Documentation for 4Sum II - Optimized Solution

Table of Contents

1. **Problem Statement**

We are given four integer arrays, nums1, nums2, nums3, and nums4, each of length n. Our task is to find the number of tuples (i, j, k, l) such that:

nums1[i]+nums2[j]+nums3[k]+nums4[l]=0 $nums1[i] + nums2[j] + nums3[k] + nums4[l] = 0$

where $0 \le i, j, k, l < n$.

Example 1:

Input:

nums1 = [1,2]
nums2 = [-2,-1]
nums3 = [-1,2]
nums4 = [0,2]

Output: 2

Explanation:

The valid tuples are:

1. (0, 0, 0, 1): 1 + (-2) + (-1) + 2 = 0
2. (1, 1, 0, 0): 2 + (-1) + (-1) + 0 = 0

Example 2:

Input:

nums1 = [0]

nums2 = [0]

nums3 = [0]

nums4 = [0]

Output: 1

Constraints:

- 1 <= n <= 200
- -2^28 <= nums1[i], nums2[i], nums3[i], nums4[i] <= 2^28

2. **Intuition**

Given four lists, a brute-force approach iterating through all possible (i, j, k, l) would take $O(n^4)$ time, which is infeasible for n = 200.

Instead, we can break the problem into two smaller subproblems:

- Compute all pairwise sums from nums1 and nums2, storing their counts in a hashmap.
- Iterate over all pairs from nums3 and nums4, checking how many times their negative sum exists in the hashmap.

This reduces the complexity to $O(n^2)$, which is much more efficient.

3. Key Observations

i. Instead of iterating over all $O(n^4)$ combinations, we precompute the sums of nums1 and nums2 and store them in a dictionary.
ii. We then search for complementary sums using nums3 and nums4 in constant time $O(1)$.
iii. Using a hashmap (dict), we efficiently count occurrences of sum pairs, making lookups faster.

4. Approach

Step 1: Compute Pair Sums for nums1 and nums2

- Iterate over all pairs (a, b) from nums1 and nums2.
- Store sum(a, b) in a hashmap with the count of occurrences.

Step 2: Find Complementary Pairs from nums3 and nums4

- Iterate over all pairs (c, d) from nums3 and nums4.
- Check if $-(c + d)$ exists in the hashmap and increment the count.

Step 3: Return the Final Count

- The total count of valid tuples is stored in count.

5. Edge Cases

i. All zeros: [0, 0, 0, 0] → The sum of any four elements is 0, resulting in n^4 valid tuples.
ii. No possible tuples: Lists where no valid (i, j, k, l) sum to zero.
iii. Mixed large positive and negative numbers: Ensures algorithm handles a wide range of values.
iv. Large n values (200): Ensures the optimized $O(n^2)$ approach performs efficiently.

6. Complexity Analysis

Time Complexity

- Constructing sum_counts: $O(n^2)$
- Checking complementary pairs: $O(n^2)$
- Total: $O(n^2) + O(n^2) = O(n^2) \rightarrow$ Efficient for $n \leq 200$

Space Complexity

- sum_counts stores $O(n^2)$ unique sums.
- Total space: $O(n^2)$.

7. Alternative Approaches

1. Brute Force ($O(n^4)$)

- Try all (i, j, k, l) combinations.
- Too slow for n = 200.

2. Sorting + Two Pointers ($O(n^3)$)

- Sort arrays and use two-pointer search.
- Still too slow for large n.

8. Test Cases

Basic Test Cases

```
solution = Solution()
print(solution.fourSumCount([1,2], [-2,-1], [-1,2], [0,2]))  # Output: 2
print(solution.fourSumCount([0], [0], [0], [0]))  # Output: 1
```

Edge Case Tests

```
print(solution.fourSumCount([1,1,1], [-1,-1,-1], [-1,-1,-1], [1,1,1]))  # Output: 27
print(solution.fourSumCount([100]*200, [-100]*200, [50]*200, [-50]*200))  # Large n = 200
```

9. Final Thoughts

- The optimized $O(n^2)$ approach using hashmaps makes this problem solvable within constraints.
- Using dictionary lookups reduces the need for nested loops, improving performance.
- Always check for edge cases like large inputs, negative values, and all-zero arrays.