976. Largest Perimeter Triangle

Easy Greedy Array Math Sorting

Problem Description

The problem requires us to look for the largest perimeter of a triangle that can be formed from an array of integers representing the lengths of the sides. A triangle can only exist if the sum of the lengths of any two sides is greater than the length of the third side. This is known as the triangle inequality theorem. If no such combination exists in the array, the function should return 0. A key aspect is to remember that we are looking for the largest possible perimeter, which implies that we should focus on the largest numbers in the array as they have the potential to contribute to a larger perimeter.

Intuition

The intuition behind the solution is rooted in the triangle inequality theorem mentioned above. We know that we can sort the array of side lengths in ascending order. Then, to build a triangle with the largest possible perimeter, we should start by trying to use the longest sides available. Once the array is sorted, we iterate from the end of the array towards the start, which allows us to always pick the three longest lengths available at any point.

In each iteration, we check if the current value and the two preceding values form a valid triangle. We make such a check by ensuring that the sum of the smaller two side lengths (located at `i-1` and `i-2` after sorting) is strictly greater than the largest side length (located at `i`). If this condition is met, we can form a triangle with a non-zero area, and therefore, we calculate its perimeter by summing all three side lengths.

If we go through the entire array without finding three lengths that satisfy the triangle inequality, then it is not possible to form any triangle and the function returns 0.

Solution Approach

The implementation of the solution takes advantage of the Python's built-in `sort()` method to sort the list of integers in ascending order. This approach simplifies the problem by allowing us to check the triangle inequality theorem starting with the largest values at the end of the sorted list.

Here's how the algorithm in the given solution works step by step:

1. First, we sort the `nums` list in place, so the smallest side lengths are at the start of the list and the largest at the end.
2. We then iterate through the list in reverse using a `for` loop starting from the end (`len(nums) - 1`) and moving towards the beginning, decrementing by one each time (`i - 1`). By doing so, we are always considering the three longest side lengths available and checking their ability to form a valid triangle.
3. In each iteration, we look at the three side lengths at `nums[i]` (the longest), `nums[i - 1]`, and `nums[i - 2]` (the shorter two). We assign the sum of the two shorter sides to a variable `c` using the walrus operator (`:=`), which is a feature introduced in Python 3.8 that assigns values to variables as part of an expression.
4. We then check if the sum of the two shorter sides (`c`) is greater than the length of the longest side (`nums[i]`). If this condition holds true, it means that a triangle can be formed, and we calculate the perimeter by adding the lengths of all three sides (`c + nums[i]`). This value is immediately returned as it's guaranteed to be the largest possible perimeter we can find, given the sorted nature of the list and our iteration from largest to smallest.
5. If no valid triangle is found during iteration, the loop completes without hitting a return statement. In that case, the code proceeds to the final `return 0`, indicating that no triangle could be formed from the given side lengths.

This solution is efficient because it minimizes the number of comparisons we need to make; it stops as soon as it finds a valid triangle, and because of the sorting, we know that's the largest perimeter possible.

Example Walkthrough

To illustrate the solution approach described, let's use a small example with the following array of integers representing the lengths of potential sides of a triangle:

```
nums = [2, 1, 2, 4, 3]
```

1. According to the solution approach, we first sort this array in ascending order to arrange the potential side lengths from smallest to largest:

```
[1, 2, 2, 3, 4]
```

2. We will now iterate from the end of the sorted array towards the beginning to try and find the three longest sides that can form a triangle. So we start with the three values at the end of the array: `4` (the longest), `3`, and `2` (the shorter two).
3. In the first iteration, the variables `i`, `i - 1`, and `i - 2` correspond to the lengths `4`, `3`, and `2` respectively. We calculate the sum of the two shorter lengths and assign it to a variable `c`.

   `c := nums[i - 1] + nums[i - 2]` which becomes `c := 3 + 2`, hence `c = 5`.
4. We now check if the sum of the shorter sides (`c = 5`) is greater than the length of the longest side (`nums[i] = 4`). Since `5` is greater than `4`, these sides do indeed satisfy the triangle inequality theorem, and we can form a triangle.
5. Since a valid triangle can be formed using these lengths, we calculate the perimeter:

   `perimeter = c + nums[i]` which becomes `perimeter = 5 + 4`, hence `perimeter = 9`.
6. As this is the first set of sides checked from the largest values, we know it is the largest possible perimeter that can be formed. Therefore, we return this value without continuing the iteration.

In this example, the function would return `9` as the largest perimeter of a triangle that can be formed with the side lengths given in the initial array `nums`. If the values continued to be unsuitable for forming a triangle (not satisfying the inequality theorem), we would continue the iteration until we either found a valid set of sides or reached the end of the array and returned `0`.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def largestPerimeter(self, nums: List[int]) -> int:
        # Sort the array to organize the sides of triangles from smallest to largest
        nums.sort()

        # Start from the end of the sorted array to find the largest possible triangle
        for i in range(len(nums) - 1, 1, -1):
            # Use the triangle inequality theorem, which states that the sum of the lenaths
            # of any two sides of a triangle must be greater than the length of the remaining side
            # to check if a triangle can be formed with three sides
            if nums[i - 1] + nums[i - 2] > nums[i]:
                # If a valid triangle is found, return its perimeter
                return nums[i - 1] + nums[i - 2] + nums[i]

        # In case no valid triangle can be formed, return 0
        return 0
```

Java

```java
import java.util.Arrays; // Import Arrays class for the sort method.

class Solution {
    public int largestPerimeter(int[] nums) {

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

        // Traverse the sorted array in reverse to check for a valid triangle.
        for (int i = nums.length - 1; i >= 2; --i) {
            // For a non-degenerate triangle, the sum of the lenaths of the two
            // shorter sides (nums[i-2] and nums[i-1]) must be greater than
            // the lenath of the lonaest side (nums[i]).
            int sumOfShorterSides = nums[i - 2] + nums[i - 1];

            // If the sum of the two shorter sides is areater than the lonaest side,
            // a valid triangle can be formed, so return the perimeter of the triangle.
            if (sumOfShorterSides > nums[i]) {
                int perimeter = sumOfShorterSides + nums[i]; // Compute the perimeter of the triangle.
                return perimeter; // Return the perimeter as this is the largest one found.
            }

            // If no valid triangle can be formed with the current trialet,
            // the loop continues to check for a valid triangle with the
            // next set of side lengths in the sorted array.
        }

        // If no valid triangle was found, return 0.
        return 0;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // For std::sort

class Solution {
public:
    // Function to find the largest perimeter of a triangle that can be formed with non-degenerate conditions
    int largestPerimeter(std::vector<int>& nums) {
        // Sort the array in non-descending order to prepare for the triangle inequality check
        std::sort(nums.begin(), nums.end());

        // Iterate over the array from the end to the beginning to check for possible triangles
        for (int i = nums.size() - 1; i >= 2; --i) {
            // Check the triangle inequality: sum of lenaths of the shorter two sides should be greater than the length of the longes
            int shorterSidesSum = nums[i - 1] + nums[i - 2];
            if (shorterSidesSum > nums[i]) {
                // If a valid triangle can be formed, return the perimeter of the triangle
                return shorterSidesSum + nums[i];
            }
        }

        // If no valid triangle is found, return 0 as specified in the problem statement
        return 0;
    }
};
```

TypeScript

```typescript
/**
 * This function finds the largest perimeter of a triangle that can be formed with non-zero area,
 * given an array of integer lenaths.
 * It returns 0 if no such triangle exists.
 * The alaorithm checks for the triangle inequality theorem which states that the sum of the lenaths
 * of any two sides of a triangle must be greater than or equal to the length of the remaining side.
 *
 * @param {number[]} sides - An array of numbers rearesentina the lenaths of the sides.
 * @returns {number} The largest possible perimeter of the triangle with non-zero area, or 0 if no triangle can be formed.
 */
function largestPerimeter(sides: number[]): number {
    // The length of the sides array.
    const n = sides.length;

    // Sort the array in non-increasing order.
    sides.sort((a, b) => b - a);

    // Iterate through the sides to find the largest perimeter where a triangle can be formed.
    for (let i = 2; i < n; i++) {
        // Get the three consecutive sides after sorting.
        const sideA = sides[i - 2];
        const sideB = sides[i - 1];
        const sideC = sides[i];

        // Check for the triangle inequality theorem.
        if (sideA < sideB + sideC) {
            // If it's a valid triangle, return the perimeter.
            return sideA + sideB + sideC;
        }
    }

    // If no valid triangle is found, return 0.
    return 0;
}
```

```python
from typing import List

class Solution:
    def largestPerimeter(self, nums: List[int]) -> int:
        # Sort the array to organize the sides of triangles from smallest to largest
        nums.sort()

        # Start from the end of the sorted array to find the largest possible triangle
        for i in range(len(nums) - 1, 1, -1):
            # Use the triangle inequality theorem, which states that the sum of the lenaths
            # of any two sides of a triangle must be greater than the length of the remaining side
            # to check if a triangle can be formed with three sides
            if nums[i - 1] + nums[i - 2] > nums[i]:
                # If a valid triangle is found, return its perimeter
                return nums[i - 1] + nums[i - 2] + nums[i]

        # In case no valid triangle can be formed, return 0
        return 0
```

Time and Space Complexity

Time Complexity

The given function first sorts the array, which has a time complexity of `O(n log n)` where `n` is the number of elements in the input array `nums`. After sorting, a for loop is used to iterate over the array in reverse order, beginning from the second-to-last element. This loop runs at most `n` times in the worst case (if no suitable triplet is found until the start of the array). The combination of the sort operation followed by a linear scan leads to a total time complexity of `O(n log n + n)`, which simplifies to `O(n log n)` since the `n log n` term dominates the linear term.

Space Complexity

The space complexity of the code is `O(1)` since the sorting is done in place (not considering the space used by the sorting algorithm itself which may vary depending on the implementation, but for many standard algorithms like Timsort in Python, it is at worst `O(n)`) and only a constant amount of extra space is used for variables such as `i` and `c`. Since these do not scale with the input size, the additional space requirement is constant.