2587. Rearrange Array to Maximize Prefix Score

Medium Greedy Array Prefix Sum Sorting

Problem Description

In this problem, you are provided with an integer array `nums`. Your goal is to rearrange the elements of this array in any order. By doing this, you want to maximize the "score" of the array. The "score" is defined as the number of positive integers in the array of prefix sums.

The prefix sums array, `prefix`, is created by summing elements from the start up to the current position after you have rearranged the array. `prefix[i]` represents the sum of elements from index `0` up to index `i` in the newly arranged array. Your task is to rearrange `nums` such that the count of positive numbers in the `prefix` array is the highest possible.

For example, if your input array `nums` is `[1, -2, 3]`, you could rearrange it to `[3, 1, -2]`. Then, your `prefix` array would be `[3, 4, 2]`, and the score of `nums` would be `3` since there are three positive numbers in the `prefix` array.

Intuition

The key intuition behind obtaining the maximum score is related to the arrangement of the numbers in descending order. By sorting the array in reverse order, we ensure that we add the largest numbers first. This has the benefit of potentially offsetting any negative numbers that come later in the sequence.

Consider an array with both positive and negative numbers. If we start by adding the largest positive numbers, the prefix sum is more likely to stay positive for a longer stretch, even if we encounter negative numbers. On the other hand, if we were to add negative numbers early on, they would decrease the overall sum, and thus there's a higher chance for the sum to drop to zero or become negative, which would reduce our score.

Thus, the approach used in the solution involves sorting `nums` in descending order and then calculating the prefix sums iteratively. With each sum, we check if the current sum `s` is less than or equal to zero. If it is, we return the index `i`, which represents the number of positive integers in the `prefix` array until that point. Otherwise, if we go through the entire list without the sum becoming non-positive, we return the length of `nums`, as all prefix sums are positive.

Solution Approach

The solution to this problem uses a greedy algorithm, which is reflected by the choice to sort the input array `nums` in reverse (i.e., descending) order. The `sort()` function in Python is used for this purpose, which typically utilizes a Timsort algorithm, a hybrid sorting algorithm derived from merge sort and insertion sort, to rearrange the elements.

Once sorted, the solution iteratively accumulates the sum `s` of the `nums` elements, using a for loop. The accumulated sum `s` represents the current prefix sum after each iteration. The data structure used here is simple, relying on integer variables to track the sum and the index.

The pseudocode pattern for the solution is as follows:

1. Sort the input array `nums` in reverse order.
2. Initialize a variable `s` to 0 to keep track of the prefix sums.
3. Iterate through the sorted array using a for loop with index `i` and value `x`: a. Increment `s` by `x`. b. If `s` becomes less than or equal to 0, return `i` as the score, because it represents the count of positive numbers in the prefix array up to this point.
4. If the loop completes without `s` becoming non-positive, return the total length of the array, because this means all prefix sums were positive.

By following these steps, we ensure that the `prefix` array starts with the largest positive numbers, therefore maximizing the possibility that the sum at every index remains positive. If a negative or zero prefix sum is encountered, the algorithm stops counting, since further sums cannot contribute positively to the score. If negative values occur after positive ones in the sorted array, their impact is minimized by the prefix sums accumulated thus far.

In Python, the implementation looks like this:

```
class Solution:
    def maxScore(self, nums: List[int]) -> int:
        nums.sort(reverse=True) # Step 1: Sort the array in reverse order.
        s = 0                   # Step 2: Initialize the [prefix sum](/problems/subarray_sum) as 0.
        for i, x in enumerate(nums): # Step 3: Iterate through the array.
            s += x              # Step 3a: Add the current element to the sum.
            if s <= 0:          # Step 3b: If the sum is non-positive, return the current score.
                return i
        return len(nums)        # Step 4: If all prefix sums are positive, return the length of the array.
```

This implementation is efficient, as it only requires a single pass through the array after sorting, which results in an overall time complexity of O(n log n) due to the sort operation, where `n` is the number of elements in the input array. The space complexity is O(1), as no additional data structures are needed beyond the input array and temporary variables.

Example Walkthrough

Let's apply the solution approach to a small example. Consider the integer array `nums` as `[2, -1, 3, -4, 1]`.

1. The first step is to sort the array in reverse order. After sorting we get `[3, 2, 1, -1, -4]`.
2. We initialize a variable `s` to track the prefix sums, starting with `s = 0`.
3. We start iterating through the sorted array and updating `s` with the current element:
   - In the first iteration `i = 0`, `x = 3`. The new value of `s` is `0 + 3 = 3` which is positive.
   - Next, `i = 1`, `x = 2`. Now `s` becomes `3 + 2 = 5`, still positive.
   - For `i = 2`, `x = 1`. We update `s` to `5 + 1 = 6`, again positive.
   - Then, `i = 3`, `x = -1`. The sum `s` will be updated to `6 - 1 = 5`, which is positive.
   - Lastly, `i = 4`, `x = -4`. Updating `s` gives us `5 - 4 = 1`, which remains positive.
4. Since we don't encounter a sum that is zero or negative, we reach the end of the loop with all positive prefix sums. Therefore, we return the length of the array `nums`, which is `5`.

So, the final score for the array `[2, -1, 3, -4, 1]` when sorted in descending order would be `5` because all prefix sums of the sorted array `[3, 2, 1, -1, -4]` are positive.

Applying this algorithm to the example provided shows the step-by-step process and confirms the efficiency of the solution, illustrating that all positive prefix sums were obtained by arranging the numbers in descending order.

Solution Implementation

Python

```
from typing import List

class Solution:
    def maxScore(self, card_points: List[int]) -> int:
        # Sort the list of card points in decreasing order
        card_points.sort(reverse=True)

        # Initialize the sum of selected card points to zero
        total_score = 0

        # Iterate over the sorted list of card points
        for index, points in enumerate(card_points):
            # Add the points of the current card to the total score
            total_score += points

            # If at any point the total score becomes non-positive,
            # return the current number of cards used (index of iteration)
            if total_score <= 0:
                return index

        # If all cards contribute positively to the score (the total score never becomes non-positive),
        # return the total number of cards as the result
        return len(card_points)

# Example of usage:
# solution = Solution()
# print(solution.maxScore([1, -2, -3, 4, 5])) # Output: 3
```

Java

```
import java.util.Arrays; // Import Arrays class for sorting

class Solution {
    // Method to calculate the maximum score that can be obtained
    // from the given array nums
    public int maxScore(int[] nums) {
        // Sort the array in ascending order
        Arrays.sort(nums);

        // n stores the total length of the array
        int n = nums.length;

        // Variable to keep track of the cumulative score
        long cumulativeScore = 0;

        // Iterate over the array from the last element to the first
        for (int i = 0; i < n; ++i) {
            // Add the value of current largest element to cumulativeScore
            cumulativeScore += nums[n - i - 1];

            // If cumulative score is less than or equal to zero,
            // the maximum score that can be obtained is the current index i.
            if (cumulativeScore <= 0) {
                return i;
            }
        }
        // If the loop completes, then all elements contributed positively,
        // hence return the total length of the array
        return n;
    }
}
```

C++

```
#include <vector>
#include <algorithm> // include necessary headers

class Solution {
public:
    int maxScore(vector<int>& numbers) {
        // Sorting the array in non-increasing order
        sort(numbers.rbegin(), numbers.rend());

        long long sum = 0; // Using long long for potential big sum
        int count = 0; // This will hold the maximum number of elements contributing to a positive sum
        int n = numbers.size(); // Get the total count of elements in the numbers vector

        // Iterate over the sorted numbers
        for (int i = 0; i < n; ++i) {
            sum += numbers[i]; // Add up the numbers starting from the largest

            // If the sum goes to zero or negative, return the current count
            if (sum <= 0) {
                return count;
            }

            count++; // Increment count as the current number contributed to a positive sum
        }

        // If the loop finishes, all elements contribute to a positive sum
        return count; // Return the total number of elements
    }
};
```

TypeScript

```
function maxScore(nums: number[]): number {
    // Sorts the array of numbers in ascending order.
    nums.sort((a, b) => a - b);

    // Stores the length of the array for convenience.
    const lengthOfNums = nums.length;

    // 'totalScore' will keep track of the aggregated score.
    let totalScore = 0;

    // Loop through each number in the array from the end towards the beginning.
    for (let i = 0; i < lengthOfNums; ++i) {
        // Accumulate the total score by adding the value of the current largest number.
        totalScore += nums[lengthOfNums - i - 1];

        // If the score becomes non-positive, return the current index as the result.
        if (totalScore <= 0) {
            return i;
        }
    }

    // If the loop completes without returning, it means all scores were positive,
    // so return the total count of numbers.
    return lengthOfNums;
}
```

```
from typing import List

class Solution:
    def maxScore(self, card_points: List[int]) -> int:
        # Sort the list of card points in decreasing order
        card_points.sort(reverse=True)

        # Initialize the sum of selected card points to zero
        total_score = 0

        # Iterate over the sorted list of card points
        for index, points in enumerate(card_points):
            # Add the points of the current card to the total score
            total_score += points

            # If at any point the total score becomes non-positive,
            # return the current number of cards used (index of iteration)
            if total_score <= 0:
                return index

        # If all cards contribute positively to the score (the total score never becomes non-positive),
        # return the total number of cards as the result
        return len(card_points)

# Example of usage:
# solution = Solution()
# print(solution.maxScore([1, -2, -3, 4, 5])) # Output: 3
```

Time and Space Complexity

Time Complexity

The time complexity of the code mainly comes from two parts:

1. Sorting the `nums` list, and
2. Iterating through the `nums` list once.

The sorting can be done in `O(n log n)` time where `n` is the length of the list `nums`. Python uses Timsort for sorting which has this time complexity for the worst case.

After sorting, the code iterates through the list once, which is an `O(n)` operation.

Combining these parts, the overall time complexity is `O(n log n + n)`, which simplifies to `O(n log n)` as the log-linear term dominates for large `n`.

Space Complexity

The space complexity of this code is `O(1)`, or constant space, not counting the input and output. This is because the sorting is done in-place (no extra space is used apart from temporary variables), and the iteration does not use additional space proportional to the input size (only a fixed number of variables `s` and `i` are used).