2945. Find Maximum Non-decreasing Array Length

Hard | Stack | Queue | Array | Binary Search | Dynamic Programming | Monotonic Queue | Monotonic Stack

Problem Description

The given problem presents an array `nums` of integers and permits operations where you select a subarray and replace it with its sum. The task is to return the maximum length of a non-decreasing array that you can achieve by performing any number of these operations. A non-decreasing array means each element is greater than or equal to the previous element.

Intuition

To solve this problem, the intuition is to use dynamic programming along with binary search. Dynamic programming will store intermediate solutions to subproblems, while binary search will help us quickly find positions in the array relevant to our calculations.

We'll keep track of a running sum of the array and construct a DP array `f` where `f[i]` represents the maximum length of the non-decreasing subarray ending at position `i`. We'll also maintain a `pre` array where `pre[i]` holds the highest index `j < i` such that `nums[j] <= 2 * nums[i]`.

While iterating over the array, we'll use binary search to find the index `j` where we can replace a subarray `[x...j]` that doubles the element `nums[x]`. We then set `pre[j]` to the current index `i` if it's not already set to a larger index. By doing this, we can efficiently extend the non-decreasing subarray as we move through `nums`.

The solution returns `f[n]`, which by the end of the algorithm, holds the maximum length of a non-decreasing array that can be created.

Solution Approach

The implementation of the solution uses dynamic programming along with a binary search to effectively tackle the problem. Here's a step-by-step breakdown of how the algorithm and its components – arrays `f` and `pre`, the sum array `s`, and binary search – work together in the solution:

1. Start by initializing an array `s` to keep the prefix sums of the `nums` array. This allows us to determine the sum of any subarray quickly.
2. Initialize two arrays – `f` which will hold the dynamic programming states representing the maximum length of the non-decreasing subarray, and `pre` which helps track the indices relevant for our subarray replacements.
3. Iterate through the array `nums` with `i` going from `1` to `n`, where `n` is the length of `nums`. For each `i`, we perform the following steps:
   - Set `pre[i]` to the maximum of `pre[i]` and `pre[i - 1]`, ensuring `pre[i]` always holds the largest index so far where subarray replacement can start.
   - Calculate `f[i]` as `f[pre[i]] + 1`. This essentially adds the length of the optimal subarray before the current index plus one for the current element.
   - Perform a binary search on the prefix sums array `s` to find the least index `j` where `s[j] >= s[i] * 2 - s[pre[i]]`. The sought `j` represents the potential end of a replacement subarray.
   - Update the `pre[j]` index to `i`, but only if `i` is greater than the current value of `pre[j]`. This prepares the `pre` array for the next elements to make decisions on whether they can extend the current non-decreasing subarray.
4. The dynamic programming array `f` accumulates the information about the maximum length of a non-decreasing array that can be achieved up to each index `i`.

After iterating over the whole array `nums`, `f[n]` will contain the length of the maximum non-decreasing subarray possible after performing the allowed subarray replacement operations. The final result is then simply `f[n]`.

The combination of prefix sums, binary search, and dynamic programming allows this algorithm to efficiently compute the maximum length of the non-decreasing subarray in an otherwise complex problem.

Example Walkthrough

Let us illustrate the solution approach with a small example:

Suppose our array `nums` is `[1, 2, 4, 1, 2, 4, 1]`.

1. We first compute the prefix sums array `s`. The prefix sum at index `i` is the sum of all `nums[j]` where `j <= i`:

```
nums: [1, 2, 4, 1, 2, 4, 1]
s:    [1, 3, 7, 8, 10, 14, 15] (each `s[i]` is the sum of all elements up to `i` in `nums`)
```

2. Initialize the `f` and `pre` arrays of the same length as `nums`, with initial values of `0` for `f` and indices of `-1` for `pre`.

```
f:   [0, 0, 0, 0, 0, 0, 0]
pre: [-1, -1, -1, -1, -1, -1, -1]
```

3. Start iterating through `nums`:
   - For `i = 1`, `pre[1]` is set to the maximum of `pre[0]` and `-1`, yielding `pre[1] = pre[0]`. For `i = 1`, the non-decreasing subarray can only include `nums[i]`, so `f[1] = 1`.
   - The same happens for `i = 2` as there's no need to replace any subarray yet.

```
nums: [1, 2, 4, 1, 2, 4, 1]
f:    [0, 1, 1, 0, 0, 0, 0]
pre:  [-1, -1, -1, -1, -1, -1, -1]
```

   - At `i = 3`, since `nums[3]` is smaller than `nums[2]`, we need to perform a replacement operation. We perform a binary search to find the least index `j` where `s[j]` is >= twice the sum before index `2`. We find that `j` is `2` itself. We set `pre[2]` to `3` and calculate `f[3]` which is `f[pre[3]] + 1 = f[-1] + 1 = 1`.

```
nums: [1, 2, 4, 1, 2, 4, 1]
f:    [0, 1, 1, 1, 0, 0, 0]
pre:  [-1, -1,  3, -1, -1, -1, -1]
```

   - For `i = 4` and `i = 5`, since the array is already non-decreasing, we just keep incrementing `f[i]` without needing any replacements.

```
nums: [1, 2, 4, 1, 2, 4, 1]
f:    [0, 1, 2, 1, 2, 3, 0]
pre:  [-1, -1,  3, -1, -1, -1, -1]
```

   - At `i = 6`, similar to `i = 3`, `nums[6]` is less than `nums[5]`, so we perform a replacement. We end up setting `pre[5]` to `6` and then `f[6]` becomes `f[pre[6]] + 1 = f[-1] + 1 = 1`.

```
nums: [1, 2, 4, 1, 2, 4, 1]
f:    [0, 1, 2, 1, 2, 3, 1]
pre:  [-1, -1,  3, -1, -1,  6, -1]
```

4. The final result is the largest value in `f`, which in this case is `3`. This indicates the maximum length of a non-decreasing subarray that we can achieve after performing these operations is `3`.

The combination of prefix sums, dynamic programming, and binary search allows us to efficiently find the maximum length of a non-decreasing subarray even in scenarios where multiple subarray replacement operations are required.

Solution Implementation

Python

```python
from itertools import accumulate
from bisect import bisect_left
from typing import List

class Solution:
    def findMaximumLength(self, nums: List[int]) -> int:
        # Calculate the length of nums list
        num_elements = len(nums)

        # Create a prefix sum list of numbers with an initial value 0 for easier index management
        prefix_sum = list(accumulate(nums, initial=0))

        # Initialize DP array f with length of num_elements + 1, to store the maximum lengths
        max_lengths = [0] * (num_elements + 1)

        # Initialize DP array pre to keep track of previous indices in the
        # prefix_sum at which new maximum lengths were calculated
        previous_indices = [0] * (num_elements + 2)

        # Iterate over the nums list to fill in the DP arrays
        for i in range(1, num_elements + 1):
            # Update the previous index with the maximum value between the current
            # and the previous maximum index.
            previous_indices[i] = max(previous_indices[i], previous_indices[i - 1])

            # Set the current maximum length to be either the same as previous max length, or 1 more
            # than the max length found at the previous maximum index.
            max_lengths[i] = max_lengths[previous_indices[i]] + 1

            # Find the next index in prefix_sum where a new segment can potentially be started
            next_index = bisect_left(prefix_sum, prefix_sum[i] * 2 - prefix_sum[previous_indices[i]])

            # Update the previous_indices list to indicate a new segment length has been found
            previous_indices[next_index] = i

        # Return the final maximum length from the last position of the max_lengths list
        return max_lengths[num_elements]
```

Java

```java
class Solution {
    public int findMaximumLength(int[] nums) {
        int length = nums.length;
        long[] prefixSum = new long[length + 1];
        // Building the prefix sum array
        for (int i = 0; i < length; ++i) {
            prefixSum[i + 1] = prefixSum[i] + nums[i];
        }

        // Initialize the array where f[i] stores the maximum length of the sequence ending at index i
        int[] maxLength = new int[length + 1];
        int[] maxIndexWithSum = new int[length + 2];
        // Iterate over the array to populate maxIndexWithSum and maxLength
        for (int i = 1; i <= length; ++i) {
            maxIndexWithSum[i] = Math.max(maxIndexWithSum[i], maxIndexWithSum[i - 1]);
            maxLength[i] = maxLength[maxIndexWithSum[i]] + 1;

            // Perform a binary search to find the index with desired sum
            int index = Arrays.binarySearch(prefixSum, prefixSum[i] * 2 - prefixSum[maxIndexWithSum[i]]);
            if (index < 0) {
                index = -index - 1; // convert to insertion point if element not found
            }
            // Update the maxIndexWithSum array with the current index
            maxIndexWithSum[index] = i;
        }

        // The last element in maxLength array holds the maximum length of the sequence for the whole array
        return maxLength[length];
    }
}
```

C++

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

class Solution {
public:
    int findMaximumLength(vector<int>& nums) {
        int n = nums.size(); // Number of elements in nums

        // f represents the maximum length ending at position i
        vector<int> max_length_ending_at(n + 1, 0);

        // pre records the furthest position that can be reached from the current position
        vector<int> furthest_reachable(n + 2, 0);

        // Prefix sums of nums, with s[0] initialized to 0 for easier calculations
        vector<long long> prefix_sum(n + 1, 0);
        for (int i = 0; i < n; ++i) {
            prefix_sum[i + 1] = prefix_sum[i] + nums[i];
        }

        // Iterate through positions of the array
        for (int i = 1; i <= n; ++i) {
            // Update the furthest reachable position for the current position i
            furthest_reachable[i] = std::max(furthest_reachable[i], furthest_reachable[i - 1]);

            // Update the maximum length at position i
            max_length_ending_at[i] = max_length_ending_at[furthest_reachable[i]] + 1;

            // Find the new furthest position that can be reached where the sum is doubled of segment up to i
            int new_position = std::lower_bound(prefix_sum.begin(), prefix_sum.end(),
                                                prefix_sum[i] * 2 - prefix_sum[furthest_reachable[i]]) - prefix_sum.begin();

            // Update the furthest reachable index for the new position
            furthest_reachable[new_position] = i;
        }

        // Return the maximum length at the last position which is the answer to the problem
        return max_length_ending_at[n];
    }
};
```

TypeScript

```typescript
// Function to find the maximum length of a subarray with equal number of 0s and 1s
function findMaximumLength(nums: number[]): number {
    const n = nums.length;
    // Initialize the array to store the furthest index with an equal number of 0s and 1s observed so far
    const furthestEqualSubarrayEnds: number[] = Array(n + 1).fill(0);
    // Initialize previous indices of the furthest index where a balanced subarray ends
    const previousIndices: number[] = Array(n + 2).fill(0);
    // Prefix sums of `nums`, s[i] is the sum of nums[0] to nums[i-1]
    const prefixSums: number[] = Array(n + 1).fill(0);

    // Compute Prefix Sums
    for (let i = 1; i <= n; ++i) {
        prefixSums[i] = prefixSums[i - 1] + nums[i - 1];
    }

    // Binary search utility method to find the leftmost position where `x` can be inserted
    // into an array `nums` without breaking the sorting
    const binarySearch = (nums: number[], x: number): number => {
        let [left, right] = [0, nums.length];
        // Perform binary search
        while (left < right) {
            // Find the mid index
            const mid = (left + right) >> 1;
            if (nums[mid] >= x) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    };

    // Iterate through the array and update the furthestEqualSubarrayEnds and previousIndices
    for (let i = 1; i <= n; ++i) {
        previousIndices[i] = Math.max(previousIndices[i], previousIndices[i - 1]);
        furthestEqualSubarrayEnds[i] = furthestEqualSubarrayEnds[previousIndices[i]] + 1;
        // Binary search for the position where a balanced subarray can potentially end
        const j = binarySearch(prefixSums, prefixSums[i] * 2 - prefixSums[previousIndices[i]]);
        previousIndices[j] = i;
    }

    // Return the last element in furthestEqualSubarrayEnds which gives the length of the longest subarray
    return furthestEqualSubarrayEnds[n];
}
```


```python
from itertools import accumulate
from bisect import bisect_left
from typing import List

class Solution:
    def findMaximumLength(self, nums: List[int]) -> int:
        # Calculate the length of nums list
        num_elements = len(nums)

        # Create a prefix sum list of numbers with an initial value 0 for easier index management
        prefix_sum = list(accumulate(nums, initial=0))

        # Initialize DP array f with length of num_elements + 1, to store the maximum lengths
        max_lengths = [0] * (num_elements + 1)

        # Initialize DP array pre to keep track of previous indices in the
        # prefix_sum at which new maximum lengths were calculated
        previous_indices = [0] * (num_elements + 2)

        # Iterate over the nums list to fill in the DP arrays
        for i in range(1, num_elements + 1):
            # Update the previous index with the maximum value between the current
            # and the previous maximum index.
            previous_indices[i] = max(previous_indices[i], previous_indices[i - 1])

            # Set the current maximum length to be either the same as previous max length, or 1 more
            # than the max length found at the previous maximum index.
            max_lengths[i] = max_lengths[previous_indices[i]] + 1

            # Find the next index in prefix_sum where a new segment can potentially be started
            next_index = bisect_left(prefix_sum, prefix_sum[i] * 2 - prefix_sum[previous_indices[i]])

            # Update the previous_indices list to indicate a new segment length has been found
            previous_indices[next_index] = i

        # Return the final maximum length from the last position of the max_lengths list
        return max_lengths[num_elements]
```


Time and Space Complexity

Time Complexity

The time complexity of the given code consists of several parts:

1. **Accumulating the sum**: The `accumulate` function is applied to the list `nums`, which takes `O(n)` time for a list of `n` elements.
2. **Iterative Calculation**:
   - There is a for loop that runs from `1` to `n`. Inside the loop, the `max` function is called, and access/update operations on arrays are performed. These operations are `O(1)`.
   - The `bisect_left` function is called within the for loop, which performs a binary search on the prefix sum array `s`. This takes `O(log n)` time. Since this is within the for loop, it contributes to `O(n log n)` time over the full loop.

Combining these, the overall time complexity is `O(n) + O(n log n)`. Since `O(n log n)` dominates `O(n)`, the final time complexity is `O(n log n)`.

Space Complexity

The space complexity of the code is due to the storage used by several arrays:

1. **Prefix Sum Array `s`**: Stores the prefix sums, requiring `O(n)` space.
2. **Array `f`**: An array of length `n+1`, also requiring `O(n)` space.
3. **Array `pre`**: Another array of length `n+2`, contributing `O(n)` space.

The total space complexity sums up to `O(n)` because adding the space requirements for `s`, `f`, and `pre` does not change the order of magnitude.