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.

```
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)
]

#task1
low_study = [];
moderate_study = [];
high_study = [];
#task2
for study,entertainment,sleep in time_data:
    if study<3:
        low_study.append(study);
    elif(study>3 and study<=5):
        moderate_study.append(study)
    elif(study>5):
        high_study.append(study)
#task3
print("Low study time (<3 hours):", low_study)
print("Moderate study time (3–5 hours):", moderate_study)
print("High study time (>5 hours):", high_study)
```

```
Low study time (<3 hours): [2.5, 1.5, 2.0, 2.5]
Moderate study time (3–5 hours): [3.5, 5.0, 4.0, 4.5, 4.0, 5.0, 3.5]
High study time (>5 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**?

```
#Task 2
print("Number of low study days:", len(low_study))
print("Number of moderate study days:", len(moderate_study))
print("Number of high study days:", len(high_study))
```

```
Number of low study days: 4
Number of moderate study days: 7
Number of high study days: 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.

```
#Task3
#task3.1
study_minutes = [];
for (study,entertainment,sleep) in time_data:
    minutes = study * 60
#task3.2
    study_minutes.append(minutes);

#task3.3
print("Study_minutes is",study_minutes)
```

```
Study_minutes is [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.

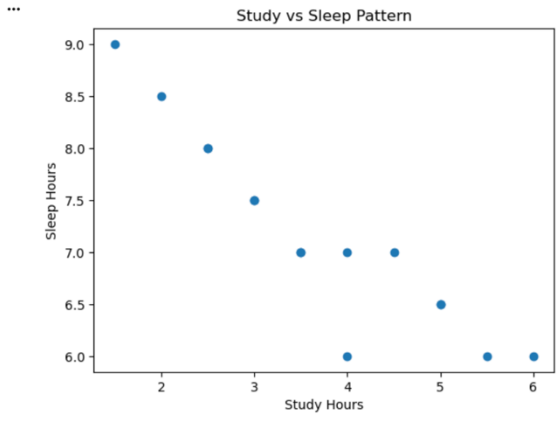
```
#Task4
#task4.1
study_hours = [];
entertainment_hours = [];
sleep_hours = [];
#task4.2
for (study,entertainement,sleep) in time_data:
    study_hours.append(study)
    entertainment_hours.append(entertainment)
    sleep_hours.append(sleep)

#task4.3
avg_study = sum(study_hours) / len(study_hours)
avg_entertainment = sum(entertainment_hours) / len(entertainment_hours)
avg_sleep = sum(sleep_hours) / len(sleep_hours)

#task4.4
print("The average_study hours is", avg_study," hours")
print("The average_entertainement hours is", avg_entertainment," hours")
print("The average_sleep hours is", avg_sleep," hours")
```

```
The average_study hours is 3.7  hours
The average_entertainement hours is 2.5  hours
The average_sleep hours is 7.166666666666667  hours
```

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
3. Add labels, title, and color.

```
import matplotlib.pyplot as plt
studyhours = [ day[0] for day in time_data]
sleephours = [day[2] for day in time_data]
plt.scatter(studyhours,sleephours)
plt.xlabel("Study Hours")
plt.ylabel("Sleep Hours")
plt.title("Study vs Sleep Pattern")
plt.show()
```



8 Problem based on Popular Algorithm.

8.1 Recursion:

1. Definition:

Recursion is a programming technique where a function calls itself to solve a problem. It's typically used when a problem can be broken down into smaller, similar sub-problems. Some key concepts in Recursion:

- Base Case: This is the condition where the recursive calls stop. It prevents infinite loops by returning a value without further recursion.
- Recursive Case: This is the part where the function calls itself with a modified argument, gradually moving towards the base case.
- Stack Memory: Each recursive call adds a layer to the call stack. Too many recursive calls without reaching a base case can lead to a "stack overflow" error due to memory limits.

Example: A classic recursive example is the factorial of a number:

Sample Code - Reccursive Approach for finding factorial of a Number.

```
def factorial(n):
"""
Calculate the factorial of a non-negative integer n.
The factorial of a number n (denoted n!) is the product of all positive integers less than or equal to n.
Specifically:
    - If n is 0, the factorial is defined as 1 (base case).
    - For any positive integer n, the factorial is calculated recursively as n * (n-1) * (n-2) * ... * 1.
Args: n (int): A non-negative integer for which to calculate the factorial.
Returns: int: The factorial of the input number n.
Raises:
    ValueError: If the input n is a negative integer, as factorial is only defined for non-negative integers.
"""
if n == 0: # Base case return 1
else: # Recursive case return n *
    factorial(n - 1)
```

2. General Algorithm for Recursion Problems:

• Identify the Base Case: Determine the simplest form of the problem. This is crucial to avoid infinite recursion.

- Divide the Problem: Think about how to reduce the problem's size. Often, the problem should become simpler with each recursive call.
- Formulate the Recursive Case: Decide how each recursive step will bring you closer to the base case.

- Combine Results: If necessary, think about how to combine the results of recursive calls to form the final result.

3. How to Identify Recursion-Based Problems:

1. Problem Can Be Split into Smaller Versions of Itself: • If the problem can be divided into smaller sub-problems that resemble the original, recursion might be suitable.
   - Example: Summing all numbers in a nested list. Each sub-list can be treated as a smaller instance of the original list.
2. Tasks with Nested or Hierarchical Structures:
   (a) Problems involving nested data structures, like nested lists or tree structures, are often recursive by nature.
   (b) Example: Calculating the size of a directory with subdirectories.
3. Repetitive or Iterative Nature with Reducing Input: • Problems that involve performing an operation repeatedly with a progressively smaller (or simpler) input are ideal candidates.
   - Example: Calculating the power of a number, where the exponent decreases with each call

4. Problems Involving Dividing and Conquering: • When a problem can be divided into parts that can be solved independently, recursion is often useful, as seen in sorting and search algorithms like quicksort and binary search.

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.

```
nested_list = [1, [2, [3, 4], 5], 6, [7, 8]]
def sum_nested_list(nested_list):
    """
    Calculate the sum of all numbers in a nested list.
    This function takes a list that may contain integers and other nested lists.
    It recursively traverses the list and sums all the integers, no matter how deeply nested they are.

    Args:
        nested_list (list): A list that may contain integers or other lists of integers.

    Returns:
        int: The total sum of all integers in the nested list, including those in sublists.

    Example:
        >>> sum_nested_list([1, [2, [3, 4], 5], 6, [7, 8]])
        36
        >>> sum_nested_list([1, [2, 3], [4, [5]]])
        15
    """
    total = 0

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

    return total
print(sum_nested_list(nested_list))
```

```
36
```

Sample Code - Sum of Nested lists.

```
def sum_nested_list(nested_list):
"""
Calculate the sum of all numbers in a nested list.
This function takes a list that may contain integers and other nested lists.
It recursively traverses the list and sums all the integers, no matter how deeply nested they are.
Args: nested_list (list): A list that may contain integers or other lists of integers.
Returns:
    int: The total sum of all integers in the nested list, including those in sublists . Example:
    >>> sum_nested_list([1, [2, [3, 4], 5], 6, [7, 8]])
    36
    >>> sum_nested_list([1, [2, 3], [4, [5]]]) 15
"""
total = 0 for element in
nested_list:
    if isinstance(element, list): # Check if the element is a list total +=
        sum_nested_list(element) # Recursively sum the nested list
    else: total += element # Add the number to the total return
total
```

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’, ’bca’, ’cab’, ’cba’]
```

```python
#task2
def generate_permutations(s):
    """
    Generate all unique permutations of a string s recursively.
    """
    # Base case: if the string has 1 or 0 characters, return it as a list
    if len(s) <= 1:
        return [s]

    permutations = set()  # Use a set to avoid duplicates
    for i, char in enumerate(s):
        # Remaining string after removing the current character
        remaining = s[:i] + s[i+1:]
        # Recursively get permutations of the remaining string
        for perm in generate_permutations(remaining):
            permutations.add(char + perm)

    # Convert set back to list before returning
    return list(permutations)


# Testing the function
print(generate_permutations("abc"))
# Expected output: ['abc', 'acb', 'bac', 'bca', 'cab', 'cba']

print(generate_permutations("aab"))
# Expected output: ['aab', 'aba', 'baa']
```

```
['bca', 'cba', 'acb', 'abc', 'bac', 'cab']
['aba', 'aab', 'baa']
```

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.

Sample directory structure.

```
# Sample directory structure directory_structure = {
    "file1.txt": 200,
    "file2.txt": 300,
    "subdir1": {
        "file3.txt": 400,
        "file4.txt": 100
    },
    "subdir2": {
        "subsubdir1": {
            "file5.txt": 250
        },
        "file6.txt": 150
    }
}
```

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. T

```
#task3
def calculate_directory_size(directory):
    total_size=0
    for element in directory.values():
        if isinstance(element,dict):
            total_size+=calculate_directory_size(element)
        else:
            total_size+=element
    return total_size

# Sample directory structure
directory_structure = {
 "file1.txt": 200,
 "file2.txt": 300,
 "subdir1": {
 "file3.txt": 400,
 "file4.txt": 100
 },
 "subdir2": {
 "subsubdir1": {
 "file5.txt": 250
 },
 "file6.txt": 150
 }
 }

size_of_directory=int(calculate_directory_size(directory_structure))
print("Size of directory is ",size_of_directory)
```

```
Size of directory is  1400
```

    est the function with a sample directory structure.

8.2 Dynamic Programming:

1. Introduction to Dynamic Programming:

Dynamic Programming is a method used for solving problems that can be broken down into smaller sub-problems. It is particularly useful when the problem exhibits overlapping sub-problems and optimal substructure.

• Overlapping Sub problems: This property occurs when the problem can be divided into smaller sub problems, and these sub problems are solved multiple times. • Optimal Substructure: The optimal solution to the problem can be constructed from optimal solutions to its sub-problems.

2. Type of Dynamic Programming:

1. Memoization: A top-down approach where we solve the problem by breaking it into sub-problems, and store the results of sub-problems to avoid redundant computations.
2. Tabulation: A bottom-up approach that starts solving from the smallest subproblem, and progressively solves larger sub-problems while storing results in a table.

3. Steps in Dynamic Programming:

1. Define the subproblem: Break the original problem into smaller sub-problems.
2. Recursive Relation: Identify the relation between sub-problems (i.e., how to compute the solution of the problem based on the solutions to smaller sub-problems).
3. Store the results: Cache or store results of sub-problems to avoid recalculating them (memoization or tabulation).
4. Build the final solution: Combine the solutions of sub-problems to get the solution to the original problem.

4. Example:

Fibonacci Sequence: The Fibonacci sequence is defined as:

$$\begin{cases} F(0) = 0 \\ F(1) = 1 \\ F(n) = F(n-1) + F(n-2) & \text{for } n > 1 \end{cases} \quad (1)$$

Solving Fibonacci Sequnce with Memoization {Top - Down Approach}.

```
def fibonacci(n, memo={}):
    """
    Computes the nth Fibonacci number using memoization to optimize the recursive solution.
```

```
This function uses memoization to store previously computed Fibonacci numbers, reducing redundant calculations
    and improving performance.
Parameters:
n (int): The index in the Fibonacci sequence for which the value is to be computed. Must be a non-negative
integer. memo (dict, optional): A dictionary used to store previously computed Fibonacci values.
                                It is initialized as an empty dictionary by default and is used during the recursive calls
to avoid recalculating results. Returns:
int: The nth Fibonacci number.
Example:
>>> fibonacci(5)
5
>>> fibonacci(10)
55
Time Complexity:
-  The time complexity is O(n) because each Fibonacci number is computed only once.
Space Complexity:
-  The space complexity is O(n) due to the memoization dictionary storing the results.
"""
if n in memo: return
memo[n] if n <= 1:
    return n
memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo) return memo[n]
```

Solving Fibonacci Sequnce with Tbulation {Bottom - Up Approach}.

```
def fibonacci(n): """
    Computes the nth Fibonacci number using dynamic programming to optimize the solution.
    This function uses a bottom-up dynamic programming approach to calculate the nth
        Fibonacci number by iteratively building up an array of Fibonacci numbers up to n, thus
    eliminating
    redundant calculations and optimizing performance.
    Parameters:
    n (int): The index in the Fibonacci sequence for which the value is to be computed.
                Must be a non-negative integer.
    Returns:
    int: The nth Fibonacci number.
    Example:
    >>> fibonacci(5)
    5
    >>> fibonacci(10)
    55
    Time Complexity:
    -  The time complexity is O(n), as the function iterates through the range 2 to n, calculating
       each Fibonacci number once.
    Space Complexity:
    -  The space complexity is O(n), due to the array used to store the Fibonacci numbers
```

```
    up to n.
"""
if n <= 1:
    return n
dp = [0] * (n + 1) dp[1] = 1 for i in
range(2, n + 1):
    dp[i] = dp[i - 1] + dp[i - 2]
return dp[n]
```

5. **Tips for Identifying DP Problems:** • Overlapping Sub-problems: If a problem requires solving the same subproblem multiple times, it is a good candidate for DP.

   - Optimal Substructure: If an optimal solution can be constructed from the optimal solutions to sub-problems, the problem might be solvable using DP.
   - Recursive Structure: If the problem can be expressed recursively, check if storing intermediate results helps in improving efficiency.

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]).

Sample Code - Coin Change Problem.

```
def min_coins(coins, amount):
"""
Finds the minimum number of coins needed to make up a given amount using dynamic programming.
This function solves the coin change problem by determining the fewest number of coins from a given set of
    coin denominations that sum up to a target amount. The solution uses dynamic
    programming(tabulation) to iteratively build up the minimum number of coins required for each amount.
Parameters:
coins (list of int): A list of coin denominations available for making change. Each coin denomination is a
    positive integer.
```

```
amount (int): The target amount for which we need to find the minimum number of coins . It must be a non-
      negative integer.
Returns:
int: The minimum number of coins required to make the given amount.
       If it is not possible to make the amount with the given coins, returns -1. Example:
>>> min_coins([1, 2, 5], 11)
3
>>> min_coins([2], 3)
-1
""" 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
```

```
# Task 1- Coin Change Problem:
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

coins=[1,2,5]
amount=11
number_of_coins_required=min_coins(coins,amount)

print("The minimum number of coins needed to make the amount ",amount," with the coin denominations ",coins, " is ",number_of_coins_required,".")
```

```
The minimum number of coins needed to make the amount  11  with the coin denominations  [1, 2, 5]  is  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").

```
#Task 2- Longest Common Subsequence (LCS):

Enter)
executed in this session
own time
                _common_subsequence(s1, s2):
                (s1),len(s2)
                "]*(n+1) for i in range(m+1)]
            for i in range(m):
                for j in range(n):
                    if s1[i]==s2[j]:
                        dp[i+1][j+1]=dp[i][j]+s1[i]
                    else:
                        if len(dp[i][j+1])>len(dp[i+1][j]):
                            dp[i+1][j+1]=dp[i][j+1]
                        else:
                            dp[i+1][j+1]=dp[i+1][j]
            return dp[m][n]

        s1="abcde"
        s2="ace"
        LCS=longest_common_subsequence(s1,s2)
        print(f"The longest common subsequence in the strings {s1} and {s2} is {LCS}.")
```

```
... The longest common subsequence in the strings abcde and ace is ace.
```

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.

```
#Task 3- 0/1 Knapsack Problem:
def knapsack(weights, values, capacity):
    n=len(weights)
    dp=[[0]*(capacity+1) for i in range(n+1)]
    for i in range(1,n+1):
        for w in range(1,capacity+1):
            if weights[i-1]<=w:
                dp[i][w]=max(dp[i-1][w],dp[i-1][w-weights[i-1]]+values[i-1])
            else:
                dp[i][w]=dp[i-1][w]
    return dp[n][capacity]

weights=[1,3,4,5]
values=[1,4,5,7]
capacity=7

weight_capacity=(knapsack(weights,values,capacity))

print("The maximum value of items that can be carried is ",weight_capacity)
```

```
... The maximum value of items that can be carried is  9
```

9 Closing Remarks on Recursive and Dynamic Algorithm.

Dynamic Programming (DP) and recursion are both techniques used to solve problems by breaking them down into smaller sub-problems. However, they differ significantly in how they approach problem-solving and how efficiently they execute. Here's a comparison of both approaches:

1. Problem Solving Approach:

- Recursion:

    1. Recursion is generally easier to implement when solving problems with a clear recursive structure, like tree traversal or calculating factorials.
    2. It is suitable for problems where each subproblem is independent and doesn't need to store its results for reuse.

- Dynamic Programming:

    1. DP is used for problems that have optimal substructure and overlapping subproblems. It's more complex to implement than simple recursion because it requires designing a table or memoization structure.
    2. DP is more suitable for optimization problems, like shortest path problems, knapsack problems, etc., where storing intermediate results is crucial for efficient computation.

2. When to Use Which?

- Use Recursion:

    1. When the problem naturally fits a recursive structure and does not involve overlapping sub-problems.
    2. 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.
    3. 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).