<https://leetcode.ca/all/170.html>

Design and implement a TwoSum class. It should support the following operations: add and find.

add - Add the number to an internal data structure.

find - Find if there exists any pair of numbers which sum is equal to the value.

**Example 1:**

add(1); add(3); add(5);

find(4) -> true

find(7) -> false

**Example 2:**

add(3); add(1); add(2);

find(3) -> true

find(6) -> false

**Attempt 1: 2024-01-010**

**Solution 1: Hash Table (10min)**

import java.util.\*;

public class Solution {

Map<Integer, Integer> map;

public Solution() {

map = new HashMap<>();

}

// Add the number to an internal data structure.

public void add(int num) {

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

}

// Find if there exists any pair of numbers which sum is equal to the value.

public boolean find(int target) {

for(Map.Entry<Integer, Integer> e : map.entrySet()) {

int key = e.getKey();

int val = e.getValue();

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

if(target - key != key) {

return true;

} else {

return val > 1;

}

}

}

return false;

}

public static void main(String[] args) {

Solution so = new Solution();

so.add(4);

so.add(4);

boolean result = so.find(8);

System.out.println(result);

}

}

Time and Space Complexity

add Method:

Time Complexity: O(1) on average for inserting the number into the Counter, though it could be in the worst case O(n) when there is a collision in the hashmap where n is the number of different elements added so far.

Space Complexity: O(n) where n is the number of different elements added to the Counter since each new addition might be a unique number requiring additional space.

find Method:

Time Complexity: O(n) where n is the number of unique numbers added to the TwoSum data structure. For each unique number x, the find method computes y and checks if y exists in the Counter. The existence check y in self.cnt take O(1) time on average.

Space Complexity: O(1) for this operation as it only stores the number y in memory and doesn't use any additional space that depends on the input size.

**Refer to**

<https://grandyang.com/leetcode/170/>

这道题让我们设计一个 Two Sum 的数据结构，跟 LeetCode 的第一道题 [Two Sum](http://www.cnblogs.com/grandyang/p/4130379.html) 没有什么太大的区别，作为 LeetCode 的首题，Two Sum 的名气不小啊，正所谓平生不会 TwoSum，刷尽 LeetCode 也枉然。记得原来在背单词的时候，总是记得第一个单词是 abandon，结果有些人背来背去还在 abandon，有时候想想刷题其实跟背 GRE 红宝书没啥太大的区别，都是一个熟练功夫，并不需要有多高的天赋，只要下足功夫，都能达到一个很不错的水平，套用一句鸡汤问来激励下吧，“有些时候我们的努力程度根本达不到需要拼天赋的地步”，好了，不闲扯了，来看题吧。不过这题也没啥可讲的，会做 Two Sum 的这题就很简单了，先来看用 HashMap 的解法，把每个数字和其出现的次数建立映射，然后遍历 HashMap，对于每个值，先求出此值和目标值之间的差值t，然后需要分两种情况来看，如果当前值不等于差值t，那么只要 HashMap 中有差值t就返回 True，或者是当差值t等于当前值时，如果此时 HashMap 的映射次数大于1，则表示 HashMap 中还有另一个和当前值相等的数字，二者相加就是目标值，参见代码如下：

class TwoSum {

public:

void add(int number) {

++m[number];

}

bool find(int value) {

for (auto a : m) {

int t = value - a.first;

if ((t != a.first && m.count(t)) || (t == a.first && a.second > 1)) {

return true;

}

}

return false;

}

private:

unordered\_map<int, int> m;

};

**Refer to**

<https://algo.monster/liteproblems/170>

**Problem Description**

The problem requires the design of a data structure to manage a stream of integers and provides two functionalities.

Adding an integer to the collection.

Querying to find out if there are any two distinct integers already in the collection that add up to a specified target sum.

This data structure should efficiently support the insertion of numbers and the checking for the existence of a pair of numbers that meet the target sum condition.

**Intuition**

The intuition behind the solution is to use a data structure that allows us to efficiently check for the existence of a specific number. A hash map or hash table comes to mind because it provides constant time complexity, O(1), lookup on average for insertions and searches.

Since we're dealing with pairs and their potential sums, when a query is made to find a pair that sums up to a value value, we can traverse each number x that has been added to the data structure and calculate its complement y (where y = value - x). Then, we can check if the complement y exists in the hash map. There are a couple of cases to consider:

If x equals y, then we need to have added this number at least twice for x + x = value to be true.

If x does not equal y, we simply need to check if y is present in the hash map.

We use a Counter to keep track of the occurrences of each number, which is essentially a specialized hash map that maps each number to the count of its occurrences. This allows us to handle both above-mentioned cases effectively.

By using this approach, we ensure that the add operation is done in constant time, while the find operation is done in O(n) time, where n is the number of unique numbers added to the data structure so far. This is an acceptable trade-off for the problem.

**Solution Approach**

The given solution makes use of the Counter class from Python's collections module. The Counter is a subclass of the dictionary that is designed to count hashable objects. It is an unordered collection where elements are stored as dictionary keys and their counts are stored as dictionary values.

**Initialization:**

The \_\_init\_\_ method initializes the TwoSum class.

def \_\_init\_\_(self):

self.cnt = Counter()

Here, an instance of Counter is created to keep track of how many times each number has been added to the data structure. This tracking is crucial for handling cases when the same number might be a part of the solution pair.

**The add Method:**

The add method takes an integer number and increments its count in the Counter.

def add(self, number: int) -> None:

self.cnt[number] += 1

Whenever a number is added, the Counter is updated so that the find method can reference the correct frequencies of the numbers.

**The find Method:**

The find method takes an integer value and tries to find if there are two distinct integers in the data structure that add up to value.

def find(self, value: int) -> bool:

for x, v in self.cnt.items():

y = value - x

if y in self.cnt:

if x != y or v > 1:

return True

return False

In this method, we iterate through each item in the Counter. For each number x, we calculate its complement y (value - x). We then check if y exists in the Counter:

If x is different from y and y exists in the counter, a pair that sums up to value was found.

If x equals y, we need to ensure that x was added at least twice (since v would be greater than 1 in such case) to say that we have a pair that adds up to value.

If no such pair is found during the iteration, the method returns False.

To summarize, this solution utilizes a hash-based data structure (Counter) for efficient lookups and count tracking to help identify if there exists a pair that sums up to the target value. The add method operates in O(1) time complexity, and the find method operates in O(n) time complexity, making the solution suitable for problems involving integer pairs with a specific sum.

**Example Walkthrough**

Let's consider a small example to illustrate the solution approach.

1. Imagine we initialize the data structure, and no numbers have been added yet. The Counter is therefore empty:

twoSum = TwoSum()

2. Now let's add some numbers to the collection:

twoSum.add(1)

twoSum.add(3)

twoSum.add(5)

3. After these operations, the internal Counter would look like this:

Counter({1: 1, 3: 1, 5: 1})

The counts reflect that each of the numbers 1, 3, and 5 has been added once.

4. Now, if we want to check if there exist any two distinct numbers that add up to 4, we call:

twoSum.find(4)

5. The find method will do the following:

For each number x in the Counter, it will calculate y = 4 - x.

If x is 1, then y will be 3, which is in the Counter. Since x is not equal to y, this is a valid pair that adds up to 4.

Since a valid pair is found, the method will return True.

6. Now let's add the number 1 once more to the collection:

twoSum.add(1)

7. Now, the Counter will be updated to:

Counter({1: 2, 3: 1, 5: 1})

8. The count for the number 1 is now 2 because we have added it twice.

If we now call find with a target sum of 2:

twoSum.find(2)

9. The method does the following:

It calculates y = 2 - x for each x.

When x is 1, y is also 1.

Since x equals y, we check the count of x in the Counter; it is 2, which means number 1 was added more than once.

Hence we have two distinct occurrences of the number 1, and they add up to 2.

The method returns True, indicating that there exists a pair that sums up to the target value.

The example demonstrates how the add method updates the Counter, and how the find method iterates through the available numbers, checking for the existence of a valid pair whose sum matches the target value.

**Java Solution**

// The TwoSum class provides a way to add numbers and find if there is a pair that sums up to a specific value.

class TwoSum {

// A HashMap to store the numbers and their frequencies.

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

// Constructor (empty since no initialization is needed here)

public TwoSum() {

}

// Adds the input number to the data structure.

public void add(int number) {

// Merges the current number into the map, incrementing its count if it already exists.

countMap.merge(number, 1, Integer::sum);

}

// Finds if there exists any pair of numbers which sum is equal to the value.

public boolean find(int value) {

// Iterates through the entries in our map

for (Map.Entry<Integer, Integer> entry : countMap.entrySet()) {

int currentKey = entry.getKey(); // The number in the pair we are currently considering.

int currentValue = entry.getValue(); // The count of how many times currentKey has been added.

int complement = value - currentKey; // The number that would complete the pair to equal 'value'.

// Check if the complement exists in our map.

if (countMap.containsKey(complement)) {

// If currentKey and complement are the same number, we need at least two instances.

if (currentKey != complement || currentValue > 1) {

return true; // Found a valid pair that sums up to the given value.

}

}

}

// If we haven't found any valid pair, return false.

return false;

}

}

// Example of how to use the TwoSum class:

// TwoSum obj = new TwoSum();

// obj.add(number);

// boolean param2 = obj.find(value);

**Time and Space Complexity**

**add Method:**

Time Complexity: O(1) on average for inserting the number into the Counter, though it could be in the worst case O(n) when there is a collision in the hashmap where n is the number of different elements added so far.

Space Complexity: O(n) where n is the number of different elements added to the Counter since each new addition might be a unique number requiring additional space.

**find Method:**

Time Complexity: O(n) where n is the number of unique numbers added to the TwoSum data structure. For each unique number x, the find method computes y and checks if y exists in the Counter. The existence check y in self.cnt take O(1) time on average.

Space Complexity: O(1) for this operation as it only stores the number y in memory and doesn't use any additional space that depends on the input size.