

# CodeSignal Python Practice Problems - 2 Week Ramp-Up

## Week 1: Core Patterns and Techniques

### Day 1-2: Array Manipulation

#### Problem 1: Neighbor Sum Array

**Difficulty:** Easy

**Time:** 15 minutes

**Pattern:** Array traversal with edge cases

Given an array of integers, create a new array where each element is the sum of itself and its immediate neighbors. Handle edge cases for first and last elements.

python

```
def neighbor_sum(arr):
```

```
    """
```

Create an array where each element is the sum of itself and its neighbors.

Examples:

Input: [1, 2, 3, 4, 5]

Output: [3, 6, 9, 12, 9]

Explanation:

- arr[0]: 1 + 2 = 3 (no left neighbor)

- arr[1]: 1 + 2 + 3 = 6

- arr[2]: 2 + 3 + 4 = 9

- arr[3]: 3 + 4 + 5 = 12

- arr[4]: 4 + 5 = 9 (no right neighbor)

```
    """
```

*# Your code here*

```
    pass
```

*# Test cases*

```
assert neighbor_sum([1, 2, 3, 4, 5]) == [3, 6, 9, 12, 9]
```

```
assert neighbor_sum([10]) == [10]
```

```
assert neighbor_sum([5, 1]) == [6, 6]
```

```
assert neighbor_sum([]) == []
```

```
print("All tests passed!")
```

#### Problem 2: Move Zeros to End

**Difficulty:** Easy

**Time:** 15 minutes

**Pattern:** Two-pointer technique (same direction)

Move all zeros in an array to the end while maintaining the relative order of non-zero elements.

python

```
def move_zeros_to_end(arr):  
    """  
    Move all zeros to the end of the array while maintaining order of non-zero elements.  
  
    Examples:  
    Input: [0, 1, 0, 3, 12]  
    Output: [1, 3, 12, 0, 0]  
  
    Input: [0, 0, 1]  
    Output: [1, 0, 0]  
    """  
    # Your code here  
    pass  
  
# Test cases  
assert move_zeros_to_end([0, 1, 0, 3, 12]) == [1, 3, 12, 0, 0]  
assert move_zeros_to_end([0, 0, 1]) == [1, 0, 0]  
assert move_zeros_to_end([1, 2, 3]) == [1, 2, 3]  
assert move_zeros_to_end([0]) == [0]  
print("All tests passed!")
```

## Day 3-4: Hash Maps/Dictionaries

### Problem 3: Character Frequency Counter

**Difficulty:** Easy

**Time:** 10 minutes

**Pattern:** Using dictionary for counting

Count the frequency of each character in a string and return the most frequent character(s).

python

```

def char_frequency(s):
    """
    Count character frequencies and find the most frequent character(s).

    Returns a tuple: (frequency_dict, list_of_most_frequent_chars)

    Example:
    Input: "programming"
    Output: ({'p': 1, 'r': 2, 'o': 1, 'g': 2, 'a': 1, 'm': 2, 'i': 1, 'n': 1}, ['r', 'g', 'm'])
    """
    # Your code here
    pass

# Test cases
freq, most = char_frequency("programming")
assert freq == {'p': 1, 'r': 2, 'o': 1, 'g': 2, 'a': 1, 'm': 2, 'i': 1, 'n': 1}
assert set(most) == {'r', 'g', 'm'}

freq, most = char_frequency("aabbcc")
assert freq == {'a': 2, 'b': 2, 'c': 2}
assert set(most) == {'a', 'b', 'c'}

freq, most = char_frequency("")
assert freq == {}
assert most == []

print("All tests passed!")

```

## Problem 4: First Non-Repeating Character

**Difficulty:** Easy-Medium

**Time:** 15 minutes

**Pattern:** Hash map for frequency + two-pass technique

Find the first non-repeating character in a string. Return its index, or -1 if none exists.

python

```
def first_non_repeating(s):
```

```
    """
```

Find the index of the first non-repeating character.

Examples:

Input: "leetcode"

Output: 0 (character 'l' at index 0)

Input: "loveleetcode"

Output: 2 (character 'v' at index 2)

Input: "aabb"

Output: -1 (no non-repeating character)

```
    """
```

*# Your code here*

```
    pass
```

*# Test cases*

```
assert first_non_repeating("leetcode") == 0
```

```
assert first_non_repeating("loveleetcode") == 2
```

```
assert first_non_repeating("aabb") == -1
```

```
assert first_non_repeating("z") == 0
```

```
assert first_non_repeating("") == -1
```

```
print("All tests passed!")
```

## Day 5-6: String Operations

### Problem 5: Valid Palindrome (with cleaning)

**Difficulty:** Easy

**Time:** 15 minutes

**Pattern:** String manipulation + two-pointer

Check if a string is a palindrome, considering only alphanumeric characters and ignoring case.

python

```
def is_valid_palindrome(s):
```

```
    """
```

Check if string is a palindrome (alphanumeric only, case-insensitive).

Examples:

Input: "A man, a plan, a canal: Panama"

Output: True

Input: "race a car"

Output: False

```
    """
```

*# Your code here*

```
    pass
```

*# Test cases*

```
assert is_valid_palindrome("A man, a plan, a canal: Panama") == True
```

```
assert is_valid_palindrome("race a car") == False
```

```
assert is_valid_palindrome("Was it a car or a cat I saw?") == True
```

```
assert is_valid_palindrome("") == True
```

```
assert is_valid_palindrome("a") == True
```

```
print("All tests passed!")
```

## Problem 6: Longest Substring Without Repeating Characters

**Difficulty:** Medium

**Time:** 20 minutes

**Pattern:** Sliding window with hash map

Find the length of the longest substring without repeating characters.

python

```
def longest_unique_substring(s):
```

```
    """
```

Find the length of the longest substring without repeating characters.

Examples:

Input: "abcabcbb"

Output: 3 (substring "abc")

Input: "bbbbbb"

Output: 1 (substring "b")

Input: "pwwkew"

Output: 3 (substring "wke" or "kew")

```
    """
```

*# Your code here*

```
    pass
```

*# Test cases*

```
assert longest_unique_substring("abcabcbb") == 3
```

```
assert longest_unique_substring("bbbbbb") == 1
```

```
assert longest_unique_substring("pwwkew") == 3
```

```
assert longest_unique_substring("") == 0
```

```
assert longest_unique_substring("dvdvf") == 3
```

```
assert longest_unique_substring("abcdef") == 6
```

```
print("All tests passed!")
```

## Week 2: Problem-Solving Patterns

### Day 1-2: Common Patterns Review

#### Problem 7: Two Sum in Sorted Array

**Difficulty:** Easy

**Time:** 15 minutes

**Pattern:** Two pointers (opposite direction)

Given a sorted array and a target sum, find two numbers that add up to the target. Return their indices.

```
python
```

```
def two_sum_sorted(arr, target):
```

```
    """
```

Find two numbers in sorted array that sum to target.

Examples:

Input: arr = [2, 7, 11, 15], target = 9

Output: [0, 1] (indices of 2 and 7)

Input: arr = [2, 3, 4], target = 6

Output: [0, 2] (indices of 2 and 4)

Return empty list if no solution exists.

```
    """
```

```
    # Your code here
```

```
    pass
```

```
# Test cases
```

```
assert two_sum_sorted([2, 7, 11, 15], 9) == [0, 1]
```

```
assert two_sum_sorted([2, 3, 4], 6) == [0, 2]
```

```
assert two_sum_sorted([1, 2, 3, 4], 10) == []
```

```
assert two_sum_sorted([1, 2], 3) == [0, 1]
```

```
print("All tests passed!")
```

## Problem 8: Maximum Subarray Sum (Fixed Size)

**Difficulty:** Easy

**Time:** 15 minutes

**Pattern:** Fixed-size sliding window

Find the maximum sum of any contiguous subarray of size k.

```
python
```

```
def max_subarray_sum_fixed(arr, k):
```

```
    """
```

Find maximum sum of any subarray of size k.

Examples:

Input: arr = [2, 1, 5, 1, 3, 2], k = 3

Output: 9 (subarray [5, 1, 3])

Input: arr = [2, 3, 4, 1, 5], k = 2

Output: 7 (subarray [3, 4])

```
    """
```

*# Your code here*

```
    pass
```

*# Test cases*

```
assert max_subarray_sum_fixed([2, 1, 5, 1, 3, 2], 3) == 9
```

```
assert max_subarray_sum_fixed([2, 3, 4, 1, 5], 2) == 7
```

```
assert max_subarray_sum_fixed([1], 1) == 1
```

```
assert max_subarray_sum_fixed([1, 2, 3], 4) == 0 # k > array length
```

```
print("All tests passed!")
```

## Day 3-4: Edge Cases and Optimization

### Problem 9: Find All Anagrams

**Difficulty:** Medium

**Time:** 20 minutes

**Pattern:** Sliding window with frequency map

Find all starting indices of anagrams of a pattern in a string.

```
python
```



```
def find_anagrams(s, p):
```

```
    """
```

Find all starting indices of p's anagrams in s.

Examples:

Input: s = "cbaebabacd", p = "abc"

Output: [0, 6]

Explanation:

- Substring at index 0: "cba" is anagram of "abc"
- Substring at index 6: "bac" is anagram of "abc"

Input: s = "abab", p = "ab"

Output: [0, 1, 2]

```
    """
```

*# Your code here*

```
    pass
```

*# Test cases*

```
assert find_anagrams("cbaebabacd", "abc") == [0, 6]
```

```
assert find_anagrams("abab", "ab") == [0, 1, 2]
```

```
assert find_anagrams("aaaa", "aa") == [0, 1, 2]
```

```
assert find_anagrams("abc", "xyz") == []
```

```
print("All tests passed!")
```

## Problem 10: Valid Parentheses

**Difficulty:** Easy

**Time:** 15 minutes

**Pattern:** Stack (bonus pattern)

Check if a string containing parentheses, brackets, and braces is valid.

python

```
def is_valid_parentheses(s):
    """
    Check if parentheses/brackets/braces are valid.

    Examples:
    Input: "()"
    Output: True

    Input: "()[]{}"
    Output: True

    Input: "["
    Output: False

    Input: "(D]"
    Output: False
    """
    # Your code here
    pass

# Test cases
assert is_valid_parentheses("()") == True
assert is_valid_parentheses "()[]{}" == True
assert is_valid_parentheses "[" == False
assert is_valid_parentheses "(D]" == False
assert is_valid_parentheses "{}[]" == True
assert is_valid_parentheses "" == True
print("All tests passed!")
```

## Day 5-6: Time Yourself Practice

### Problem 11: Group Anagrams

**Difficulty:** Medium

**Time:** 20 minutes

**Pattern:** Hash map with sorted key

Group strings that are anagrams of each other.

python

```

def group_anagrams(strs):
    """
    Group strings that are anagrams of each other.

    Example:
    Input: ["eat", "tea", "tan", "ate", "nat", "bat"]
    Output: [["eat", "tea", "ate"], ["tan", "nat"], ["bat"]]

    Note: Order of groups and order within groups may vary.
    """
    # Your code here
    pass

# Test cases
result = group_anagrams(["eat", "tea", "tan", "ate", "nat", "bat"])
expected = [["eat", "tea", "ate"], ["tan", "nat"], ["bat"]]
# Convert to sets for comparison (order doesn't matter)
result_sets = [set(group) for group in result]
expected_sets = [set(group) for group in expected]
assert sorted(result_sets, key=str) == sorted(expected_sets, key=str)

result = group_anagrams([""])
assert result == [[""]]

result = group_anagrams(["a"])
assert result == [["a"]]

print("All tests passed!")

```

## Problem 12: Container With Most Water

**Difficulty:** Medium

**Time:** 20 minutes

**Pattern:** Two pointers (optimization problem)

Given an array of heights, find two lines that together with the x-axis form a container that holds the most water.

python

```
def max_water_container(heights):
```

```
    """
```

Find maximum water that can be contained.

Example:

Input: [1, 8, 6, 2, 5, 4, 8, 3, 7]

Output: 49

Explanation: Lines at index 1 (height=8) and index 8 (height=7)

form container with area =  $7 * 7 = 49$

The area is calculated as:

```
min(height[i], height[j]) * (j - i)
```

```
    """
```

```
# Your code here
```

```
pass
```

```
# Test cases
```

```
assert max_water_container([1, 8, 6, 2, 5, 4, 8, 3, 7]) == 49
```

```
assert max_water_container([1, 1]) == 1
```

```
assert max_water_container([4, 3, 2, 1, 4]) == 16
```

```
assert max_water_container([1, 2, 1]) == 2
```

```
print("All tests passed!")
```

## Tips for Practice

1. **Time yourself** - Try to complete each problem within the suggested time
2. **Think before coding** - Spend 2-3 minutes understanding the problem and planning your approach
3. **Handle edge cases** - Empty arrays, single elements, None values
4. **Test your code** - Run the test cases before checking solutions
5. **Learn from mistakes** - If stuck, try for 5 more minutes before looking at hints

## Difficulty Progression

- Start with Easy problems (1-5, 7-8, 10)
- Move to Easy-Medium (4)
- Tackle Medium problems (6, 9, 11-12)

Good luck with your CodeSignal assessment!