2971. Find Polygon With the Largest Perimeter

Medium Topics Array Prefix Sum Sorting

Problem Description

In this problem, we're given an array `nums` consisting of positive integers which represent potential side lengths of a polygon. A polygon is a closed plane figure with at least three sides, and a crucial property of a valid polygon is that its longest side must be shorter than the sum of its other sides. This problem asks us to determine the largest possible perimeter of a polygon that can be formed using the lengths provided in the `nums` array. If it's not possible to form a polygon from the given lengths, we must return `-1`.

The perimeter of a polygon is simply the sum of the lengths of all its sides. The goal is to select three or more lengths from `nums` that meet the polygon inequality (sum of smaller sides greater than the longest side) and maximize their sum.

Intuition

To solve this problem, we start by sorting the `nums` array in non-decreasing order. This makes it easier to check the polygon inequality: for any three consecutive lengths in this sorted array, if the sum of the first two is greater than the third, they can form a polygon. Once the array is sorted, we want to check every possible set of three consecutive lengths starting from the longest set and going down.

To achieve this efficiently, we accumulate the sums of `nums` elements by using the `accumulate` function from the `itertools` module, storing prefix sums in array `s`. E.g., `s[k]` is the sum from `nums[0]` to `nums[k-1]`.

We start our check from a polygon with `k` sides. If the sum of the first `k-1` sides (`s[k - 1]`) is greater than the `k`-th side (`nums[k - 1]`), this means there can be a polygon with `s[k]` as its perimeter. If no such `k` exists, it means we cannot form a polygon, and thus, we return `-1`. The answer keeps track of the max perimeter found while satisfying the condition.

Solution Approach

The solution provided applies a greedy approach combined with sort and prefix sum techniques to solve the problem efficiently.

Here is a step-by-step breakdown of the solution implementation.

1. **Sorting**: To facilitate the inequality check for forming a polygon, the first step is to sort the array of possible side lengths `nums` in non-decreasing order. This helps to easily identify the potential largest side for any choice of sides and apply the polygon inequality, which is fundamental for the solution.

   ```
   nums.sort()
   ```

2. **Prefix Sums**: We utilize the `accumulate` function from Python's `itertools` module to compute the prefix sum of the sorted array `nums`. Prefix sums are helpful to quickly calculate the sum of elements that could form the sides of a potential polygon without repeatedly adding elements in every iteration.

   ```
   s = list(accumulate(nums, initial=0))
   ```

   Note that `initial=0` is provided so that the accumulation array `s` starts with a `0`, syncing the index of `s` with the corresponding sum in `nums`.

3. **Iterating and Checking for a Polygon**: The algorithm iterates over the sorted `nums` array, considering subsets starting from the third element (the smallest possible polygon has three sides) up to the end, checking if the current set can form a polygon.

   ```
   for k in range(3, len(nums) + 1):
   ```

   For each subset, the algorithm checks if the sum of the first `k-1` side lengths is greater than the `k`-th side. If this inequality holds, a valid polygon can be formed, and we update the maximum perimeter `ans`.

   ```
   if s[k - 1] > nums[k - 1]:
       ans = max(ans, s[k])
   ```

4. **Returning the Largest Perimeter**: After the iterations, the variable `ans`, which is initialized with `-1`, will contain the largest perimeter found, or `-1` if no polygon can be formed.

   ```
   return ans
   ```

In this solution, the greedy choice is in selecting the largest potential side combinations starting from the end of the sorted array. The algorithm stops at the largest perimeter found that satisfies the polygon inequality, without the need to check all possible combinations, thus reducing the complexity.

Data structures used include:

- A list to hold the prefix sums (`s`).
- The sorted version of the initial array (`nums`).

These tools combined allow the solution to efficiently find the largest perimeter of a polygon with sides found in the input array or determine that such a polygon cannot exist.

Example Walkthrough

Let's illustrate the solution approach using a small example. Consider the array of sides `nums = [2, 1, 2, 4, 3]`.

1. **Sorting**: First, we sort the array to simplify the checking of potential polygons.

   ```
   nums.sort()  # Becomes [1, 2, 2, 3, 4]
   ```

2. **Prefix Sums**: Next, we use the `accumulate` function to get the prefix sums.

   ```
   s = list(accumulate(nums, initial=0))  # s becomes [0, 1, 3, 5, 8, 12]
   ```

3. **Iterating and Checking for a Polygon**: We then iterate over the array, starting from the three smallest sides and move towards the larger ones to check for the possibility of a polygon. The `k`-th index in `nums` corresponds to `s[k+1]` because of the `initial=0` in the prefix sum array.

   For `k = 3` (first iteration):

   ```
   if s[3 - 1] > nums[3 - 1]:
       if 3 > 2:
   ```

   This inequality holds, so a polygon can be formed with sides `[1, 2, 2]` and its perimeter is `3 + 2 = 5`.

   For `k = 4` (second iteration):

   ```
   if s[4 - 1] > nums[4 - 1]:
       if 5 > 3:
   ```

   This inequality also holds, so a polygon can be formed with sides `[1, 2, 2, 3]` and its perimeter is `5 + 3 = 8`.

   For `k = 5` (third iteration, checking for a triangle):

   ```
   if s[5 - 1] > nums[5 - 1]:
       if 8 > 4:
   ```

   This inequality holds as well and represents a triangle with sides `[2, 3, 4]` which has the largest perimeter so far of `8 + 4 = 12`.

4. **Returning the Largest Perimeter**: The largest perimeter found is `12`, so our function would return `12`.

In this example, the sorted side lengths allow us to efficiently find the combination `[2, 3, 4]` with the largest perimeter that satisfies the polygon inequality, without having to check all possible combinations.

Solution Implementation

Python

```
from itertools import accumulate
from typing import List

class Solution:
    def largest_perimeter(self, nums: List[int]) -> int:
        # Sort the numbers in non-decreasing order
        nums.sort()

        # Generate the cumulative sum of the sorted list.
        # with an initial value of 0 to make indices line up
        cumulative_sum = list(accumulate(nums, initial=0))

        # Initialize the answer to be 0, where 0 will indicate
        # that no triangle can be formed
        largest_perimeter = 0

        # Iterate over the sorted numbers from the third element to the end
        for k in range(3, len(nums) + 1):
            # Check if the sum of the two smaller sides is greater than the largest side
            if cumulative_sum[k - 1] > nums[k - 1]:
                # Update the largest perimeter found so far if the above condition holds
                # cumulative_sum[k] is the perimeter of a triangle formed by nums[k-3], nums[k-2], nums[k-1]
                largest_perimeter = max(largest_perimeter, cumulative_sum[k])

        # Final result, will be 0 if no triangle can be formed
        return largest_perimeter
```

Java

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

// Define the Solution class
class Solution {

    // Method to find the largest perimeter of a triangle that can be formed with given side lengths
    public long largestPerimeter(int[] nums) {

        // Sort the array in non-decreasing order.
        Arrays.sort(nums);

        // Get the number of elements in the array.
        int n = nums.length;

        // Create an array to hold the sum of lengths up to the current index.
        long[] prefixSums = new long[n + 1];

        // Compute the prefix sums.
        for (int i = 1; i <= n; ++i) {
            prefixSums[i] = prefixSums[i - 1] + nums[i - 1];
        }

        // Initialize the variable to store the maximum perimeter found.
        long maxPerimeter = -1;

        // Loop over the array to find the maximum perimeter of any triangle that can be formed.
        for (int i = 3; i <= n; ++i) {
            // Check if the sum of the two smaller sides is greater than the largest side.
            if (prefixSums[i - 1] > nums[i - 1]) {
                // Update the maximum perimeter with the new larger perimeter.
                maxPerimeter = Math.max(maxPerimeter, prefixSums[i]);
            }
        }

        // Return the maximum perimeter found, or -1 if no triangle can be formed.
        return maxPerimeter;
    }
}
```

C++

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

using namespace std;

class Solution {
public:
    // Function to calculate the largest perimeter of a non-degenerate triangle
    long long largestPerimeter(vector<int>& nums) {
        // First, sort the array in non-decreasing order
        sort(nums.begin(), nums.end());

        int n = nums.size(); // Size of the input array
        vector<long long> prefixSums(n + 1); // Vector to store prefix sums

        // Compute the prefix sums
        for (int i = 1; i <= n; ++i) {
            prefixSums[i] = prefixSums[i - 1] + nums[i - 1];
        }

        long long maxPerimeter = -1; // Initialize the maximum perimeter as -1

        // Iterate through the array considering each number as the longest side of the triangle
        for (int k = 3; k <= n; ++k) {
            // Check if a non-degenerate triangle can be formed
            // The sum of the two smaller sides must be greater than the third side
            if (prefixSums[k - 1] > nums[k - 1]) {
                // Update the maximum perimeter with the current perimeter if it is larger
                maxPerimeter = max(maxPerimeter, prefixSums[k]);
            }
        }

        // Return the maximum perimeter found, or -1 if no such triangle exists
        return maxPerimeter;
    }
};
```

TypeScript

```
function largestPerimeter(nums: number[]): number {
    // Sort the input array in non-decreasing order
    nums.sort((a, b) => a - b);

    // Get the total number of elements in the array
    const numElements = nums.length;

    // Initialize a new array to store the cumulative sums
    // The cumulative sum will be stored such that index 'i' of sumArray
    // contains the sum of the first 'i' elements of the sorted 'nums' array
    const sumArray: number[] = Array(numElements + 1).fill(0);

    // Compute the cumulative sums
    for (let i = 1; i <= numElements; ++i) {
        sumArray[i + 1] = sumArray[i] + nums[i];
    }

    // Initialize the answer as -1, which will indicate no triangle can be formed
    let maxPerimeter = -1;

    // Loop through the array to find a valid triangle
    // Starting from the third element since a triangle needs 3 sides
    for (let index = 3; index <= numElements; ++index) {
        // Check if the sum of the smaller two sides is greater than the current side,
        // which is a necessary and sufficient condition for a non-degenerate triangle.
        if (sumArray[index - 1] > nums[index - 1]) {
            // Update the maximum perimeter found so far
            maxPerimeter = Math.max(maxPerimeter, sumArray[index]);
        }
    }

    // Return the maximum perimeter, or -1 if no valid triangle can be formed
    return maxPerimeter;
}
```

```
from itertools import accumulate
from typing import List

class Solution:
    def largest_perimeter(self, nums: List[int]) -> int:
        # Sort the numbers in non-decreasing order
        nums.sort()

        # Generate the cumulative sum of the sorted list.
        # with an initial value of 0 to make indices line up
        cumulative_sum = list(accumulate(nums, initial=0))

        # Initialize the answer to be 0, where 0 will indicate
        # that no triangle can be formed
        largest_perimeter = 0

        # Iterate over the sorted numbers from the third element to the end
        for k in range(3, len(nums) + 1):
            # Check if the sum of the two smaller sides is greater than the largest side
            if cumulative_sum[k - 1] > nums[k - 1]:
                # Update the largest perimeter found so far if the above condition holds
                # cumulative_sum[k] is the perimeter of a triangle formed by nums[k-3], nums[k-2], nums[k-1]
                largest_perimeter = max(largest_perimeter, cumulative_sum[k])

        # Final result, will be 0 if no triangle can be formed
        return largest_perimeter
```

Time and Space Complexity

The time complexity of the provided code is `O(n log n)` and the space complexity is `O(n)`.

Time Complexity:

1. Sorting the list (`nums.sort()`): Sorting an array of `n` numbers has a time complexity of `O(n log n)`.
2. Accumulating the sorted array (`list(accumulate(nums, initial=0))`): The `accumulate` function generates a sum of the elements sequentially, hence it has a time complexity of `O(n)`.
3. The `for` loop (`for k in range(3, len(nums) + 1)`): This loop runs `(n - 2)` times (from `3` to `len(nums)`), which is `O(n)`.

In the loop, other than the condition check, we have an `O(1)` assignment for `ans` variable (`ans = max(ans, s[k])`).

The dominant term here is from the sorting step, therefore the overall time complexity is `O(n log n)`.

Space Complexity:

1. The sorted list does not require additional space since sorting is done in-place, hence the space complexity is `O(1)` for this step.
2. When creating a list from the `accumulate` function, we're generating a new list of the same size as `nums`, hence we have a space complexity of `O(n)` for this list.
3. The variables `ans` and `k` use constant space, contributing `O(1)`.

Adding these up, the main contribution to space complexity comes from the list generated by the `accumulate` function, therefore the overall space complexity is `O(n)`.