medium.com

Leetcode 454. 4-Sum II - HAOYU LEI - Medium

HAOYU LEI

3-4 minutes

Link:

Problem Description

> 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 -2^{28} to $2^{28}–1$ and the result is guaranteed to be at most $2^{31}–1$.

Analysis

On first thought, it is very similar to 3-Sum problem. We break down 4-Sum into `O(n^2)` 2-Sum problems, which suffices an overall complexity of `O(n^3)`. However, this solution cannot pass all the test cases. Hence a more efficient solution is needed.

Since 4 = 2 + 2, we can think of 4-Sum problem as a variant of 2-Sum problem. Let E = A + B, F = C + D, the lengths of E and F are `O(n^2)`. Then we can find E + F = 0 in `O(O(n^2))=O(n^2)` time, given E and F are both sorted, which is true if we performed the addition cleverly. Bravo!

So it requires a careful design of how we do A+B and C+D, as well as how to store the result, to limit the time complexity within `O(n^2)` and make the overall algorithm faster than the initial proposal.

There are several ways to achieve this, in different languages:

1. Java: HashMap
2. Python: collections.counter
3. C++: unordered_map

Using map as data structure is ideal because there we care about time complexity of two operations: looking up a sum in a the data structure, if it exists, increase its counter by one; if it doesn’t, insert it. Looking up and insertion usually takes `O(1)` time, because our use case is simple.

Solution: Java

```
public int fourSumCount(int[] A, int[] B, int[] C, int[] D) {    // first part
    HashMap<Integer, Integer> E = new HashMap<Integer, Integer>();
    for (int a: A) {
        for (int b: B) {
            if ( Objects.isNull(E.get(a+b)) ) {
                E.put(a+b, 1);
            }
            else {
                int count = E.get(a+b);
                E.put(a+b, count+1);
            }
        }
    }
            HashMap<Integer, Integer> F = new HashMap<Integer,
Integer>();
    for (int c: C) {
        for (int d: D) {
            if ( Objects.isNull(F.get(c+d)) ) {
                F.put(c+d, 1);
            }
            else {
                int count = F.get(c+d);
                F.put(c+d, count+1);
            }
        }
    }
    // second part
    int count = 0;
    for ( int key: E.keySet() ) {
        if ( Objects.nonNull(F.get(-key))) {
            count += E.get(key) * F.get(-key);
        }
    }
    return count;
}
```

Final Note

1. For the second part, I wanted to use `forEach` method in HashMap

at first, before encountering the problem that we cannot modify a local variable neither in a lambda function or an inner class (`accept` method in`BiConsumer<Integer, Integer>`). So I used `keySet` instead.

2. Optimizing code: `HashMap.getOrDefault` method can be used to make the above code more efficient. Besides, introducing `F` is unnecessary, if we think more deeply about its usage.

   So the last optimization:

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

            return count;
}
```