# Question

Given four lists A, B, C, D of integer values, compute how many tuples (i, j, k, l) there are such that A[i] + B[j] + C[k] + D[l] is zero.

To make problem a bit easier, all A, B, C, D have same length of N where 0 ≤ N ≤ 500. All integers are in the range of -228 to 228 - 1 and the result is guaranteed to be at most 231 - 1.

**Example:**

**Input:**

A = [ 1, 2]

B = [-2,-1]

C = [-1, 2]

D = [ 0, 2]

**Output:**

2

**Explanation:**

The two tuples are:

1. (0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0

2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0

# Solution

## **Solution**

This problem is a variation of [4Sum](https://leetcode.com/articles/4sum/), and we recommend checking that problem first. The main difference is that here we pick each element from a different array, while in 4Sum all elements come from the same array. For that reason, we cannot use the [Two Pointers](https://leetcode.com/articles/4sum/#approach-1-two-pointers) approach, where elements must be in the same sorted array.

On the bright side, we do not need to worry about using the same element twice - we pick one element at a time from each array. As you will see later, this help reduce the time complexity.

Finally, we do not need to return actual values and ensure they are unique; we just count each combination of four elements that sums to zero.

#### **Approach 1: Hashmap**

A brute force solution will be to enumerate all combinations of elements using four nested loops, which results in \mathcal{O}(n^4)O(*n*4) time complexity. A faster approach is to use three nested loops, and, for each sum a + b + c, search for a complementary value d == -(a + b + c) in the fourth array. We can do the search in \mathcal{O}(1)O(1) if we populate the fourth array into a hashmap.

Note that we need to track the frequency of each element in the fourth array. If an element is repeated multiple times, it will form multiple quadruples. Therefore, we will use hashmap values to store counts.

Building further on this idea, we can observe that a + b == -(c + d). First, we will count sums of elements a + b from the first two arrays using a hashmap. Then, we will enumerate elements from the third and fourth arrays, and search for a complementary sum a + b == -(c + d) in the hashmap.

1 / 8

**Algorithm**

1. For each a in A.
   * For each b in B.
     + If a + b exists in the hashmap m, increment the value.
     + Else add a new key a + b with the value 1.
2. For each c in C.
   * For each d in D.
     + Lookup key -(c + d) in the hashmap m.
     + Add its value to the count cnt.
3. Return the count cnt.

|  |
| --- |
| public int fourSumCount(int[] A, int[] B, int[] C, int[] D) {  int cnt = 0;  Map<Integer, Integer> m = new HashMap<>();  for (int a : A)  for (int b : B)  m.put(a + b, m.getOrDefault(a + b, 0) + 1);  for (int c : C)  for (int d : D)  cnt += m.getOrDefault(-(c + d), 0);  return cnt;  } |

**Complexity Analysis**

* Time Complexity: O(n^2). We have 2 nested loops to count sums, and another 2 nested loops to find complements.
* Space Complexity: O(n^2) for the hashmap. There could be up to O(n^2) distinct a + b keys.

#### **Approach 2: kSum II**

After you solve 4Sum II, an interviewer can follow-up with 5Sum II, 6Sum II, and so on. What they are really expecting is a generalized solution for k input arrays. Fortunately, the hashmap approach can be easily extended to handle more than 4 arrays.

Above, we divided 4 arrays into two equal groups, and processed each group independently. Same way, we will divide k*k* arrays into two groups. For the first group, we will have \frac{k}{2}2*k*​ nested loops to count sums. Another \frac{k}{2}2*k*​ nested loops will enumerate arrays in the second group and search for complements.

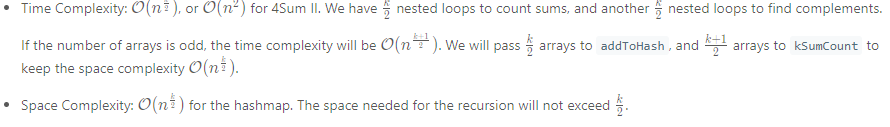
**Algorithm**

We can implement \frac{k}{2}2*k*​ nested loops using a recursion, passing the index i of the current list as the parameter. The first group will be processed by addToHash recursive function, which accumulates sum and terminates when adding the final sum to a hashmap m.

The second function, countComplements, will process the other group, accumulating the complement value. In the end, it searches for the final complement value in the hashmap and adds its count to the result.

|  |
| --- |
| public int fourSumCount(int[] A, int[] B, int[] C, int[] D) {  return kSumCount(new int[][]{A, B, C, D});  }  public int kSumCount(int[][] lists) {  Map<Integer, Integer> m = new HashMap<>();  addToHash(lists, m, 0, 0);  return countComplements(lists, m, lists.length / 2, 0);  }  void addToHash(int[][] lists, Map<Integer, Integer> m, int i, int sum) {  if (i == lists.length / 2)  m.put(sum, m.getOrDefault(sum, 0) + 1);  else  for (int a : lists[i])  addToHash(lists, m, i + 1, sum + a);  }  int countComplements(int[][] lists, Map<Integer, Integer> m, int i, int complement) {  if (i == lists.length)  return m.getOrDefault(complement, 0);  int cnt = 0;  for (int a : lists[i])  cnt += countComplements(lists, m, i + 1, complement - a);  return cnt;  } |

**Complexity Analysis**



#### **Further Thoughts**

For an interview, keep in mind the generalized implementation. Even if your interviewer is OK with a simpler code, you'll get some extra points by describing how your solution can handle more than 4 arrays.

It's also important to discuss trade-offs with your interviewer. If we are tight on memory, we can move some arrays from the first group to the second. This, of course, will increase the time complexity.

