CodeSignal Python Practice Solutions - 2 Week Ramp-Up

Week 1: Core Patterns and Techniques

Day 1-2: Array Manipulation

Problem 1: Neighbor Sum Array - SOLUTION

Pattern: Array traversal with edge cases

python

```python
def neighbor_sum(arr):
    """
    Create an array where each element is the sum of itself and its neighbors.
    """
    if not arr:
        return []

    n = len(arr)
    result = [0] * n

    for i in range(n):
        # Add current element
        result[i] = arr[i]

        # Add left neighbor if exists
        if i > 0:
            result[i] += arr[i - 1]

        # Add right neighbor if exists
        if i < n - 1:
            result[i] += arr[i + 1]

    return result

# 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("Problem 1: All tests passed!")
```

Problem 2: Move Zeros to End - SOLUTION

Pattern: Two-pointer technique (same direction)

```python
```

```
def move_zeros_to_end(arr):
    """
    Move all zeros to the end while maintaining order.
    Two-pointer approach: one for iteration, one for non-zero placement.
    """
    if not arr:
        return arr

    # Pointer for position of next non-zero element
    insert_pos = 0

    # Move all non-zero elements to the front
    for i in range(len(arr)):
        if arr[i] != 0:
            arr[insert_pos] = arr[i]
            insert_pos += 1

    # Fill remaining positions with zeros
    while insert_pos < len(arr):
        arr[insert_pos] = 0
        insert_pos += 1

    return arr

# Alternative solution using swap
def move_zeros_to_end_v2(arr):
    """
    Alternative: Swap non-zero elements to front
    """
    insert_pos = 0

    for i in range(len(arr)):
        if arr[i] != 0:
            arr[i], arr[insert_pos] = arr[insert_pos], arr[i]
            insert_pos += 1

    return arr

# 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("Problem 2: All tests passed!")
```

Day 3-4: Hash Maps/Dictionaries

Problem 3: Character Frequency Counter - SOLUTION

Pattern: Dictionary for counting

python

```python
def char_frequency(s):
    """
    Count frequencies and find most frequent characters.
    """
    # Count frequencies
    freq = {}
    for char in s:
        if char in freq:
            freq[char] += 1
        else:
            freq[char] = 1

    # Handle empty string
    if not freq:
        return {}, []

    # Find maximum frequency
    max_freq = max(freq.values())

    # Find all characters with max frequency
    most_frequent = [char for char, count in freq.items() if count == max_freq]

    return freq, most_frequent

# Alternative using collections
from collections import defaultdict, Counter

def char_frequency_v2(s):
    """Using defaultdict"""
    freq = defaultdict(int)
    for char in s:
        freq[char] += 1

    if not freq:
        return {}, []

    max_freq = max(freq.values())
    most_frequent = [char for char, count in freq.items() if count == max_freq]

    return dict(freq), most_frequent

def char_frequency_v3(s):
    """Using Counter (most Pythonic)"""
    freq = Counter(s)

    if not freq:
        return {}, []

    max_count = freq.most_common(1)[0][1] if freq else 0
    most_frequent = [char for char, count in freq.items() if count == max_count]

    return dict(freq), most_frequent

# 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("Problem 3: All tests passed!")
```

Problem 4: First Non-Repeating Character - SOLUTION

Pattern: Two-pass with hash map

python

```python
def first_non_repeating(s):
    """
    Find first non-repeating character's index.
    Two-pass: first count, then find first unique.
    """
    # Handle empty string
    if not s:
        return -1

    # First pass: count frequencies
    char_count = {}
    for char in s:
        char_count[char] = char_count.get(char, 0) + 1

    # Second pass: find first character with count 1
    for i, char in enumerate(s):
        if char_count[char] == 1:
            return i

    return -1

# Alternative using Counter
from collections import Counter

def first_non_repeating_v2(s):
    """Using Counter for cleaner code"""
    if not s:
        return -1

    char_count = Counter(s)

    for i, char in enumerate(s):
        if char_count[char] == 1:
            return i

    return -1

# 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("Problem 4: All tests passed!")
```

Day 5-6: String Operations

Problem 5: Valid Palindrome - SOLUTION

Pattern: String cleaning + two-pointer

python

```python
def is_valid_palindrome(s):
    """
    Check palindrome with only alphanumeric, case-insensitive.
    Method 1: Clean first, then check.
    """
    # Clean the string
    cleaned = ""
    for char in s:
        if char.isalnum():
            cleaned += char.lower()

    # Check palindrome using two pointers
    left = 0
    right = len(cleaned) - 1

    while left < right:
        if cleaned[left] != cleaned[right]:
            return False
        left += 1
        right -= 1

    return True

def is_valid_palindrome_v2(s):
    """
    Method 2: Two pointers without creating new string.
    More memory efficient.
    """
    left = 0
    right = len(s) - 1

    while left < right:
        # Skip non-alphanumeric from left
        while left < right and not s[left].isalnum():
            left += 1

        # Skip non-alphanumeric from right
        while left < right and not s[right].isalnum():
            right -= 1

        # Compare characters
        if s[left].lower() != s[right].lower():
            return False
        left += 1
        right -= 1

    return True

def is_valid_palindrome_v3(s):
    """
    Method 3: Pythonic one-liner approach
    """
    cleaned = ''.join(char.lower() for char in s if char.isalnum())
    return cleaned == cleaned[::-1]

# 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("Problem 5: All tests passed!")
```

Problem 6: Longest Substring Without Repeating - SOLUTION

Pattern: Sliding window with hash map

python

```
def longest_unique_substring(s):
    """
    Find longest substring without repeating characters.
    Sliding window: expand right, contract left when duplicate found.
    """
    if not s:
        return 0

    # Track character positions
    char_index = {}
    max_length = 0
    start = 0  # Start of current window

    for end in range(len(s)):
        char = s[end]

        # If character is repeated and within current window
        if char in char_index and char_index[char] >= start:
            # Move start to position after the repeated character
            start = char_index[char] + 1

        # Update character's latest position
        char_index[char] = end

        # Update max length
        current_length = end - start + 1
        max_length = max(max_length, current_length)

    return max_length

def longest_unique_substring_v2(s):
    """
    Alternative: Using set for cleaner logic
    """
    char_set = set()
    max_length = 0
    left = 0

    for right in range(len(s)):
        # Shrink window from left while duplicate exists
        while s[right] in char_set:
            char_set.remove(s[left])
            left += 1
        char_set.add(s[right])
        max_length = max(max_length, right - left + 1)

    return max_length

# Test cases
assert longest_unique_substring("abcabcbb") == 3
assert longest_unique_substring("bbbbb") == 1
assert longest_unique_substring("pwwkew") == 3
assert longest_unique_substring("") == 0
assert longest_unique_substring("dvdf") == 3
assert longest_unique_substring("abcdef") == 6
print("Problem 6: All tests passed!")
```

Week 2: Problem-Solving Patterns

Day 1-2: Common Patterns Review

Problem 7: Two Sum in Sorted Array - SOLUTION

Pattern: Two pointers (opposite direction)

python

```python
def two_sum_sorted(arr, target):
    """
    Two pointers from both ends.
    If sum too small, move left pointer right.
    If sum too large, move right pointer left.
    """
    left = 0
    right = len(arr) - 1

    while left < right:
        current_sum = arr[left] + arr[right]

        if current_sum == target:
            return [left, right]
        elif current_sum < target:
            left += 1  # Need larger sum
        else:
            right -= 1  # Need smaller sum

    return []  # No solution found

# 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("Problem 7: All tests passed!")
```

Problem 8: Maximum Subarray Sum (Fixed Size) - SOLUTION

Pattern: Fixed-size sliding window

python

```
def max_subarray_sum_fixed(arr, k):
    """
    Sliding window of size k.
    Calculate first window, then slide by removing left and adding right.
    """
    if len(arr) < k:
        return 0

    # Calculate sum of first window
    window_sum = sum(arr[:k])
    max_sum = window_sum

    # Slide the window
    for i in range(k, len(arr)):
        # Remove leftmost element of previous window
        # Add rightmost element of new window
        window_sum = window_sum - arr[i - k] + arr[i]
        max_sum = max(max_sum, window_sum)

    return max_sum

# 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
print("Problem 8: All tests passed!")
```

Day 3-4: Edge Cases and Optimization

Problem 9: Find All Anagrams - SOLUTION

Pattern: Sliding window with frequency comparison

```
python
```

```python
def find_anagrams(s, p):
    """
    Sliding window with character frequency comparison.
    Window size = len(p), compare frequencies at each position.
    """
    from collections import Counter

    if len(p) > len(s):
        return []

    result = []
    p_count = Counter(p)
    window_count = Counter(s[:len(p)])

    # Check first window
    if window_count == p_count:
        result.append(0)

    # Slide the window
    for i in range(len(p), len(s)):
        # Add new character
        window_count[s[i]] = window_count.get(s[i], 0) + 1

        # Remove old character
        old_char = s[i - len(p)]
        window_count[old_char] -= 1
        if window_count[old_char] == 0:
            del window_count[old_char]

        # Check if current window is anagram
        if window_count == p_count:
            result.append(i - len(p) + 1)

    return result

# 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("Problem 9: All tests passed!")
```

Problem 10: Valid Parentheses - SOLUTION

Pattern: Stack for matching pairs

python

```python
def is_valid_parentheses(s):
    """
    Use stack to track opening brackets.
    When closing bracket found, check if it matches the last opening.
    """
    stack = []
    mapping = {')': '(', '}': '{', ']': '['}

    for char in s:
        if char in mapping:
            # Closing bracket
            if not stack or stack[-1] != mapping[char]:
                return False
            stack.pop()
        else:
            # Opening bracket
            stack.append(char)

    # Valid if stack is empty
    return len(stack) == 0

# Test cases
assert is_valid_parentheses("()") == True
assert is_valid_parentheses("()[]{}") == True
assert is_valid_parentheses("(]") == False
assert is_valid_parentheses("([)]") == False
assert is_valid_parentheses("{[]}") == True
assert is_valid_parentheses("") == True
print("Problem 10: All tests passed!")
```

Day 5-6: Time Yourself Practice

Problem 11: Group Anagrams - SOLUTION

Pattern: Hash map with sorted key

python

```
def group_anagrams(strs):
    """
    Group anagrams using sorted string as key.
    All anagrams will have the same sorted characters.
    """
    from collections import defaultdict

    anagram_groups = defaultdict(list)

    for s in strs:
        # Sort the string to create a key
        key = ''.join(sorted(s))
        anagram_groups[key].append(s)

    return list(anagram_groups.values())

def group_anagrams_v2(strs):
    """
    Alternative: Using tuple of character counts as key
    More efficient for long strings
    """
    from collections import defaultdict

    anagram_groups = defaultdict(list)

    for s in strs:
        # Count frequency of each character
        count = [0] * 26  # for lowercase letters
        for char in s:
            count[ord(char) - ord('a')] += 1

        # Use tuple of counts as key
        key = tuple(count)
        anagram_groups[key].append(s)

    return list(anagram_groups.values())

# 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("Problem 11: All tests passed!")
```

Problem 12: Container With Most Water - SOLUTION

Pattern: Two pointers (optimization problem)

python

```python
def max_water_container(heights):
    """
    Two pointers: start at widest container.
    Move pointer with smaller height inward.
    Logic: Moving the taller line inward can't increase area
    (width decreases, height limited by shorter line).
    """
    if len(heights) < 2:
        return 0

    max_area = 0
    left = 0
    right = len(heights) - 1

    while left < right:
        # Calculate current area
        width = right - left
        height = min(heights[left], heights[right])
        current_area = width * height
        max_area = max(max_area, current_area)

        # Move pointer with smaller height
        if heights[left] < heights[right]:
            left += 1
        else:
            right -= 1

    return max_area

def max_water_container_v2(heights):
    """
    Slight optimization: skip smaller heights when moving pointers
    """
    max_area = 0
    left = 0
    right = len(heights) - 1

    while left < right:
        # Calculate area
        if heights[left] < heights[right]:
            area = heights[left] * (right - left)
            max_area = max(max_area, area)
            # Skip all heights smaller than current left
            current_height = heights[left]
            while left < right and heights[left] <= current_height:
                left += 1
        else:
            area = heights[right] * (right - left)
            max_area = max(max_area, area)
            # Skip all heights smaller than current right
            current_height = heights[right]
            while left < right and heights[right] <= current_height:
                right -= 1

    return max_area

# 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("Problem 12: All tests passed!")
```

Key Patterns Summary

1. Two Pointers

- **Same direction**: Move zeros (Problem 2)
- **Opposite direction**: Two sum sorted (Problem 7), Container with water (Problem 12)
- **String palindrome**: Valid palindrome (Problem 5)

2. Sliding Window

- **Fixed size**: Max subarray sum (Problem 8)
- **Variable size**: Longest unique substring (Problem 6)
- **With frequency map**: Find anagrams (Problem 9)

3. Hash Maps/Dictionaries

- **Counting**: Character frequency (Problem 3)
- **Lookup**: First non-repeating (Problem 4)
- **Grouping**: Group anagrams (Problem 11)
- **Position tracking**: Longest unique substring (Problem 6)

4. Additional Patterns

- **Stack**: Valid parentheses (Problem 10)
- **Array traversal**: Neighbor sum (Problem 1)

Time Complexity Analysis

Problem	Time Complexity	Space Complexity
1. Neighbor Sum	O(n)	O(n)
2. Move Zeros	O(n)	O(1)
3. Character Frequency	O(n)	O(k) where k = unique chars
4. First Non-Repeating	O(n)	O(k) where k = unique chars
5. Valid Palindrome	O(n)	O(1) or O(n)
6. Longest Unique Substring	O(n)	O(min(n, m)) where m = charset size
7. Two Sum Sorted	O(n)	O(1)
8. Max Subarray Fixed	O(n)	O(1)
9. Find Anagrams	O(n)	O(k) where k = pattern size
10. Valid Parentheses	O(n)	O(n)
11. Group Anagrams	O(n*m*log(m))	O(n*m) where m = avg string length
12. Container With Water	O(n)	O(1)

Tips for CodeSignal Success

1. **Read carefully**: Understand what the problem is asking for
2. **Consider edge cases**: Empty inputs, single elements, duplicates
3. **Choose appropriate data structures**: Dict for counting, Set for uniqueness
4. **Optimize after solving**: Get it working first, then optimize
5. **Use Python built-ins**: Counter, defaultdict, enumerate, zip
6. **Test incrementally**: Test with simple cases first

Good luck with your assessment!