1885. Count Pairs in Two Arrays

Medium Array Binary Search Sorting Leetcode Link

Problem Description

The problem is to determine the number of specific pairs of indices in two arrays, `nums1` and `nums2`. Both arrays have the same length `n`. A valid pair `(i, j)` must satisfy two conditions: `i < j` (meaning the first index must be less than the second one), and `nums1[i] + nums1[j] > nums2[i] + nums2[j]` (meaning the sum of `nums1` elements at these indices is greater than the sum of `nums2` elements at the same indices). We need to count how many such pairs exist.

The task is, given the two arrays `nums1` and `nums2`, to return the total count of pairs `(i, j)` that meet these conditions.

Intuition

To solve this problem, we leverage the fact that if we fix one index and sort all the potential pair values, we can then use binary search to efficiently find how many values meet our condition for each fixed index. Here's a step-by-step explanation of the solution:

1. First, compute the difference between the elements at the same indices in `nums1` and `nums2` and store these differences in a new array `d`. This transformation simplifies the problem, as we're now looking for indices where `d[i] + d[j] > 0`.
2. Sort the array `d` in non-decreasing order. Sorting enables us to use binary search to efficiently find indices satisfying our condition.
3. For each element `d[i]` in `d`, we want to find the count of elements `d[j]` such that `j > i` and `d[i] + d[j] > 0`. We can rewrite this condition to `d[j] > -d[i]`. Using binary search on the sorted array `d`, we look for the right-most position to which `-d[i]` could be inserted while maintaining the sorted order. This gives us the index beyond which all elements of `d[j]` would result in `d[i] + d[j] > 0`.
4. The `bisect_right` function from Python's `bisect` module is used for this purpose. For each `i`, it returns the index beyond which `-d[i]` would go in the sorted array.
5. The count of valid `j` for each `i` is the number of elements in `d` beyond the index found in step 4, which is simply `n - (index found by bisect_right)`.
6. The total count of valid pairs is obtained by summing the count from step 5 for each `i`.

Using this method, we reduce a potentially O(n^2) problem (checking each pair directly) to O(n log n) due to sorting and binary search for each element.

Solution Approach

The implemented solution follows these steps, using a mix of algorithmic techniques and Python-specific functionalities:

1. **Difference Calculation and Store in Array `d`:** The first step is to calculate the difference array `d`, where each element `d[i]` is the difference between `nums1[i]` and `nums2[i]` for `i` from `0` to `n-1`. This subtraction is done using a list comprehension, which is a concise way to create lists in Python.

```
d = [nums1[i] - nums2[i] for i in range(n)]
```

2. **Sorting the Array `d`:** The array `d` is then sorted in non-decreasing order. This sorting is crucial as it prepares the array for a binary search operation. The sorted property of `d` allows us to apply the bisect algorithm effectively.

```
d.sort()
```

3. **Using Binary Search to Find Count of Valid Pairs:** For each element in `d`, we use the `bisect_right` function from the `bisect` module to find the insertion point for `-d[i]` into `d` such that the array remains sorted.

   The function `bisect_right` is a binary search algorithm that returns the index in the sorted list `d`, where the value `-d[i]` should be inserted to maintain the sorted order. The `lo` parameter signifies the start position for the search which in this case is `i + 1`, ensuring that `j > i`.

   The subtraction from `n` gives us the number of elements larger than `-d[i]`, effectively counting how many `j` indices will satisfy the condition `d[i] + d[j] > 0`.

```
sum(n - bisect_right(d, -v, lo=i + 1) for i, v in enumerate(d))
```

4. **Summing the Counts for Each `i`:** The sum operation in the final line adds up the valid pairs count for each value of `i`. It iterates over the sorted array `d` and for each element calculates the number of valid pairs `(i, j)` where `i < j` and the sum of the original elements from `nums1` at these indices is greater than the sum of the elements from `nums2` at the same indices, which corresponds to the condition `d[i] + d[j] > 0` post-transformation.

This solution uses a combination of algorithmic concepts:

- **Transformation**: to simplify the original condition to a more manageable form.
- **Sorting**: to prepare data for efficient searching.
- **Binary Search**: to reduce the search space for the pairs from O(n) to O(log n), greatly enhancing the overall algorithm efficiency.
- **Prefix Sum**: implicit in the adding up of counts for each index, effectively reducing the number of direct comparisons needed.

Returning the sum at the end gives us the desired count of pairs that fulfill the problem's conditions efficiently.

Example Walkthrough

Let's use a small example with the arrays `nums1 = [3, -1, 7]` and `nums2 = [4, 0, 5]` to illustrate the solution approach step-by-step. The length of both arrays, `n`, is 3. Our goal is to find the count of valid pairs `(i, j)` for which `i < j` and `nums1[i] + nums1[j] > nums2[i] + nums2[j]`.

Step 1: Calculate difference array `d`

First, find the difference between corresponding elements of `nums1` and `nums2`:

- `d[0] = nums1[0] - nums2[0] = 3 - 4 = -1`
- `d[1] = nums1[1] - nums2[1] = -1 - 0 = -1`
- `d[2] = nums1[2] - nums2[2] = 7 - 5 = 2`

So the difference array `d` is `[-1, -1, 2]`.

Step 2: Sort the array `d`

We sort the array `d` to get `[-1, -1, 2]`. In this small case, sorting does not change the order, as the list is already in non-decreasing order.

Step 3: Use binary search for each `i`

We use `bisect_right` to find where `-d[i]` can be inserted:

- For `i = 0 (d[0] = -1)`: We search for where `1` can be inserted after index 0. `bisect_right([-1, -1, 2], 1, lo=0 + 1) = 3`.
- For `i = 1 (d[1] = -1)`: We search for where `1` can be inserted after index 1. `bisect_right([-1, -1, 2], 1, lo=1 + 1) = 3`.
- We do not search for `i = 2` because it's the last element, and no `j` can satisfy `i < j`.

Step 4: Calculate the valid `j` indices and sum them up

- For `i = 0`: The count of valid `j` indices is `n - 3 = 3 - 3 = 0`.
- For `i = 1`: The count of valid `j` indices is `n - 3 = 3 - 3 = 0`.

The total count of valid pairs `(i, j)` is the sum of the counts above: `0 + 0 = 0`.

Therefore, for the given arrays `nums1` and `nums2`, there are no valid pairs `(i, j)` that meet the condition `nums1[i] + nums1[j] > nums2[i] + nums2[j]`.

Python Solution

```python
from typing import List
from bisect import bisect_right

class Solution:
    def countPairs(self, nums1: List[int], nums2: List[int]) -> int:
        # Length of the input lists
        length = len(nums1)
        # Calculate the difference between the two lists element-wise
        differences = [nums1[i] - nums2[i] for i in range(length)]
        # Sort the differences to prepare for binary search
        differences.sort()

        # Initialize count of pairs to 0
        count = 0
        # Loop through the sorted differences list
        for i, value in enumerate(differences):
            # For each element, find the number of elements in the sorted
            # list that would create a negative sum with the current element.
            # The `bisect_right` function is used to find the position to
            # insert `-value` which gives the number of such elements.
            # Subtract this position from the total number of elements that
            # can be paired with the current element, which is (length - i - 1).
            # We use `lo=i+1` because we shouldn't pair an element with itself.
            count += length - bisect_right(differences, -value, lo=i + 1)

        # Return the total count of valid pairs
        return count

```

Java Solution

```java
class Solution {
    public long countPairs(int[] nums1, int[] nums2) {
        // Get the length of the arrays
        int n = nums1.length;

        // Create a new array to store the differences between nums1 and nums2
        int[] differences = new int[n];
        for (int i = 0; i < n; ++i) {
            differences[i] = nums1[i] - nums2[i];
        }

        // Sort the array of differences
        Arrays.sort(differences);

        // Initialize answer to count the valid pairs
        long answer = 0;

        // Iterate over each element in the differences array
        for (int i = 0; i < n; ++i) {
            // Use binary search to find the number of valid pairs
            int left = i + 1, right = n;
            while (left < right) {
                int mid = (left + right) / 2;
                // Check if this position contributes to a valid pair
                if (differences[mid] > -differences[i]) {
                    right = mid;
                } else {
                    left = mid + 1;
                }
            }
            // Add the count of valid pairs for this position to the answer
            answer += n - left;
        }

        // Return the total number of valid pairs
        return answer;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    long long countPairs(vector<int>& nums1, vector<int>& nums2) {
        // Get the size of the input vectors
        int size = nums1.size();

        // Create a difference vector to store differences of nums1[i] - nums2[i]
        vector<int> diff(size);

        // Populate the difference vector
        for (int i = 0; i < size; ++i) {
            diff[i] = nums1[i] - nums2[i];
        }

        // Sort the difference vector in non-decreasing order
        sort(diff.begin(), diff.end());

        // Initialize result variable to store the final count of pairs
        long long result = 0;

        // Iterate through each element in the difference vector
        for (int i = 0; i < size; ++i) {
            // Find the index of the first element that is greater than -diff[i]
            // This is done to ensure that for any pair (i, j), diff[i] + diff[j] > 0
            int j = upper_bound(diff.begin() + i + 1, diff.end(), -diff[i]) - diff.begin();

            // Increment the result by the number of valid pairs with the current element at index i
            result += size - j;
        }

        // Return the computed number of valid pairs
        return result;
    }
};

```

Typescript Solution

```typescript
function countPairs(nums1: number[], nums2: number[]): bigint {
    // Get the size of the input arrays
    const size: number = nums1.length;

    // Create a difference array to store differences of nums1[i] - nums2[i]
    let diff: number[] = new Array<number>(size);

    // Populate the difference array
    for (let i = 0; i < size; i++) {
        diff[i] = nums1[i] - nums2[i];
    }

    // Sort the difference array in non-decreasing order (ascending)
    diff.sort((a, b) => a - b);

    // Initialize result variable to store the final count of pairs
    let result: bigint = BigInt(0);

    // Iterate through each element in the difference array
    for (let i = 0; i < size; i++) {
        // Find the index of the first element that is strictly greater than -diff[i]
        // This is done to ensure that for any pair (i, j), diff[i] + diff[j] > 0
        let j: number = findUpperBound(diff, i + 1, size, -diff[i]);

        // Increment the result by the number of valid pairs with the current element at index i
        result += BigInt(size - j);
    }

    // Return the computed number of valid pairs
    return result;
}

function findUpperBound(arr: number[], start: number, end: number, value: number): number {
    // Binary search for the first element in the sorted array that is strictly greater than the given value
    let low: number = start;
    let high: number = end;

    while (low < high) {
        const mid: number = low + Math.floor((high - low) / 2);
        if (arr[mid] <= value) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }

    // Return the index where the value would be inserted (first index greater than the value)
    return low;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code can be broken down into several steps:

1. Create a difference list `d` by subtracting `nums2` from `nums1`. This step is `O(n)` where `n` is the length of the input lists.
2. Sort the difference list `d`. Sorting algorithms generally have a time complexity of `O(n log n)`.
3. For each element in `d`, perform a binary search using `bisect_right`. Since we perform a binary search (`O(log n)`) for each element in the list, this step has a time complexity of `O(n log n)`.

Adding these up, the overall time complexity is dominated by the sorting and binary search steps, which leads to `O(n log n)`.

Space Complexity

The space complexity is evaluated as follows:

1. We are creating a difference list `d` of size `n`, therefore requiring `O(n)` additional space.
2. Sorting the list in-place (as Python's sort does) has a space complexity of `O(log n)`, as typical implementations of sorting algorithms, like Timsort (used by Python's sort method), use `O(log n)` space.
3. The binary search itself does not use additional space (aside from a few pointers), so the space used remains `O(log n)`.

As the additional space required for the difference list is the largest contributor, the overall space complexity is `O(n)`.