Hello everyone, today we're going to talk about a very common type of LeetCode problem: using HashTable to optimize solutions for array problems.

Before we dive into specific problems, we need to make sure we really understand two core data structures: **arrays** and **hash tables**. These two often show up together, but their characteristics and use cases are quite different.

Basics

Arrays

Let's start with arrays.

The biggest feature of an array is **contiguous storage**. That means in memory, the elements are stored right next to each other, in a single block. Because of that, we can directly access any element by its index in **O(1)** time.

For example, if I have `arr = [2, 7, 11, 15]` and I want the element at index 2, I can just do `arr[2]` and instantly get 11. No need to scan through one by one.

But arrays also have a clear downside: **insertion and deletion**. Since elements are stored contiguously, if I insert a new number in the middle, everything after it has to be shifted down. Same thing with deletion—everything after has to be shifted up. That's why these operations are usually **O(n)**.

So arrays are really good for sequential operations like traversal, but not great if you're frequently inserting or deleting in the middle.

Hash Tables

Now let's talk about hash tables.

A **hash table** is a general data structure. The core idea is to use a hash function to map a key into an array index, so you can quickly store or retrieve the value.

The main advantage of a hash table is that, on average, **lookups, insertions, and deletions all take O(1)** time. You can think of it like a real dictionary: given a word (the key), you can immediately find its definition (the value) without flipping through every page.

- A **HashMap** is an implementation of a hash table that stores **key** → **value** pairs. In Python, this is just a `dict`.
- A **HashSet** is a specialized version of a hash table that stores only keys, no values. It represents a collection where you only care about whether an element exists. In Python, that's just `set`. This is especially useful for tasks like deduplication or checking membership.

In algorithm problems, hash tables are most commonly used for two things:

1. **Counting frequency –** for example, counting how many times each element appears. You can just do something like `map[num] += 1`, and in the end you know which number is most frequent or which ones are duplicates.
2. **Recording positions –** for example, storing the index where each element first appeared: `map[num] = index`. That way, you can later look it up in **O(1)** time.

So overall, the hash table gives us:

- Quick existence checks
- Quick frequency counts
- Quick position lookups

Arrays vs. Hash Tables

So here's the key question: when is an array alone enough, and when should we bring in a hash table?

Here's a simple way to decide:

If the problem is about **sequential processing**, like maximum subarray sum or sliding window maximum, an array by itself is usually enough.

But if the problem mentions things like:

- "check quickly if something has appeared before,"
- "count frequencies,"
- "find the position of a target value,"
- or explicitly "achieve O(1) time complexity,"

then it's time to consider a hash table.

Example: Two Sum

Let's look at a classic problem — **Two Sum**.

The problem says:

> Given an array of integers `nums` and an integer `target`, return the **indices** of the two numbers such that they add up to `target`.
>
> You may assume that each input has exactly one solution, and you may not use the same element twice.
>
> You can return the answer in any order.

> Example:
>
> ```
> Input: nums = [2,7,11,15], target = 9
> Output: [0,1]
> Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].
> ```

Brute Force

The most straightforward idea is brute force: just try every possible pair.

1. Use the outer loop to pick the first number.
2. Use the inner loop to pick the second number.
3. If their sum equals `target`, return their indices.

That works, but the time complexity is **O(n²)**, which gets very slow for large arrays.

```
def twoSum(nums, target):
    n = len(nums)
    for i in range(n):
        for j in range(i + 1, n):  # start j from i+1 to avoid duplicates
            if nums[i] + nums[j] == target:
                return [i, j]
```

Optimized with a Hash Table

Can we do better? Yes!

We can use a hash table to store the mapping from **value** → **index** while we traverse the array:

1. For each number `num`, compute the partner we need: `complement = target - num`.
2. Check if this complement is already in the hash table.
3. If it is, that means we've seen it before, so we can immediately return `[hashmap[complement], i]`.
4. Otherwise, store the current number and its index in the hash table, so future numbers can find it.

This way, each number is processed once, and each lookup in the hash table is O(1). So the total complexity is **O(n)**.

```python
def twoSum(nums, target):
    hashmap = {}  # key = value, value = index
    for i, num in enumerate(nums):
        complement = target - num
        if complement in hashmap:
            return [hashmap[complement], i]
        hashmap[num] = i
```

This is the power of a hash table: It turns a problem that would normally require nested loops into a **single pass** with O(n) time complexity. And that's why Two Sum is often the very first problem people solve with a hash table.

Why Store Indices?

Now, here's an important detail:

If the question only asked, *"Does the array contain two numbers that add up to target?"* Then we don't need indices at all. We could just use a **hash set** to check if `target - num` has appeared before, and return True if it has.

But this problem specifically asks for **indices**. That's why we need a **hash map**, not just a hash set. The hash map doesn't just store values — it stores their positions. This way, when we find a complement, we can return the exact pair of indices.

Handy Libraries: Counter & defaultdict

So, we just talked about how hash tables are used in problems.

In Python, a hash table is basically just a dictionary (`dict`). But here's the thing: sometimes using a plain `dict` can feel a bit clunky.

For example, when you're counting frequencies, you always need to check if a key exists first before updating the count. Or when grouping items, you need to set up an empty list before you can append things to it.

```python
# Using a normal dict to count frequencies
nums = [1, 1, 2, 3, 3, 3]
count = {}

for num in nums:
    if num not in count:   # need to check manually
        count[num] = 0
    count[num] += 1

print(count)  # {1: 2, 2: 1, 3: 3}
```

This works fine, but writing this boilerplate again and again gets annoying. That's why Python gives us two super useful tools in `collections`: **`Counter`** and **`defaultdict`**. They make the code a lot cleaner.

`collections.Counter`

`Counter` lets you count frequencies in one line, instead of writing loops and conditionals yourself.

```python
from collections import Counter

nums = [1,1,2,3,3,3]
counter = Counter(nums)
print(counter)   # Output: Counter({3: 3, 1: 2, 2: 1})
```

It immediately gives you how many times each number appears. For example, `3` shows up 3 times, `1` shows up 2 times, and so on.

Things to keep in mind:

- `Counter` is a subclass of `dict`, so underneath it's still a hash table. Time complexity is O(n).
- You can use it just like a normal dictionary (`counter[key]`). The only difference is the print format looks like `Counter({...})`.
- Missing keys default to **0**, instead of raising an error like normal `dict`.
- It also supports operations like add/subtract, union/intersection:
    - `Counter(a) + Counter(b)` adds frequencies.
    - `Counter(a) - Counter(b)` subtracts frequencies (and drops keys with ≤ 0).
    - These are neat, but in interviews, make sure you explain clearly what's happening.
- Because `Counter` is feature-rich, there's a little overhead. For huge datasets, using a plain `dict` or `defaultdict(int)` might actually be faster.

`collections.defaultdict`

Now let's look at `defaultdict`.

Normally with a `dict`, if a key doesn't exist, you have to do:

```python
if key not in dict:
    dict[key] = []
```

With `defaultdict`, you don't need this check. It automatically initializes a default value for you.

```python
from collections import defaultdict

groups = defaultdict(list)
words = ["eat","tea","tan","ate","nat","bat"]

for w in words:
    key = "".join(sorted(w))
    groups[key].append(w)

print(groups)
# Output: {'aet': ['eat', 'tea', 'ate'], 'ant': ['tan', 'nat'], 'abt': ['bat']}
```

`defaultdict(list)` automatically sets up an empty list, so you can just append directly. No need for manual initialization.

Things to keep in mind:

- You must pass a **factory function** when creating it:
    - `defaultdict(int)` → default is 0
    - `defaultdict(list)` → default is []
    - `defaultdict(set)` → default is set()
    - You **cannot** pass a raw value like `defaultdict(0)` — that will raise an error.
- It **automatically creates keys** when accessed:

    > ```python
    > from collections import defaultdict
    > d = defaultdict(int)
    > print(d[1])   # prints 0, and also creates key=1
    > print(d)      # defaultdict(<class 'int'>, {1: 0})
    > ```
    >
    > So if you just want to "check" whether a key exists, you might end up creating it by accident.

- Printing looks like `defaultdict(<class 'list'>, {...})`, not a plain dict. If you need to return the result to a platform like LeetCode, better convert it: `dict(d)`.

- Performance-wise, it's the same as a normal dict — O(1) for insert/lookup. The only difference is the automatic initialization logic.

Both `Counter` and `defaultdict` make your code shorter and cleaner. But in interviews, it's important to understand what's happening under the hood.

Common Problem Types

Today we focused on how hash tables are applied in array problems. Here's a practice list you can work on after class to really apply the ideas to actual problems.

Today's Problems

- 1.Two Sum
- 128. Longest Consecutive Sequence
- 36. Valid Sudoku

Hash Table Counting + Array Traversal

Core idea: first use a hash table to count frequencies or features, then traverse the array to do grouping or pairing.

- Group Anagrams
- Top K Frequent Elements

Two Pointers + Hash Table

Core idea: use two pointers to maintain a sliding window, and use a hash table to keep track of element frequencies inside the window. This way you can dynamically expand/shrink the window while quickly checking conditions.

- 3. Longest Substring Without Repeating Characters

Subarray / Substring Problems

Core idea: use a hash table to store element positions or prefix-sum counts, then traverse the array.

- [560. Subarray Sum Equals K]
- [523. Continuous Subarray Sum]

Design Problems

These test whether you can use a hash table to implement your own data structure.

- [380. Insert Delete GetRandom O(1)]