4Sum II

Problem Description

Given four integer arrays `nums1`, `nums2`, `nums3`, and `nums4`, all of length **n**, return the number of tuples `(i, j, k, l)` such that:

- $0 \leq i, j, k, l < n$
- `nums1[i] + nums2[j] + nums3[k] + nums4[l] == 0`

Example Explanation

Input

```
2
1 2
-2 -1
-1 2
0 2
```

Explanation

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

We need to find combinations that sum to 0:

1. `nums1[0] + nums2[1] + nums3[0] + nums4[1] = 1 + (-1) + (-1) + 1 = 0` (Wait, let's re-verify sample values based on input provided in prompt)

- Sample Logic verification:
- Tuple 1: (0, 0, 0, 1) -> `1 + (-2) + (-1) + 2 = 0`
- Tuple 2: (1, 1, 0, 0) -> `2 + (-1) + (-1) + 0 = 0`

There are **2** such tuples.

Output

```
2
```

Input

```
1
0
0
0
0
```

Explanation

- All arrays are `[0]`.
- Sum `0 + 0 + 0 + 0 = 0`.
- Only 1 tuple exists.

Output

```
1
```

Constraints & Key Observations

- `1 <= n <= 200`
- `-2^28 <= values <= 2^28`
- Time Limit: 1000 ms (1 second)
- **Naive Approach:** Four nested loops would result in . With , operations, which will definitely exceed the time limit (TLE).
- **Target Complexity:** We need something significantly faster, likely .
- **Memory:** We can trade space for time using a Hash Map.

Intuition

The equation is .

We can rearrange this equation to:

This simple algebraic manipulation suggests a **"Meet-in-the-Middle"** strategy:

1. Calculate all possible sums of pairs from the first two arrays (`nums1` and `nums2`).
2. Store the frequency of each sum in a Hash Map.
3. Calculate all possible sums of pairs from the last two arrays (`nums3` and `nums4`).
4. For each sum from the second half, check if its negation exists in the Hash Map.

This breaks the problem from one big task into two smaller tasks.

Approaches

Approach 1: Brute Force (Naive)

Explanation Iterate through every possible combination of indices `i, j, k, l` using four nested loops.

Why It Works It checks every single possibility, so correctness is guaranteed.

Why It Fails

- **Time Complexity:** .
- For , operations billion. A typical judge handles operations per second. This is far too slow.

Code

```
# Pseudo-code for logic
count = 0
for a in nums1:
    for b in nums2:
        for c in nums3:
            for d in nums4:
                if a + b + c + d == 0:
                    count += 1
print(count)
```

Approach 2: Hash Map / Meet-in-the-Middle (Optimal)

Explanation

1. Create a dictionary (Hash Map) to store sums from `nums1` and `nums2`.
2. Iterate through `nums1` and `nums2`. For every pair sum `s = a + b`:

- Increment the count of `s` in the map.

3. Initialize `total_count = 0`.
4. Iterate through `nums3` and `nums4`. For every pair sum `s = c + d`:

- We need `(a + b)` to equal `-(c + d)`.
- Look up `target = -s` in the map.
- Add `map[target]` to `total_count`.

Why It Works We effectively check all combinations, but we group them. Instead of matching one element against three others (), we match pairs against pairs ().

Code

```
import sys

def solve():
    try:
        # Handling input based on the sample format
        input_data = sys.stdin.read().split()
        if not input_data: return

        iterator = iter(input_data)
        n = int(next(iterator))

        nums1 = [int(next(iterator)) for _ in range(n)]
        nums2 = [int(next(iterator)) for _ in range(n)]
        nums3 = [int(next(iterator)) for _ in range(n)]
        nums4 = [int(next(iterator)) for _ in range(n)]

    except StopIteration:
        return

    # 1. Store sums of first two arrays
    sum_map = {}
    for a in nums1:
        for b in nums2:
            s = a + b
            sum_map[s] = sum_map.get(s, 0) + 1

    # 2. Check sums of last two arrays against the map
    count = 0
    for c in nums3:
        for d in nums4:
            target = -(c + d)
            if target in sum_map:
                count += sum_map[target]

    print(count)

if __name__ == "__main__":
    solve()
```

Time Complexity

- Building Map: (nested loop over first two arrays).
- Checking Map: (nested loop over last two arrays).
- Total: ****. For , operations , which is instant.

Space Complexity

- **** — In the worst case (all pair sums are unique), the hash map stores entries. With , this is integers, which fits easily in memory.

Edge Cases & Common Pitfalls

- **Zero Sums:** Multiple different combinations might sum to the same value. Using a frequency map (count of occurrences) handles this correctly compared to a simple set.
- **Integer Overflow:** The problem constraints say values are up to . The sum of four such values fits within a standard 64-bit integer (and even 32-bit if signs cancel out, but Python handles arbitrary precision integers automatically).
- **N=0:** If arrays are empty, loops won't execute, result 0 is printed correct. (Though constraint says).

When Not to Use This Approach

- If **Space** is extremely tight and you cannot afford memory. In that case, you might sort lists and use pointers, but for 4Sum, space is standard.

"'