611. Valid Triangle Number

Medium Greedy Array Two Pointers Binary Search Sorting Leetcode Link

Problem Description

The problem presents us with an integer array `nums` and asks us to find the count of all possible triplets in this array that can form a valid triangle. A set of three numbers forms the sides of a triangle if and only if the sum of any two sides is greater than the third side. This is known as the triangle inequality theorem.

Intuition

To solve this problem, we can start by sorting the array. Sorting helps us to apply the triangle inequality more effectively by having an ordered list to work with. In this way, when we pick two numbers, the smaller two sides of a potential triangle, we already know that any number following them in the array will be greater or equal than these two.

Once we have the sorted array, the intuition behind the solution is as follows:

- The outer loop picks the first side `a` of the potential triangle.
- The second, inner loop picks the second side `b`, which is just the next number in the sorted array after the first side.
- Using the triangle inequality theorem, we know that in order to form a triangle, the third side `c` must be less than the sum of `a` and `b` but also greater than their difference. Since the array is sorted and we only need the sum condition ("less than the sum"), we can use binary search to quickly find the largest index `k` such that `nums[k]` is less than the sum of `nums[i]` plus `nums[j]` (where `nums[i]` is the first side and `nums[j]` is the second side).

With this approach, for each pair of `a` and `b`, binary search helps us find the count of possible third sides. We then subtract the index of the second side `j` from `k` to get the number of valid third sides `c`, and add this to our answer `ans`. We repeat this process until we have considered all possible pairs of sides `a` and `b`.

To sum up, the solution uses a combination of sorting, iterating through pairs of potential sides, and binary search to efficiently calculate the number of valid triangles that can be formed from the array.

Solution Approach

The solution begins with sorting the array `nums`. Sorting is a critical step because it allows us to take advantage of the triangle inequality in a sorted sequence, which makes it easier to reason about the relationship between the sides. With sorting, we guarantee that if `nums[i]` and `nums[j]` (where `i < j`) are two sides of a potential triangle, then any `nums[k]` (where `k > j`) is at least as large as `nums[j]`. This means that for the triangle inequality to be satisfied (`nums[i] + nums[j] > nums[k]`), we only need to find the first `k` where the inequality does not hold and then all indices before `k` will form valid triangles with `nums[i]` and `nums[j]`.

Here is how the solution is implemented:

- After sorting, a `for` loop is used to iterate through the array as the outer loop. This loop starts from the first element and stops at the third-to-last element, with `i` representing the index of the first side of a potential triangle.

- Inside the outer loop, a nested `for` loop starts from `i+1`, iterates over the remaining elements, and stops at the second-to-last element, with `j` representing the index of the second side of a potential triangle.

- For each pair of sides represented by `nums[i]` and `nums[j]`, binary search is used to find the right-most element which could be a candidate for the third side of the triangle. The `bisect_left` function from the Python `bisect` module is used to find the index `k`, where `nums[k]` is the smallest number greater than or equal to the sum of `nums[i]` and `nums[j]`. Since `nums[k]` itself cannot form a triangle with `nums[i]` and `nums[j]` (it's not strictly less than the sum), we subtract one to get the index of the largest valid third side.

- Now with `k-1` being the largest index of a valid third side and `j` the index of the second side, all numbers in the range `(j, k)` will satisfy the triangle inequality. Thus, `ans` (the count of valid triangles) is incremented by `k - j - 1`.

- By the end of these loops, `ans` will contain the total number of triplets that can form valid triangles.

The solution applies the sorting algorithm, iteration through nested loops, and binary search to solve the problem efficiently.

Example Walkthrough

Let's apply our solution approach on a small example array, `nums = [4, 2, 3, 4]`.

First, we sort the array to get `nums = [2, 3, 4, 4]`. Sorting helps us apply the triangle inequality theorem effectively.

Next, we begin our outer loop with `i = 0`, where `nums[i] = 2`.

Inside our outer loop, we start our inner loop with `j = i + 1`, which makes `j = 1`, and `nums[j] = 3`.

For `nums[i] = 2` and `nums[j] = 3`, we perform a binary search to find the largest index `k` where `nums[k]` is less than the sum of `nums[i]` and `nums[j]`, which is `2 + 3 = 5`. In this sorted array, `nums[k]` could be either `4` or `4`, both are less than `5`. Binary search finds that `k = 3` (the last index of `4` in the sorted array). Since the actual element at `k` cannot be part of the triangle, to get the largest valid `k`, we subtract one from it, making `k = 2`.

Now we calculate the number of valid `c` sides between `j` and `k`, which is `k - j - 1 = 2 - 1 - 1 = 0`. There is no third side that can form a triangle with `nums[i] = 2` and `nums[j] = 3`.

Next, we increment `j` to `2`, making `nums[j] = 4`, and repeat the binary search. This finds that `k = 3` (as `4 + 4` is not less than any element in the array). Subtracting one from `k` gives us `k = 2`. The number of valid third sides `c` is `k - j - 1 = 2 - 2 - 1 = -1`. No valid triangle can be formed in this case as well.

Then we increment `i` to `1` and `j` to `i + 1` which is `2`, and now we have `nums[i] = 3` and `nums[j] = 4`. Repeating the binary search for the sum `3 + 4 = 7`, we find that no such `k` exists since all existing elements are less than `7`. Thus, `k` is the length of the array, `k = 4`. And the number of valid `c` sides is `k - j - 1 = 4 - 2 - 1 = 1`, so we can form one triangle using the indices `(1, 2, 3)`.

Our next pair would be `nums[i] = 3` and `nums[j] = 4` (again, but with `j = 3`), but since there's no element after `nums[j]` when `j = 3`, we cannot form a triangle, and the process stops here.

Therefore, through our example, we found only one valid triplet `(3, 4, 4)` which can form a triangle. Our answer `ans` would be `1` for this example.

Python Solution

```python
from bisect import bisect_left

class Solution:
    def triangleNumber(self, nums: List[int]) -> int:
        # First, sort the input array to arrange numbers in non-decreasing order
        nums.sort()

        # Initialize the answer and get the length of the array
        triangle_count, n = 0, len(nums)

        # Loop over each triple. For triangles, we just need to ensure that
        # the sum of the lengths of any two sides is greater than the length of the third side.
        for i in range(n - 2):  # The last two numbers are not needed as they are candidates for the longest side of the triangle
            for j in range(i + 1, n - 1):  # j is always after i

                # Find the index of the smallest number that is greater than
                # the sum of nums[i] and nums[j] using binary search.
                # This determines the right boundary of possible valid triangles.
                # We subtract 1 because bisect_left returns the index where we could insert the
                # number keeping the list sorted, but we want the last valid index.
                k = bisect_left(nums, nums[i] + nums[j], lo=j + 1) - 1

                # The count of triangles for this specific (i, j) pair is k - j
                # because any index between j and k (exclusive) can be chosen as the
                # third side of the triangle.
                triangle_count += k - j

        # Return the total count of triangles that can be formed
        return triangle_count

```

Java Solution

```java
class Solution {
    public int triangleNumber(int[] nums) {
        // Sort the array in non-decreasing order
        Arrays.sort(nums);
        int count = nums.length;
        // Initialize result to 0
        int result = 0;

        // Iterate over the array starting from the end
        for (int i = count - 1; i >= 2; --i) {
            // Set two pointers, one at the beginning and one before the current element
            int left = 0, right = i - 1;

            // Iterate as long as left pointer is less than right pointer
            while (left < right) {
                // Check if the sum of the elements at left and right pointers is greater than
                // the current element to form a triangle
                if (nums[left] + nums[right] > nums[i]) {
                    // If condition is satisfied, we can form triangles with any index between left and right
                    result += right - left;
                    // Move the right pointer downwards
                    --right;
                } else {
                    // If condition is not satisfied, move the left pointer upwards
                    ++left;
                }
            }
        }
        // Return the total count of triangles found
        return result;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int triangleNumber(vector<int>& nums) {
        // Sort the numbers in non-decreasing order
        sort(nums.begin(), nums.end());

        // Initialize the count of triangles
        int count = 0;

        // The total number of elements in the vector
        int n = nums.size();

        // The outer loop goes through each number starting from the first until the third last
        for (int i = 0; i < n - 2; ++i) {

            // The second loop goes through numbers after the current number chosen by the outer loop
            for (int j = i + 1; j < n - 1; ++j) {

                // Find the rightmost position where the sum of nums[i] and nums[j] is not less than the element at that position
                // This will ensure that the triangle inequality holds => nums[i] + nums[j] > nums[k]
                int k = lower_bound(nums.begin() + j + 1, nums.end(), nums[i] + nums[j]) - nums.begin() - 1;

                // Increase the count by the number of valid triangles found with the base[i,j] pair
                // k is the rightmost position before which all elements can form a triangle with nums[i] and nums[j], hence k - j
                count += k - j;
            }
        }

        // Return the count of triangles
        return count;
    }
};

```

Typescript Solution

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

    let count = nums.length;
    let validTriangles = 0; // Initialize the count of valid triangles

    // Iterate through each number starting from the end
    for (let i = count - 1; i >= 2; i--) {
        // Initialize two pointers
        let left = 0;
        let right = i - 1;

        // Use the two pointers to find valid triangles
        while (left < right) {
            // Check if the sum of the two sides is greater than the third side
            if (nums[left] + nums[right] > nums[i]) {
                // Count the number of triangles with nums[i] as the longest side
                validTriangles += right - left;
                // Move the right pointer to check for other possible triangles
                right--;
            } else {
                // Move the left pointer to increase the sum of nums[left] + nums[right]
                left++;
            }
        }
    }

    // Return the total count of valid triangles
    return validTriangles;
}

```

Time and Space Complexity

The given Python code sorts an array and then uses a double loop to find every triplet that can form a valid triangle, using binary search to optimize the finding of the third side.

Time Complexity

The time complexity of the code can be broken down as follows:

1. `nums.sort()` has a time complexity of `O(n log n)` where `n` is the number of elements in nums.
2. The outer for-loop runs `(n - 2)` times.
3. The inner for-loop runs `(n - i - 2)` times per iteration of the outer loop, summing up to roughly `(1/2) * n^2` in the order of growth.
4. The `bisect_left` call within the inner loop does a binary search, which has a time complexity of `O(log n)`.

Thus, the total time complexity of the nested loop (excluding the sort) would be `O((n^2/2) * log n)`, as each iteration of `j` could potentially result in a binary search. When adding the sort, the overall time complexity is `O(n log n + (n^2/2) * log n)` which simplifies to `O(n^2 log n)` when n is large.

Space Complexity

Space complexity is easier to analyze since the only extra space used is for sorting, which in Python is `O(n)` because Timsort (the algorithm behind Python's sort) can require this much space. No other significant extra space is used, as the variables `i`, `j`, `k`, and `ans` use constant space. Therefore, the space complexity of the algorithm is `O(n)`.