<https://leetcode.com/problems/finding-pairs-with-a-certain-sum/description/>

You are given two integer arrays nums1 and nums2. You are tasked to implement a data structure that supports queries of two types:

Add a positive integer to an element of a given index in the array nums2.

Count the number of pairs (i, j) such that nums1[i] + nums2[j] equals a given value (0 <= i < nums1.length and 0 <= j < nums2.length).

Implement the FindSumPairs class:

FindSumPairs(int[] nums1, int[] nums2) Initializes the FindSumPairs object with two integer arrays nums1 and nums2.

void add(int index, int val) Adds val to nums2[index], i.e., apply nums2[index] += val.

int count(int tot) Returns the number of pairs (i, j) such that nums1[i] + nums2[j] == tot.

**Example 1:**

Input

["FindSumPairs", "count", "add", "count", "count", "add", "add", "count"]

[[[1, 1, 2, 2, 2, 3], [1, 4, 5, 2, 5, 4]], [7], [3, 2], [8], [4], [0, 1], [1, 1], [7]]

Output

[null, 8, null, 2, 1, null, null, 11]

Explanation

FindSumPairs findSumPairs = new FindSumPairs([1, 1, 2, 2, 2, 3], [1, 4, 5, 2, 5, 4]);

findSumPairs.count(7); // return 8; pairs (2,2), (3,2), (4,2), (2,4), (3,4), (4,4) make 2 + 5 and pairs (5,1), (5,5) make 3 + 4

findSumPairs.add(3, 2); // now nums2 = [1,4,5,4,5,4]

findSumPairs.count(8); // return 2; pairs (5,2), (5,4) make 3 + 5

findSumPairs.count(4); // return 1; pair (5,0) makes 3 + 1

findSumPairs.add(0, 1); // now nums2 = [2,4,5,4,5,4]

findSumPairs.add(1, 1); // now nums2 = [2,5,5,4,5,4]

findSumPairs.count(7); // return 11; pairs (2,1), (2,2), (2,4), (3,1), (3,2), (3,4), (4,1), (4,2), (4,4) make 2 + 5 and pairs (5,3), (5,5) make 3 + 4

**Constraints:**

1 <= nums1.length <= 1000

1 <= nums2.length <= 10^5

1 <= nums1[i] <= 10^9

1 <= nums2[i] <= 10^5

0 <= index < nums2.length

1 <= val <= 10^5

1 <= tot <= 10^9

At most 1000 calls are made to add and count each.

**Attempt 1: 2024-01-16**

**Wrong Solution**

**错误的原因是在add(index, val)方法中没有更新原来的nums2(或者其指针copy)，而是仅仅获取了nums2[index](或者copy[index])的值，这是错误的，因为add和count方法都是多次调用的，后面的add和count方法基于前面变更后的nums2数组**

class FindSumPairs {

    Map<Integer, Integer> map1;

    Map<Integer, Integer> map2;

    int[] copy;

    public FindSumPairs(int[] nums1, int[] nums2) {

        map1 = new HashMap<>();

        map2 = new HashMap<>();

        for(int num1 : nums1) {

            map1.put(num1, map1.getOrDefault(num1, 0) + 1);

        }

        for(int num2 : nums2) {

            map2.put(num2, map2.getOrDefault(num2, 0) + 1);

        }

        copy = nums2;

    }

    public void add(int index, int val) {

        //int key = nums2[index];

        int key = copy[index];

        int newKey = key + val;

        map2.put(key, map2.get(key) - 1);

        map2.put(newKey, map2.getOrDefault(newKey, 0) + 1);

    }

    public int count(int tot) {

        int result = 0;

        for(int key : map1.keySet()) {

            if(map2.containsKey(tot - key)) {

                result += map1.get(key) \* map2.get(tot - key);

            }

        }

        return result;

    }

}

/\*\*

 \* Your FindSumPairs object will be instantiated and called as such:

 \* FindSumPairs obj = new FindSumPairs(nums1, nums2);

 \* obj.add(index,val);

 \* int param\_2 = obj.count(tot);

 \*/

**Solution 1: Hash Table (30 min)**

**Because we don't need to print out what exactly index of num been update or used, we just update the frequency in 'add' method by decreasing the old value frequency by 1 and increasing the new value frequency by 1, and one more critical thing is we have to update the original 'nums2' (or its 'copy' as same object reference) for subsequence calls, if only get value such as nums2[index] (or copy[index]) and update that value it won't work, the change must on original array object**

**Style 1: Create map for 'nums1'**

class FindSumPairs {

    Map<Integer, Integer> map1;

    Map<Integer, Integer> map2;

    int[] copy;

    public FindSumPairs(int[] nums1, int[] nums2) {

        map1 = new HashMap<>();

        map2 = new HashMap<>();

        for(int num1 : nums1) {

            map1.put(num1, map1.getOrDefault(num1, 0) + 1);

        }

        for(int num2 : nums2) {

            map2.put(num2, map2.getOrDefault(num2, 0) + 1);

        }

        copy = nums2;

    }

    public void add(int index, int val) {

        //int key = copy[index];

        //int newKey = key + val;

        map2.put(copy[index], map2.get(copy[index]) - 1);

        // Critical: We have to update the original 'nums2'(or 'copy'

        // as same reference object) prepare for subsequent invocations,

        // only get value and no change on original 'nums2'(or 'copy'

        // as same reference object) won't work

        copy[index] += val;

        //map2.put(newKey, map2.getOrDefault(newKey, 0) + 1);

        map2.put(copy[index], map2.getOrDefault(copy[index], 0) + 1);

    }

    public int count(int tot) {

        int result = 0;

        for(int key : map1.keySet()) {

            if(map2.containsKey(tot - key)) {

                result += map1.get(key) \* map2.get(tot - key);

            }

        }

        return result;

    }

}

/\*\*

 \* Your FindSumPairs object will be instantiated and called as such:

 \* FindSumPairs obj = new FindSumPairs(nums1, nums2);

 \* obj.add(index,val);

 \* int param\_2 = obj.count(tot);

 \*/

Time Complexity: O(N)

Space Complexity: O(N)

**Style 2: No need create map for 'nums1', beacuse 1 <= nums1.length <= 1000, nums1 only have 1k as maximum length**

class FindSumPairs {

    Map<Integer, Integer> map;

    int[] copy1;

    int[] copy2;

    public FindSumPairs(int[] nums1, int[] nums2) {

        map = new HashMap<>();

        for(int num : nums2) {

            map.put(num, map.getOrDefault(num, 0) + 1);

        }

        copy1 = nums1;

        copy2 = nums2;

    }

    public void add(int index, int val) {

        map.put(copy2[index], map.get(copy2[index]) - 1);

        // Critical: We have to update the original 'nums2'(or 'copy'

        // as same reference object) prepare for subsequent invocations,

        // only get value and no change on original 'nums2'(or 'copy'

        // as same reference object) won't work

        copy2[index] += val;

        map.put(copy2[index], map.getOrDefault(copy2[index], 0) + 1);

    }

    public int count(int tot) {

        int result = 0;

        for(int key : copy1) {

            if(map.containsKey(tot - key)) {

                result += map.get(tot - key);

            }

        }

        return result;

    }

}

/\*\*

 \* Your FindSumPairs object will be instantiated and called as such:

 \* FindSumPairs obj = new FindSumPairs(nums1, nums2);

 \* obj.add(index,val);

 \* int param\_2 = obj.count(tot);

 \*/

Time Complexity: O(N)

Space Complexity: O(N)

**Step by Step**

nums1 = [1, 1, 2, 2, 2, 3]

nums2 = [1, 4, 5, 2, 5, 4]

-------------------------------------

Round 1: count(7)

nums1[2] + nums2[2] = 2 + 5 = 7

nums1[3]

nums1[4]

nums1[2] + nums2[4] = 2 + 5 = 7

nums1[3]

nums1[4]

nums1[5] + nums2[1] = 3 + 4 = 7

           nums2[5]

return 3 + 3 + 2 = 8

-------------------------------------

Round 2: add(3, 2)

nums1 = [1, 1, 2, 2, 2, 3]

nums2 = [1, 4, 5, 2 + 2, 5, 4]

     => [1, 4, 5, 4, 5, 4]

-------------------------------------

Ruond 3: count(8)

nums1[5] + nums2[2] = 3 + 5 = 8

           nums2[4]

return 2

-------------------------------------

Round 4: count(4)

nums1[5] + nums2[0] = 3 + 1 = 4

return 1

-------------------------------------

Round 5: add(0, 1)

nums1 = [1, 1, 2, 2, 2, 3]

nums2 = [1 + 1, 4, 5, 4, 5, 4]

     => [2, 4, 5, 4, 5, 4]

-------------------------------------

Round 6: add(1, 1)

nums1 = [1, 1, 2, 2, 2, 3]

nums2 = [2, 4 + 1, 5, 4, 5, 4]

     => [2, 5, 5, 4, 5, 4]

-------------------------------------

Round 7: count(7)

nums1[2] + nums2[1] = 2 + 5 = 7

nums1[3]

nums1[4]

nums1[2] + nums2[2] = 2 + 5 = 7

nums1[3]

nums1[4]

nums1[2] + nums2[4] = 2 + 5 = 7

nums1[3]

nums1[4]

nums1[5] + nums2[3] = 3 + 4 = 7

           nums2[5]

return 3 + 3 + 3 + 2 = 11

-------------------------------------

Because we don't need to print out what exactly index of num been

update or used, we just update the frequency in 'add' method by

decreasing the old value frequency by 1 and increasing the new

value frequency by 1, and one more critical thing is we have to

update the original 'nums2' (or its 'copy' as same object reference)

for subsequence calls, if only get value such as nums2[index] (or

copy[index]) and update that value it won't work, the change must

on original array object

**Refer to**

<https://algo.monster/liteproblems/1865>

**Problem Description**

You are provided with two integer arrays, nums1 and nums2, and the goal is to create a data structure that supports two types of queries. Firstly, you should be able to add a positive integer to an element at a given index in nums2. Secondly, you need to count the number of pairs (i, j) where the sum of nums1[i] and nums2[j] is equal to a specified value (tot). The pairs should only be considered if i and j are valid indexes within nums1 and nums2, respectively.

To accomplish the task, you should implement a class FindSumPairs with the following methods:

FindSumPairs(int[] nums1, int[] nums2): a constructor that initializes the object with the two integer arrays, nums1 and nums2.

add(int index, int val): a method that adds the integer val to the element at index in nums2.

count(int tot): a method that returns the number of pairs (i, j) where nums1[i] + nums2[j] equals the integer tot.

**Intuition**

The intuitive approach to solving this problem involves efficient data retrieval and update methods. If we were to perform a brute force approach for the count operation, we would iterate through all possible (i, j) pairs to find their sum and compare with tot, which would result in a time-consuming process, especially with large arrays.

A better way is to use a hash table to keep track of the frequency of each number in nums2. This allows us to quickly calculate how many times a certain number that can be added to elements of nums1 appears in nums2 to reach the required sum (tot).

Here's the intuition for the methods:

The constructor initializes the object and also creates a counter (Counter from collections in Python) that maps each number in nums2 to its frequency.

The add method updates the specific element in nums2 and adjusts the counter correspondingly. If a number's frequency changes (because it's being increased by val), we decrease the count of the old number and increase the count for the new, updated number.

The count method calculates the required pairs by traversing through nums1 and checking if the complement to reach tot (calculated as tot - nums1[i]) exists in the hash table (counter for nums2). The sum of all the frequencies of these complements gives us the total number of valid pairs that meet the condition.

**Solution Approach**

The implementation of the FindSumPairs class makes use of hash tables to store elements and their frequencies from nums2. This is essentially a mapping from each unique integer in nums2 to the number of times it appears. The Python Counter class from the collections module is used here for this purpose as it automatically counts the frequency of items in a list.

Here's the breakdown of the implementation:

The \_\_init\_\_ function of the FindSumPairs class initializes two properties, nums1 and nums2, with the corresponding input arrays. Additionally, a Counter object named cnt is created to store the frequency count of elements in nums2.

def \_\_init\_\_(self, nums1: List[int], nums2: List[int]):

self.nums1 = nums1

self.nums2 = nums2

self.cnt = Counter(nums2)

The add function takes in an index and a value val to add to nums2 at the specified index. Before updating nums2[index] with the new value, it decreases the count of the old value in cnt. Then, it increases the count of nums2[index] plus val. After updating the cnt, nums2 is updated with the new value.

def add(self, index: int, val: int) -> None:

old = self.nums2[index]

self.cnt[old] -= 1

if self.cnt[old] == 0:

del self.cnt[old] # Remove the entry from the counter if the frequency is zero

self.cnt[old + val] += 1

self.nums2[index] += val

The count function takes in a value tot and returns the number of pairs (i, j) where nums1[i] + nums2[j] == tot. To do this, it sums up the counts of tot - nums1[i] in cnt for each element i in nums1. In other words, for every number in nums1, it calculates the complement (the number that needs to be added to it in order to reach tot) and uses this to get the frequency of such complements from the Counter.

def count(self, tot: int) -> int:

return sum(self.cnt[tot - v] for v in self.nums1)

By utilizing the Counter, the class is able to execute the count function in O(n) time relative to the size of nums1, as it only has to iterate over nums1 and can retrieve the complement counts in constant time from the hash table. This is significantly more efficient than attempting to calculate the pairs through nested loops, which would be O(n \* m) where n and m are the sizes of nums1 and nums2.

**Example Walkthrough**

Let's step through the problem using a small example to illustrate the solution approach.

Suppose nums1 is [1, 2, 3] and nums2 is [1, 4, 5, 2], and we are interested in finding pairs whose sum equals tot = 5.

Upon class initialization:

nums1 remains [1, 2, 3].

nums2 remains [1, 4, 5, 2].

A Counter is created from nums2, resulting in {1: 1, 4: 1, 5: 1, 2: 1}.

Now, let's say we perform an add operation – add(2, 2) – which adds 2 to the element at index 2 in nums2.

The old value at nums2[2] is 5, so the counter for 5 is decreased from 1 to 0 and consequently removed since its count is now zero.

The value at nums2[2] is updated to 7 (5 + 2), so the counter for 7 is increased from 0 to 1.

Now nums2 becomes [1, 4, 7, 2], and the Counter reflects {1: 1, 4: 1, 7: 1, 2: 1}.

Next, we perform a count operation – count(5) to find pairs summing up to 5.

We iterate through nums1, looking for values that, when added to values from nums2, yield 5.

For nums1[0] = 1, we find tot - nums1[0] = 5 - 1 = 4; cnt[4] equals 1 indicating a pair (1, 4).

For nums1[1] = 2, tot - nums1[1] = 5 - 2 = 3; cnt[3] equals 0 indicating no valid pair exists with the second element 2.

For nums1[2] = 3, tot - nums1[2] = 5 - 3 = 2; cnt[2] equals 1 indicating a pair (3, 2).

Sum up the counts of valid pairs for all elements in nums1, giving us cnt[4] + cnt[3] + cnt[2] = 1 + 0 + 1 = 2.

Therefore, the count(5) operation tells us there are 2 valid pairs (1, 4) and (3, 2) whose sum equals 5.

This example demonstrates the efficiency of the approach using a Counter to avoid iterating over nums2 each time we call count. Instead, we make use of the precomputed frequencies to quickly tally the number of pairs that sum up to tot.

**Java Solution**

import java.util.HashMap;

import java.util.Map;

// Class to find count of pairs from two arrays that sum up to a given value

class FindSumPairs {

// Arrays to store the two input integer arrays

private int[] nums1;

private int[] nums2;

// Map to keep the frequency count of elements in the second array

private Map<Integer, Integer> frequencyMap = new HashMap<>();

// Constructor initializes the class with two integer arrays

public FindSumPairs(int[] nums1, int[] nums2) {

this.nums1 = nums1;

this.nums2 = nums2;

// Populating the frequency map with the count of each number in nums2

for (int value : nums2) {

frequencyMap.put(value, frequencyMap.getOrDefault(value, 0) + 1);

}

}

// Method that increments an element of nums2 at a given index by a given value

public void add(int index, int value) {

// Obtain the original value at the given index in nums2

int originalValue = nums2[index];

// Decrement the frequency of the original value in the frequency map

frequencyMap.put(originalValue, frequencyMap.get(originalValue) - 1);

// Increment the original value by the given value and update in nums2

nums2[index] += value;

// Increment the frequency of the new value in the frequency map

frequencyMap.put(nums2[index], frequencyMap.getOrDefault(nums2[index], 0) + 1);

}

// Method that counts the pairs across nums1 and nums2 that sum up to a given total

public int count(int total) {

int count = 0;

// Iterate through each value in nums1

for (int value : nums1) {

// For the current value in nums1, check if there's a complement in nums2 that sums up to total

count += frequencyMap.getOrDefault(total - value, 0);

}

// Return the count of such pairs

return count;

}

}

/\*

\* The usage of the FindSumPairs class:

\*

\* FindSumPairs obj = new FindSumPairs(nums1, nums2);

\* obj.add(index, value);

\* int result = obj.count(total);

\*/

**Time and Space Complexity**

**Time Complexity**

**init**(self, nums1: List[int], nums2: List[int]): The constructor initializes two lists nums1 and nums2. It also counts the elements of nums2 using Counter which takes O(n) time where n is the length of nums2.

add(self, index: int, val: int) -> None: This method updates an element in nums2 and modifies the count of the old and the new value in cnt. The update operation and the changes in the counter have a constant time complexity O(1) since dictionary (counter) operations in Python have an average-case complexity of O(1).

count(self, tot: int) -> int: The count method iterates over nums1 and for each element v in nums1, it accesses the counter cnt to find the count of (tot - v). If nums1 has m elements, the time complexity is O(m) since each lookup in the counter is O(1) on average.

**Space Complexity**

**init**(self, nums1: List[int], nums2: List[int]): Apart from the input lists nums1 and nums2, a counter cnt is created to store the frequency of each element in nums2. The space complexity is O(n) where n is the number of unique elements in nums2.

add(self, index: int, val: int) -> None: This method uses no extra space and thus has a space complexity of O(1).

count(self, tot: int) -> int: This method does not use extra space apart from a few variables for its operation, hence the space complexity is O(1).

Overall, the space complexity of the FindSumPairs class is determined by the space required for storing the input lists and the counter, which is O(n) where n is the size of nums2 and the number of unique elements it contains.