# 题目

设计一个接收整数流的数据结构，该数据结构支持检查是否存在两数之和等于特定值。

实现 TwoSum 类：

TwoSum() 使用空数组初始化 TwoSum 对象

void add(int number) 向数据结构添加一个数 number

boolean find(int value) 寻找数据结构中是否存在一对整数，使得两数之和与给定的值相等。如果存在，返回 true ；否则，返回 false 。

示例：

输入：

["TwoSum", "add", "add", "add", "find", "find"]

[[], [1], [3], [5], [4], [7]]

输出：

[null, null, null, null, true, false]

解释：

TwoSum twoSum = new TwoSum();

twoSum.add(1); // [] --> [1]

twoSum.add(3); // [1] --> [1,3]

twoSum.add(5); // [1,3] --> [1,3,5]

twoSum.find(4); // 1 + 3 = 4，返回 true

twoSum.find(7); // 没有两个整数加起来等于 7 ，返回 false

提示：

-105 <= number <= 105

-231 <= value <= 231 - 1

最多调用 5 \* 104 次 add 和 find。

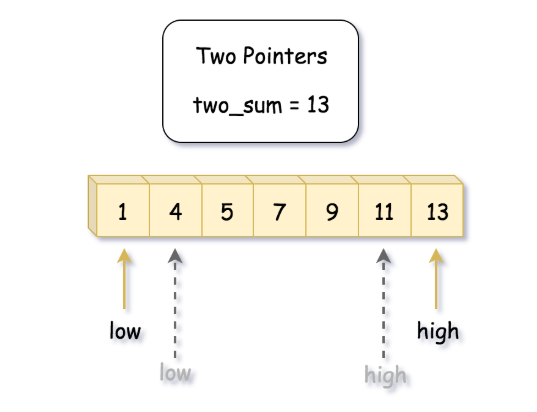
# 分析

## 方法一：双指针

这是第一题两数之和的后续问题之一，两数之和要求一个返回列表中两个数字的索引，这两个数字的和等于目标值。

我们可以从两数之和中汲取灵感，把所有的输入数字保存在一个列表中。

两数之和的一个解决方法是双指针法，指针从两个方向迭代，互相逼近。



但是，双指针法的前提条件之一是输入列表有序。

现在有几个问题：

函数 add(number) 在列表中插入新数字时，是否应该保证列表有序。

或者在调回 find(value) 时按需进行排序？

我们将在后面的算法部分讨论上述两个问题。

算法：

先给出双指针在有序列表中找到两数之和的算法。

初始化两个指针 low 和 high 分别指向列表的头尾。

从两个方向同时进行迭代，要么找到两数之和的解，要么两个指针相遇。

在每个步骤中，我们将根据不同的条件移动指针：

如果当前指针指向元素的和小于目标值，则应该增加总和来满足目标值，即将 low 指针向前移动获得更大的值。

如果当前指针指向元素的和大于目标值，则应该减少总和来满足目标值，即将 high 向 low 靠近来减少总和。

如果当前指针指向元素的和等于目标值，则直接返回结果。

如果两个指针相交，说明当前列表不存在组合成目标值的两个数。

**代码：**

import java.util.Collections;

class TwoSum {

private ArrayList<Integer> nums;

private boolean is\_sorted;

/\*\* Initialize your data structure here. \*/

public TwoSum() {

this.nums = new ArrayList<Integer>();

this.is\_sorted = false;

}

/\*\* Add the number to an internal data structure.. \*/

public void add(int number) {

this.nums.add(number);

this.is\_sorted = false;

}

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

public boolean find(int value) {

if (!this.is\_sorted) {

Collections.sort(this.nums);

this.is\_sorted = true;

}

int low = 0, high = this.nums.size() - 1;

while (low < high) {

int twosum = this.nums.get(low) + this.nums.get(high);

if (twosum < value)

low += 1;

else if (twosum > value)

high -= 1;

else

return true;

}

return false;

}

}

我们会发现，add(number) 函数将被频繁调用，而 find(value) 将不被那么频繁调用。

这样的使用模式下，意味着我们应该减少 add(number) 函数的时间消耗，因而我们是在 find(value) 对列表进行排序，而不是在 add(number)。

在哪个函数进行排序，都是可行的，只是对应该使用模式下在 add(number) 下进行排序就不是最佳的方案了。

并且，我们在 find(value) 中是按需排序，也就是当列表更新时，才进行排序。

复杂度分析：

时间复杂度：

add(number)：O(1)

find(value)：O(Nlog(N))，在最坏的情况下，我们需要对列表进行排序和遍历整个列表，这需要O(Nlog(N))和O(N) 的时间。因此总的时间复杂度为O(Nlog(N))。

空间复杂度：O(N)，其中N指的是列表中的元素个数。

## 方法二：哈希表

**官方答案：**

两数之和的另一个解决方案是用哈希表存储值到索引的映射关系。

给定一个目标值 S，对于每个数字 a，我们只需要验证哈希表中是否存在 S - a。

众所周知，哈希表可以提供快速的查找和插入操作，非常适合上述要求。

**算法：**

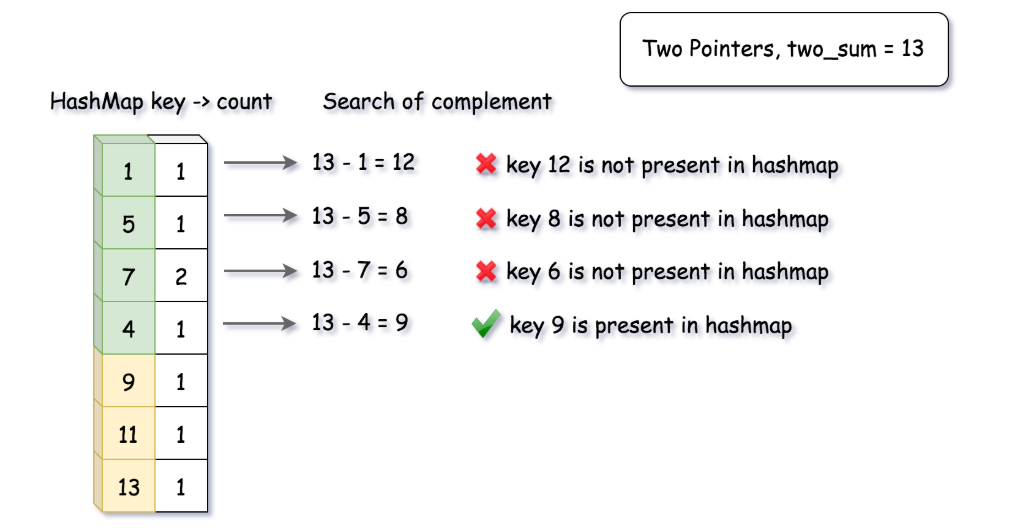
首先初始化一个哈希表。

在 add(number) 函数中：在哈希表中添加 number 到 number 频数之间的映射关系。

在 find(value) 函数中：遍历哈希表，对于每个键值（number），我们检查哈希表中是否存在 value - number。如果存在，我们终止循环并返回结果。

当 number = value - number 时，在哈希表中 number 对应的值应大于 2。

我们在下图中演示了算法：



import java.util.HashMap;

class TwoSum {

private HashMap<Integer, Integer> num\_counts;

/\*\* Initialize your data structure here. \*/

public TwoSum() {

this.num\_counts = new HashMap<Integer, Integer>();

}

/\*\* Add the number to an internal data structure.. \*/

public void add(int number) {

if (this.num\_counts.containsKey(number))

this.num\_counts.replace(number, this.num\_counts.get(number) + 1);

else

this.num\_counts.put(number, 1);

}

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

public boolean find(int value) {

for (Map.Entry<Integer, Integer> entry : this.num\_counts.entrySet()) {

int complement = value - entry.getKey();

if (complement != entry.getKey()) {

if (this.num\_counts.containsKey(complement))

return true;

} else {

if (entry.getValue() > 1)

return true;

}

}

return false;

}

}

**复杂度分析：**

时间复杂度：

在add(number)中：O(1)，只进行了哈希表的更新。

在find(value) 中：O(N)，其中N指的是哈希表中键值对的数量。在最坏的情况下，会遍历整个表。

空间复杂度：O(N)，哈希表所使用的空间大小。

**思路：**

使用hash表，key为数字，value为数字出现次数  
 注意：可能存在添加多个相同元素 如add(2)、add(2) find(4)

**代码：**

class TwoSum {

public:

/\*\* Initialize your data structure here. \*/

TwoSum() {

m\_hash.reserve(4\*1024);

}

/\*\* Add the number to an internal data structure.. \*/

void add(int number) {

++m\_hash[number];

}

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

bool find(int value) {

for (auto iter = m\_hash.begin(); iter != m\_hash.end(); ++iter)

{

auto found = m\_hash.find(value - iter->first);

if (found != m\_hash.end())

{

//两个数不相等 或者 两个数相等并且统计的出现次数大于1

if(found->first != iter->first || (found->first == iter->first && found->second > 1))

return true;

}

}

return false;

}

private:

unordered\_map<int, int> m\_hash;

};

## 方法三：multiset

class TwoSum {

private:

multiset<int> data;

public:

TwoSum() {}

void add(int number) {

data.insert(number);

}

bool find(int value) {

if (data.empty()) return false;

auto p = data.begin(), q = --data.end();

while (p != q) {

int sum = \*p + \*q;

if (sum < value) ++p;

else if (sum > value) --q;

else return true;

}

return false;

}

};