1403. Minimum Subsequence in Non-Increasing Order

Easy Greedy Array Sorting

Problem Description

The task is to find a subsequence from a given array `nums` such that the sum of this subsequence's elements is strictly greater than the sum of the elements that are not included in the subsequence. A subsequence is defined as a sequence that can be derived from the array by removing some or no elements without changing the order of the remaining elements.

The constraints for the solution are:

- If there are multiple subsequences that satisfy the condition, the one with the smallest number of elements should be returned.
- If there is still more than one subsequence with the minimum number of elements, the subsequence among them with the maximum total sum should be returned.
- The returned subsequence should be sorted in non-increasing order (from the largest to the smallest element).
- It is guaranteed that a unique solution exists for the given constraints.

Intuition

To arrive at the solution, we should understand the following points:

- A subsequence with the largest possible elements will contribute more towards making its sum greater than the rest of the elements.
- Sorting the array in non-increasing order helps in picking out the subsequence with the maximum possible sum in the smallest size.
- Accumulating the items from the sorted array until the running total sum (`t`) of these items is greater than the sum of the remaining items (`s - t`) gives a subsequence that fulfills the condition.

Now, considering these points, the algorithm follows these steps:

1. Sort the array in non-increasing order.
2. Iterate over the sorted array, adding each element to a running total (`t`) and storing the element in a subsequence result (`ans`).
3. After each addition, check if the sum of elements in this subsequence result (`t`) is strictly greater than the sum of the remaining elements in the original array (`s - t`).
4. Continue accumulating elements until the condition is met.

Once the condition is met, the current subsequence result (`ans`) contains the minimum number of elements with the maximum sum, sorted in non-increasing order. This is the desired subsequence to be returned.

Solution Approach

The solution approach takes advantage of the sorting algorithm and a greedy strategy to create the desired subsequence. Here's a walkthrough of the implementation based on the reference solution provided:

1. First, we need to know the sum of all elements in the given array `nums`. This is calculated with the expression `sum(nums)` and stored in the variable `s`.
2. We then define a variable `t` to keep track of the total sum of the selected subsequence as we build it. Initially, `t` is set to 0 since we haven't selected any elements yet.
3. The input array `nums` is then sorted in non-increasing order using `sorted(nums, reverse=True)`. Sorting the array allows us to consider larger elements first, aligning with the greedy approach of maximizing the sum of the subsequence.
4. We initialize an empty list `ans` to store the elements of the subsequence.
5. Next, we iterate over the sorted array. For each element `x` in the array:
   - We add the element `x` to the total `t` and also append `x` to our answer list `ans`.
   - After each addition, we check if the sum `t` is strictly greater than the sum of the remaining elements in the original array (`s - t`).
   - If this condition is true, it means the selected elements in `ans` now have a sum greater than that of the remaining elements of `nums`, fulfilling our requirement.
6. The iteration is terminated immediately once our condition is met (i.e., when `t > s - t`). Since we are iterating over the array in non-increasing order, the first time this condition is met, we have the smallest subsequence with the greatest total sum.
7. Lastly, the subsequence stored in `ans` is returned. This is the subsequence with the minimum size and maximum total sum, sorted in non-increasing order – as dictated by our sorting of `nums`.

By utilizing a greedy algorithmic approach along with elementary data structures like lists and performing array sorting, we have an efficient implementation that ensures the strict constraints of the problem are met and a unique solution is returned.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Consider the array `nums = [4, 3, 1, 2]`. We need to find a subsequence such that its sum is greater than the sum of the elements not included in it.

1. Calculate the sum of all elements in `nums`: `s = 4 + 3 + 1 + 2 = 10`.
2. Initialize `t` to 0, which will represent the total sum of our subsequence.
3. Sort the array in non-increasing order: `sorted_nums = [4, 3, 2, 1]`.
4. Initialize an empty list `ans` to store our subsequence.
5. Start iterating over the sorted array. We pick elements one by one and add to both `t` and `ans`.
   - First element `x = 4`: `t = 0 + 4 = 4`, `ans = [4]`, and remaining sum `s - t = 6`. The sum of `ans` is not yet greater than the remaining sum.
   - Second element `x = 3`: `t = 4 + 3 = 7`, `ans = [4, 3]`, and remaining sum `s - t = 3`. Now, the sum of `ans` is greater than the remaining sum (`7 > 3`).
6. The condition `t > s - t` is met, so we stop iterating.
7. The resulting subsequence `ans = [4, 3]` is already sorted in non-increasing order, which is also the subsequence with the smallest number of elements and the maximum total sum out of all valid subsequences.

We return `ans = [4, 3]` as our final answer. This subsequence has a sum of `7`, which is greater than the sum of the non-included elements `(2 + 1 = 3)`. It's the smallest such subsequence with the largest sum, fulfilling all the problem's requirements.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def minSubsequence(self, nums: List[int]) -> List[int]:
        # Initialize the variable to store the resulting subsequence
        result_subsequence = []

        # Calculate the total sum of the elements in nums
        total_sum = sum(nums)

        # Initialize the variable to track the running total of the subsequence
        subsequence_sum = 0

        # Sort the nums array in decreasing order
        for num in sorted(nums, reverse=True):
            # Add the current number to the running total
            subsequence_sum += num

            # Append the current number to the result subsequence
            result_subsequence.append(num)

            # Check if the sum of the subsequence is greater than the remaining elements
            if subsequence_sum > total_sum - subsequence_sum:
                # If true, break out of the loop
                break

        # Return the result subsequence
        return result_subsequence
```

Java

```java
import java.util.Arrays;  // Import Arrays utility for sorting and streaming
import java.util.List;    // Import List interface
import java.util.ArrayList;  // Import ArrayList class for dynamic arrays

class Solution {
    public List<Integer> minSubsequence(int[] nums) {
        // Sort the array in non-decreasing order
        Arrays.sort(nums);

        // Prepare a list to store the result
        List<Integer> result = new ArrayList<>();

        // Calculate the sum of all elements in the array
        int totalSum = Arrays.stream(nums).sum();

        // Initialize a variable to keep track of the running sum of the subsequence
        int subsequenceSum = 0;

        // Iterate over the array from the last element to the first
        for (int i = nums.length - 1; i >= 0; i--) {
            // Add the current element to the subsequence sum
            subsequenceSum += nums[i];
            // Add the current element to the result list
            result.add(nums[i]);
            // Check if the subsequence sum is greater than the remaining elements' sum
            if (subsequenceSum > totalSum - subsequenceSum) {
                // Break the loop if the subsequence sum is greater as we have the minimum subsequence
                break;
            }
        }

        // Return the result list which now contains the minimum subsequence
        return result;
    }
}
```

C++

```cpp
#include <vector>
#include <numeric> // For accumulate
#include <algorithm> // For sort

class Solution {
public:
    // Function to find the smallest subsequence with a sum greater than the sum of the remaining elements.
    vector<int> minSubsequence(vector<int>& nums) {
        // Sort the input vector in non-increasing order.
        sort(nums.rbegin(), nums.rend());

        // Calculate the total sum of all elements in the vector.
        int totalSum = accumulate(nums.begin(), nums.end(), 0);
        // This will keep track of the sum of the current subsequence.
        int subsequenceSum = 0;
        // This vector will store the elements of the smallest subsequence.
        vector<int> answer;

        // Iterate over the sorted array to accumulate a sum greater than the half of total sum.
        for (int num : nums) { // Using num instead of x for clarity.
            subsequenceSum += num; // Add the current element to the subsequence sum.
            answer.push_back(num); // Add the current element to the subsequence.
            // Check if the current subsequence sum is greater than the sum of the remaining elements.
            if (subsequenceSum > totalSum - subsequenceSum) {
                break; // Terminate the loop if the condition is true.
            }
        }

        // Return the resulting subsequence.
        return answer;
    }
};
```

TypeScript

```typescript
function minSubsequence(nums: number[]): number[] {
    // Sort the array in non-increasing order (largest to smallest)
    nums.sort((a, b) => b - a);

    // Calculate the sum of all elements in the array
    const totalSum = nums.reduce((sum, current) => sum + current, 0);

    let subsequenceSum = 0;

    // Iterate through the sorted array
    for (let i = 0; i < nums.length; i++) {
        // Add the current element to the subsequence sum
        subsequenceSum += nums[i];

        // Check if the sum of the elements of the subsequence is greater than
        // the sum of the remaining elements in the array
        if (subsequenceSum > totalSum - subsequenceSum) {
            // If the condition is met, return the subsequence from the start
            // up to the current element
            return nums.slice(0, i + 1);
        }
    }

    // In case no subsequence is found that satisfies the condition
    // Though this case won't occur as per the problem's constraints,
    // we need to return something to satisfy TypeScript's return type.
    // An empty subsequence signifies failure to find the desired subset.
    return [];
}
```


```python
from typing import List

class Solution:
    def minSubsequence(self, nums: List[int]) -> List[int]:
        # Initialize the variable to store the resulting subsequence
        result_subsequence = []

        # Calculate the total sum of the elements in nums
        total_sum = sum(nums)

        # Initialize the variable to track the running total of the subsequence
        subsequence_sum = 0

        # Sort the nums array in decreasing order
        for num in sorted(nums, reverse=True):
            # Add the current number to the running total
            subsequence_sum += num

            # Append the current number to the result subsequence
            result_subsequence.append(num)

            # Check if the sum of the subsequence is greater than the remaining elements
            if subsequence_sum > total_sum - subsequence_sum:
                # If true, break out of the loop
                break

        # Return the result subsequence
        return result_subsequence
```


Time and Space Complexity

Time Complexity

The time complexity of the given code is primarily determined by the sorting operation. Here, `sorted(nums, reverse=True)` sorts the input list `nums` in descending order. The sorting algorithm used in Python's `sorted` function is Timsort, which has a time complexity of `O(n log n)` for the average and worst case, where `n` is the number of elements in the input list.

After sorting, the code iterates through the sorted list once. This single pass through the list has a time complexity of `O(n)`.

Therefore, the overall time complexity of the code is dominated by the sorting operation and is `O(n log n)`.

Space Complexity

The space complexity of the code involves the space needed for the output list `ans` and the temporary space used by the sorting operation.

The output list `ans` grows proportionally with the number of elements it includes, which in the worst case, can be all the elements of the input list `nums`. Hence, the space complexity for `ans` is `O(n)`.

The sorting operation may require additional space for its internal workings (temp arrays for mergesort or hybrid sorts like Timsort). In Python, the `sorted` function does not sort in-place; instead, it returns a new sorted list. However, since this sorted list is not stored in an additional variable and is used in the same loop for iteration, the dominant additional space is the same as for `ans`, which has already been accounted for as `O(n)`.

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