# Question

Design a data structure that accepts a stream of integers and checks if it has a pair of integers that sum up to a particular value.

Implement the TwoSum class:

* TwoSum() Initializes the TwoSum object, with an empty array initially.
* void add(int number) Adds number to the data structure.
* boolean find(int value) Returns true if there exists any pair of numbers whose sum is equal to value, otherwise, it returns false.

**Example 1:**

**Input**

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

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

**Output**

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

**Explanation**

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, return true

twoSum.find(7); // No two integers sum up to 7, return false

**Constraints:**

* -105 <= number <= 105
* -231 <= value <= 231 - 1
* At most 5 \* 104 calls will be made to add and find.

# Solution

#### **Approach 1: Sorted List**

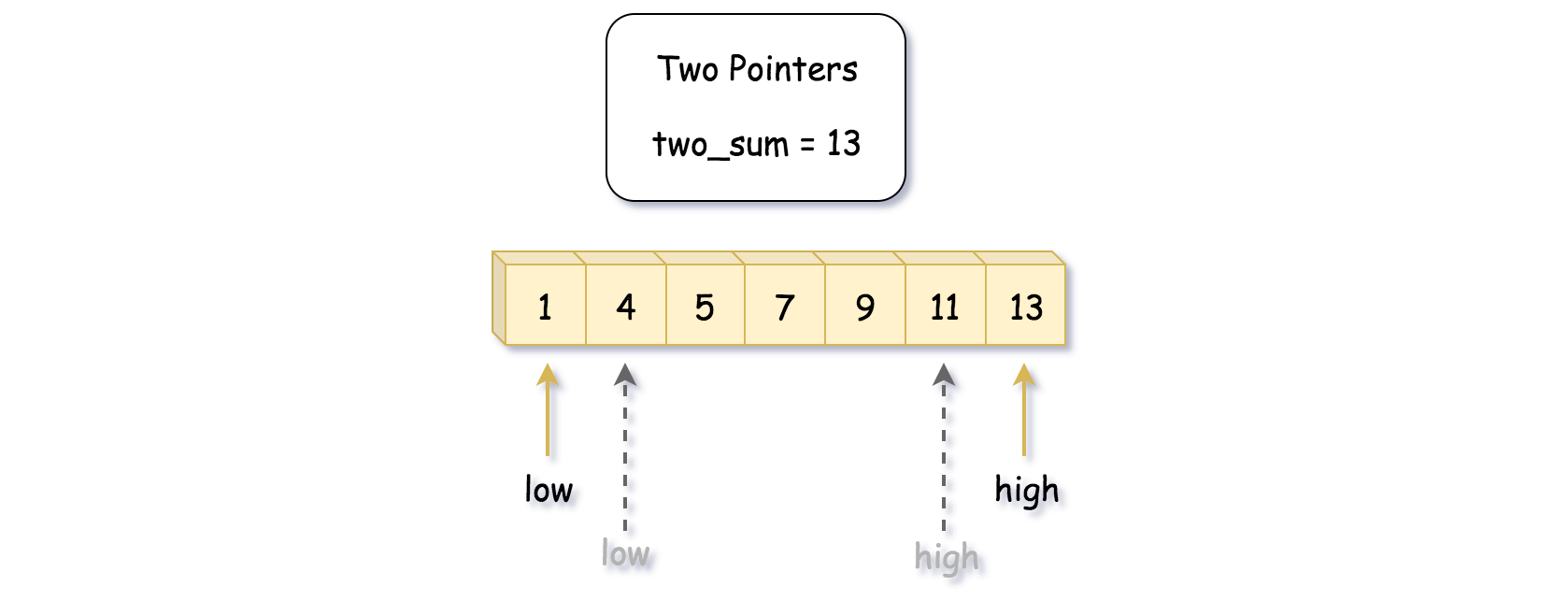
**Intuition**

First of all, the problem description is not terribly clear on the requirements of time and space complexity. But let us consider this as part of the challenge or a freedom of design. We could figure out the desired complexity for each function, by trial and error.

This is one of the followup problems to the first programming problem on LeetCode called [Two Sum](https://leetcode.com/problems/two-sum/), where one is asked to return the indice of two numbers from a **list** that could sum up to a given value.

Let us take the inspiration from the origin problem, by keeping all the incoming numbers in a list.

Given a list, one of the solutions to the Two Sum problem is called **Two-Pointers Iteration** where we iterate through the list from two directions with two pointers approaching each other.



However, one of the preconditions for the Two-Pointers Iteration solution is that the input list should be ***sorted***.

So now, here are the questions:

* Should we keep the list in order while inserting new numbers in the function add(number) ?
* Or should we do the sorting on demand, i.e. at the invocation of find(value) ?

We will address the above two questions later in the Algorithm section.

**Algorithm**

Let us first give the algorithm of Two-Pointers Iteration to find the two-sum solution from a sorted list:

* We initialize **two pointers** low and high which point to the head and the tail elements of the list respectively.
* With the two pointers, we start a **loop** to iterate the list. The loop would terminate either we find the two-sum solution or the two pointers meet each other.
* Within the loop, at each step, we would move either of the pointers, according to different conditions:
  + If the sum of the elements pointed by the current pointers is ***less than*** the desired value, then we should try to increase the sum to meet the desired value, i.e. we should move the low pointer forwards to have a larger value.
  + Similarly if the sum of the elements pointed by the current pointers is ***greater than*** the desired value, we then should try to reduce the sum by moving the high pointer towards the low pointer.
  + If the sum happen to the desired value, then we could simply do an **early return** of the function.
* If the loop is terminated at the case where the two pointers meet each other, then we can be sure that there is no solution to the desired value.

|  |
| --- |
| 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;  }  } |

The usage pattern of the desired data structure in the online judge, as we would discover, is that the add(number) function would be called **frequently** which might be followed a less frequent call of find(value) function.

The usage pattern implies that we should try to minimize the cost of add(number) function. As a result, we sort the list within the find(value) function instead of the add(number) function.

So to the above questions about where to place the sort operation, actually both options are valid and correct. Due to the usage pattern of the two functions though, it is **less optimal** to sort the list at each add operation.

On the other hand, we do not do sorting at each occasion of find(value) neither. But rather, we sort on demand, i.e. only when the list is updated. As a result, we **amortize** the cost of the sorting over the time. And this is the optimization trick for the solution to pass the online judge.

**Complexity Analysis**

* Time Complexity:
  + For the add(number) function: \mathcal{O}(1)O(1), since we simply append the element into the list.
  + For the find(value) function: \mathcal{O}(N \cdot \log(N))O(*N*⋅log(*N*)). In the worst case, we would need to sort the list first, which is of \mathcal{O}(N \cdot \log(N))O(*N*⋅log(*N*)) time complexity normally. And later, again in the worst case we need to iterate through the entire list, which is of \mathcal{O}(N)O(*N*) time complexity. As a result, the overall time complexity of the function lies on \mathcal{O}(N \cdot \log(N))O(*N*⋅log(*N*)) of the sorting operation, which dominates over the later iteration part.
* Space Complexity: the overall space complexity of the data structure is \mathcal{O}(N)O(*N*) where N*N* is the total number of numbers that have been added.

#### **Approach 2: HashTable**

**Intuition**

As an alternative solution to the original [Two Sum](https://leetcode.com/problems/two-sum/) problem, one could employ the HashTable to index each number.

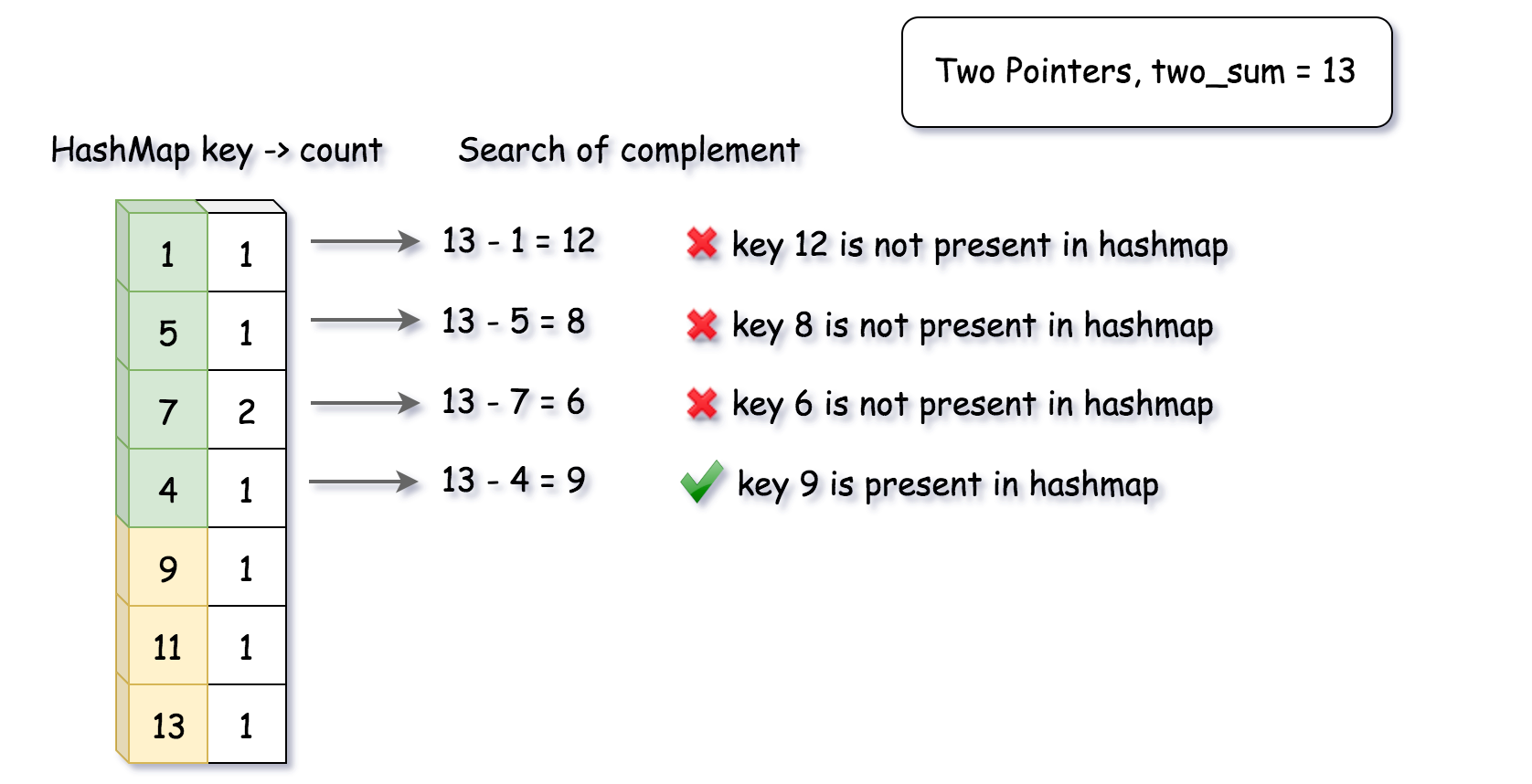
Given a desired sum value S, for each number a, we just need to verify if there exists a complement number (S-a) in the table.

As we know, the data structure of hashtable could offer us a quick lookup as well as insertion operations, which fits well with the above requirements.

**Algorithm**

* First, we initialize a hashtable container in our data structure.
* For the add(number) function, we build a frequency hashtable with the number as key and the frequency of the number as the value in the table.
* For the find(value) function, we then iterate through the hashtable over the keys. For each key (number), we check if there exists a complement (value - number) in the table. If so, we could terminate the loop and return the result.
* In a particular case, where the number and its complement are equal, we then need to check if there exists at least **two copies** of the number in the table.

We illustrate the algorithm in the following figure:



|  |
| --- |
| **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;**  **}**  **}** |

**Complexity Analysis**

* Time Complexity:
  + For the add(number) function: \mathcal{O}(1)O(1), since it takes a constant time to update an entry in hashtable.
  + For the find(value) function: \mathcal{O}(N)O(*N*), where N*N* is the total number of **unique** numbers. In the worst case, we would iterate through the entire table.
* Space Complexity: \mathcal{O}(N)O(*N*), where N*N* is the total number of **unique** numbers that we will see during the usage of the data structure.