Basics

Arrays

Contiguous storage: elements stored sequentially in memory.

O(1) access: direct access by index.

```
Memory Layout:

Index:   0     1     2     3
Value:   2     7    11    15
         ↑     ↑     ↑     ↑
       arr[0] arr[1] arr[2] arr[3]
```

Drawback: insertion/deletion requires shifting elements → **O(n)**.

```
Before:
Index:   0     1     2     3
Value:   2     7    11    15

Insert 99 at index 2 → shift [11, 15] to the right

After:
Index:   0     1     2     3     4
Value:   2     7    99    11    15
```

```
Before:
Index:   0     1     2     3
Value:   2     7    11    15

Delete 7 at index 1 → shift [11, 15] to the left

After:
Index:   0     1     2
Value:   2    11    15
```

Use case:

- efficient for sequential traversal;
- inefficient for frequent middle insertions/deletions.

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.

```
Hash Function: key → index

Keys:    [apple]   [banana]   [cat]
Index:      2          5          8
Value:   "red"     "yellow"   "black"
```

Advantage: O(1) average time for lookup, insertion, deletion

- **HashMap**: stores key → value (Python: `dict`)
- **HashSet**: stores only keys (Python: `set`), used for deduplication / membership check

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

1. **Counting frequency** – find most frequent elements, detect duplicates

   ```
   map[num] += 1
   ```

2. **Recording positions** – quick lookup of element's index

   ```
   map[num] = index
   ```

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

> 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

Idea: try every possible pair

1. Outer loop picks the first number
2. Inner loop picks the second number
3. If their sum equals `target`, return indices

Time complexity: **$O(n^2)$** → inefficient for large arrays

```python
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

Store mapping **value** → **index** while traversing

1. For each number `num`, compute `complement = target - num`
2. Check if `complement` exists in hash table
3. If yes → return `[hashmap[complement], i]`
4. If no → store current `num: i` in hash table

Each number processed once, each lookup O(1). Total time complexity: **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
```

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

In Python, a hash table is a `dict`, but using plain `dict` can be verbose:

- Counting frequencies → need to check key existence before updating.
- Grouping items → need to initialize empty lists manually.

```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}
```

To simplify:

- **`Counter`**: one-line frequency counting.
- **`defaultdict`**: automatic initialization of default values (e.g., `int`, `list`, `set`).

These tools reduce boilerplate and make code 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})
```

Things to keep in mind:

- Subclass of `dict` → implemented as a hash table, **O(n)** time.
- Usable like a normal dict (`counter[key]`). The only difference is the print format looks like `Counter({...})`.
- Missing keys default to **0** (no KeyError).

- Supports arithmetic & set-like operations:
  - `Counter(a) + Counter(b)` → add counts
  - `Counter(a) - Counter(b)` → subtract counts (removes ≤ 0)
- Slight overhead due to extra features → for large datasets, `dict` or `defaultdict(int)` may be faster.

`collections.defaultdict`

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

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

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

```
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']}
```

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:

  > ```
  > from collections import defaultdict
  > d = defaultdict(int)
  > print(d[1])   # prints 0, and also creates key=1
  > print(d)      # defaultdict(<class 'int'>, {1: 0})
  > ```

- 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.

Common Problem Types

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]

- [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)]