iterate through `nums`, we keep track of the maximum value found so far; this value represents `max(nums[0..i])` at each step.
2. **Computing the Conversion Values**: For each index `i`, we simply add `nums[i]` to this maximum value to get our current `conver[i]`.
3. **Accumulating the Scores**: To calculate the score of the prefix, we also need to accumulate the scores obtained from previous prefixes. This can be done by maintaining a running sum of the conversion values as we proceed. The score up to the current prefix is the sum of the current `conver[i]` and the score of the previous prefix (which we have been accumulating). For the first element in the array, since there is no previous prefix, the score is simply `nums[0] + max(nums[0])`, which is just `nums[0] * 2`.
4. **Handling the First Element**: We handle the first element as a special case, our code checks if `i` is `0` and, if so, it doesn't add the score of the previous prefix (since there is none) but simply the `nums[i] * 2`.

This process ensures that we only need to traverse the array once. We do not need to recalculate the maximum or the sum for each prefix from scratch.

The code implements these steps through the following logic:

- We initialize the maximum value `mx` as `0`, and we start with an answer array `ans` of zeros.
- We iterate through the input array `nums`, update `mx` to the maximum value so far, and calculate `ans[i]`.
- Finally, return the populated `ans` array as the result.

Solution Approach

The implementation uses a simple `for` loop to iterate over the elements of the input array `nums`, while maintaining two key pieces of information: the maximum value encountered so far (`mx`) and the accumulated score for each prefix (`ans[i]`). Here's a detailed walkthrough of the implementation:

1. **Initialization**: Before the loop starts, an array `ans` of zeros with the same length as `nums` is created to store the scores for each prefix. The variable `mx` is initialized to `0` to keep track of the maximum value seen at each step during the iteration.
2. **Enumeration**: The code uses `enumerate` to iterate over `nums`, giving us both the index `i` and the value `x` at each step. The `enumerate` function is a common pattern when both the element and its index are needed.
3. **Updating the Maximum Value**: As we traverse the array, the maximum value so far is updated by comparing the current value `x` with the last recorded maximum `mx` and assigning the greater of the two to `mx`.

   ```
   mx = max(mx, x)
   ```

4. **Calculation of Conversion Values and Accumulating Scores**: For each index `i`, the current conversion value is computed as `x + mx`. The score of the current prefix is the sum of this conversion value and the score of the previous prefix, which we have stored in `ans[i - 1]`. Since `ans` is initialized to have zeros, when `i` is `0`, `ans[i - 1]` does not exist and we handle this by adding `0` instead.

   ```
   ans[i] = x + mx + (0 if i == 0 else ans[i - 1])
   ```

5. **Data Structures**: The solution uses two main data structures: a list `ans` to hold the scores of the prefixes and a simple variable `mx` to keep track of the running maximum. No additional complex data structures are required to implement this solution effectively.
6. **Algorithm Complexity**: The time complexity of this implementation is O(n) where n is the length of the `nums` because it makes a single pass through the input array. The space complexity is also O(n) due to the space used by the `ans` array. This efficient use of time and space makes the solution optimal for this problem.

In conclusion, the solution leverages a single linear scan, a common algorithmic pattern used to minimize the time complexity by avoiding nested loops or recursive calls that could potentially increase the complexity. It efficiently computes the required maximal values and accumulated sums in one iteration by using constant space (apart from the output array) and in linear time.

Example Walkthrough

Let's walk through an example to illustrate the solution approach using the following input array `nums`:

```
nums = [1, 3, 5, 2]
```

1. **Initialization**: Before we begin, we create an array `ans` of zeros with the same length as `nums` to store the scores for each prefix. Also, we initialize the variable `mx` to `0`.

   ```
   ans = [0, 0, 0, 0]  // to hold the results
   mx = 0  // maximum value encountered so far
   ```

2. **Processing the First Element**: We start with the element at index `0` which is `1`.

   ```
   mx = max(mx, nums[0])  // mx becomes 1, since max(0, 1) is 1
   ans[0] = nums[0] + mx  // ans[0] becomes 1 + 1 = 2
   ```

   After processing the first element, our array `ans` and `mx` variable are updated as:

   ```
   ans = [2, 0, 0, 0]
   mx = 1
   ```

3. **Processing the Second Element**: Moving on to the element at index `1` which is `3`.

   ```
   mx = max(mx, nums[1])  // mx is updated to 3
   ans[1] = nums[1] + mx + ans[0]  // ans[1] is 3 + 3 + 2 = 8
   ```

   Now `ans` and `mx` are:

   ```
   ans = [2, 8, 0, 0]
   mx = 3
   ```

4. **Processing the Third Element**: For the element at index `2` which is `5`.

   ```
   mx = max(mx, nums[2])  // mx stays at 5
   ans[2] = nums[2] + mx + ans[1]  // ans[2] is 5 + 5 + 8 = 18
   ```

   The updated `ans` and `mx`:

   ```
   ans = [2, 8, 18, 0]
   mx = 5
   ```

5. **Processing the Last Element**: Finally, for the element at index `3` which is `2`.

   ```
   mx = max(mx, nums[3])  // mx stays at 5, since max(5, 2) is 5
   ans[3] = nums[3] + mx + ans[2]  // ans[3] is 2 + 5 + 18 = 25
   ```

   We have our final `ans` array:

   ```
   ans = [2, 8, 18, 25]
   ```

As a result, for each prefix of the array `nums`, we have calculated the corresponding scores and stored them in the `ans` array. The scores indicate the sum of the values in the conversion array for each prefix. Thus, the final output for the input `nums = [1, 3, 5, 2]` is `[2, 8, 18, 25]`. This walkthrough demonstrates how our approach effectively processes each element of the input array to compute the scores in a step-by-step manner, updating the maximum value encountered and accumulating the scores for each prefix.

Python Solution

```python
class Solution:
    def findPrefixScore(self, nums: List[int]) -> List[int]:
        # Get the number of elements in the input list 'nums'.
        num_elements = len(nums)

        # Initialize a result list 'prefix_scores' with zeros of the same length as 'nums'.
        prefix_scores = [0] * num_elements

        # Initialize a variable 'max_prefix' that will hold the maximum prefix score so far.
        max_prefix = 0

        # Iterate over each number in 'nums' using its index and value.
        for index, number in enumerate(nums):
            # Update 'max_prefix' with the maximum between 'max_prefix' and the current number.
            max_prefix = max(max_prefix, number)

            # Compute the current prefix score.
            # If it's the first element, then 'max_prefix' is the current number.
            # Otherwise, it's the sum of the maximum prefix so far, the current number,
            # and the previous prefix score.
            if index == 0:
                prefix_scores[index] = number + max_prefix
            else:
                prefix_scores[index] = number + max_prefix + prefix_scores[index - 1]

        # Return the resulting list of prefix scores.
        return prefix_scores
```

Java Solution

```java
class Solution {
    // Method to find the prefix score for each number in the array
    public long[] findPrefixScore(int[] nums) {
        // n holds the length of the input array
        int n = nums.length;

        // Initialize the answer array with the same length as the input array
        long[] answer = new long[n];

        // Initialize a variable to keep track of the maximum value seen so far
        int max = 0;

        // Iterate over the input array to compute the prefix scores
        for (int i = 0; i < n; ++i) {
            // Update max if the current number is greater than the current max
            max = Math.max(max, nums[i]);

            // Calculate the prefix score for the current index i
            // It is the sum of the current number, the maximum number seen so far,
            // and the previous prefix score (if not the first element)
            answer[i] = nums[i] + max + (i == 0 ? 0 : answer[i - 1]);
        }

        // Return the computed prefix scores
        return answer;
    }
}
```

C++ Solution

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

class Solution {
public:
    // Function to find the prefix score of a given array of integers
    vector<long long> findPrefixScore(vector<int>& nums) {
        int size = nums.size(); // Get the size of the input vector.
        vector<long long> prefixScore(size); // Initialize the result vector to store prefix scores.

        int maxElement = 0; // Variable to keep track of the maximum element in the prefix.

        // Iterate over the array to calculate prefix scores
        for (int i = 0; i < size; ++i) {
            maxElement = max(maxElement, nums[i]); // Update the maximum element in the prefix.

            // The prefix score of the current position is the sum of the current element,
            // the maximum element seen so far, and the previous prefix score (if not the first element).
            prefixScore[i] = nums[i] + maxElement + (i == 0 ? 0 : prefixScore[i - 1]);
        }

        return prefixScore; // Return the final computed prefix scores.
    }
};
```

Typescript Solution

```typescript
function findPrefixScore(nums: number[]): number[] {
    const numsLength = nums.length; // Length of the input array.
    const prefixScores: number[] = new Array(numsLength); // Array to hold the prefix scores.

    let maxNum = 0; // Variable to keep track of the maximum number encountered so far.

    // Iterate through the array of numbers.
    for (let index = 0; index < numsLength; ++index) {
        // Update the maximum number if the current number is greater.
        maxNum = Math.max(maxNum, nums[index]);

        // Calculate the prefix score:
        // It's the sum of the current number, the maximum number so far,
        // and the prefix score of the previous element (if it exists).
        prefixScores[index] = nums[index] + maxNum;
        if (index > 0) {
            prefixScores[index] += prefixScores[index - 1];
        }
    }

    // Return the array of calculated prefix scores.
    return prefixScores;
}
```

Time and Space Complexity

Time Complexity

The provided code calculates the prefix score for a list of integers. The time complexity can be broken down as follows:

- There is a single loop that iterates over the list `nums`, which contains `n` elements.
- Inside the loop, the code performs constant time operations, including assignment, the `max` function on two integers, and arithmetic operations.

As a result, the loop will have `n` iterations, and each iteration consists of constant time operations. This gives the time complexity as `O(n)`.

Space Complexity

Analyzing the space complexity:

- The list `ans` is initialized with the same length as `nums`, which adds `O(n)` space.
- The variables `s`, `mx`, and `i` use constant space (`O(1)`).
- No additional data structures or recursive calls that use space proportional to the size of the input.

Thus, the space complexity is `O(n)` primarily due to the space required for the `ans` list.