Two Sum Problem

Problem Description:

“Given an array of integers `nums` and an integer `target`, return the **indices** of two numbers such that they add up to `target`. Assume exactly one solution exists.”

Q1: Why is "Two Sum" so famous in interviews?

1. **Simple to state, tricky to solve efficiently.**
2. Tests your knowledge of **arrays**, **hashmaps**, and **runtime optimization**.
3. A classic warm-up problem to set the tone for tougher algorithm questions.

Q2: Naive or clever? Why not just brute force?

Sure, brute force works:

python
Copier le code

```
for i in range(len(nums)):
    for j in range(i + 1, len(nums)):
        if nums[i] + nums[j] == target:
            return [i, j]
```

But that’s **O(n²)**! 🔥 On a big array, your interviewer will grill you for inefficiency.

Q3: What’s the optimized approach?

The **hashmap** is your bestie:

- While looping through the array, calculate the complement (`target - nums[i]`).
- Check if this complement exists in the hashmap.
- If yes, 💥 return indices. Else, store the number in the map for future reference.

Q4: Does HashMap magic make it linear?

YES, because:

- HashMap lookups are **O(1)** on average.
- You only loop the array once. Total runtime? **O(n)**.

Optimal Solution Code

python
Copier le code

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

Q5: What are the edge cases to consider?

- Negative numbers (`[-1, -2, -3], target = -5`).
- Duplicates (`[3, 3, 4, 2], target = 6`).
- Large input size—optimize for **O(n)**!

Facts About Arrays and Strings (For Two Sum Context)

Arrays 🛠️

1. **Dynamic vs Fixed:** Python arrays (lists) are **dynamic**; C++/Java arrays are often fixed-size.
2. **Indexing is fast!** Access time is **O(1)**.
3. **Hashmaps + Arrays = Power Combo**:
    - Arrays → store order.
    - HashMaps → fast lookups.

Strings 📜

1. **Immutable:** Python strings can't be changed—creates a new one if altered.
2. **Sliding Window Pattern**: Useful for problems like **Longest Substring Without Repeating Characters.**
3. **Efficient Methods**:
    - `s.split()` or `s.replace()`
    - Checking membership: `'a' in s` is **O(n)**.

Q6: Two Sum Variations to Practice

🟢 Two Sum - Sorted Array

- **Twist**: Array is sorted.
- **Solution**: Use two pointers from either end. **O(n)** time:

python
Copier le code
```python
def two_sum_sorted(nums, target):
    left, right = 0, len(nums) - 1
    while left < right:
        total = nums[left] + nums[right]
        if total == target:
            return [left, right]
        elif total < target:
            left += 1
        else:
            right -= 1
```

🟡 Three Sum

- Find **three numbers** that add to `target`.
- Sort array, fix one element, and use two pointers for the rest.
- **O(n²)** solution:

python
Copier le code
```python
def three_sum(nums):
    nums.sort()
    res = []
    for i in range(len(nums)):
        if i > 0 and nums[i] == nums[i-1]:  # Skip duplicates
            continue
        left, right = i+1, len(nums)-1
        while left < right:
            s = nums[i] + nums[left] + nums[right]
            if s == 0:
                res.append([nums[i], nums[left], nums[right]])
                left += 1
                right -= 1
```

```
            elif s < 0:
                left += 1
            else:
                right -= 1
    return res
```

🔴 **Four Sum**

- Similar to Three Sum. Fix two numbers, use a two-pointer strategy. Complexity: **O(n³)**.

Key Insights from Two Sum

1. **HashMaps are MVPs:** Optimize problems with key-value lookups.
2. **Know trade-offs**: Arrays store order; hashmaps provide quick access.
3. **Patterns Rule**: Brute force first for clarity, optimize with a well-known pattern like **hashmaps** or **two pointers**.