Python Data Structures: Practice Solutions

Practice Assignment 1: Finding the Farthest Point

Problem: Write a function that takes a list of tuples representing (x, y) coordinates and returns the tuple with the largest distance from the origin (0, 0).

python

```python
def find_farthest_point(coordinates):
    """
    Takes a list of tuples representing (x, y) coordinates and returns the tuple
    with the largest distance from the origin (0, 0).

    Args:
        coordinates: A list of tuples, each containing x and y coordinates

    Returns:
        The tuple with the largest distance from origin, or None if list is empty
    """
    if not coordinates:
        return None

    max_distance = 0
    farthest_point = None

    for point in coordinates:
        x, y = point
        # Calculate Euclidean distance: sqrt(x^2 + y^2)
        distance = (x**2 + y**2)**0.5

        if distance > max_distance:
            max_distance = distance
            farthest_point = point

    return farthest_point
```

Example usage:

python

```python
points = [(1, 2), (3, 4), (-5, 12), (0, 0)]
result = find_farthest_point(points)  # Returns (-5, 12)
```

Practice Assignment 2: Set Operations on Two Lists

Problem: Write a function that takes two lists and returns a tuple containing: 1) A set of elements common to both lists, 2) A set of elements unique to the first list, 3) A set of elements unique to the second list.

python

```python
def analyze_lists(list1, list2):
    """
    Takes two lists and returns a tuple containing three sets:
    1. Elements common to both lists
    2. Elements unique to the first list
    3. Elements unique to the second list

    Args:
        list1: The first list
        list2: The second list

    Returns:
        A tuple of three sets: (common, unique_to_list1, unique_to_list2)
    """
    # Convert lists to sets for set operations
    set1 = set(list1)
    set2 = set(list2)

    # Find common elements (intersection)
    common = set1 & set2

    # Find elements unique to first list (difference)
    unique_to_list1 = set1 - set2

    # Find elements unique to second list (difference)
    unique_to_list2 = set2 - set1

    return (common, unique_to_list1, unique_to_list2)
```

Example usage:

python

```python
first_list = [1, 2, 3, 4, 5]
second_list = [4, 5, 6, 7, 8]
common, unique1, unique2 = analyze_lists(first_list, second_list)
# common = {4, 5}
# unique1 = {1, 2, 3}
# unique2 = {6, 7, 8}
```

Practice Assignment 3: Character Frequency Counter

Problem: Write a function that takes a string as input and returns a dictionary where keys are characters and values are the number of times each character appears in the string (character frequency).

python

```python
def character_frequency(text):
    """
    Takes a string as input and returns a dictionary where keys are characters
    and values are the number of times each character appears in the string.

    Args:
        text: The input string

    Returns:
        A dictionary mapping characters to their frequencies
    """
    frequency = {}

    for char in text:
        if char in frequency:
            frequency[char] += 1
        else:
            frequency[char] = 1

    # Alternative one-liner using get with default value:
    # for char in text:
    #     frequency[char] = frequency.get(char, 0) + 1

    return frequency
```

Example usage:

python

```python
text = "hello world"
freq = character_frequency(text)
# freq = {'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1}
```

Practice Assignment 4: Element Index Mapping

Problem: Write a function that takes a sequence (list, tuple, or string) and returns a dictionary that maps each unique element to all the indices where it appears in the sequence.

python

```python
def element_indices(sequence):
    """
    Takes a sequence (list, tuple, or string) and returns a dictionary
    that maps each unique element to all the indices where it appears.

    Args:
        sequence: A list, tuple, or string

    Returns:
        A dictionary mapping elements to lists of indices
    """
    index_map = {}

    for i, element in enumerate(sequence):
        if element in index_map:
            index_map[element].append(i)
        else:
            index_map[element] = [i]

    return index_map
```

Example usage:

```python
text = "banana"
indices = element_indices(text)
# indices = {'b': [0], 'a': [1, 3, 5], 'n': [2, 4]}

numbers = [1, 2, 3, 2, 1, 4, 5]
indices = element_indices(numbers)
# indices = {1: [0, 4], 2: [1, 3], 3: [2], 4: [5], 5: [6]}
```

Challenging Problem 1: Skill Mapping

Problem: Write a function that takes a list of dictionaries (each representing a person with 'name' and 'skills' keys, where 'skills' is a list of strings) and returns: 1) A set of all unique skills across all people, 2) A dictionary mapping each skill to a list of people who have that skill.

python

```python
def analyze_skills(people):
    """
    Takes a list of dictionaries (each representing a person with 'name' and 'skills' keys)
    and returns:
    1. A set of all unique skills across all people
    2. A dictionary mapping each skill to a list of people who have that skill

    Args:
        people: A list of dictionaries, each with 'name' and 'skills' keys

    Returns:
        A tuple with (set of unique skills, dictionary mapping skills to people)
    """
    all_skills = set()
    skill_to_people = {}

    for person in people:
        name = person['name']
        skills = person['skills']

        # Add all skills to the set of unique skills
        all_skills.update(skills)

        # Map each skill to the person
        for skill in skills:
            if skill in skill_to_people:
                skill_to_people[skill].append(name)
            else:
                skill_to_people[skill] = [name]

    return (all_skills, skill_to_people)
```

Example usage:

python

```python
people = [
    {'name': 'Alice', 'skills': ['Python', 'SQL', 'JavaScript']},
    {'name': 'Bob', 'skills': ['Java', 'C++', 'Python']},
    {'name': 'Charlie', 'skills': ['JavaScript', 'HTML', 'CSS']}
]

unique_skills, skill_map = analyze_skills(people)
# unique_skills = {'Python', 'SQL', 'JavaScript', 'Java', 'C++', 'HTML', 'CSS'}
# skill_map = {
#     'Python': ['Alice', 'Bob'],
#     'SQL': ['Alice'],
#     'JavaScript': ['Alice', 'Charlie'],
#     'Java': ['Bob'],
#     'C++': ['Bob'],
#     'HTML': ['Charlie'],
#     'CSS': ['Charlie']
# }
```

Challenging Problem 2: Matrix Transposition

Problem: Write a function that performs a "matrix transposition" by taking a tuple of tuples (representing a matrix) and returning a new tuple of tuples where rows and columns are swapped.

python

```python
def transpose_matrix(matrix):
    """
    Takes a tuple of tuples (representing a matrix) and returns a new tuple of tuples
    where rows and columns are swapped.

    Args:
        matrix: A tuple of tuples, each inner tuple representing a row

    Returns:
        A transposed tuple of tuples
    """
    # Check if matrix is empty
    if not matrix or not matrix[0]:
        return ()

    # Number of rows and columns in the original matrix
    rows = len(matrix)
    cols = len(matrix[0])

    # Create transposed matrix using tuple comprehension
    transposed = tuple(
        tuple(matrix[row][col] for row in range(rows))
        for col in range(cols)
    )

    return transposed
```

Example usage:

python

```python
matrix = (
    (1, 2, 3),
    (4, 5, 6)
)
transposed = transpose_matrix(matrix)
# transposed = ((1, 4), (2, 5), (3, 6))
```

Challenging Problem 3: Nested Dictionary for Student Data

Problem: Create a function that processes student data in the form of a list of dictionaries. Each dictionary contains a student's name, grade, and scores in different subjects. The function should return a nested dictionary where the outer keys are grades, the inner keys are student names, and the values are the average scores for each student.

python

```python
def organize_student_data(students):
    """
    Processes student data and returns a nested dictionary where:
    - Outer keys are grades
    - Inner keys are student names
    - Values are the average scores for each student

    Args:
        students: A list of dictionaries with 'name', 'grade', and 'scores' keys

    Returns:
        A nested dictionary organized by grade and student name
    """
    result = {}

    for student in students:
        name = student['name']
        grade = student['grade']
        scores = student['scores']

        # Calculate average score
        average_score = sum(scores.values()) / len(scores)

        # Initialize grade dictionary if not present
        if grade not in result:
            result[grade] = {}

        # Add student with average score
        result[grade][name] = average_score

    return result
```

Example usage:

python

```python
students = [
    {
        'name': 'Alice',
        'grade': 10,
        'scores': {'Math': 90, 'Science': 85, 'English': 92}
    },
    {
        'name': 'Bob',
        'grade': 9,
        'scores': {'Math': 88, 'Science': 79, 'English': 85}
    },
    {
        'name': 'Charlie',
        'grade': 10,
        'scores': {'Math': 75, 'Science': 80, 'English': 88}
    }
]

organized_data = organize_student_data(students)
# organized_data = {
#     10: {
#         'Alice': 89.0,
#         'Charlie': 81.0
#     },
#     9: {
#         'Bob': 84.0
#     }
# }
```

Challenging Problem 4: Longest Common Subsequence

Problem: Write a function that takes two sequences (strings, lists, or tuples) and finds the longest common subsequence between them.

python

```python
def longest_common_subsequence(seq1, seq2):
    """
    Finds the longest common subsequence between two sequences.

    Args:
        seq1: First sequence (string, list, or tuple)
        seq2: Second sequence (string, list, or tuple)

    Returns:
        The longest common subsequence as the same type as the first sequence
    """
    # Convert inputs to lists for consistent handling
    list1 = list(seq1)
    list2 = list(seq2)

    # Initialize the DP table with zeros
    m, n = len(list1), len(list2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Fill the DP table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if list1[i-1] == list2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    # Backtrack to find the LCS
    lcs = []
    i, j = m, n
    while i > 0 and j > 0:
        if list1[i-1] == list2[j-1]:
            lcs.append(list1[i-1])
            i -= 1
            j -= 1
        elif dp[i-1][j] > dp[i][j-1]:
            i -= 1
        else:
            j -= 1

    # Reverse the LCS and convert to the same type as the first sequence
    lcs.reverse()
```

```python
    # Return the LCS in the same type as the first input
    if isinstance(seq1, str):
        return ''.join(lcs)
    elif isinstance(seq1, tuple):
        return tuple(lcs)
    else:
        return lcs
```

Example usage:

```python
str1 = "ABCBDAB"
str2 = "BDCABA"
lcs = longest_common_subsequence(str1, str2)  # Returns "BCBA"

list1 = [1, 2, 3, 4, 5]
list2 = [3, 4, 1, 2, 5]
lcs = longest_common_subsequence(list1, list2)  # Returns [3, 4, 5]
```

Challenging Problem 5: Product Category Analysis

Problem: Create a function that analyzes a dictionary of products where keys are product IDs and values are dictionaries containing: name (product name), category (product category), and price (product price). The function should return a dictionary where keys are categories and values are sets of product IDs in that category, sorted by price (highest to lowest).

python

```python
def analyze_products(products):
    """
    Analyzes a dictionary of products and returns a dictionary where keys are
    categories and values are sets of product IDs in that category, sorted by price.

    Args:
        products: A dictionary mapping product IDs to product info dictionaries

    Returns:
        A dictionary mapping categories to sorted sets of product IDs
    """
    # Dictionary to store products by category
    categories = {}

    # Process each product
    for product_id, product_info in products.items():
        category = product_info['category']

        # Add category if not present
        if category not in categories:
            categories[category] = []

        # Add product ID and price to the category list
        categories[category].append((product_id, product_info['price']))

    # Sort products by price (highest to lowest) and convert to sets
    result = {}
    for category, products_list in categories.items():
        # Sort by price in descending order
        sorted_products = sorted(products_list, key=lambda x: x[1], reverse=True)
        # Extract just the product IDs and convert to a set
        result[category] = set(product_id for product_id, _ in sorted_products)

    return result
```

Example usage:

python

```python
products = {
    'p1': {'name': 'Laptop', 'category': 'Electronics', 'price': 1200},
    'p2': {'name': 'Headphones', 'category': 'Electronics', 'price': 250},
    'p3': {'name': 'Chair', 'category': 'Furniture', 'price': 300},
    'p4': {'name': 'Table', 'category': 'Furniture', 'price': 500},
    'p5': {'name': 'Phone', 'category': 'Electronics', 'price': 800}
}

categorized = analyze_products(products)
# categorized = {
#     'Electronics': {'p1', 'p5', 'p2'},  # Sorted by price: Laptop, Phone, Headphones
#     'Furniture': {'p4', 'p3'}           # Sorted by price: Table, Chair
# }
```