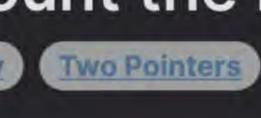
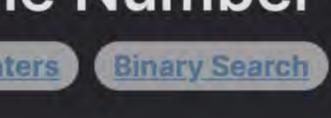
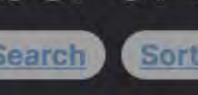
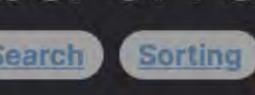


**Problem Description** 









Leetcode Link

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:

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

- 1. 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. 2. 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.</pre> 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

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

sum falls within the specified range. This would result in an O(n^2) time complexity, which is not efficient for large arrays.

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.

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

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

Solution Approach

merge sort and insertion sort.

function is called twice:

starting at index i+1.

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

- 3. 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.
- our answer for each i. 5. 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

4. 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

- 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.
- 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:

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

1. 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

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

o 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

all elements nums[j] ... nums[k-1] will satisfy the condition with nums[i]. 5. 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

4. 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,

**Example Walkthrough** 

• nums = [1, 3, 5, 7]• lower = 4 upper = 8 Steps:

### 2. Iterate and Binary Search:

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

length is 4 (0 indexed).

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

loop.

**Given Input:** 

■ 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.

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

pairs in the given array where the sum of pairs is within the range [lower, upper].

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

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

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

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

Python Solution

9

10

11

12

13

14

21

22

23

9

12

13

nums.sort()

fair\_pairs\_count = 0

return fair\_pairs\_count

Arrays.sort(nums);

int n = nums.length;

for (int i = 0; i < n; ++i) {

for index, num in enumerate(nums):

# Return the total number of fair pairs.

// Sort the array to enable binary search

// Iterate over each element in the array

# Iterate over each number to find suitable pairs.

# Find the left boundary for fair pairs.

# Find the right boundary for fair pairs.

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.

3. 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

■ 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

- from bisect import bisect\_left class Solution: def count\_fair\_pairs(self, nums: List[int], lower: int, upper: int) -> int: # Sort the list of numbers to leverage binary search advantage.
- right\_index = bisect\_left(nums, upper num + 1, lo=index + 1) 15 16 # Update the count of fair pairs by the number of elements that 17 # fall between the calculated left and right boundaries. 18 fair\_pairs\_count += right\_index - left\_index 19 20

## Java Solution 1 class Solution {

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

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

// Find the left boundary for the fair sum range

2 #include <algorithm> // Include algorithm library for sort, lower\_bound

// Initialize the answer (count of fair pairs) to 0

// Iterate through each element in the vector nums

// A fair pair (i, j) satisfies: lower <= nums[i] + nums[j] <= upper

long long countFairPairs(vector<int>& nums, int lower, int upper) {

// Find the first element in the range [i+1, nums.size()) which

// Find the first element in the range [i+1, nums.size()) which

// [lowerBoundIt, upperBoundIt), which are the eligible pairs.

// Increment the fair pair count by the number of elements in the range

// would form a pair with a sum just above upper limit.

fairPairCount += (upperBoundIt - lowerBoundIt);

auto lowerBoundIt = lower\_bound(nums.begin() + i + 1, nums.end(), lower - nums[i]);

auto upperBoundIt = upper\_bound(nums.begin() + i + 1, nums.end(), upper - nums[i]);

// could form a fair pair with nums[i], having a sum >= lower.

// Function to count the number of "fair" pairs

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

// Return the final count of fair pairs

return fairPairCount;

long long fairPairCount = 0;

// Sort the input vector nums

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

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

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

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

left\_index = bisect\_left(nums, lower - num, lo=index + 1)

```
14
15
               // Find the right boundary for the fair sum range
               int rightBoundaryIndex = binarySearch(nums, upper - nums[i] + 1, i + 1);
16
17
18
               // Calculate the number of fair pairs with the current element
19
               count += rightBoundaryIndex - leftBoundaryIndex;
20
21
22
           // Return the total count of fair pairs
23
           return count;
24
25
26
       // Performs a binary search to find the index of the smallest number in 'nums'
27
       // starting from 'startIdx' that is greater or equal to 'target'.
       private int binarySearch(int[] nums, int target, int startIdx) {
28
29
           int endIdx = nums.length; // Sets the end index of the search range
30
31
           // Continue the loop until the search range is exhausted
32
           while (startIdx < endIdx) {</pre>
33
               int midIdx = (startIdx + endIdx) >> 1; // Calculate the mid index
34
               // If the mid element is greater or equal to target,
35
               // we need to continue in the left part of the array
36
               if (nums[midIdx] >= target) {
37
                   endIdx = midIdx;
               } else {
38
                   // Otherwise, continue in the right part
39
40
                   startIdx = midIdx + 1;
41
42
43
44
           // Return the start index which is the index of the smallest
45
           // number greater or equal to 'target'.
46
           return startIdx;
47
48 }
49
C++ Solution
```

### 32 33 }; 34

Typescript Solution

1 #include <vector>

class Solution {

5 public:

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

```
1 // Counts the number of fair pairs in an array where the pairs (i, j) satisfy
 2 // lower <= nums[i] + nums[j] <= upper and i < j.</pre>
   function countFairPairs(nums: number[], lower: number, upper: number): number {
       // Binary search function to find the index of the first number in `sortedNums`
       // that is greater than or equal to `target`, starting the search from index `left`.
       const binarySearch = (target: number, left: number): number => {
            let right = nums.length;
           while (left < right) {</pre>
                const mid = (left + right) >> 1;
                if (nums[mid] >= target) {
10
                    right = mid;
11
12
                } else {
13
                    left = mid + 1;
14
15
            return left;
16
       };
17
18
       // Sort the array in non-descending order.
19
       nums.sort((a, b) \Rightarrow a - b);
20
21
22
       // Initialize the count of fair pairs to zero.
23
       let fairPairCount = 0;
24
25
       // Iterate through the array to count fair pairs.
       for (let i = 0; i < nums.length; ++i) {
26
           // Find the starting index 'j' for the valid pairs with nums[i]
            const startIdx = binarySearch(lower - nums[i], i + 1);
28
29
           // Find the ending index 'k' for the valid pairs with nums[i]
30
            const endIdx = binarySearch(upper - nums[i] + 1, i + 1);
31
           // The number of valid pairs with nums[i] is the difference between these indices
32
            fairPairCount += endIdx - startIdx;
33
34
35
       // Return the total count of fair pairs.
36
       return fairPairCount;
37 }
```

# Time Complexity

# Time and Space Complexity

The given Python code performs the sorting of the nums list, which takes 0(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.

k of the first number not less than upper -x + 1 from the same index i + 1. The binary searches take  $0(\log n)$  time each. Since the binary searches are inside a loop that runs n times, the total time for all binary searches combined is 0(n log n). This

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

**Space Complexity** 

The space complexity of the algorithm is 0(1) if we disregard the input and only consider additional space because the sorting is

means the overall time complexity of the function is dominated by the sorting and binary searches, which results in O(n log n).

# 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 0(1).