454. 4Sum II

June 6, 2020 | 3.3K views

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 $0 \le N \le 500$. All integers are in the range of -2^{28} to $2^{28} - 1$ and the result is guaranteed to be at most $2^{31} - 1$.

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

This problem is a variation of 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 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.

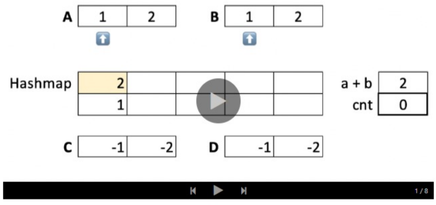
Finally, we do not need to return actual values and ensure they are unique; we just count each combination of four elements that sums to zero.

Approach 1: Hashmap

A brute force solution will be to enumerate all combinations of elements using four nested loops, which results in $\mathcal{O}(n^4)$ 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 $\mathcal{O}(1)$ 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.



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`.

C++ | Java | Python3 — Copy

```
class Solution:
    def fourSumCount(self, A: List[int], B: List[int], C: List[int], D: List[int]) -> int:
        cnt = 0
        m = {}
        for a in A:
            for b in B:
                m[a + b] = m.get(a + b, 0) + 1
        for c in C:
            for d in D:
                cnt += m.get(-(c + d), 0)
        return cnt
```

Complexity Analysis

- Time Complexity: $\mathcal{O}(n^2)$. We have 2 nested loops to count sums, and another 2 nested loops to find complements.
- Space Complexity: $\mathcal{O}(n^2)$ for the hashmap. There could be up to $\mathcal{O}(n^2)$ 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 arrays into two groups. For the first group, we will have $\frac{k}{2}$ nested loops to count sums. Another $\frac{k}{2}$ nested loops will enumerate arrays in the second group and search for complements.

Algorithm

We can implement $\frac{k}{2}$ 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.

C++ | Java | Python3 — Copy

```
class Solution:
    def fourSumCount(self, A: List[int], B: List[int], C: List[int], D: List[int]) -> int:
        m = {}

        def nSumCount(lists: List[List[int]]) -> int:
            addToHash(lists, 0, 0)
            return countComplements(lists, len(lists) // 2, 0)

        def addToHash(lists: List[List[int]], i: int, sum: int) -> None:
            if i == len(lists) // 2:
                m[sum] = m.get(sum, 0) + 1
            else:
                for a in lists[i]:
                    addToHash(lists, i + 1, sum + a)

        def countComplements(lists: List[List[int]], i: int, complement: int) -> int:
            if i == len(lists):
                return m.get(complement, 0)
            cnt = 0
            for a in lists[i]:
                cnt += countComplements(lists, i + 1, complement - a)
            return cnt

        return nSumCount([A, B, C, D])
```

Complexity Analysis

- Time Complexity: $\mathcal{O}(n^{\frac{k}{2}})$, or $\mathcal{O}(n^2)$ for 4Sum II. We have $\frac{k}{2}$ nested loops to count sums, and another $\frac{k}{2}$ nested loops to find complements.

  If the number of arrays is odd, the time complexity will be $\mathcal{O}(n^{\frac{k+1}{2}})$. We will pass $\frac{k}{2}$ arrays to `addToHash`, and $\frac{k+1}{2}$ arrays to `kSumCount` to keep the space complexity $\mathcal{O}(n^{\frac{k}{2}})$.
- Space Complexity: $\mathcal{O}(n^{\frac{k}{2}})$ for the hashmap. The space needed for the recursion will not exceed $\frac{k}{2}$.

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.

In other words, the time complexity can range from $\mathcal{O}(n^k)$ to $\mathcal{O}(n^{\frac{k}{2}})$, and the memory complexity ranges from $\mathcal{O}(1)$ to $\mathcal{O}(n^{\frac{k}{2}})$ accordingly.