<https://leetcode.com/problems/count-the-number-of-fair-pairs/description/>

Given a 0-indexed integer array nums of size n and two integers lower and upper, return the number of fair pairs.

A pair (i, j) is fair if:

0 <= i < j < n, and

lower <= nums[i] + nums[j] <= upper

**Example 1:**

**Input:** nums = [0,1,7,4,4,5], lower = 3, upper = 6

**Output:** 6

**Explanation:** There are 6 fair pairs: (0,3), (0,4), (0,5), (1,3), (1,4), and (1,5).

**Example 2:**

**Input:** nums = [1,7,9,2,5], lower = 11, upper = 11

**Output:** 1

**Explanation:** There is a single fair pair: (2,3).

**Constraints:**

1 <= nums.length <= 10^5

nums.length == n

-10^9 <= nums[i] <= 10^9

-10^9 <= lower <= upper <= 10^9

**Attempt 1: 2024-01-23**

**Solution 1: Sorting + Binary Search (60 min)**

**注意：在nums[i]做binary search的时候为了避免重复计算，binary search的区间是[i + 1, n - 1]而不是[0, n - 1]**

**Wrong Solution**

**错误原因：在nums[i]做binary search的时候重复计算，binary search的区间是[0, n - 1]**

**Test out by**

Input: nums = [1,7,9,2,5], lower = 11, upper = 11

Output: -3, Expect: 1

**Code**

class Solution {

    public long countFairPairs(int[] nums, int lower, int upper) {

        long count = 0;

        Arrays.sort(nums);

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

            // lower <= nums[i] + nums[j] <= upper

            // -> lower - nums[i] <= nums[j] <= upper - nums[i]

            // 'left' is the left most index 'j' satisfy nums[j] >= lower - nums[i]

            int left = findLowerBoundary(nums, lower - nums[i]);

            // 'right' is the right most index 'j' satisfy nums[j] <= upper - nums[i]

            int right = findUpperBoundary(nums, upper - nums[i]);

            count += right - left;

        }

        return count;

    }

    private int findLowerBoundary(int[] nums, int val) {

        int lo = 0;

        int hi = nums.length - 1;

        while(lo <= hi) {

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

            if(nums[mid] >= val) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        //if(lo == nums.length || nums[lo] != val) {

        //    return lo + 1;

        //}

        return lo;

    }

    private int findUpperBoundary(int[] nums, int val) {

        int lo = 0;

        int hi = nums.length - 1;

        while(lo <= hi) {

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

            if(nums[mid] > val) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        //if(lo == nums.length || nums[lo] != val) {

        //    return lo;

        //}

        return lo - 1;

    }

}

**Correct Solution**

class Solution {

        public long countFairPairs(int[] nums, int lower, int upper) {

        long count = 0;

        Arrays.sort(nums);

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

            // lower <= nums[i] + nums[j] <= upper

            // -> lower - nums[i] <= nums[j] <= upper - nums[i]

            // 'left' is the left most index 'j' satisfy nums[j] >= lower - nums[i]

            int left = findLowerBoundary(nums, lower - nums[i], i + 1);

            // 'right' is the right most index 'j' satisfy nums[j] <= upper - nums[i]

            int right = findUpperBoundary(nums, upper - nums[i], i + 1);

            count += right - left + 1;

        }

        return count;

    }

    private int findLowerBoundary(int[] nums, int val, int searchStartIndex) {

        int lo = searchStartIndex;

        int hi = nums.length - 1;

        while(lo <= hi) {

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

            if(nums[mid] >= val) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // No need to separate corner case out because if index

        // exceeds array length or not equal to target value, 'lo'

        // will be the position to insert that will satisfy

        // nums[lo] >= lower - nums[i]

        //if(lo == nums.length || nums[lo] != val) {

        //    return lo;

        //}

        return lo;

    }

    private int findUpperBoundary(int[] nums, int val, int searchStartIndex) {

        int lo = searchStartIndex;

        int hi = nums.length - 1;

        while(lo <= hi) {

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

            if(nums[mid] > val) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // No need to separate corner case out because if index

        // exceeds array length or not equal to target value,

        // 'lo - 1' will be the position to insert that will

        // satisfy nums[lo - 1] <= upper - nums[i]

        //if(lo == nums.length || nums[lo] != val) {

        //    return lo - 1;

        //}

        return lo - 1;

    }

}

Time Complexity: O(N\*logN)

Space Complexity: O(1)

**Why we start binary search with i + 1 ?**

**Refer to**

<https://leetcode.com/problems/count-the-number-of-fair-pairs/solutions/3174418/why-sorting-does-not-affect-order-intuition-and-thoughtprocess-explained-sorting-binary-search/>

**Idea of Why this way?**

Why to go with sorting and binsearch? we will see next

Does sorting Affect ordering? Yah, but here its NOT.

See they said 0<= i < j <= N

Say after SORTING, u affected the order and now j < i.

No issues at all, just assume j as i and i as j, so that j<i => i<j.

So basically the **breakdown of the Condition is i!=j**.

Why SORT?

By sorting we group all the ans to one side and making further process easy. TOWARDS RIGHT here.

Now we need to find HOW MANY VALUES are there in right to satisfy the range [lower, upper]

So, WHY SORTING is sorted in brain.

As we fixed Nums[i], So lowval to be searched is lower - nums[i].

Also upval to be searched is upper - nums[i].

**Why we do so? [1..i i+1......N-1]**

**We should not search 1...i-1, because its just DUPLICATION of Pairs, useless. As already one i selected and searched and counted**

In [i+1. .... N-1] find j and k such that [i+1...j....k....N-1]

nums[j] >= lowval and nums[k] <= upval so that we are always in RANGE [lower, upper]

DONE.

**Algo :**

sort it so that we group the ans to right side

now pick one value for i idx, then search for 2 Values to the right side.

Lower - nums[i] = Lower bound value

Upper - nums[i] = Upper bound value

Now Do 2 Binary search, as the part of array is sorted. [i+1...N-1]

Find value in nums that is >= Lower bound value. Minimize it.

Find value in nums that is <= Upper bound value. Maximize it.

So that we have bigger range.[j....k]. j and k should be as far as possible to get maximum number of pairs

k-j+1 will give you total nos of nums that satisfy the condition with inclusive of the nums in j and k indx too.

Add on to final ans.

Do the same for every i in nums.

We got the ans.

**Complexity**

Time complexity: O(NLogN+N∗2∗LogN)=O(NLogN)

Sorting

For every value in list, doing bin search from i+1 to N

Space complexity:O(1)

**Refer to**

<https://algo.monster/liteproblems/2563>

**Problem Description**

In this problem, you are given an array nums of integers that has n elements, and you're also given two integers lower and upper. Your task is to count the number of "fair pairs" in this array. A pair of elements (nums[i], nums[j]) is considered a "fair pair" if it fulfills two conditions:

The indices i and j must satisfy 0 <= i < j < n, meaning that i is strictly less than j and both are within the bounds of the array indices.

The sum of the elements at these indices, nums[i] + nums[j], must be between lower and upper, inclusive. That is, lower <= nums[i] + nums[j] <= upper.

You have to calculate and return the total number of such fair pairs present in the array.

**Intuition**

To approach the problem, we first observe that a brute-force solution would require checking all possible pairs and seeing if their sum falls within the specified range. This would result in an O(n^2) time complexity, which is not efficient for large arrays.

We can do better by first sorting nums. Once the array is sorted, we can use the two-pointer technique or binary search to find the range of elements that can pair with each nums[i] to form a fair pair. This is more efficient because when the array is sorted, we can make certain that if nums[i] + nums[j] is within the range, then nums[i] + nums[j+1] will only grow larger.

The provided solution takes advantage of the bisect\_left method from Python's bisect module. This method is used to find the insertion point for a given element in a sorted array to maintain the array's sorted order.

Here's the intuition behind the steps in the provided solution:

We first sort nums. Sorting allows us to use binary search, which dramatically reduces the number of comparisons needed to find fair pairs.

We iterate through nums with enumerate which gives us both the index i and the value x of each element in nums.

For each x in nums, we want to find the range of elements within nums that can be added to x to make a sum between lower and upper. To do this, we perform two binary searches with bisect\_left. The first binary search finds j, the smallest index such that the sum of x and nums[j] is at least lower. The second search finds k, the smallest index such that the sum of x and nums[k] is greater than upper.

The range of indices [j, k) in nums gives us all the valid j's that can pair with our current i to form fair pairs. We add k - j to our answer for each i.

Finally, after completing the loop, ans holds the total count of fair pairs, which we return.

By sorting the array and using binary search, we reduce the complexity of the problem. The sorting step is O(n log n) and the binary search inside the loop runs in O(log n) time for each element, so overall the algorithm runs significantly faster than a naive pairwise comparison approach.

**Solution Approach**

The solution uses Python's sorting algorithm and the bisect module as its primary tools. Here's a detailed walk-through of how the code works, with reference to the patterns, data structures, and algorithms used:

Sorting the Array: The nums.sort() line sorts the array in non-decreasing order. This is critical because it allows us to use binary search in the following steps. Sorting in Python uses the Timsort algorithm, which is a hybrid sorting algorithm derived from merge sort and insertion sort.

Enumerating through Sorted Elements: The for i, x in enumerate(nums): line iterates over the elements of the sorted nums array, obtaining both the index i and value x of each element.

Binary Search with bisect\_left: Uses the bisect\_left function from Python's bisect module to perform binary searches. This function is called twice:

Once to find j, the index of the first element in nums such that when added to x, the sum is not less than lower. The call is bisect\_left(nums, lower - x, lo=i + 1), which looks for the "left-most" position to insert lower - x in nums starting at index i+1 to keep the sorted order.

A second time to find k, the index where the sum of x and the element at this index is just greater than upper. The call is bisect\_left(nums, upper - x + 1, lo=i + 1), which is looking for the "left-most" insertion point for upper - x + 1 in nums starting at index i+1.

Counting Fair Pairs: The line ans += k - j calculates the number of elements between indices j and k, which is the count of all j indices that pair with the current i index to form a fair pair where lower <= nums[i] + nums[j] <= upper. Since nums is sorted, all elements nums[j] ... nums[k-1] will satisfy the condition with nums[i].

Return Final Count: After completing the loop over all elements, the ans variable holds the total count, which is then returned by the function return ans.

By utilizing the bisect\_left function for binary search, the code efficiently narrows down the search space for potential pairs, which is faster than a linear search. Moreover, the use of enumeration and range-based counting (k - j) makes the solution concise and readable. The overall complexity of the solution is O(n log n) due to the initial sorting and the subsequent binary searches inside the loop.

**Example Walkthrough**

Let's walk through a small example to illustrate how the solution finds the number of fair pairs.

**Given Input:**

nums = [1, 3, 5, 7]

lower = 4

upper = 8

**Steps:**

Sort the Array: First, we sort nums. The array nums is already in sorted order, so no changes are made here.

Iterate and Binary Search:

When i = 0 and x = 1, we search for:

j: Using bisect\_left(nums, lower - x, lo=i + 1), which evaluates to bisect\_left(nums, 3, lo=1). The function returns j = 1 because nums[1] = 3 is the first value where nums[1] + x >= lower.

k: Using bisect\_left(nums, upper - x + 1, lo=i + 1), we get bisect\_left(nums, 8, lo=1). This returns k = 4 because that's the index where inserting 8 would keep the array sorted, and there's no actual index 4 since the array length is 4 (0 indexed).

We calculate ans += k - j which is ans += 4 - 1, adding 3 to ans.

When i = 1 and x = 3, we search for:

j: bisect\_left(nums, 1, lo=2) and the function returns j = 2.

k: bisect\_left(nums, 6, lo=2) which returns k = 3 because that's the fitting place to insert 6 (just before 7).

We update ans to ans += 3 - 2, adding 1 to ans.

When i = 2 and x = 5, we do similar searches. No fair pairs can be made as there is only one element (7) after i, which does not satisfy the conditions, and the ans is not updated.

When i = 3 and x = 7, this is the last element, so no pairs can be made, and we don't update ans.

Return Final Count: Summing all the valid pairs, we have ans = 3 + 1 = 4. The function returns 4, which is the total count of fair pairs in the given array where the sum of pairs is within the range [lower, upper].

**Java Solution**

class Solution {

    // Counts the number of 'fair' pairs in the array, where a pair is considered fair

    // if the sum of its elements is between 'lower' and 'upper' (inclusive).

    public long countFairPairs(int[] nums, int lower, int upper) {

        // Sort the array to enable binary search

        Arrays.sort(nums);

        long count = 0; // Initialize count of fair pairs

        int n = nums.length;

        // Iterate over each element in the array

        for (int i = 0; i < n; ++i) {

            // Find the left boundary for the fair sum range

            int leftBoundaryIndex = binarySearch(nums, lower - nums[i], i + 1);

            // Find the right boundary for the fair sum range

            int rightBoundaryIndex = binarySearch(nums, upper - nums[i] + 1, i + 1);

            // Calculate the number of fair pairs with the current element

            count += rightBoundaryIndex - leftBoundaryIndex;

        }

        // Return the total count of fair pairs

        return count;

    }

    // Performs a binary search to find the index of the smallest number in 'nums'

    // starting from 'startIdx' that is greater or equal to 'target'.

    private int binarySearch(int[] nums, int target, int startIdx) {

        int endIdx = nums.length; // Sets the end index of the search range

        // Continue the loop until the search range is exhausted

        while (startIdx < endIdx) {

            int midIdx = (startIdx + endIdx) >> 1; // Calculate the mid index

            // If the mid element is greater or equal to target,

            // we need to continue in the left part of the array

            if (nums[midIdx] >= target) {

                endIdx = midIdx;

            } else {

                // Otherwise, continue in the right part

                startIdx = midIdx + 1;

            }

        }

        // Return the start index which is the index of the smallest

        // number greater or equal to 'target'.

        return startIdx;

    }

}

**Time and Space Complexity**

**Time Complexity**

The given Python code performs the sorting of the nums list, which takes O(n log n) time, where n is the number of elements in the list. After sorting, it iterates over each element in nums and performs two binary searches using the bisect\_left function.

For each element x in the list, it finds the index j of the first number not less than lower - x starting from index i + 1 and the index k of the first number not less than upper - x + 1 from the same index i + 1. The binary searches take O(log n) time each.

Since the binary searches are inside a loop that runs n times, the total time for all binary searches combined is O(n log n). This means the overall time complexity of the function is dominated by the sorting and binary searches, which results in O(n log n).

**Space Complexity**

The space complexity of the algorithm is O(1) if we disregard the input and only consider additional space because the sorting is done in-place and the only other variables are used for iteration and counting.

In the case where the sorting is not done in-place (depending on the Python implementation), the space complexity would be O(n) due to the space needed to create a sorted copy of the list. However, typically, the .sort() method on a list in Python sorts in-place, thus the typical space complexity remains O(1).