Previous Next

170. Two Sum III - Data Structure Design

Dec. 1, 2019 | 7.1K views

Average Rating: 5 (4 votes)

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
```

Solution

Approach 1: Sorted List

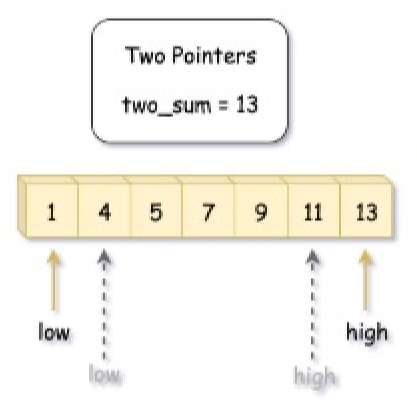
Intuition

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, 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.

Java | Python | Copy

```java
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);
    }
    int low = 0, high = this.nums.size() - 1;
    while (low < high) {
      int twosum = this.nums.get(low) + this.nums.get(high);
      if (twosum < value)
```

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

Approach 2: HashTable

Intuition

As an alternative solution to the original Two Sum problem, one could employ the *HashTable* to index each number.

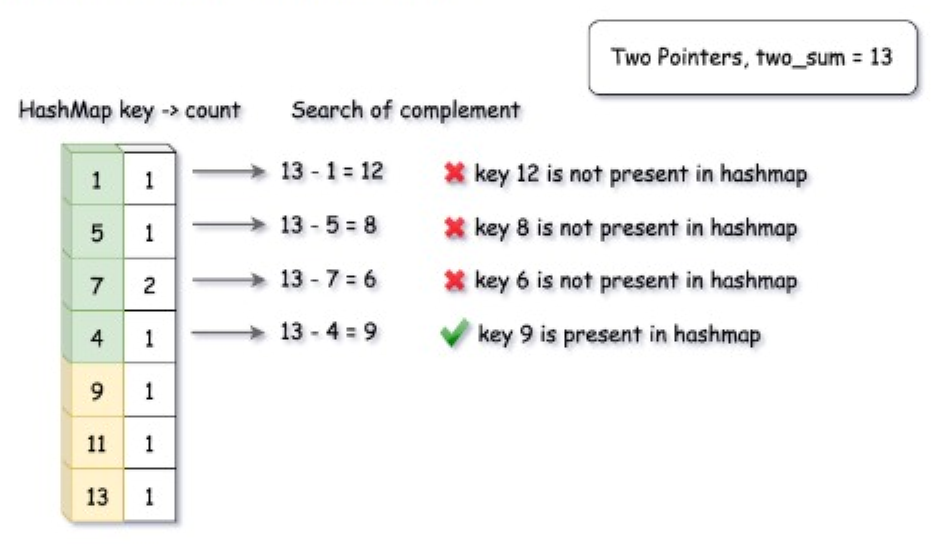
> Given a desired sum value `S`, for each number `a`, we just need to verify if there exists a complement number (`S-a`) in the table.

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:



Java | Python | Copy

```java
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)
```

Complexity Analysis

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

Analysis written by @liaison and @andvary

Rate this article:

Previous Next

Comments: 7 Sort By

Type comment here... (Markdown is supported)

Preview Post

byronshilly ★ 5 December 2, 2019 2:12 AM

Isn't the time complexity for Approach 2's `find` function linear? Looking up the compliment is done in constant time, but in the event that no valid pair exists, the algorithm will loop through the entire hash table looking for compliments.

4 Share Reply

SHOW 2 REPLIES

Ayamin ★ 14 January 11, 2020 11:41 PM Report

I think we can get linear add() and constant time find(), if we keep a Set of all the sums.
When we add, compute the pair-wise sum of the new element with all the existing elements, and add those pair-wise sums to the set.

Just something to consider, depending on the behavior in traffic on those 2 API endpoints of this class

Read More

1 Share Reply

SHOW 3 REPLIES

tangy321 ★ 4 April 19, 2020 10:33 AM

Two easy

0 Share Reply

undefitied ★ 73 February 15, 2020 1:12 AM

You can do sorted array in O(n) time, if you will maintain it with each addition.
Then `add(number)` -> O(n)
and `find(value)` -> O(n)
Space for the sorted array O(number of unique items)

0 Share Reply

koushirou ★ 4 February 3, 2020 3:22 AM

In Approach 1, we could use divide and conquer when inserting an element into the sorted array. So the complexity would be O(logN) for add and O(N) for find. This way is better than the original Approach 1 if there are more find than add.

0 Share Reply

SHOW 1 REPLY

robinali34 ★ 3 January 28, 2020 11:35 AM

The HashMap solutions is hard to read, modified a bit:

```
class TwoSum {
    private HashMap<Integer, Integer> num_counts;
```

Read More

0 Share Reply

Swapnil510 ★ 18 January 17, 2020 10:03 AM

No need of writing if else in add function. You can simply use map.getOrDefault(key,default_value)

0 Share Reply