2824. Count Pairs Whose Sum is Less than Target

Easy Array Two Pointers Sorting Leetcode Link

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:

1. **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.
2. **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.
3. **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`.
4. **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:

1. **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.

   ```python
   nums.sort()
   ```

2. **Binary Search**: The binary search is done using Python's `bisect_left` method from the `bisect` module.

   ```python
   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.

3. **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`.

   ```python
   ans = 0
   for j, x in enumerate(nums):
       i = bisect_left(nums, target - x, hi=j)
       ans += i
   ```

4. **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.

   ```python
   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:

1. **Sort the Array**: We start by sorting `nums` to get `[1, 3, 5, 7]`.
2. **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.
3. **Counting Pairs**:
   - For each `j`, we add `i` to our total count `ans`.
   - From our steps: `ans = 1 + 1 + 0 = 2`.
4. **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.

Python Solution

```python
from bisect import bisect_left
from typing import List

class Solution:
    def countPairs(self, nums: List[int], target: int) -> int:
        # Sort the list of numbers first to use binary search
        nums.sort()
        count = 0  # Initialize count of pairs

        # Iterate through the sorted list
        for index, value in enumerate(nums):
            # Determine the index in the list where the pair's complement would be inserted
            # to maintain sorted order. Only consider elements before the current one.
            insertion_point = bisect_left(nums, target - value, hi=index)

            # Add the number of eligible pair counts.
            # Since we're searching in a sorted list up to the current index, all indices
            # before the insertion point are valid pairs with the current value.
            count += insertion_point

        return count  # Return the total count of pairs

```

Java Solution

```java
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;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to count pairs with a sum equal to a given target.
    int countPairs(vector<int>& nums, int target) {
        // First, we sort the input vector which enables us to use binary search.
        sort(nums.begin(), nums.end());

        // This variable will hold the count of valid pairs.
        int pairCount = 0;

        // Iterate through the sorted vector to find valid pairs.
        for (int rightIndex = 0; rightIndex < nums.size(); ++rightIndex) {
            // For each element at rightIndex, find the first number in the range [0, rightIndex)
            // that, when added to nums[rightIndex], would equal at least the target.
            // lower_bound returns an iterator pointing to the first element not less than target - nums[rightIndex].
            int leftIndex = lower_bound(nums.begin(), nums.begin() + rightIndex, target - nums[rightIndex]) - nums.begin();

            // The number of valid pairs for this iteration is the index found by lower_bound (leftIndex),
            // because all previous elements (0 to leftIndex-1) paired with nums[rightIndex] will have a sum less than target.
            pairCount += leftIndex;
        }

        // Return the total count of valid pairs.
        return pairCount;
    }
};

```

Typescript Solution

```typescript
// Counts the number of pairs in the 'nums' array that add up to the given 'target'.
function countPairs(nums: number[], target: number): number {
    // Sort the array in ascending order to facilitate binary search.
    nums.sort((a, b) => a - b);
    let pairCount = 0; // Initialize the count of pairs.

    // A binary search function to find the index of the smallest number in 'nums'
    // that is greater than or equal to 'x', up to but not including index 'rightLimit'.
    function binarySearch(x: number, rightLimit: number): number {
        let left = 0;
        let right = rightLimit;
        while (left < right) {
            // Calculate the middle index.
            const mid = Math.floor((left + right) / 2);
            if (nums[mid] >= x) {
                // If the element at 'mid' is greater than or equal to 'x',
                // narrow down the search to the left half including 'mid'.
                right = mid;
            } else {
                // Otherwise, narrow down the search to the right half excluding 'mid'.
                left = mid + 1;
            }
        }
        // Return the index of the smallest number greater than or equal to 'x'.
        return left;
    }

    // Iterate through the sorted array to find all pairs that meet the condition.
    for (let j = 0; j < nums.length; ++j) {
        // Use the binary search function to find the number of elements
        // that can be paired with 'nums[j]' to be less than the 'target'.
        const index = binarySearch(target - nums[j], j);
        // Add the number of valid pairs to 'pairCount'.
        pairCount += index;
    }

    // Return the total count of valid pairs.
    return pairCount;
}

```

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.