2599. Make the Prefix Sum Non-negative

Medium Greedy Array Heap (Priority Queue)

Leetcode Link

Problem Description

This problem presents an optimization challenge with an integer array `nums`. The main objective is to transform this array so that its prefix sum array contains no negative integers. To clarify, a prefix sum array is one where each element at index `i` is the total sum of all elements from the start of the array to that index, inclusive. The transformation of the array `nums` can only be achieved through a series of operations, each of which involves selecting one element from the array and moving it to the end.

The task is to figure out the minimum number of such operations needed to ensure that all the sums in the prefix sum array are non-negative. It is confirmed that there is always a way to rearrange the elements of the initial array in order to achieve this non-negative prefix sum property.

Intuition

The solution hinges on the greedy algorithm and a priority queue (which is implemented as a min heap in Python). The intuition behind the greedy approach is that by continually relocating the most negative value from the array to the end, we decrease the sum in the most effective way possible, which in turn quickly leads to a non-negative prefix sum at each step.

We start by traversing the array and calculating the prefix sums using a running sum variable `s`. When we come across a negative number, we put it into our priority queue (min heap). If at any point our running sum `s` becomes negative, it means our current prefix sum array contains negative values. In order to fix this, we repeatedly remove the smallest (most negative) number from the heap, since removing this number will give us the largest positive change towards making the sum non-negative. The number of times we need to remove an element from the heap to make the running sum non-negative at each step is added to our answer total, `ans`.

The loop structure ensures that we only pop from the heap when necessary, i.e., when the running sum is negative. This combination of a greedy approach with a priority queue allows us to efficiently manage and adjust the most negative elements which are contributing to a negative prefix sum. The counting of heap removals gives us the minimum number of operations required to prevent any negatives in the prefix sum array.

Solution Approach

The provided solution uses a greedy approach in conjunction with a priority queue (min heap) data structure from Python's `heapq` library, which enables efficient retrieval of the smallest element.

Following are step-by-step implementation details:

1. Initialize an empty min heap `h`, an accumulator `ans` to count the number of moves, and a sum `s` to keep track of the prefix sum.
2. Iterate over each element `x` in the array `nums`:
   - Add the current element to the prefix sum `s` (`s += x`).
   - If `x` is negative, push it onto the min heap `h` using `heappush(h, x)`.
3. After each addition of an element to `s`, check if `s` is negative:
   - While `s` is negative, repeatedly:
     - Pop the smallest (most negative) element from the min heap `h` using `heappop(h)`.
     - Subtract the popped element from `s`, which will make `s` less negative or non-negative.
     - Increment `ans` by 1 representing an operation.
4. Once all elements have been processed, return `ans`, which now contains the count of the minimum number of operations needed to ensure the prefix sum array is non-negative.

The algorithm efficiently rearranges the elements by virtually moving the most negative values to the end of the array, thus not needing to manipulate the array directly. Instead, we operate on the prefix sum and extract the negative impact whenever it threatens to pull the sum below zero.

As we traverse `nums`, we accumulate negative values into the min heap. Popping from the min heap gives us the smallest negative number quickly, which is the ideal candidate to "move" to the end (in a virtual sense) because its removal will have the biggest immediate positive (or least negative) impact on `s`.

The solution approach takes advantage of the properties of a min heap, where ensuring the heap structure after each insertion or deletion operation (i.e., `heappush` and `heappop`) takes O(log n) time, thus each operation on the heap is efficient even as `nums` grows in size.

In summary, the solution applies algorithms and data structures (greedy technique and priority queue) to cleverly and efficiently find the minimum set of adjustments to `nums` that guarantees a non-negative prefix sum array.

Example Walkthrough

Consider the following small example array `nums`: [3, -7, 4, -2, 1, 2]

1. We initialize the following:
   - Min heap `h`: []
   - Operation counter `ans`: 0
   - Prefix sum `s`: 0
2. Process the array `nums`:
   - For the first element (3):
     - `s` becomes 3 (0 + 3). `s` is non-negative, so no need for changes.
     - Min heap `h` remains empty.
   - For the second element (-7):
     - `s` becomes -4 (3 - 7). `s` is negative.
     - We push -7 onto the min heap `h`: [-7]
     - Since `s` is negative, we pop from `h` (-7), add it back to `s` (+7), and increment `ans` by 1.
     - `s` is now 3 (-4 + 7), and `ans` is 1.
     - Min heap `h` is now empty.
   - For the third element (4):
     - `s` becomes 7 (3 + 4). `s` is non-negative.
     - Min heap `h` remains empty.
   - For the fourth element (-2):
     - `s` becomes 5 (7 - 2). `s` is non-negative.
     - We push -2 onto the min heap `h`: [-2]
   - For the fifth element (1):
     - `s` becomes 6 (5 + 1). `s` is non-negative.
     - Min heap `h` remains as: [-2]
   - For the sixth element (2):
     - `s` becomes 8 (6 + 2). `s` is non-negative.
     - Min heap `h` remains as: [-2]
3. Since we've finished processing `nums` and `s` is non-negative, `ans` remains at 1.
4. Thus, the minimum number of operations needed to ensure the prefix sum array is non-negative for `nums` is 1.

In this example, only one operation was needed, which involved moving the `-7` to the end of the array to ensure all prefix sums were non-negative. This step is conceptual as we actually just remove the negative influence of `-7` from the running sum `s`.

Python Solution

```python
from heapq import heappush, heappop
from typing import List

class Solution:
    def makePrefSumNonNegative(self, nums: List[int]) -> int:
        # A min-heap to store the negative numbers encountered
        min_heap = []

        # The variable 'total_operations' represents the number of operations performed to make the prefix sums non-negative
        total_operations = 0

        # The variable 'current_sum' is used to store the running sum of numbers from the array
        current_sum = 0

        # Iterate through each number in the list
        for number in nums:
            # Update the running sum
            current_sum += number

            # If the number is negative, add it to the min-heap
            if number < 0:
                heappush(min_heap, number)

            # If current_sum drops below zero, we need to perform operations to make it non-negative
            while current_sum < 0:
                # The operation involves removing the smallest element (top of the min-heap) from the running sum
                current_sum -= heappop(min_heap)

                # Increment the count of operations needed
                total_operations += 1

        # Return the total number of operations performed to ensure all prefix sums are non-negative
        return total_operations
```

Java Solution

```java
class Solution {
    public int makePrefSumNonNegative(int[] nums) {
        // A priority queue to store negative numbers, it will heapify them based on their natural order, i.e., the smallest number will be at the top
        PriorityQueue<Integer> negativeNumbers = new PriorityQueue<>();

        // 'operations' will hold the count of the number of negative numbers removed from the prefix sum to make it non-negative.
        int operations = 0;

        // 'sum' is used to store the running prefix sum of the array.
        long sum = 0;

        // Iterate over each number in the array.
        for (int number : nums) {
            // Add the current number to the prefix sum.
            sum += number;

            // If the number is negative, add it to the priority queue.
            if (number < 0) {
                negativeNumbers.offer(number);
            }

            // If the prefix sum is negative, we need to perform operations.
            while (sum < 0) {
                // Remove the smallest negative number from the prefix sum to try and make it non-negative.
                sum -= negativeNumbers.poll();

                // Increment the count of operations.
                ++operations;
            }
        }

        // Return the total number of operations performed to make the prefix sum non-negative.
        return operations;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int makePrefSumNonNegative(vector<int>& nums) {
        // Initialize a min-heap to keep track of negative numbers
        priority_queue<int, vector<int>, greater<int>> minHeap;

        int operations = 0; // Count the number of operations needed
        long long prefixSum = 0; // This will store the prefix sum of the array

        // Iterate through the vector of numbers
        for (int& num : nums) {
            prefixSum += num; // Add current number to the prefix sum

            // If the number is negative, add it to the min-heap
            if (num < 0) {
                minHeap.push(num);
            }

            // If the prefix sum is negative, we need to make operations
            while (prefixSum < 0) {
                // Remove the smallest negative number from prefix sum and from the min-heap
                prefixSum -= minHeap.top();
                minHeap.pop();

                // Increment the operation count as we removed one element
                ++operations;
            }
        }

        // Return the total number of operations performed
        return operations;
    }
};
```

Typescript Solution

```typescript
import { MinPriorityQueue } from 'datastructures-js/priority-queue'; // Make sure to import the priority queue data structure

// This function adjusts the input array `nums` in such a way that the prefix sum never goes negative.
// If necessary, it removes the smallest elements until the sum is non-negative.
// It returns the number of elements removed to achieve this.
function makePrefSumNonNegative(nums: number[]): number {
    const priorityQueue = new MinPriorityQueue<number>(); // Initialize a minimum priority queue for numbers
    let removals = 0; // Counter for the number of removed elements
    let sum = 0; // Sum of elements encountered so far

    // Iterate through each element in the `nums` array
    for (const num of nums) {
        sum += num; // Add current element to the sum
        // If current element is negative, add it to the priority queue
        if (num < 0) {
            priorityQueue.enqueue(num);
        }
        // While the sum is negative, remove the smallest element from the priority queue to increase the sum
        while (sum < 0) {
            sum -= priorityQueue.dequeue().element; // Subtract the smallest element from sum
            removals++; // Increment the removal counter
        }
    }

    return removals; // Return the total number of elements removed
}
```

Time and Space Complexity

The time complexity of the given code is `O(n * log n)`. This is because the function iterates through all `n` elements of the input list `nums`. For each negative number encountered, it performs a `heappush` operation which is `O(log k)`, where `k` is the number of negative numbers encountered so far, leading to a maximum of `O(log n)` when all elements are negative. Additionally, when the sum `s` becomes negative, the code performs a `heappop` in a while loop until `s` is non-negative again. In the worst case, this could involve popping every number that was pushed, resulting in a sequence of `heappop` operations. Since each `heappop` operation is `O(log k)`, and you could theoretically pop every element once, the total time for all the heap operations across the entire list will be `O(n * log n)`.

The space complexity of the given code is `O(n)`. This is due to the additional heap `h` that is used to store the negative numbers. In the worst-case scenario, all elements in the list are negative and will be added to the heap. Since the heap can contain all negative numbers of the list at once, the space required for the heap is proportional to the input size, hence the space complexity is `O(n)`.