<https://leetcode.com/problems/count-pairs-whose-sum-is-less-than-target/description/>

Given a 0-indexed integer array nums of length n and an integer target, return the number of pairs (i, j) where 0 <= i < j < n and nums[i] + nums[j] < target.

**Example 1:**

**Input:** nums = [-1,1,2,3,1], target = 2

**Output:** 3

**Explanation:** There are 3 pairs of indices that satisfy the conditions in the statement:

- (0, 1) since 0 < 1 and nums[0] + nums[1] = 0 < target

- (0, 2) since 0 < 2 and nums[0] + nums[2] = 1 < target

- (0, 4) since 0 < 4 and nums[0] + nums[4] = 0 < target

Note that (0, 3) is not counted since nums[0] + nums[3] is not strictly less than the target.

**Example 2:**

**Input:** nums = [-6,2,5,-2,-7,-1,3], target = -2

**Output:** 10

**Explanation:** There are 10 pairs of indices that satisfy the conditions in the statement:

- (0, 1) since 0 < 1 and nums[0] + nums[1] = -4 < target

- (0, 3) since 0 < 3 and nums[0] + nums[3] = -8 < target

- (0, 4) since 0 < 4 and nums[0] + nums[4] = -13 < target

- (0, 5) since 0 < 5 and nums[0] + nums[5] = -7 < target

- (0, 6) since 0 < 6 and nums[0] + nums[6] = -3 < target

- (1, 4) since 1 < 4 and nums[1] + nums[4] = -5 < target

- (3, 4) since 3 < 4 and nums[3] + nums[4] = -9 < target

- (3, 5) since 3 < 5 and nums[3] + nums[5] = -3 < target

- (4, 5) since 4 < 5 and nums[4] + nums[5] = -8 < target

- (4, 6) since 4 < 6 and nums[4] + nums[6] = -4 < target

**Constraints:**

1 <= nums.length == n <= 50

-50 <= nums[i], target <= 50

**Attempt 1: 2024-01-27**

**Solution 1: Sorting + Binary Search (10 min)**

class Solution {

    public int countPairs(List<Integer> nums, int target) {

        int count = 0;

        Collections.sort(nums);

        for(int i = 0; i < nums.size(); i++) {

            int index = binarySearch(nums, i + 1, target - nums.get(i));

            // 'index - 1' means the maximum value we can pick

            // in 'nums' list should only before 'index', because

            // if at or above 'index' will make sum up >= target

            // 'index - 1 - (i + 1) + 1' means 'end(=index - 1) - start(=i + 1) + 1'

            // for the count of pairs available during the [i + 1, index - 1] range

            count += index - 1 - (i + 1) + 1;

        }

        return count;

    }

    // Find lower boundary for 'target - nums.get(i)'

    private int binarySearch(List<Integer> nums, int startIndex, int val) {

        int lo = startIndex;

        int hi = nums.size() - 1;

        while(lo <= hi) {

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

            if(nums.get(mid) >= val) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        return lo;

    }

}

Time Complexity: O(N\*logN)

Space Complexity: O(1)

**Refer to**

<https://algo.monster/liteproblems/2824>

**Problem Description**

The task at hand is to find the number of unique pairs (i, j) within an array nums, where nums has n elements, such that when we add the elements at positions i and j, the sum is less than a given target value. It's important to note that the array is 0-indexed (meaning indexing starts from 0), and the pairs must satisfy 0 <= i < j < n, which ensures that i is strictly less than j, and j is within the bounds of the array.

To simplify, given an array and a numeric target, we're looking for pairs of numbers in the array that add up to a number less than the target. The problem asks for a count of such pairs.

**Intuition**

When tackling this problem, the intuition is that if we have an array sorted in increasing order, we can efficiently find the threshold beyond which a pair of numbers would exceed the target value. Sorting helps constrain the search space when looking for the second number of the pair.

Here's the step-by-step approach to arrive at the solution:

Sort the Array: Start by sorting nums in non-decreasing order. This allows us to use the property that if nums[k] is too large for some i when paired with nums[j], then nums[k+1], nums[k+2], ..., nums[n-1] will also be too large.

Two-Pointer Strategy: We could use a two-pointer strategy to find the count of valid pairs, but the issue is that it runs in O(n^2) time in its naïve form because we'd check pairs (i, j) exhaustively.

Binary Search Optimization: To optimize, we turn to binary search (bisect\_left in Python). For each number x in our sorted nums at index j, we want to find the largest index i such that i < j and nums[i] + x < target, which gives us the number of valid pairs with the second number being x.

Counting Pairs: The function bisect\_left returns the index where target - x would be inserted to maintain the sorted order, which is conveniently the index i we are looking for. The value of i represents how many numbers in the sorted array are less than target - x when x is the second element of the pair. Since j is the current index, and we're interested in indices less than j, by passing hi=j to bisect\_left, we ensure that.

By looping through all elements x of the sorted nums and applying binary search, we get the count of valid pairs for each element. Summing these counts gives us the total number of pairs that satisfy the problem's criteria.

The elegance of this solution lies in effectively reducing the complexity from O(n^2) to O(n log n) due to sorting and the binary search, which takes O(log n) time per element.

**Solution Approach**

The given solution implements the optimized approach using sorting and binary search as follows:

Sorting: First, the list nums is sorted in non-decreasing order. This allows us to leverage the fact that once we find a pair that satisfies our condition (nums[i] + nums[j] < target), any smaller i for the same j will also satisfy the condition, since the array is sorted.

nums.sort()

Binary Search: The binary search is done using Python's bisect\_left method from the bisect module.

i = bisect\_left(nums, target - x, hi=j)

Here, the bisect\_left method is used to find the index i at which we could insert target - x while maintaining the sorted order of the array. It searches in the slice of nums up to the index j, which ensures that we are only considering elements at indices less than j. The element x corresponds to the second number in our pair, and the index j is its position in the sorted array.

Loop and Count: For every number x in our sorted nums, represented by the loop index j, we find how many numbers are to the left of j that could form a valid pair with x. This is done by adding the result of the binary search to our answer ans.

ans = 0

for j, x in enumerate(nums):

i = bisect\_left(nums, target - x, hi=j)

ans += i

Return Result: After iterating through all the elements of the sorted array and accumulating the valid pairs count in ans, the final step is to return ans, which holds the total number of valid pairs found.

return ans

In summary, the solution harnesses the binary search algorithm to efficiently find for each element x in nums the number of elements to the left that can be paired with x to form a sum less than target. The sorting step beforehand ensures that the binary search operation is possible. The time complexity of this algorithm is O(n log n), with O(n log n) for the sorting step and O(n log n) for the binary searches (O(log n) for each of the n elements).

**Example Walkthrough**

Let's say we have an array nums = [7, 3, 5, 1] and the target = 8. We want to find the number of unique pairs (i, j) such that nums[i] + nums[j] < target and 0 <= i < j < n.

Here's how we apply the solution approach to our example:

**Sort the Array:** We start by sorting nums to get [1, 3, 5, 7].

**Binary Search and Loop:**

Let's begin the loop with j = 1 (x = 3), since i < j. For x = 3, we want to find how many numbers to the left are less than target - x (8 - 3 = 5). We use bisect\_left and obtain i = bisect\_left([1, 3, 5, 7], 5, hi=1) = 1. This means starting from the index 0, there is 1 number that can be paired with 3 to have a sum less than 8.

Next, j = 2 (x = 5). We're looking for numbers less than 8 - 5 = 3. Index i is found by bisect\_left([1, 3, 5, 7], 3, hi=2) = 1. Again, 1 number left of index 2 can pair with 5.

Then, for j = 3 (x = 7), target - x is 8 - 7 = 1. Calling bisect\_left([1, 3, 5, 7], 1, hi=3) = 0 gives i = 0, but there are no numbers less than 1 in the array, so we cannot form any new pairs with 7.

**Counting Pairs:**

For each j, we add i to our total count ans.

From our steps: ans = 1 + 1 + 0 = 2.

**Return Result:** With the loop completed, we've determined there are 2 unique pairs that meet the criteria: (1, 3) and (1, 5). Thus, we return ans = 2.

This example illustrates how sorting the array, using a two-pointer approach, and optimizing with binary search allows us to efficiently solve this problem.

**Java Solution**

class Solution {

    // Method to count the number of pairs that, when added, equals the target value

    public int countPairs(List<Integer> nums, int target) {

        // Sort the list first to apply binary search

        Collections.sort(nums);

        int pairCount = 0;

        // Iterate through each element in the list to find valid pairs

        for (int j = 0; j < nums.size(); ++j) {

            int currentVal = nums.get(j);

            // Search for index of the first number that is greater than or equal to (target - currentVal)

            int index = binarySearch(nums, target - currentVal, j);

            // Increment the pair count by the number of valid pairs found

            pairCount += index;

        }

        return pairCount;

    }

    // Helper method to perform a binary search and find the first element greater than or equal to x before index r

    private int binarySearch(List<Integer> nums, int x, int rightBound) {

        int left = 0;

        while (left < rightBound) {

            // Find the middle index between left and rightBound

            int mid = (left + rightBound) >> 1; // equivalent to (left + rightBound) / 2

            // If the value at mid is greater than or equal to x, move the rightBound to mid

            if (nums.get(mid) >= x) {

                rightBound = mid;

            } else {

                // Otherwise, move the left bound just beyond mid

                left = mid + 1;

            }

        }

        // Return the left bound as the first index greater than or equal to x

        return left;

    }

}

**Time and Space Complexity**

The code provided is using a sorted array to count pairs that add up to a specific target value.

**Time Complexity:**

The time complexity of the sorting operation at the beginning is O(n log n) where n is the total number of elements in the nums list.

The for loop runs in O(n) time since it iterates over each element in the list once.

Inside the loop, the bisect\_left function is called, which performs a binary search and runs in O(log j) time where j is the current index of the loop.

Since bisect\_left is called inside the loop, we need to consider its time complexity for each iteration. The average time complexity of bisect\_left across all iterations is O(log n), making the for loop's total time complexity O(n log n).

Hence, the overall time complexity, considering both the sort and the for loop operations, is O(n log n) because they are not nested but sequential.

**Space Complexity:**

The space complexity is O(1) assuming the sort is done in-place (Python's Timsort, which is typically used in .sort(), can be O(n) in the worst case for space, but this does not count the input space). If we consider the input space as well, then the space complexity is O(n). There are no additional data structures that grow with input size n used in the algorithm outside of the sorting algorithm's temporary space. The variables ans, j, and x use constant space.

**Solution 2: Sorting + Two Pointers (10 min)**

class Solution {

    public int countPairs(List<Integer> nums, int target) {

        int count = 0;

        Collections.sort(nums);

        int i = 0;

        int j = nums.size() - 1;

        while(i < j) {

            if(nums.get(i) + nums.get(j) < target) {

                count += j - i;

                i++;

            } else {

                j--;

            }

        }

        return count;

    }

}

**Refer to**

<https://leetcode.com/problems/count-pairs-whose-sum-is-less-than-target/solutions/3933451/two-pointers-approach-easy-to-understand-in-9-languages/>

**Intuition**

Problem: You're given an array of integers and a target value. The task is to count the number of pairs of elements in the array such that the sum of the pair is less than the target value.

The key insight for solving this problem is recognizing that if you have an array sorted in ascending order, you can efficiently find pairs that meet the given condition using a Two-Pointers Approach.

**Approach**

The Two-Pointers Approach is a common technique to solve problems involving arrays or sequences. In this case, you can use two pointers, often referred to as the "left" and "right" pointers, to traverse the array and find pairs that satisfy the given condition.

Here's a high-level overview of the approach:

1.) Sort the array in ascending order. Sorting helps in efficiently exploring pairs with sums less than the target value.

2.) Initialize two pointers, left and right, pointing to the first and last elements of the sorted array, respectively.

3.) Initialize a variable count to keep track of the count of valid pairs.

While the left pointer is less than the right pointer:

If the sum of the elements at left and right is less than the target value, it means all pairs with the current left element will also satisfy the condition. So, increment the count by right - left and move the left pointer to the right.

If the sum is greater than or equal to the target, move the right pointer to the left.

4.) Continue this process until the left pointer crosses the right pointer.

**Complexity**

Time complexity: O(nlogn)

Sorting the array takes O(n log n) time, where n is the number of elements in the array. The two-pointer traversal takes linear time, O(n), as each element is visited once.

Overall, the time complexity is dominated by the sorting step, making it O(n log n).

Space complexity: O(1)

The space complexity depends on the sorting algorithm used. If you're using an in-place sorting algorithm like Quicksort, the space complexity will be O(1) (negligible). If you're using an algorithm that requires additional space, the space complexity might be higher.

class Solution {

public:

int countPairs(vector<int>& nums , int target){ // function to count the pairs whose sum is less than target

sort(nums.begin() , nums.end()); // sort the vector nums

int count = 0; // variable to store the count

int left = 0; // variable to store the left

int right = nums.size()-1; // variable to store the right

while(left < right){ // loop until left is less than right

if(nums[left] + nums[right] < target){ // if nums[left] + nums[right] is less than target

count += right-left; // update the count

left++; // increment the left

}

else{ // else

right--; // decrement the right

}

}

return count; // return the count

}

};