2563. Count the Number of Fair Pairs

Medium | Array | Two Pointers | Binary Search | Sorting | Leetcode Link

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:

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`.

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:

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`.
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`.
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 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 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:

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 merge sort and insertion sort.
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 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`.
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, 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 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:

1. **Sort the Array**: First, we sort `nums`. The array `nums` is already in sorted order, so no changes are made here.
2. **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`.
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 pairs in the given array where the sum of pairs is within the range `[lower, upper]`.

Python Solution

```python
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.
        nums.sort()

        fair_pairs_count = 0

        # Iterate over each number to find suitable pairs.
        for index, num in enumerate(nums):
            # Find the left boundary for fair pairs.
            left_index = bisect_left(nums, lower - num, lo=index + 1)
            # Find the right boundary for fair pairs.
            right_index = bisect_left(nums, upper - num + 1, lo=index + 1)

            # Update the count of fair pairs by the number of elements that
            # fall between the calculated left and right boundaries.
            fair_pairs_count += right_index - left_index

        # Return the total number of fair pairs.
        return fair_pairs_count

```

Java Solution

```java
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;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include algorithm library for sort, lower_bound

class Solution {
public:
    // Function to count the number of "fair" pairs
    // A fair pair (i, j) satisfies: lower <= nums[i] + nums[j] <= upper
    long long countFairPairs(vector<int>& nums, int lower, int upper) {
        // Initialize the answer (count of fair pairs) to 0
        long long fairPairCount = 0;

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

        // Iterate through each element in the vector nums
        for (int i = 0; i < nums.size(); ++i) {
            // Find the first element in the range [i+1, nums.size()) which
            // could form a fair pair with nums[i], having a sum >= lower.
            auto lowerBoundIt = lower_bound(nums.begin() + i + 1, nums.end(), lower - nums[i]);

            // Find the first element in the range [i+1, nums.size()) which
            // would form a pair with a sum just above upper limit.
            auto upperBoundIt = upper_bound(nums.begin() + i + 1, nums.end(), upper - nums[i]);

            // Increment the fair pair count by the number of elements in the range
            // [lowerBoundIt, upperBoundIt), which are the eligible pairs.
            fairPairCount += (upperBoundIt - lowerBoundIt);
        }

        // Return the final count of fair pairs
        return fairPairCount;
    }
};

```

Typescript Solution

```typescript
// Counts the number of fair pairs in an array where the pairs (i, j) satisfy
// lower <= nums[i] + nums[j] <= upper and i < j.
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) {
            const mid = (left + right) >> 1;
            if (nums[mid] >= target) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    };

    // Sort the array in non-descending order.
    nums.sort((a, b) => a - b);

    // Initialize the count of fair pairs to zero.
    let fairPairCount = 0;

    // Iterate through the array to count fair pairs.
    for (let i = 0; i < nums.length; ++i) {
        // Find the starting index 'j' for the valid pairs with nums[i]
        const startIdx = binarySearch(lower - nums[i], i + 1);
        // Find the ending index 'k' for the valid pairs with nums[i]
        const endIdx = binarySearch(upper - nums[i] + 1, i + 1);
        // The number of valid pairs with nums[i] is the difference between these indices
        fairPairCount += endIdx - startIdx;
    }

    // Return the total count of fair pairs.
    return fairPairCount;
}

```

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)`.