7 TO - DO - Task

Please complete all the problem listed below.

7.1 Warming Up Exercise:

In this exercise, you'll work with daily time allocation data recorded by a group of students. Each student tracked the number of hours spent studying, watching entertainment, and sleeping for 15 consecutive days.

- **Dataset:** Each record represents one day's data in the format:

(study _hours,entertainment hours,sleep hours) • A

sample dataset is provided below.

Daily Student Productivity Data

```
# Daily time (in hours): [study, entertainment, sleep] time_data = [
    (3.5, 2.0, 7.0), (5.0, 1.5, 6.5), (2.5, 3.0, 8.0),
    (4.0, 2.0, 6.0), (1.5, 4.5, 9.0), (3.0, 2.5, 7.5),
    (5.5, 1.0, 6.0), (2.0, 3.5, 8.5), (4.5, 2.0, 7.0),
    (3.0, 3.0, 7.5), (6.0, 1.5, 6.0), (2.5, 4.0, 8.0),
    (4.0, 2.5, 7.0), (5.0, 2.0, 6.5), (3.5, 2.5, 7.0)
]
```

Complete all the tasks below:

Task 1. Classify Study Time:

1. Create empty lists for study time classifications:
   (a) Low: less than 3 hours.
   (b) Moderate: between 3 and 5 hours.
   (c) High: more than 5 hours.
2. Iterate over the time_data list and add each study hour to the appropriate category.
3. Print the lists to verify the classifications.

```python
#1
time_data = [
(3.5, 2.0, 7.0), (5.0, 1.5, 6.5), (2.5, 3.0, 8.0),
(4.0, 2.0, 6.0), (1.5, 4.5, 9.0), (3.0, 2.5, 7.5),
(5.5, 1.0, 6.0), (2.0, 3.5, 8.5), (4.5, 2.0, 7.0),
(3.0, 3.0, 7.5), (6.0, 1.5, 6.0), (2.5, 4.0, 8.0),
(4.0, 2.5, 7.0), (5.0, 2.0, 6.5), (3.5, 2.5, 7.0)
]
low = []
moderate = []
high = []

for study, entertainment, sleep in time_data:
    if study < 3:
        low.append(study)
    elif 3 <= study <= 5:
        moderate.append(study)
    else:
        high.append(study)

print("Low study hours:", low)
print("Moderate study hours:", moderate)
print("High study hours:", high)
```

```
Low study hours: [2.5, 1.5, 2.0, 2.5]
Moderate study hours: [3.5, 5.0, 4.0, 3.0, 4.5, 3.0, 4.0, 5.0, 3.
High study hours: [5.5, 6.0]
```

Task 2. Based on Data – Answer all the Questions:

1. How many days had **low study time**?

    (a) Hint: Count the number of items in the low study list and print the result.

2. How many days had **moderate study time**?

3. How many days had **high study time**?

```
#2
print("Days with low study time:", len(low))
print("Days with moderate study time:", len(moderate))
print("Days with high study time:", len(high))
```

```
Days with low study time: 4
Days with moderate study time: 9
Days with high study time: 2
```

Task 3. Convert Study Hours to Minutes:

Convert each study hour value into minutes and store it in a new list called study_minutes.

Formula: Minutes = Hours× 60

1. Iterate over the time_data list and apply the formula to the study hours.
2. Store the results in the new list.
3. Print the converted values.

```python
#3
study_minutes = []

for study, entertainment, sleep in time_data:
    study_minutes.append(study * 60)

print("Study minutes:", study_minutes)
```

```
Study minutes: [210.0, 300.0, 150.0, 240.0, 90.0, 180.0, 330.0, 120.0, 270.0, 180.0, 360.0, 150.0, 240.0, 300.0, 210.0]
```

Task 4. Analyze Average Time Use:

Scenario: Each record contains daily hours of study, entertainment, and sleep.

1. Create empty lists for study_hours, entertainment_hours, and sleep_hours.
2. Iterate over time_data and extract values into each list.
3. Calculate and print:
   (a) Average hours spent studying.
   (b) Average hours spent on entertainment.
   (c) Average hours spent sleeping.

```python
#4
study_hours = []
entertainment_hours = []
sleep_hours = []

for study, entertainment, sleep in time_data:
    study_hours.append(study)
    entertainment_hours.append(entertainment)
    sleep_hours.append(sleep)

avg_study = sum(study_hours) / len(study_hours)
avg_entertainment = sum(entertainment_hours) / len(entertainment_hours)
avg_sleep = sum(sleep_hours) / len(sleep_hours)

print("Average Study Hours:", avg_study)
print("Average Entertainment Hours:", avg_entertainment)
print("Average Sleep Hours:", avg_sleep)
```

```
Average Study Hours: 3.7
Average Entertainment Hours: 2.5
Average Sleep Hours: 7.166666666666667
```

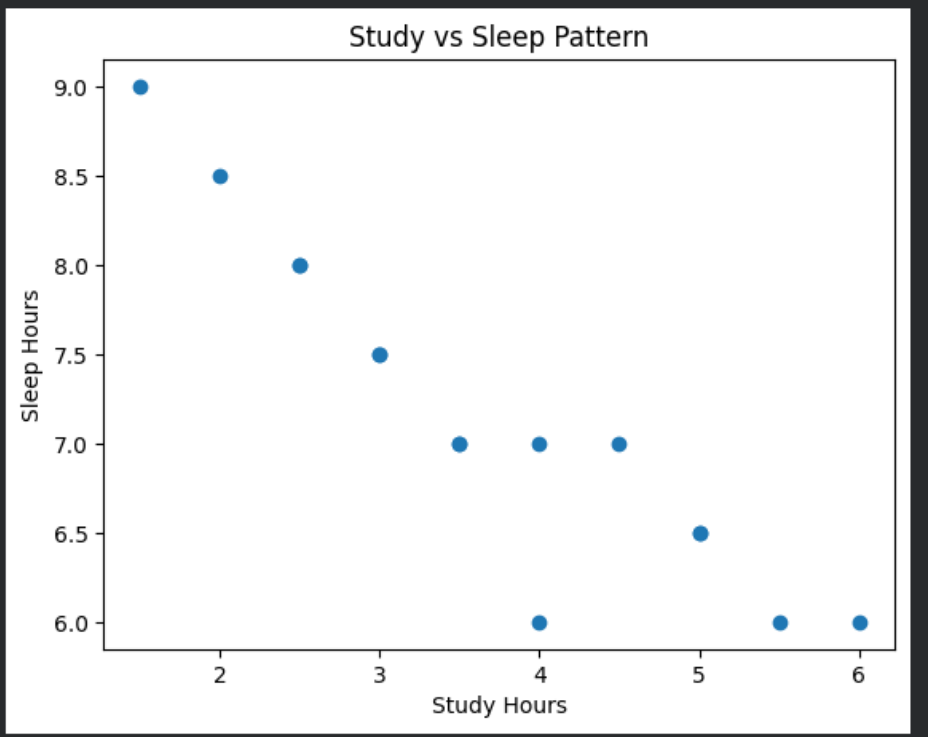
Task 5. Visualization - Study vs Sleep Pattern:

1. Import matplotlib.pyplot as plt.
2. Plot a scatter plot with:
    - x-axis: Study hours
    - y-axis: Sleep hours
    - Add labels, title, and color.

```python
#5
import matplotlib.pyplot as plt

study_hours = [t[0] for t in time_data]
sleep_hours = [t[2] for t in time_data]

plt.scatter(study_hours, sleep_hours)
plt.xlabel("Study Hours")
plt.ylabel("Sleep Hours")
plt.title("Study vs Sleep Pattern")
plt.show()
```



8.1.1 Exercise - Recursion:

Task 1 - Sum of Nested Lists:

Scenario: You have a list that contains numbers and other lists of numbers (nested lists). You want to find the total sum of all the numbers in this structure. Task:

- Write a recursive function sum_nested_list(nested_list) that:
  1. Takes a nested list (a list that can contain numbers or other lists of numbers) as input.
  2. Sums all numbers at every depth level of the list, regardless of how deeply nested the numbers are.
- Test the function with a sample nested list, such as nested_list = [1, [2, [3, 4], 5], 6, [7, 8]].
  The result should be the total sum of all the numbers.

```python
#1
nested_list = [1, [2, [3, 4], 5], 6, [7, 8]]
def sum_nested_list(nested_list):
    total = 0

    for element in nested_list:
        if isinstance(element, list):
            total += sum_nested_list(element)
        else:
            total += element
    return total

print("Sum of nested list =", sum_nested_list(nested_list))
```

```
Sum of nested list = 36
```

Task 2 - Generate All Permutations of a String:

Scenario: Given a string, generate all possible permutations of its characters. This is useful for understanding backtracking and recursive depth-first search. Task:

- Write a recursive function generate_permutations(s) that:
  - Takes a string s as input and returns a list of all unique permutations.
- Test with strings like ”abc” and ”aab”.

print(generate_permutations("abc"))
# Should return [’abc’, ’acb’, ’bac’, ’b ca’, ’cab’, ’cba’]

```python
#2
def generate_permutations(s):
    if len(s) == 1:
        return [s]

    permutation = []
    for i in range(len(s)):
        fixed = s[i]
        remaining = s[:i] + s[i+1:]
        for p in generate_permutations(remaining):
            permutation.append(fixed + p)

    return list(set(permutation))

print(generate_permutations("abc"))
```

```
['bac', 'cab', 'abc', 'cba', 'bca', 'acb']
```

Task 3 - Directory Size Calculation:

Directory Size Calculation Scenario: Imagine a file system where directories can contain files (with sizes in KB) and other directories. You want to calculate the total size of a directory, including all nested files and subdirectories.

Task:

1. Write a recursive function calculate_directory_size(directory) where:

    - directory is a dictionary where keys represent file names (with values as sizes in KB) or directory names (with values as another dictionary representing a subdirectory).
    - The function should return the total size of the directory, including all nested subdirectories.

    3. Test the function with a sample directory structure.

```python
#3
def calculate_directory_size(directory):
    total = 0

    for key, value in directory.items():
        if isinstance(value, dict):
            total += calculate_directory_size(value)
        else:
            total += value
    return total
directory_structure = {
    "file1.txt": 200,
    "file2.txt": 300,
    "subdir1": {
        "file3.txt": 400,
        "file4.txt": 100
    },
    "subdir2": {
        "subsubdir1": {
            "file5.txt": 250
        },
        "file6.txt": 150
    }
}
print("Total directory size:", calculate_directory_size(directory_structure))
```

```
Total directory size: 1400
```

8.2.1 Exercises - Dynamic Programming:

Task 1 - Coin Change Problem:

Scenario: Given a set of coin denominations and a target amount, find the minimum number of coins needed to make the amount. If it's not possible, return - 1. Task:

1. Write a function min_coins(coins, amount) that: • Uses DP to calculate the minimum number of coins needed to make up the amount.
2. Test with coins = [1, 2, 5] and amount = 11. The result should be 3 (using coins [5, 5, 1]).

```python
#1
def min_coins(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0

    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] = min(dp[i], dp[i - coin] + 1)

    return dp[amount] if dp[amount] != float('inf') else -1


print(min_coins([1, 2, 5], 11))
```

```
3
```

Task 2 - Longest Common Subsequence (LCS):

Scenario: Given two strings, find the length of their longest common subsequence (LCS). This is useful in text comparison. Task:

1. Write a function longest_common_subsequence(s1, s2) that:
    - Uses DP to find the length of the LCS of two strings s1 and s2.
2. Test with strings like "abcde" and "ace"; the LCS length should be 3 ("ace").

```python
#2
def longest_common_subsequence(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i-1] == s2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[m][n]

print(longest_common_subsequence("abcde", "ace"))
```

```
3
```

Task 3 - 0/1 Knapsack Problem:

Scenario: You have a list of items, each with a weight and a value. Given a weight capacity, maximize the total value of items you can carry without exceeding the weight capacity. Task:

1. Write a function knapsack(weights, values, capacity) that: • Uses DP to determine the maximum value that can be achieved within the given weight capacity.
2. Test with weights [1, 3, 4, 5], values [1, 4, 5, 7], and capacity 7. The result should be 9.

1. When the problem has a small problem size (so the overhead of recursion is minimal) or when an elegant, simple solution is needed without worrying about performance.
2. Examples: Tree traversal, divide-and-conquer algorithms (like merge sort).

- Use Dynamic Programming:

1. When the problem involves overlapping sub-problems and optimal substructure.
2. When you need to optimize recursive solutions to avoid redundant work.
3. Examples: Knapsack problem, Fibonacci sequence, shortest path algorithms (e.g., Dijkstra's, Bellman-Ford).

```python
#3
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        w = weights[i-1]
        v = values[i-1]

        for cap in range(1, capacity + 1):
            if w <= cap:
                dp[i][cap] = max(dp[i-1][cap], dp[i-1][cap-w] + v)
            else:
                dp[i][cap] = dp[i-1][cap]

    return dp[n][capacity]

print(knapsack([1, 3, 4, 5], [1, 4, 5, 7], 7))
```

```
9
```

———————————— Fasten Your Seat Belt and Enjoy the Ride. ————————————-