<https://leetcode.com/problems/valid-triangle-number/>

Given an integer array nums, return *the number of triplets chosen from the array that can make triangles if we take them as side lengths of a triangle*.

**Example 1:**

Input: nums = [2,2,3,4]

Output: 3

Explanation: Valid combinations are:

2,3,4 (using the first 2)

2,3,4 (using the second 2)

2,2,3

**Example 2:**

Input: nums = [4,2,3,4]

Output: 4

**Constraints:**

1 <= nums.length <= 1000

0 <= nums[i] <= 1000

**Attempt 1: 2022-09-17**

**Solution 1:  Two Pointers solution (10 min, similar to L15 3Sum Two Pointers solution)**

class Solution {

    public int triangleNumber(int[] nums) {

        if(nums.length < 3) {

            return 0;

        }

        Arrays.sort(nums);

        int count = 0;

        for(int i = 2; i < nums.length; i++) {

            int lo = 0;

            int hi = i - 1;

            while(lo < hi) {

                // Assume a is the longest edge, b and c are shorter ones, to form a triangle,

                // they need to satisfy len(b) + len(c) > len(a)

                if(nums[lo] + nums[hi] > nums[i]) {

                    // Since the array is sorted and nums[lo] + nums[hi] > nums[i],

                    // we know that all elements from lo, lo+1, lo+2 till just

                    // before hi will also satisfy the condition. hence we

                    // directly add (hi-lo) to the result. After that, we

                    // reduce hi by one place and follow the same process

                    count += hi - lo;

                    hi--;

                } else {

                    lo++;

                }

            }

        }

        return count;

    }

}

Space Complexity: O(1)

Time Complexity: O(n^2)

**Refer to**

<https://leetcode.com/problems/valid-triangle-number/discuss/104169/Java-Solution-3-pointers>

**Solution 2: Binary Search solution (360min, too long to figure out how to transfer problem into Find Lower Boundary template)**

class Solution {

    public int triangleNumber(int[] nums) {

        Arrays.sort(nums);

        int len = nums.length;

        if(len < 3) {

            return 0;

        }

        int count = 0;

        for(int i = 0; i < len - 2; i++) {

            for(int j = i + 1; j < len - 1; j++) {

                int target = nums[i] + nums[j];

                int index = findLowerBoundary(nums, j, target);

                // Important: The return from find lower boundary as -1

                // means no lower boundary found, equal to not able to

                // find first element d >= a + b (target), which means all

                // elements from index (j+1) till the end of nums (full

                // len) < a + b (target). So how many 'c' match the condition ?

                // Just calculate how many elements between (j+1) to (len-1)

                // as (len-1) - (j+1) + 1 = (len-j-1)

                if(index != -1) {

                    count += (index - j - 1);

                } else {

                    count += (len - j - 1);

                }

            }

        }

        return count;

    }

    // Why we have to find lower boundary first ?

    // Return the right most number 'c' such that a + b > c, return -1 otherwise

    // ==>

    // In an ascending array, recognize (a + b) as target, equal to find first element

    // d >= a + b (target), then all left elements start from (j + 1) to (index of 'd'

    // - 1) are match the requirement as a + b > c

    // =========================================

    // Example 1:

    // nums={2,2,3,4} sort -> nums={2,2,3,4}

    // i=0,j=1

    // target=nums[0]+nums[1]=2+2=4

    // lo=1+1=2,hi=3 -> mid=2 -> nums[2]=3 < target=4 -> lo=2+1=3

    // lo=3,hi=3 -> mid=3 -> nums[3]=4 = target=4 -> hi=3-1=2

    // lo > hi -> while loop done -> return lo=3

    // so the left adjacent element of nums[lo] is the right most number 'c' match

    // condition a + b > c, as index=lo-1=2, nums[2]=3. So how many 'c' match the

    // condition ? Just calculate the range between -> (index-(j+1)+1) = (lo-1-j-1+1) =

    // (lo-j-1) = (3-1-1) = 1, which means from j+1=2 to index=lo-1=2 there is only 1

    // number match the requirement

    // -----------------------------------------

    // i=0,j=2

    // target=nums[0]+nums[2]=2+3=5

    // lo=2+1=3,hi=3 -> mid=3 -> nums[3]=4 < target=5 -> lo=3+1=4

    // lo > hi -> while loop done -> but because lo==nums.length=4 we will return -1

    // The return from find lower boundary as -1 means no lower boundary found, equal

    // to not able to find first element d >= a + b (target), which means all elements

    // from index (j+1) till the end of nums (full len) < a + b (target). So how many

    // 'c' match the condition ? Just calculate the range between -> (len-1-(j+1)+1) =

    // (len-1-j-1+1) = (len-j-1) = (4-2-1) = 1

    // -----------------------------------------

    // Another solution comes from i=1,j=2

    // =========================================

    // Example 2:

    // nums={7,0,0,0} sort -> nums={0,0,0,7}

    // i=0,j=1

    // target=nums[0]+nums[1]=0+0=0

    // lo=1+1=2,hi=3 -> mid=2 -> nums[2]=0 >= target=0 -> hi=2-1=1

    // lo > hi -> while loop done -> return lo=2

    // count+=(2-1-1)=count+=0 no change

    // -----------------------------------------

    // i=0,j=2

    // target=nums[0]+nums[2]=0+0=0

    // lo=2+1=3,hi=3 -> mid=3 -> nums[3]=7 >= target=0 -> hi=3-1=2

    // lo > hi -> while loop done -> return lo=3

    // count+=(3-2-1)=count+=0 no change

    // -----------------------------------------

    // i=1,j=2

    // target=nums[1]+nums[2]=0+0=0

    // lo=2+1=3,hi=3 -> mid=3 -> nums[3]=7 >= target=0 -> hi=3-1=2

    // lo > hi -> while loop done -> return lo=3

    // count+=(3-2-1)=count+=0 no change

    // =========================================

    private int findLowerBoundary(int[] nums, int j, int target) {

        int lo = j + 1;

        int hi = nums.length - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(nums[mid] >= target) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        if(lo == nums.length || nums[lo] < target) {

            return -1;

        }

        return lo;

    }

}

Space Complexity: O(1)

Time Complexity: O(n^2logn)

**Refer to**

<https://leetcode.com/problems/valid-triangle-number/discuss/2203786/Binary-Search-Solution-oror-C%2B%2B-oror-O(n2logn)-Solution>

class Solution {

public:

    int findIdx(int lo, int hi, int target, vector<int> &nums){

        while(lo <= hi){

            int mid = (lo + hi) / 2;

            if(nums[mid] >= target){

                hi = mid - 1;

            }

            else{

                lo = mid + 1;

            }

        }

        return hi;

    }

    int triangleNumber(vector<int>& nums) {

        sort(nums.begin(), nums.end());

        int n = nums.size();

        // 3 sides of triangle are valid if a + b > c

        // We can select a & b using 2 for loops

        // To check the third side, we first sort the given array

        // Since c > a && c > b, we definitely know that c will lie beyond b's index

        // So, we search for c such that it is the largest value less than (a + b) and return its index

        // From the b's index upto this index will be the choices we can have for our side

        int ans = 0;

        for(int i = 0; i < n; i++){

            for(int j = i + 1; j < n; j++){

                int idx = findIdx(j + 1, n - 1, nums[i] + nums[j], nums);

                int k = idx - j;

                ans += k;

            }

        }

        return ans;

    }

};

**Refer to**

[L15.P3.4.3Sum](note://EC0082E510FD4360AD113BFC40B23D1F)

[L18.P3.9.4Sum(Ref.L15)](note://C61608B7F28E422BA14F37ABF6331F2A)