# 454. 4Sum II

Medium Array Hash Table Leetcode Link

## Problem Description

The problem provides us with four integer arrays `nums1`, `nums2`, `nums3`, and `nums4`, each of the same length `n`. Your task is to find the number of quadruplets `(i, j, k, l)` such that the sum of the elements at these indices from each array equals zero, that is `nums1[i] + nums2[j] + nums3[k] + nums4[l] == 0`. The indices `i, j, k, l` vary from `0` to `n-1`.

## Intuition

To find quadruplets that sum up to zero, a naive approach would involve checking all possible combinations of indices `(i, j, k, l)` from the four arrays, which leads to a solution with a time complexity of `O(n^4)`. However, this isn't efficient when `n` is large.

Instead, we can first consider pairs of elements from `nums1` and `nums2`. We record the possible sums that these pairs can make and the frequency of each sum. A `Counter` in Python can be used effectively for this purpose.

Then, we can look for the pairs of elements from `nums3` and `nums4` that, when added together, create a sum that negates the sum we obtained from `nums1` and `nums2`. This way, the total sum will be zero. We find these complementary sums by iterating through all combinations of `nums3` and `nums4` and checking if the negated sum is present in our `Counter` (which is populated by sums of `nums1` and `nums2`).

Using this method significantly reduces the time complexity because we only consider two loops of `n^2` operations each, hence `O(n^2)`. Furthermore, since we are storing the sum and frequency in a `Counter` (which is a hashmap, effectively), our lookups for the complementary sum are done in constant time.

## Solution Approach

The implementation of the solution uses a two-step process that takes advantage of the `Counter` class in Python, which is a type of dictionary designed for counting hashable objects, a subclass of `dict`. Here's a breakdown of the algorithm:

### Step 1: Compute Pair Sums from `nums1` and `nums2`

Firstly, we iterate over all pairs of elements `(a, b)` where `a` is from `nums1` and `b` is from `nums2`. We calculate their sum and store this in a `Counter` dictionary, which will hold the sum as keys and the frequency of these sums as values. This dictionary will capture all possible pairwise sums along with how many times each sum occurs.

```python
cnt = Counter(a + b for a in nums1 for b in nums2)
```

### Step 2: Find Complementary Sums from `nums3` and `nums4`

Next, for each combination of elements `(c, d)` from `nums3` and `nums4`, we calculate their sum and look for the complement of this sum in our previously populated `Counter` (specifically, we're looking for `-(c + d)`). If found, it means there exists at least one pair from `nums1` and `nums2` that can be added to this `nums3` and `nums4` pair to make the total sum zero. The frequency stored in the `Counter` for `-(c + d)` represents how many such pairs we have from `nums1` and `nums2`.

```python
return sum(cnt[-(c + d)] for c in nums3 for d in nums4)
```

By iterating over all pairs from `nums3` and `nums4` and summing up the counts from the `Counter`, we get the total number of valid quadruplets that meet our condition.

This approach effectively reduces a complex problem into a simpler one that requires fewer computations by dividing it into two parts. It utilizes the power of hashmaps to speed up lookups for complementary pairs, ensuring an overall time complexity of `O(n^2)` which is a significant improvement over the brute force approach with `O(n^4)`. Also, because we are storing up to `n^2` pairs in the `Counter`, the space complexity is also `O(n^2)`.

```python
class Solution:
    def fourSumCount(self, nums1: List[int], nums2: List[int], nums3: List[int], nums4: List[int]) -> int:
        cnt = Counter(a + b for a in nums1 for b in nums2)
        return sum(cnt[-(c + d)] for c in nums3 for d in nums4)
```

Notice how the implementation is cogent and does not require nested loops over all four arrays, significantly enhancing efficiency.

### Example Walkthrough

Let's go through a small example using the provided solution approach to understand how it works in practice. We will take four arrays with a smaller length for simplicity.

Suppose `nums1 = [1, -1]`, `nums2 = [-1, 1]`, `nums3 = [0, 1]`, and `nums4 = [1, -1]`, and we need to find the number of quadruplets such that `nums1[i] + nums2[j] + nums3[k] + nums4[l] == 0`.

#### Step 1: Compute Pair Sums from `nums1` and `nums2`

Firstly, we calculate all possible sums of pairs `(a, b)` where `a` is from `nums1` and `b` is from `nums2`, and store these sums in the `Counter`:

- Pair `(1, -1)`: sum is `0`
- Pair `(1, 1)`: sum is `2`
- Pair `(-1, -1)`: sum is `-2`
- Pair `(-1, 1)`: sum is `0`

Thus, the `Counter` after step 1 will look like this: `{0: 2, 2: 1, -2: 1}`, which tells us that the sum `0` appears twice and the sums `2` and `-2` appear once.

#### Step 2: Find Complementary Sums from `nums3` and `nums4`

Now, we need to find pairs `(c, d)` where `c` is from `nums3` and `d` is from `nums4` such that `- (c + d)` is in the `Counter`.

- Pair `(0, 1)`: sum is `1` and its complement `-1` is not in the `Counter`, so this pair does not contribute.
- Pair `(0, -1)`: sum is `-1` and its complement `1` is not in the `Counter`, so no contribution from this pair either.
- Pair `(1, 1)`: sum is `2` and its complement `-2` is in the `Counter` and it appears once, so we have one quadruplet: `(1,1,1,1)`.
- Pair `(1, -1)`: sum is `0` and its complement `0` is in the `Counter` appearing twice, contributing two quadruplets: `(1,-1,1,-1)` and `(-1,1,1,-1)`.

By iterating over the last two arrays and summing up the counts for each valid complement in the `Counter`, we find a total of 3 quadruplets.

Hence, for the given small example, the `fourSumCount` method would return `3`. This illustrates that even for small arrays, the algorithm effectively combines pairs from the first two and last two arrays, utilizing the `Counter` to manage and efficiently count the complementary pairs, achieving a significant performance gain compared to the brute force approach.

## Python Solution

```python
from collections import Counter
from typing import List

class Solution:
    def fourSumCount(self, nums1: List[int], nums2: List[int], nums3: List[int], nums4: List[int]) -> int:
        # Create a Counter to store the frequency of the sums of pairs taken from nums1 and nums2
        pairwise_sum_count = Counter(a + b for a in nums1 for b in nums2)

        # Accumulate the number of times the current sum of pairs from nums3 and nums4
        # when added to the pair sums from nums1 and nums2 gives zero (i.e., sum to zero).
        count = 0
        for c in nums3:
            for d in nums4:
                # The target is the negative of the sum of c and d which would give zero when added to a pair sum from nums1 and nums
                target = -(c + d)
                # If the target exists in the Counter, add the frequency to the count
                count += pairwise_sum_count[target]

        return count

```

## Java Solution

```java
class Solution {

    // This method finds the count of tuples (i, j, k, l) such that nums1[i] + nums2[j] + nums3[k] + nums4[l] is zero.
    public int fourSumCount(int[] nums1, int[] nums2, int[] nums3, int[] nums4) {
        // HashMap to store the frequency of the sum of elements from nums1 and nums2
        Map<Integer, Integer> sumFrequencyMap = new HashMap<>();

        // Calculate all possible sums of pairs from nums1 and nums2 and store frequencies in the map
        for (int num1 : nums1) {
            for (int num2 : nums2) {
                sumFrequencyMap.merge(num1 + num2, 1, Integer::sum);
            }
        }

        // Initialize the count of valid tuples to 0
        int countOfValidTuples = 0;

        // For each possible pair of nums3 and nums4, check if the negative sum already exists in our map
        for (int num3 : nums3) {
            for (int num4 : nums4) {
                // The count is incremented by the frequency of the negated sum,
                // if it exists, indicating valid tuples that add up to zero
                countOfValidTuples += sumFrequencyMap.getOrDefault(-(num3 + num4), 0);
            }
        }

        // Return the total count of valid tuples
        return countOfValidTuples;
    }
}

```

## C++ Solution

```cpp
#include <vector>
#include <unordered_map>
using namespace std;

class Solution {
public:
    int fourSumCount(vector<int>& nums1, vector<int>& nums2, vector<int>& nums3, vector<int>& nums4) {
        // Create a hash map to store the frequency of the sum of pairs from nums1 and nums2
        unordered_map<int, int> sumCount;

        // Calculate all possible sums of pairs from nums1 and nums2, and record the frequency
        for (int num1 : nums1) {
            for (int num2 : nums2) {
                sumCount[num1 + num2]++;
            }
        }

        // Initialize the answer to 0. This will hold the number of tuples such that the sum is 0
        int count = 0;

        // For every pair from nums3 and nums4, check if the opposite number exist in the sumCount map
        for (int num3 : nums3) {
            for (int num4 : nums4) {
                // Find the complement of the current sum in the hash map
                auto it = sumCount.find(-(num3 + num4));

                // If the complement is found, this means there are tuples from nums1 and nums2
                // that, when added with the current pair from nums3 and num4, sum to 0
                if (it != sumCount.end()) {
                    // Add the frequency of the complement to the count
                    count += it->second;
                }
            }
        }

        // Return the total count of tuples resulting in a sum of 0
        return count;
    }
};

```

## Typescript Solution

```typescript
// Function to count the number of tuples (a, b, c, d) from four lists
// such that a + b + c + d is equal to 0.
function fourSumCount(nums1: number[], nums2: number[], nums3: number[], nums4: number[]): number {
    // Create a map to store the frequency of sums of pairs from the first two lists.
    const sumFrequency: Map<number, number> = new Map();

    // Calculate all possible sums from nums1 and nums2 and update the frequency map.
    for (const num1 of nums1) {
        for (const num2 of nums2) {
            const sum = num1 + num2;
            sumFrequency.set(sum, (sumFrequency.get(sum) || 0) + 1);
        }
    }

    // Initialize a variable to keep track of the number of valid tuples found.
    let tupleCount = 0;

    // For each possible sum of pairs from nums3 and nums4, check if the negation
    // of the sum is present in the frequency map. If so, increase the count of valid tuples.
    for (const num3 of nums3) {
        for (const num4 of nums4) {
            const sum = num3 + num4;
            tupleCount += sumFrequency.get(-sum) || 0;
        }
    }

    // Return the total count of valid tuples.
    return tupleCount;
}

```

## Time and Space Complexity

The given Python code is designed to find the number of tuples `(i, j, k, l)` such that `nums1[i] + nums2[j] + nums3[k] + nums4[l]` is zero.

### Time Complexity:

The time complexity is analyzed based on the number of operations performed by the code:

- The first part of the code creates a `Counter` object to count the frequency of sums of pairs taken from `nums1` and `nums2`. This involves iterating over each element in `nums1` and `nums2`, which, if the length of the lists is `n`, results in `n * n` or `n^2` operations.
- The second part computes the sum of frequencies of the complement of each possible sum in `nums3` and `nums4` that makes the total sum zero. This also requires `n * n` or `n^2` operations.

Combining these two, we get a total of `2 * n^2` operations, but since constants are neglected in Big O notation, the time complexity of the code is `O(n^2)`.

### Space Complexity:

The space complexity is considered based on the additional memory used by the program:

- The `Counter` object holds at most `n^2` entries, corresponding to each unique sum of elements from `nums1` and `nums2`.
- No other significant extra space is used for computation since the second sum is computed iteratively and sums are not stored.

Hence, the space complexity of the code is `O(n^2)`.