Ashutosh dulal

Workshop 0

7.1 Warming Up Exercise: In this exercise, you'll work with temperature data from Tribhuwan International Airport, Kathmandu. The data spans one month and represents typical early winter temperatures. • Datasets: The temperatures list contains daily temperature readings in Celsius for one month in Kathmandu. Each day includes three readings representing night (00-08), evening (08-16), and day (16-24) temperatures. Sample Code - List of temperature measured at Tribhuwan International Airport. temperatures = [8.2, 17.4, 14.1, 7.9, 18.0, 13.5, 9.0, 17.8, 13.0, 8.5, 16.5, 12.9, 7.7, 17.2, 13.3, 8.4, 16.7, 14.0, 9.5, 18.3, 13.4, 8.1, 17.9, 14.2, 7.6, 17.0, 12.8, 8.0, 16.8, 13.7, 7.8, 17.5, 13.6, 8.7, 17.1, 13.8, 9.2, 18.1, 13.9, 8.3, 16.4, 12.7, 8.9, 18.2, 13.1, 7.8, 16.6, 12.5] Complete all the task below: Task 1. Classify Temperatures: 1. Create empty lists for temperature classifications: (a) Cold: temperatures below 10°C. (b) Mild: temperatures between 10°C and 15°C. (c) Comfortable: temperatures between 15°C and 20°C. 2. Iterate over the temperatures list and add each temperature to the appropriate category. 3. Print the lists to verify the classifications.

Ans:

```
#Temprature
s
temperatures =
[
```

1

```
    8.2, 17.4, 14.1, 7.9, 18.0, 13.5, 9.0, 17.8, 13.0, 8.5,
    16.5, 12.9, 7.7, 17.2, 13.3, 8.4, 16.7, 14.0, 9.5, 18.3, 13.4, 8.1,
    17.9, 14.2, 7.6, 17.0, 12.8, 8.0, 16.8, 13.7, 7.8, 17.5, 13.6, 8.7,
    17.1, 13.8, 9.2, 18.1, 13.9, 8.3, 16.4, 12.7, 8.9, 18.2, 13.1, 7.8, 16.6, 12.5
```

```
#empty list to push different type of
temprature
cold = []
mild = []
comfortable =
[]
```

```
#Condition to set different
temperature:

for temp in
temperatures:
    if temp < 10:
    cold.append(tem
    p) elif 10 <= temp <
    15:
        mild.append(te
        mp)
    elif 15 <= temp < 20:
        comfortable.append(t
        emp)

#print
output
print("Cold Temperatures (<10°C):", cold) print("Mild
Temperatures (10°C to 15°C):", mild) print("Comfortable
Temperatures (15°C to 20°C):", comfortable)
```

Task 2:

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

1. How many times was it mild?

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

2. How many times was it comfortable?

3. How many times was it cold?

```
print("The length of Mild temprature is",
len(mild))
print("The length of Comfortable temprature is",
len(comfortable))
print("The length of Cold temprature is",
len(cold))
```

Cold Temperatures (<10°C): [8.2, 7.9, 9.0, 8.5, 7.7, 8.4, 9.5, 8.1, 7.6, 8.0, 7. 8, 8.7, 9.2, 8.3, 8.9, 7.8]

Mild Temperatures (10°C to 15°C): [14.1, 13.5, 13.0, 12.9, 13.3, 14.0, 13.4, 14. 2, 12.8, 13.7, 13.6, 13.8, 13.9, 12.7, 13.1, 12.5]

Comfortable Temperatures (15°C to 20°C): [17.4, 18.0, 17.8, 16.5, 17.2, 16.7, 18 .3, 17.9, 17.0, 16.8, 17.5, 17.1, 18.1, 16.4, 18.2, 16.6]

The length of Mild temprature is 16

The length of Comfortable temprature is 16

The length of Cold temprature is 16

Google Colab:

QN 1

[3] # Given temperature data

```
temperatures = [
    8.2, 17.4, 14.1, 7.9, 18.0, 13.5, 9.0, 17.8, 13.0, 8.5,
16.5, 12.9, 7.7, 17.2, 13.3, 8.4, 16.7, 14.0, 9.5, 18.3, 13.4, 8.1, 17.9, 14.2, 7.6, 17.0, 12.8, 8.0, 16.8, 13.7, 7.8, 17.5, 13.6,
    8.7, 17.1, 13.8, 9.2, 18.1, 13.9, 8.3, 16.4, 12.7, 8.9, 18.2, 13.1, 7.8, 16.6, 12.5
]

# Create empty lists for classification
cold = []
mild = []
comfortable = []

# Classify each temperature
for temp in temperatures:
    if temp 10:
        cold.append(temp)
    elif 10 <= temp < 15:
        mild.append(temp)
    elif 15 <= temp < 20:
        comfortable.append(temp)

# Print the classified lists
print("Cold Temperatures (<10°C):", cold)
print("Mild Temperatures (10°C to 15°C):", mild)
print("Comfortable Temperatures (15°C to 20°C):", comfortable)

#Task 2

print("The length of Mild temprature is", len (mild))
print("The length of Comfortable temprature is", len (comfortable))
print("The length of cold temprature is", len (cold))
```

Cold Temperatures (<10°C): [8.2, 7.9, 9.0, 8.5, 7.7, 8.4, 9.5, 8.1, 7.6, 8.0, 7.8, 8.7, 9.2, 8.3, 8.9, 7.8]

```
#Task 2

print("The length of Mild temprature is", len (mild))
print("The length of Comfortable temprature is", len(comfortable))
print("The length of cold temprature is", len(cold))
```

Cold Temperatures (<10°C): [8.2, 7.9, 9.0, 8.5, 7.7, 8.4, 9.5, 8.1, 7.6, 8.0, 7.8, 8.7, 9.2, 8.3, 8.9, 7.8]
Mild Temperatures (10°C to 15°C): [14.1, 13.5, 13.0, 12.9, 13.3, 14.8, 13.4, 14.2, 12.8, 13.7, 13.6, 13.8, 13.9, 12.7, 13.1, 12.5]
Comfortable Temperatures (15°c to 20°C): [17.4, 18.0, 17.8, 16.5, 17.2, 16.7, 18.3, 17.9, 17.0, 16.8, 17.5, 17.1, 18.1, 16.4, 18.2, 16.6] The length of Mild temprature is 16

```
The length of Comfortable temprature is 16
The length of cold temprature is 16
```

Task 3. Convert Temperatures from Celsius to Fahrenheit Using the formula for temperature conversion, convert each reading from Celsius to Fahrenheit and store it in a new list called temperatures_fahrenheit. Formula: Fahrenheit = (Celsius × 95) + 32 1. Iterate over the temperatures list and apply the formula to convert each temperature. 2. Store the results in the new list. 3. Print the converted Fahrenheit values.

```
# temperature data in
Celsius
temperatures =
[
```

1

```
    8.2, 17.4, 14.1, 7.9, 18.0, 13.5, 9.0, 17.8, 13.0, 8.5,
    16.5, 12.9, 7.7, 17.2, 13.3, 8.4, 16.7, 14.0, 9.5, 18.3, 13.4, 8.1, 17.9, 14.2, 7.6, 17.0,
    12.8, 8.0, 16.8, 13.7, 7.8, 17.5, 13.6, 8.7,

    17.1, 13.8, 9.2, 18.1, 13.9, 8.3, 16.4, 12.7, 8.9, 18.2, 13.1, 7.8, 16.6, 12.5
```

```
# empty list to store Fahrenheit
values temperatures_fahrenheit =
[]

# Celsius temperature to
Fahrenheit for temp in
temperatures:
    fahrenheit = (temp * 9/5) +
    32
    temperatures_fahrenheit.append(fahren
    heit)

# converted Fahrenheit values
print("Temperatures in Fahrenheit:",
temperatures_fahrenheit)
```

Temperatures in Fahrenheit: [46.76, 63.32, 57.379999999999995, 46.22, 64.4, 56.3 48.2, 64.04, 55.4, 47.3, 61.7, 55.22, 45.86, 62.959999999999994, 55.94, 47.120 000000000005, 62.059999999999995, 57.2, 49.1, 64.94, 56.120000000000005, 46.58, 64.22, 57.56, 45.68, 62.6, 55.04, 46.4, 62.24, 56.66, 46.04, 63.5, 56.48, 47.66, 62.78, 56.84, 48.56, 64.58, 57.02, 46.94, 61.51999999 999996, 54.86, 48.02, 64. 75999999999999, 55.58, 46.04, 61.88, 54.5]

Google Colab:

```
]
#Task 3
#temperature data in Celsius
temperatures = [
    8.2, 17.4, 14.1, 7.9, 18.0, 13.5, 9.0, 17.8, 13.0, 8.5,
    16.5, 12.9, 7.7, 17.2, 13.3, 8.4, 16.7, 14.0, 9.5, 18.3, 13.4, 8.1, 17.9, 14.2, 7.6, 17.0, 12.8,
    8.0, 16.8, 13.7, 7.8, 17.5, 13.6, 8.7, 17.1, 13.8, 9.2, 18.1, 13.9, 8.3, 16.4, 12.7, 8.9, 18.2,
    13.1, 7.8, 16.6, 12.5

# empty list to store Fahrenheit values
temperatures_fahrenheit = []

# Celsius temperature to Fahrenheit
for temp in temperatures:
    fahrenheit (temp* 9/5) + 32
    temperatures_fahrenheit.append(fahrenhei
    t)

#converted Fahrenheit values
print('Temperatures in Fahrenheit: ', temperatures_fahrenheit)

Temperatures in Fahrenheit: [46.76, 63.32, 57.379999999999995, 46.22, 64.4, 56.3, 48.2, 64.04, 55.4, 47.3, 61.7, 55.22, 45.86, 62.959999999999994, 55.94, 47.120000000000005, 62.059999999999995, 57.2, 49.1, 64.94.
```

Task 4. Analyze Temperature Patterns by Time of Day: Scenario: Each day's readings are

grouped as: Night (00-08), Evening (08-16), Day (16-24). 1. Create empty lists for night, day, and evening temperatures. 2. Iterate over the temperatures list, assigning values to each time-of-day list based on their position. 3. Calculate and print the average day-time temperature. 4. (Optional) Plot "day vs. temperature" using matplotlib.

temperature data for one month

temperatures = [

]

8.2, 17.4, 14.1, 7.9, 18.0, 13.5, 9.0, 17.8, 13.0, 8.5,

16.5, 12.9, 7.7, 17.2, 13.3, 8.4, 16.7, 14.0, 9.5, 18.3, 13.4, 8.1, 17.9, 14.2, 7.6, 17.0,

```
    12.8, 8.0, 16.8, 13.7, 7.8, 17.5, 13.6, 8.7, 17.1, 13.8, 9.2, 18.1, 13.9, 8.3, 16.4, 12.7,
    8.9, 18.2, 13.1, 7.8, 16.6, 12.5


# empty lists for each time
of day
night_temps =
[]
evening_temps =
[]
day_temps =
[]

# temperatures by time
of day
for i in range(0,
len(temperatures), 3):
night_temps.append(temperatures[i])
    evening_temps.append(temperature
    s [i+1])
    day_temps.append(temperatures
    [i+2])

# average temperatures for each time
of day
average_night = sum (night_temps) / len(night_temps)
average_evening = sum (evening_temps) /
len(evening_temps) average_day = sum (day_temps) /
len(day_temps)

# the average
temperatures
print(f"Average Night Temperature:
{average_night:.2f}°C")
print(f"Average Evening Temperature:
{average_evening:.2f}°C") print(f"Average Day
Temperature: {average_day:.2f}°C")

# plotting day-time
temperatures
plt.plot(day_temps, marker='o', label='Day
```

Temperatures') plt.title('Day-Time Temperatures Over One Month')

plt.xlabel('Days'
)

plt.ylabel('Temperature (°C)')

plt.legen d()

plt.grid(Tr ue)

plt.show ()

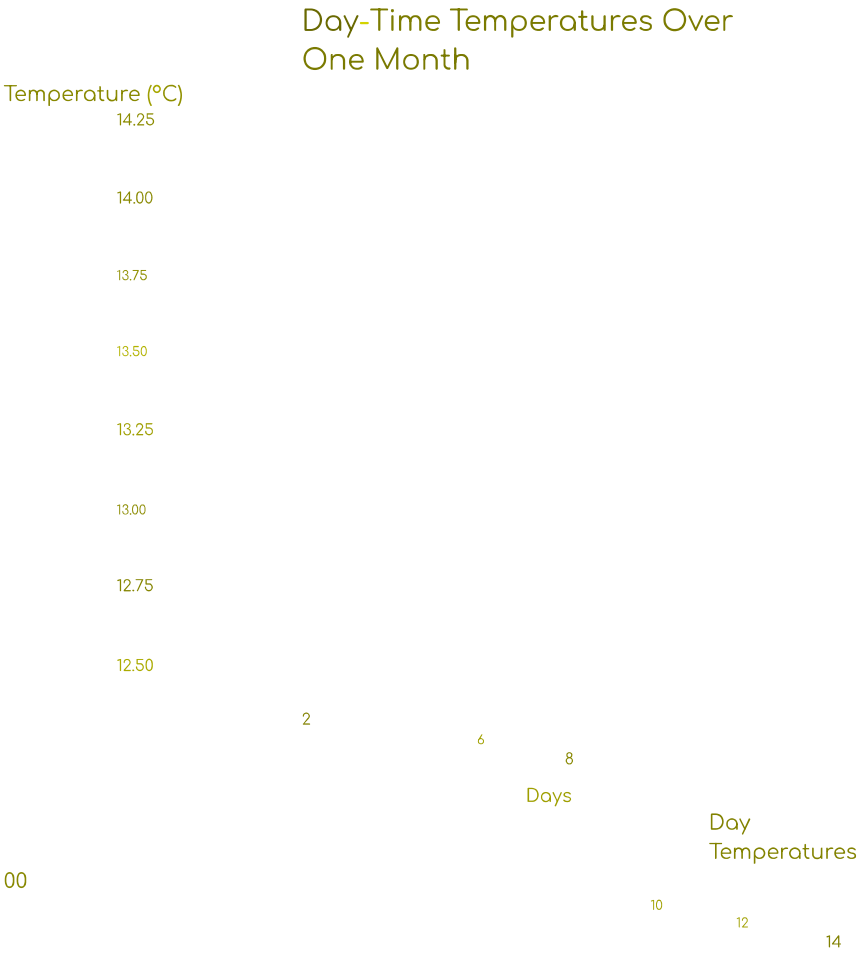
Google Colab:

```
#Task 4
import matplotlib.pyplot as plt

# temperature data for one
month (
temperatures = [
    8.2, 17.4, 14.1, 7.9, 18.0, 13.5, 9.0, 17.8, 13.0, 8.5,
    16.5, 12.9, 7.7, 17.2, 13.3, 8.4, 16.7, 14.0, 9.5, 18.3, 13.4, 8.1, 17.9, 14.2, 7.6, 17.0, 12.8, 8.0, 16.8, 13.7,
    7.8, 17.5, 13.6, 8.7,
    17.1, 13.8, 9.2, 18.1, 13.9, 8.3, 16.4, 12.7, 8.9, 18.2, 13.1, 7.8, 16.6, 12.5
]

# empty lists for each time of day
night_temps =
[]
evening_temps = []
day_temps = []

# temperatures by time of day
for i in range(0, len(temperatures), 3):
    night_temps.append(temperatures[1])
    evening temps.append(temperatures [i+1])
    day_temps.append(temperatures
    [i+2])

# average temperatures for each time
of day
average night=sum(night_temps) / len(night_temps)
average evening = sum(evening temps) / len
(evening_temps) average_day = sum(day_temps) / len
(day_temps)

# the average temperatures
print (f'Average Night Temperature: {average_night:.2f}°C')
print (f'Average Evening Temperature: average evening:.2f}°C')
print (f'Average Day Temperature:
average_day:.2f}°c")
```

```
#plotting day-time
temperatures
plt.plot(day_temps, marker='0', label=' Day Temperatures')
plt.title('Day-Time Temperatures Over One Month')
plt.xlabel('Days')
plt.ylabel('Temperature
```

```
(°C)')
plt.legend()
plt.grid(True)
plt.show()
```

```
Average Night Temperature: 8.35°C
    Average Evening Temperature:
    17.34°C Average Day Temperature:
    13.41°C
```



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.

```
def
```

```
sum_nested_list(nested_list):
    total = 0
    for element in nested_list:
        if is instance(element,
        list):
            # Recursive call for nested lists
            total +=
            sum_nested_list(element)
        else:
            total += element
    return total

# Test cases
nested_list1 = [1, [2, [3, 4], 5], 6, [7,
8]]
nested_list2 = [1, [2, 3], [4, [5]]]
nested_list3 = [10, [20, [30, 40], [50]], 60]

# the results
print("Sum of nested_list1:", sum_nested_list(nested_list1))
print("Sum of nested_list2:",
sum_nested_list(nested_list2)) print("Sum of
nested_list3:", sum_nested_list(nested_list3))
```

```
Sum of nested list1: 36
Sum of nested list2: 15
Sum of nested list3: 210
```

Google colab:

```
[6] #Task 5
    def sum nested list (nested_list):
        total = 0
        for element in nested list:
            if isinstance(element, list):
                # Recursive call for nested lists
                total +=
                sum_nested_list(element)
            else:
                total element
        return total

    # Test cases
    nested list1 = [1, [2, [3, 4], 5], 6, [7, 8]]
    nested_list2 = [1, [2, 3], [4, [5]]]
```

```
nested_list3

# the results
                [10, 20, 30, 40], [50]], 60]


print("Sum of nested list1: ", sum_nested_list(nested_list1))
print("Sum of nested list 2: ", sum_nested_list(nested_list2))
print("Sum of nested list3:", sum_nested_list(nested_list3))

Sum of nested list1: 36 Sum
of nested list2: 15 Sum of
nested list3: 210
```

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']

```
def
generate_permutations(s)
:
    # Base case: a single character or empty
    string
    if len(s) <= 1:
        return
        [s]

    # set to store unique
    permutations
    permutations =
    set()

    for i, char in
    enumerate(s):

        remaining = s[i] + s[i+1:]
        for perm in generate_permutations
        (remaining):
            permutations.add(char +
```

```
        perm)
    # Conversion set to list before
    returning
    return
    list(permutations)

#Test cases
print("Permutations of 'abc':",
generate_permutations("abc")) print("Permutations of
'aab':", generate_permutations("aab"))
```

```
Permutations of 'abc': ['acb',
                        abc': ['acb', 'cba', 'bca', 'bac', Permutations
of 'aab': ['baa', 'aba', aab']
```

Google
Colab:

```
[7] # Task 6
    def generate permutations(s):



        # Base case: a single character or empty string
        if len(s) <= 1:
            return [s]

        # set to store unique permutations
        permutations = set()

        for 1, char in enumerate(s):

            remaining s[i] + s[i+1]

            for perm in generate per mutations
            (remaining):
                permutations.add(char +
                perm)

        # Conversion set to list before returning
        return list (permutations)

    # Test cases
    print("Permutations
    of
    print ("Permutations of abc", generate
    permutations ("abc"))
```

aab'", generate_permutations ("aab"))

Permutations of 'abc': ['abc', 'bac', 'cab', 'acb', 'bca', 'cba'] Permutations of 'aab': ['baa', 'aba', 'aab']

'cab'

abc']

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.

```
def
calculate_directory_size(director
y):
    total_size = 0
```

```
    for item, value in
    directory.items():
```

```
        if is instance(value,
        dict):

            total_size +=
            calculate_directory_size(value)
        else:
            # add the file size to the total
            total_size += value
    #the final total size
    return total_size

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

# print the total
directory size
total_size =
calculate_directory_size(directory_structure)
print(f"Total directory size: {total_size}
KB")
```

```
Total directory size: 1400 KB
```

Google Colab:

```
#Task 7
def calculate_directory_size(directory):
    total_size = 0

    for item, value in directory.items():
        if isinstance(value, dict):

            total_size = calculate_directory_size(value)
        else:
            # add the file size to the total
            total_size += value
      #the final total size
    return total_size

# directory structure
directory structure = {
    "file1.txt": 200,
    "file2.txt": 300,
    "subdir1": {

        "file3.txt": 400,

    },
        "file4.txt": 100

    "subdir2": {
        "subsubdir1": {
            "files.txt": 250
        },
        "file6.txt": 150
    }
}

# print the total directory size
total_size = calculate_directory_size(directory_structure) print
(f"Total directory size: {total_size} KB")
```

2. Test the function with a sample directory structure.

```
# Sample test
cases
test_directories =
[
  {
```

```
    "name": "Simple
    Directory",
    "structure": {
        "file1.txt": 100,
        "file2.txt": 200,
        "file3.txt": 300
    },
    "expected":
    600
},
{
    "name": "Nested
    Directory",
    "structure": {
        "file1.txt": 150,
        "folder1": {
            "file2.txt": 200,
            "folder2": {
                "file3.txt": 250,
                "file4.txt": 100
            }
        }
    },
    "expected": 700
},
{
    "name": "Deeply Nested
    Directory",
    "structure":
    {
        "folder1": {
            "folder2": {
                "folder3": {
                    "file1.txt": 50
                }
            }
        }
```

```
        },
        "expected":
        50
    },
    {
        "name": "Mixed
        Directory",
        "structure":
        {
            "file1.txt": 120,
            "file2.txt": 80,
            "folder1": {
                "file3.txt": 100,
                "file4.txt": 300
            },
            "folder2": {
                "file5.txt": 50
            }
        },
        "expected":
        650
    }
```

```
for test in test_directories:
    result =
    calculate_directory_size(test["structure"])
print(f"{test['name']} Size: {result} KB (Expected: {test['expected']}
KB)") assert result == test["expected"], f"Test failed for {test['name']}"
print("All tests passed!")
```

```
Simple Directory Size: 600 KB Nested
Directory Size: 700 KB
                                                    (Expected: 600 KB)
                                                    (Expected: 700 KB)
Deeply Nested Directory Size: 50 KB (Expected: 50 KB) Mixed Directory
Size: 650 KB (Expected: 650 KB)
All tests passed!
```

Google Colab:

```
#Sample test cases
test_directories
[9]
    {
                [
    "name": "Simple Directory",
    "structure": {
        "file1.txt": 100,
        "file2.txt": 200,
        "file3.txt": 300
    },
    "expected":
    600

    {
    "name": "Nested Directory",









    {
    "structure": {
        "file1.txt": 150,
        "folder1": {

            "folder2": {
            "file2.txt": 200,

                "file3.txt": 250,
                "file4.txt": 100
            }
        }
    },
    "expected": 700.


    "name": "Deeply Nested Directory",
    "structure": {
        "folder1": {
            "folder2": {
```

```
                    "folder3": {

                    }
                }
            }
                    "file1.txt": 50
[9]
        },
        "expected": 50

        "name": "Mixed Directory",
        "structure":{
            "file1.txt": 120,
            "file2.txt": 80,
            "folder1": {
                "files.txt": 100,
                "file4.txt": 300
            },
            "folder2": {
                "files.txt": 50
            }
        },
        "expected": 650
    }
]

for test in test_directories:
    result = calculate_directory_size(test["structure"])
    print (f" test['name']} Size: [result} KB (Expected: [test['expected' ]} KB) ") assert result
    == test ["expected"], f"Test failed for {test['name']}"
print("All tests passed!")
```

```
Simple Directory Size: 600 KB (Expected: 600 KB) Nested Directory
    Size: 700 KB (Expected: 700 KB) Deeply Nested Directory
    Size: 50 KB (Expected: 50 KB) Mixed Directory Size: 650 KB
    (Expected: 650 KB) All tests passed!
```

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

```
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


coins1 = [1, 2, 5]
amount1 = 11
print(f"Minimum coins to make {amount1}:", min_coins (coins1,
amount1))

coins2 = [2]
amount2 = 3
print(f"Minimum coins to make {amount2}:", min_coins (coins2,
amount2))

coins3 = [1, 3, 4]
amount3 = 6
print(f"Minimum coins to make {amount3}:", min_coins (coins3,
amount3))
```

2. Test with coins = [1, 2, 5] and amount = 11. The result should be 3 (using coins [5, 5,

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

coins =
        = [1, 2, 5]
amount = 11
result = min_coins (coins,
amount)
print(f"Minimum coins to make {amount}:
{result}")
```

```
Minimum coins to make 11: 3
Minimum coins to make 3: -1
Minimum coins to make 6: 2 Minimum
coins to make 11: 3
```

Google Colab:

```
#Task 9
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
```

```
coins1= [1, 2, 5]
amount1 = 11
print (f"Minimum coins to make {amount1}:", min_coins (coins1,
amount1))

coins2 = [2]
amount2 = 3
print (f"Minimum coins to make {amount2}:", min_coins (coins2, amount2))

coins3 = [1, 3, 4]
amount3 = 6
print (f"Minimum coins to make {amount3}:", min_coins (coins3,
amount3))

#Task 10
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.
```

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.

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

    # Fill the dp
    array
    for i in range(1, len(s1) +
    1):
        for j in range(1, len(s2) +
        1):
            # Characters match
            if s1[i-1]==
```

```
                == s2[j-1]:
            dp[i][j] = dp[i - 1][j - 1]
            + 1
                                1][j-1]+
        else: # Characters don't match
            dp[i][j] = max(dp[i - 1][j],
            dp[i][j-1])
    # length of the
    LCS
    return
    dp[len(s1)][len(s2)]

s1="AGGTAB"
s2= "GXTXAYB"
print(f"Length of LCS between '{s1}' and '{s2}':", longest_common_subsequence(s1, s2))
```

```
Length of LCS between AGGTAB' and 'GXTXAYB': 4
```

Google Colab:

```
[10] #Task 11
    def longest_common_subsequence(s1, s2):
        dp = [[0] * (len (s2) + 1) for in range(len(s1) + 1)]

        # Fill the dp array
        for i in range(1, len(s1) + 1):
            for i in range(1, len (s2) + 1):
                #Characters match
                if s1[i - 1] == s2[j-1]:
                    dp[i][j] = dp[i - 1][j-1] + 1
                else: # Characters don't match
                    dp[i][j] = max(dp[i - 1][j], dp[i][j-1])

        # length of the LCS
        return dp[len (s1)][len (s2)]

    s1="AGGTAB"
    $2 = "GXTXAYB"
    print (f"Length of LCS between '{s1}' and '{s2}:", longest_common_subsequence(s1, s2))

    Length of LCS between 'AGGTAB' and 'GXTXAYB': 4
```

2. Test with strings like "abcde" and "ace"; the LCS length should be 3 ("ace").

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

    for i in range(1, len(s1) +
    1):
        for j in range(1, len(s2) +
        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 the length of the
    LCS return
    dp[len(s1)][len(s2)]

#with strings "abcde" and
"ace" s1 = "abcde"
s2 = "ace"
result =
longest_common_subsequence(s1, s2)
print(f"Length of LCS between '{s1}' and '{s2}':
{result}")
```

```
Length of LCS between abcde' and 1 ace': 3
```

Google
Colab:

```
[11] #Task 12
        def longest_common_subsequence(s1, s2):
            dp = [[0] * (len(s2) + 1) for in range(len(s1) + 1)]

            for i in range(1, len(s1) + 1):
                for i in range(1, len(s2) + 1):
                    if s1[i - 1]

                    else:
                                    == s2[j-1]:
```

```
                dp[i][j] = dp[i - 1][j-1] + 1

                dp[i][j] = max(dp[i - 1][j], dp[i][j-1])

    # Return the length of the LCS
    return dp[len (s1)][len (s2)]

#with strings "abcde" and
"ace"
s1 = "abcde"
s2 = "ace"
result longest_common_subsequence(s1,
52)
print (f"Length of LCS between '{s1}' and '{s2} {result}")

Length of LCS between 'abcde' and '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

```
def knapsack(weights, values,
capacity):
    n =
    len(weights)
    dp = [[0] * (capacity + 1) for _ 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], values[i - 1] + dp[i - 1][w - weights[i - 1]]) else:
                dp[i][w] = dp[i - 1][w]

    return
    dp[n][capacity]
```

```
weights = [2, 3, 4, 5]
values = [3, 4, 5, 6]
capacity =
5
result = knapsack(weights, values, capacity)
print(f"Maximum value that can be achieved:
{result}")
```

```
Maximum value that can be achieved: 7
```

Google
Colab:

```
[12] #Task 13
    def knapsack (weights, values,
    capacity):
        n = len(weights)
        dp = [[0]* (capacity + 1) for 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], values[i - 1] + dp[i - 1] [w weights[i - 1]]) else:

                    dp[i] [w] dp[i - 1] [w]

        return
        dp[n][capacity]

    weights [2, 3, 4, 5]
    values = [3, 4, 5, 6]
    capacity = 5.
    result knapsack (weights, values, capacity)
    print (f"Maximum value that can be achieved:
    {result}")

    Maximum value that can be achieved: 7
```

2. Test with weights [1, 3, 4, 5], values [1, 4, 5, 7], and capacity 7. The result should be 9.

```
def knapsack(weights, values,
capacity):
    n =
```

```
    len(weights)
    dp = [[0] * (capacity + 1) for _ 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], values[i - 1] + dp[i - 1][w - weights[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
result = knapsack(weights, values,
capacity)
print(f"Maximum value that can be achieved:
{result}")
```

```
Maximum value that can be achieved: 9
```

```
[13] #Task 14
        def knapsack (weights, values, capacity):
            n = len(weights)
            dp = [[0]* (capacity + 1) for 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], values[i - 1] + dp[i - 1] [w weights [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
result knapsack (weights, values, capacity)
print (f"Maximum value that can be achieved:
{result}")
```

```
Maximum value that can be achieved: 9
```