170. Two Sum III - Data structure design

Easy | Design | Array | Hash Table | Two Pointers | Data Stream | Leetcode Link

Problem Description

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

1. Adding an integer to the collection.
2. 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:

1. If `x` equals `y`, then we need to have added this number at least twice for `x + x = value` to be true.
2. 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.

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

```
twoSum = TwoSum()
```

Now let's add some numbers to the collection:

```
twoSum.add(1)
twoSum.add(3)
twoSum.add(5)
```

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.

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

```
twoSum.find(4)
```

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`.

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

```
twoSum.add(1)
```

Now, the `Counter` will be updated to:

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

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)
```

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.

Python Solution

```python
from collections import Counter

class TwoSum:
    def __init__(self):
        # Initialize a counter to keep track of the number of occurrences of each number
        self.num_counter = Counter()

    def add(self, number: int) -> None:
        # Increment the count for the added number.
        self.num_counter[number] += 1

    def find(self, value: int) -> bool:
        # Check if there are any two numbers that add up to the given value.
        for num, count in self.num_counter.items():
            # The required partner number to reach the value
            complement = value - num
            # Check if the complement exists in the counter
            if complement in self.num_counter:
                # If the number and complement are the same, ensure there are at least two occurrences
                if num != complement or count > 1:
                    return True
        # If no valid pair is found, return False
        return False

# Example usage:
# two_sum_instance = TwoSum()
# two_sum_instance.add(number)
# result = two_sum_instance.find(value)

```

Java Solution

```java
// 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);

```

C++ Solution

```cpp
#include <unordered_map>
using namespace std;

// Class to store numbers and find if there are two numbers that add up to a specific value.
class TwoSum {
public:
    // Constructor initializes the TwoSum object.
    TwoSum() {
    }

    // Add the number to the internal data structure.
    void add(int number) {
        // Increment the count of this number in the hash map.
        ++numCount[number];
    }

    // Find if there exists any pair of numbers which sum is equal to the value.
    bool find(int value) {
        // Iterate through the hash map using a range-based for loop.
        for (const auto& numPair : numCount) {
            const int& num = numPair.first;       // The current number from the hash map.
            const int& frequency = numPair.second; // The frequency of the current number.

            // Calculate the complement that we need to find.
            long complement = (long)value - num;

            // Check if the complement exists in the hash map.
            if (numCount.count(complement)) {
                // If the number and its complement are the same, check if the number occurs at least twice.
                // Otherwise, we found a pair with the required sum.
                if (num != complement || frequency > 1) {
                    return true;
                }
            }
        }

        // If we reach here, no such pair exists.
        return false;
    }

private:
    // Hash map to store the count of each number added.
    unordered_map<int, int> numCount;
};

// Usage example:
// TwoSum* obj = new TwoSum();
// obj->add(number);
// bool param_2 = obj->find(value);

```

Typescript Solution

```typescript
// In a TypeScript environment, we do not often use global variables and functions,
// as it is considered bad practice. However, here is how a similar functionality
// could be implemented in TypeScript without a class.

// An object to store the count of each number added.
const numCount: Record<number, number> = {};

// Adds the number to the internal data structure by incrementing its count.
function addNumber(number: number): void {
  if (numCount[number] !== undefined) {
    numCount[number] += 1;
  } else {
    numCount[number] = 1;
  }
}

// Finds if there exists any pair of numbers which sum is equal to the value.
function find(value: number): boolean {
  for (const num in numCount) {
    // Parse the key as an integer since object keys are always strings.
    const number = parseInt(num);

    // Calculate the complement that is needed to find.
    const complement = value - number;

    // Check if the complement exists in the internal data structure.
    if (numCount[complement] !== undefined) {
      // If the number and its complement are the same, check if the number occurs at least twice.
      // Otherwise, we have found a pair with the required sum.
      if (number !== complement || numCount[number] > 1) {
        return true;
      }
    }
  }

  // If we reach here, no such pair exists.
  return false;
}

// Usage example (uncomment the following lines to use it in an environment that supports execution):
// addNumber(1);
// addNumber(3);
// addNumber(5);
// let result = find(4); // Should return true, because 1 + 3 = 4.
// console.log(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.