PYTHON PROGRAMMING LAB 2024-25

Python Numbers:

a) You are developing a program to determine whether a given year is a leap year, using the following formula: a leap year is one that is divisible by four, but not by one hundred, unless it is also divisible by four hundred. For example, 1992, 1996, and 2000 are leap years, but 1967 and 1900 are not. The next leap year falling on a century is 2400.

```
# Input year from user
year = int(input("Enter a year: "))
# Check if it's a leap year
if year % 400 == 0:
    print(f"{year} is a leap year.")
elif year % 100 == 0:
    print(f"{year} is not a leap year.")
elif year % 4 == 0:
    print(f"{year} is a leap year.")
else:
    print(f"{year} is not a leap year.")
```

Output:

```
Enter a year: 1920
1920 is a leap year.
Enter a year: 1900
1900 is not a leap year.
```

b) You are developing a program to determine the greatest common divisor and least common multiple of a pair of integers.

```
# Input two integers
a = int(input("Enter the first number: "))
b = int(input("Enter the second number: "))
# Determine the GCD using the Euclidean algorithm
```

```
x, y = a, b
while y != 0:
    x, y = y, x % y
gcd = x
# Determine the LCM using the formula: LCM(a, b) = abs(a * b) / GCD(a, b)
lcm = abs(a * b) // gcd
# Output the results
print(f"The Greatest Common Divisor (GCD) of {a} and {b} is: {gcd}")
print(f"The Least Common Multiple (LCM) of {a} and {b} is: {lcm}")
```

Output:

```
Enter the first number: 4
Enter the second number: 6
The Greatest Common Divisor (GCD) of 4 and 6 is: 2
The Least Common Multiple (LCM) of 4 and 6 is: 12
```

c) You are developing a program to create a calculator application. Write code that will take two numbers and an operator in the format: N1 OP N2. where N1 and N2 are floating point or integer values, and OP is one of the following: +,- , *, /, %, **, representing addition, subtraction, multiplication, division, modulus/remainder, and exponentiation, respectively, and displays the result of carrying out that operation on the input operands.

```
# Take input in the format: N1 OP N2
input_string = input("Enter an expression (N1 OP N2): ")

# Split the input string into operands and operator
operands = input_string.split()
N1 = float(operands[0])  # Convert the first operand to float
operator = operands[1]   # The operator as a string
N2 = float(operands[2])  # Convert the second operand to float

# Perform the operation based on the operator
if operator == '+':
```

```
    result = N1 + N2
elif operator == '-':
    result = N1 - N2
elif operator == '*':
    result = N1 * N2
elif operator == '/':
    if N2 != 0:
        result = N1 / N2
    else:
        result = "Error: Division by zero"
elif operator == '%':
    result = N1 % N2
elif operator == '**':
    result = N1 ** N2
else:
    result = "Error: Invalid operator"

# Display the result
print(f"Result: {result}")
```

Output:

```
Enter an expression (N1 OP N2): 2 + 6
Result: 8.0
Enter an expression (N1 OP N2): 2 - 6
Result: -4.0
Enter an expression (N1 OP N2): 2 * 6
Result: 12.0
Enter an expression (N1 OP N2): 2 / 6
Result: 0.3333333333333333
Enter an expression (N1 OP N2): 2 % 6
Result: 2.0
Enter an expression (N1 OP N2): 2 ** 6
Result: 64.0
```

Week 1: Viva Questions

Topic : Python Numbers

1. What are the different types of numbers in Python, and how do they differ?
2. How does Python handle type conversion between different numeric types?
3. What is the difference between the // and / operators in Python?
4. How can you perform exponentiation in Python?
5. Explain the use of the round() function in Python.
6. What are the built-in methods available to check the properties of numbers in Python?
7. What is the difference between int and long types in Python 2, and how has this changed in Python 3?
8. Can you perform mathematical operations with complex numbers in Python?
9. What is the math module in Python, and what are some commonly used functions from it?
10. What is the significance of float('inf') and float('-inf') in Python?

WEEK 2: Control Flow

a) Write a Program for checking whether the given number is a prime number or not.

```
num=int(input("enter a value:"))
flag=False
if num>1:
 for i in range(2,num):
  if(num%i)==0:
   flag=True
   break
if flag:
 print(num,"is not a prime number")
else:
 print(num,"is a prime number")
```

Output:

```
enter a value:5
5 is a prime number
enter a value:12
12 is not a prime number
```

b) Write a program to print Fibonacci series up to given n value

```
n=int(input("enter a value:"))
f1=0
f2=1
print(f1,f2,end=' ')
f3=f1+f2
while(f3<=n):
 print(f3,end=' ')
 f1=f2
 f2=f3
```

```
 f3=f1+f3
```

Output:

```
enter a value:5
0 1 1 2 3 5
enter a value:9
0 1 1 2 3 5 8
```

3. write a program to calculate factorial of given integer number,

```
n=int(input("enter a value :"))
fact=1
for i in range (1,n+1):
 fact=fact*i
print("the factorial of",n,"is",fact)
```

Output:

```
enter a value :5
the factorial of 5 is 120
```

4. Write a program to calculate value of the following series $1+x-x^2-+x^3-x^4+\ldots\ldots x^n$

```
# Function to calculate the series sum
def calculate_series(x, n):
    result = 0  # Initial result
    for i in range(n + 1):
        result += (-1) ** i * (x ** i)
    return result

# Input values for x and n
x = float(input("Enter the value of x: "))
n = int(input("Enter the value of n: "))

# Calculate the series
```

```
series_sum = calculate_series(x, n)

# Output the result
print(f"The value of the series is: {series_sum}")
```

Output:

Enter the value of x: 2

Enter the value of n: 8

The value of the series is: 171.0

5. Write a program to print Pascal triangle.

Each number in Pascal's Triangle is the sum of the two numbers directly above it. The topmost row is considered the 0th row, and each row represents the coefficients in the binomial expansion.

Steps to Print Pascal's Triangle:

1. The number of rows in Pascal's Triangle can be specified.
2. Each row consists of binomial coefficients. The number at position $C(n, r)$ in row n and column r is calculated using the binomial coefficient formula:

$$C(n, r) = \frac{n!}{r! \times (n - r)!}$$

3. We can also calculate each number using the previous number in the row.

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

```
# Input: Number of rows for Pascal's Triangle
rows = int(input("Enter the number of rows for Pascal's Triangle: "))

# Generate and print Pascal's Triangle
for n in range(rows):
```

```
    # Initialize a list for the current row
    row = [1] * (n + 1)

    # Compute the values in the current row using the previous row
    for r in range(1, n):
        row[r] = row[r - 1] * (n - r) // r

    # Print the row with appropriate spacing
    print(' ' * (rows - n - 1), end='')  # For alignment
    print(' '.join(map(str, row)))  # Print the row values
```

Output:

```
Enter the number of rows for Pascal's Triangle: 5
    1
   1 1
  1 1 1
 1 2 1 1
1 3 3 1 1
```

Week 2: Viva Questions

Topic : Control Flow

1. What is control flow in Python, and how does it affect the execution of a program?
2. Explain the purpose of if, elif, and else statements in Python.
3. How does Python handle nested if statements?
4. What is the difference between while and for loops in Python?
5. What is a break statement, and how is it used in Python loops?
6. What does the continue statement do in a loop?
7. How does the else clause work with loops in Python?
8. What is the purpose of the pass statement in Python?
9. How would you implement a simple while loop that counts from 1 to 5 in Python?
10. Explain how to use try, except, and finally in Python for error handling.

Week-3 Python Sequences

a) Write a program to sort the numbers in ascending order and strings in reverse alphabetical order

```
# Input list with both numbers and strings
input_list = [5, "banana", 3, "apple", 8, "orange", 1]

# Separate numbers and strings
numbers = [item for item in input_list if isinstance(item, (int, float))]
strings = [item for item in input_list if isinstance(item, str)]

# Sort numbers in ascending order
numbers.sort()

# Sort strings in reverse alphabetical order
strings.sort(reverse=True)

# Combine the sorted lists
sorted_list = numbers + strings

# Print the sorted list
print("Sorted List:", sorted_list)
```

Output:

Sorted List: [1, 3, 5, 8, 'orange', 'banana', 'apple']

b) Given an integer value, return a string with the equivalent English text of each digit. For example, an input of 89 results in "eight-nine" being returned. Write a program to implement it.

```
# Create a dictionary to map digits to their corresponding English words
digit_to_word = {
```

```
    '0': 'zero', '1': 'one', '2': 'two', '3': 'three', '4': 'four',
    '5': 'five', '6': 'six', '7': 'seven', '8': 'eight', '9': 'nine'
}

# Input: integer value
num = 89

# Convert the integer to a string to handle each digit
num_str = str(num)

# Convert each digit into its English equivalent using the dictionary
word_list = [digit_to_word[digit] for digit in num_str]

# Join the words with a hyphen to match the desired output format
result = '-'.join(word_list)

# Print the result
print(result)
```

Output:

eight-nine

c) Write a program to create a function that will return another string similar to the input string, but with its case inverted. For example, input of "Mr. Ed" will result.

```
def invert_case(input_string):
    # Invert the case using the swapcase() method
    return input_string.swapcase()

# Example usage
input_string = "Mr. Ed"
result = invert_case(input_string)

# Print the result
```

```
print(result)
```

Output:

```
mR. eD
```

d) Write a program to take a string and append a backward copy of that string, making a palindrome.

```
# Input string
input_string = "race"

# Create a palindrome by appending the reverse of the string
palindrome = input_string + input_string[::-1]

# Print the result
print(palindrome)
```

Output:

```
raceecar
```

Week 3: Viva Questions

Topic : Python Sequences

1. What are sequences in Python, and what types of sequences are available?
2. What is the difference between lists and tuples in Python?
3. How do you access elements of a sequence in Python?
4. How do you slice a sequence in Python?
5. What is the purpose of the len() function when working with sequences?

6. Can you modify a string in Python? Why or why not?
7. How would you concatenate two lists in Python?
   What are some common methods available for lists and tuples?
8. What is the difference between a sequence and a collection in Python?
9. How do you check if an element exists in a sequence in Python?

Week-4 Python Dictionaries

a) Write a program to create a dictionary and display its keys alphabetically.

```
# Create a dictionary
my_dict = {
    'banana': 3,
    'apple': 5,
    'cherry': 2,
    'date': 4
}

# Get the keys and sort them alphabetically
sorted_keys = sorted(my_dict.keys())

# Print the sorted keys
print("Keys in alphabetical order:", sorted_keys)
```

Output:

Keys in alphabetical order: ['apple', 'banana', 'cherry', 'date']

b) Write a program to take a dictionary as input and return one as output, but the values are now the keys and vice versa.

```
# Input dictionary
input_dict = {
    'a': 1,
    'b': 2,
    'c': 3,
    'd': 4
}

# Swap keys and values using dictionary comprehension
```

```
swapped_dict = {v: k for k, v in input_dict.items()}

# Print the swapped dictionary
print("Swapped dictionary:", swapped_dict)
```

Output:

Swapped dictionary: {1: 'a', 2: 'b', 3: 'c', 4: 'd'}

c) Given a List, extract all elements whose frequency is greater than K. Ex: Input test list= (4,6,4,3,3,4,3,4,3,8], k=3 Output -[4,3] from collections import Counter.

```
# Input list and value of K
test_list = [5, 6, 9, 3, 2, 5, 6, 4, 5, 8, 6, 6, 5, 6, 5]
k = 3

# Count the frequency of each element in the list
freq = Counter(test_list)

# Extract elements whose frequency is greater than k
result = [element for element, count in freq.items() if count > k]

# Print the result
print("Elements whose frequency is greater than", k, ":", result)
```

Output:

Elements whose frequency is greater than 3 : [5, 6]

Week 4: Viva Questions

Topic : Python Dictionaries

1.What is a Python dictionary, and how does it differ from a list?

2.How do you define a dictionary in Python?

3. How can you access the value of a key in a Python dictionary?

4. What happens if you try to access a key that doesn't exist in a dictionary?

5. How can you add or update an item in a Python dictionary?

6. How do you remove a key-value pair from a dictionary?

7. What are some common methods available for dictionaries in Python?

8. Explain the difference between the pop() and popitem() methods in Python dictionaries.

9. Can a dictionary in Python have mutable types like lists or other dictionaries as keys? Why or why not?

10. How would you merge two dictionaries in Python?

Week 5 Files:

a) Write a program to compare two text files. If they are different, give the line and column numbers in the files where the first difference occurs.

```
# File paths
file1 = "file1.txt"
file2 = "file2.txt"
```

```
# Open both files in read mode
with open(file1, 'r') as f1, open(file2, 'r') as f2:
    line_num = 1  # Line counter starts at 1
    while True:
        # Read one line from each file
        line1 = f1.readline()
        line2 = f2.readline()

        # If both files are exhausted, they are identical
        if not line1 and not line2:
            print("The files are identical.")
            break

        # If one file ends before the other, they are different
        if not line1 or not line2:
            print(f"Files differ at line {line_num}.")
            break

        # Compare the lines character by character
        for col_num, (char1, char2) in enumerate(zip(line1, line2), start=1):
            if char1 != char2:
                print(f"Files differ at line {line_num}, column {col_num}.")
                print(f"File1: {line1.strip()}")
                print(f"File2: {line2.strip()}")
                break
        else:
            # Only increment line_num if no difference was found in the current line
            line_num += 1
```

Output:

For the following content in file1.txt:

Hello World

This is a test

Goodbye

And file2.txt containing:
Hello World
This is a text
Goodbye

The output will be:
Files differ at line 2, column 19.
File1: This is a test
File2: This is a text

b) Write a program to compute the number of characters, words and lines in a file.

```
# File path
file_path = "sample.txt"  # Change this to the path of your file

# Initialize counters
num_characters = 0
num_words = 0
num_lines = 0

# Open the file and read it
with open(file_path, 'r') as file:
    for line in file:
        num_lines += 1  # Increment line counter

        # Count characters in the current line
        num_characters += len(line)

        # Count words in the current line by splitting on whitespace
        num_words += len(line.split())

# Print the results
print(f"Number of characters: {num_characters}")
```

```
print(f"Number of words: {num_words}")
print(f"Number of lines: {num_lines}")
```

Output:

For a file sample.txt containing:

Hello world

This is a test file

Counting words and characters

The output will be:

Number of characters: 54

Number of words: 10

Number of lines: 3

Week 5: Viva Questions

Topic : Files

1. What are files in Python, and how are they used?
2. How do you open a file in Python?
3. What are the different modes in which a file can be opened in Python?
4. How do you read data from a file in Python?
5. How do you write data to a file in Python?
6. What is the purpose of with statement when working with files in Python?
7. How can you check if a file exists before opening it in Python?
8. What is the difference between read() and readlines() when reading a file?
9. How do you close a file in Python, and why is it important?
10. How can you handle exceptions when working with files in Python?

Week- 6&7 Functions

a) Write a function ball collide that takes two balls as parameters and computes if they are colliding. Your function should return a Boolean representing whether or not the balls are colliding. Hint: Represent a ball on a plane as a tuple of (x, y, r), r being the radius If (distance between two balls centres) <= (sum of their radii) then (they are colliding).

```
import math

def ball_collide(ball1, ball2):
    # Extract coordinates and radii from the tuples
    x1, y1, r1 = ball1
    x2, y2, r2 = ball2

    # Compute the distance between the centers of the two balls
    distance = math.sqrt((x2 - x1)**2 + (y2 - y1)**2)

    # Check if the distance is less than or equal to the sum of the radii
    if distance <= (r1 + r2):
        return True  # The balls are colliding
    else:
        return False  # The balls are not colliding

# Example usage:
ball1 = (0, 0, 5)  # Ball 1 with center at (0, 0) and radius 5
ball2 = (3, 4, 5)  # Ball 2 with center at (3, 4) and radius 5

# Check if the balls are colliding
if ball_collide(ball1, ball2):
    print("The balls are colliding.")
else:
```

```
    print("The balls are not colliding.")
```

Output:

The balls are colliding.

b) Find mean, median, mode for the given set of numbers in a list.

```
import statistics

# Given list of numbers
numbers = [1, 2, 2, 3, 4, 5, 5, 6, 7, 8]

# Mean (average)
mean = statistics.mean(numbers)

# Median (middle value)
median = statistics.median(numbers)

# Mode (most frequent value)
try:
    mode = statistics.mode(numbers)
except statistics.StatisticsError:
    mode = "No unique mode"

# Print the results
print(f"Mean: {mean}")
print(f"Median: {median}")
print(f"Mode: {mode}")
```

Output:

Mean: 4.3

Median: 4.5

Mode: 2

c) Write simple functions max2() and min2() that take two items and return the larger and smaller item, respectively. They should work on arbitrary Python objects. For example, max2(4, 8) and min2(4, 8) would each return 8 and 4, respectively.

```
def max2(a, b):
    """Returns the larger of two items."""
    if a > b:
        return a
    else:
        return b

def min2(a, b):
    """Returns the smaller of two items."""
    if a < b:
        return a
    else:
        return b

# Example usage
print(max2(4, 8))  # Output: 8
print(min2(4, 8))  # Output: 4
```

Output:

```
8
4
```

d) Write a function nearly equal to test whether two strings are nearly equal. Two strings a and bare nearly equal when a can be generated by a single mutation on b.

```
def nearly_equal(a, b):
    # If the length difference is greater than 1, they can't be nearly equal
    if abs(len(a) - len(b)) > 1:
        return False
```

```
    # Case 1: If strings are the same length, check for a single substitution
    if len(a) == len(b):
        diff_count = 0
        for i in range(len(a)):
            if a[i] != b[i]:
                diff_count += 1
            if diff_count > 1:
                return False
        return diff_count == 1

    # Case 2: If strings have length difference of 1, check for a single insertion or deletion
    if len(a) + 1 == len(b):  # b is longer by one character
        shorter, longer = a, b
    elif len(a) == len(b) + 1:  # a is longer by one character
        shorter, longer = b, a
    else:
        return False

    # Check if we can make b equal to a by removing one character
    for i in range(len(longer)):
        if shorter == longer[:i] + longer[i+1:]:
            return True
    return False

# Example usage:
print(nearly_equal("hello", "helo"))  # Output: True (deletion)
print(nearly_equal("hello", "hullo"))  # Output: True (substitution)
print(nearly_equal("hello", "helloo"))  # Output: True (insertion)
print(nearly_equal("hello", "world"))  # Output: False (multiple changes)
```

Output:

```
True
True
```

True
False

e) Write a function dups to find all duplicates in the list.

```
def dups(lst):
    # Create a dictionary to keep track of the frequency of elements
    freq = {}
    duplicates = []

    # Loop through the list and track occurrences
    for item in lst:
        if item in freq:
            freq[item] += 1
        else:
            freq[item] = 1

    # Add items that appear more than once to the duplicates list
    for item, count in freq.items():
        if count > 1:
            duplicates.append(item)

    return duplicates

# Example usage:
my_list = [1, 2, 3, 4, 2, 5, 3, 6, 7, 8, 3]
print(dups(my_list))  # Output: [2, 3]
```

Output:
[2, 3]

f) Write a function unique to find all the unique elements of a list.

```
def unique(lst):
```

```
    # Create a dictionary to keep track of the frequency of elements
    freq = {}
    unique_elements = []

    # Loop through the list and track occurrences
    for item in lst:
        if item in freq:
            freq[item] += 1
        else:
            freq[item] = 1

    # Add items that appear exactly once to the unique_elements list
    for item, count in freq.items():
        if count == 1:
            unique_elements.append(item)

    return unique_elements

# Example usage:
my_list = [1, 2, 3, 4, 2, 5, 3, 6, 7, 8, 3]
print(unique(my_list))  # Output: [1, 4, 5, 6, 7, 8]
```

Output:

[1, 4, 5, 6, 7, 8]

g) Write a function cumulative_ product to compute cumulative product of a list of numbers.

```
def cumulative_product(lst):
    # Initialize a list to store the cumulative products
    result = []

    # Variable to keep track of the running product
    product = 1
```

```
    # Loop through the input list
    for num in lst:
        product *= num  # Multiply the current number to the running product
        result.append(product)  # Append the running product to the result list

    return result

# Example usage:
my_list = [1, 2, 3, 4]
print(cumulative_product(my_list))  # Output: [1, 2, 6, 24]
```

Output:

```
[1, 2, 6, 24]
```

h) Write a function reverse to reverse a list. Without using the reverse function.

```
def reverse(lst):
    # Initialize an empty list to store the reversed list
    reversed_list = []

    # Loop through the original list from the end to the start
    for i in range(len(lst)-1, -1, -1):
        reversed_list.append(lst[i])  # Append each element to the new list

    return reversed_list

# Example usage:
my_list = [1, 2, 3, 4, 5]
print(reverse(my_list))  # Output: [5, 4, 3, 2, 1]
```

Output:

```
[5, 4, 3, 2, 1]
```

i) Write function to compute GCD, LCM of two numbers. Each function shouldn't exceed one line.

```
import math

# Function to compute GCD of two numbers
def gcd(a, b):
    return math.gcd(a, b)

# Function to compute LCM of two numbers
def lcm(a, b):
    return abs(a * b) // math.gcd(a, b)

# Example usage:
a, b = 12, 15
print("GCD:", gcd(a, b))  # Output: 3
print("LCM:", lcm(a, b))  # Output: 60
```

Output:

GCD: 3

LCM: 60

Week 6&7: Viva questions

1. **What is a function in Python?**
2. **How do you define a function in Python?**
3. **What is the syntax of a Python function?**
4. **What is the use of the def keyword?**
5. **How do you call a function in Python?**
6. **What is the purpose of the return statement?**
7. **Can a Python function return multiple values?**
8. **What is the difference between return and print?**
9. **What are arguments in a function?**
10. **What is the difference between parameters and arguments?**

Week- 8 Multithreading

a) Write a program to create thread using thread module.

```
import threading
import time

# Function to be executed by the thread
def print_numbers():
    for i in range(1, 6):
        print(f"Number {i}")
        time.sleep(1)  # Sleep for 1 second

# Create a thread
thread = threading.Thread(target=print_numbers)

# Start the thread
thread.start()

# Main thread continues to run
print("Main thread is running...")

# Wait for the thread to complete
thread.join()

print("Thread has finished execution.")
```

Output:

```
Main thread is running...
Number 1
Number 2
Number 3
Number 4
```

Number 5
Thread has finished execution.

b) Write a program to create thread using threading module.

```
import threading
import time

# Function that will run in the thread
def print_numbers():
    for i in range(1, 6):
        print(f"Number {i}")
        time.sleep(1)  # Delay for 1 second

# Create a thread and pass the function to be executed
thread = threading.Thread(target=print_numbers)

# Start the thread
thread.start()

# Continue with the main thread
print("Main thread continues to run...")

# Wait for the thread to finish before the program ends
thread.join()

print("Thread execution finished!")
```

Output:

Main thread continues to run...
Number 1
Number 2
Number 3
Number 4

Number 5
Thread execution finished!

c) Write a Program to use Python's threading module to calculate the square and cube of a number concurrently.

```
import threading

# Function to calculate the square of a number
def calculate_square(number):
    square = number ** 2
    print(f"Square of {number} is {square}")

# Function to calculate the cube of a number
def calculate_cube(number):
    cube = number ** 3
    print(f"Cube of {number} is {cube}")

# Number to calculate square and cube for
number = 5

# Create threads for both square and cube calculations
thread_square = threading.Thread(target=calculate_square, args=(number,))
thread_cube = threading.Thread(target=calculate_cube, args=(number,))

# Start both threads
thread_square.start()
thread_cube.start()

# Wait for both threads to complete
thread_square.join()
thread_cube.join()

print("Both square and cube calculations are done.")
```

Output:

Square of 5 is 25

Cube of 5 is 125

Both square and cube calculations are done.

- **What is multithreading?**
- **What is the difference between a process and a thread?**
- **Why do we use multithreading in Python?**
- **Which module is used for multithreading in Python?**
- **How do you create a thread in Python?**
- **What is the use of the threading module?**
- **What is the difference between thread and threading modules?**
- **What is the basic syntax of creating a thread using the threading.Thread class?**
- **How do you start a thread in Python?**
- **What is the use of the start() and run() methods?**

Week 9:Graphs

a) Write a Python program to implement Euler Circuit.

```
class Graph:
    def __init__(self, vertices):
        # Number of vertices
        self.V = vertices
        # Adjacency list for the graph
        self.graph = {i: [] for i in range(vertices)}

    # Add an edge to the graph
    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)
```

```
# Check if all vertices have even degree
def has_even_degree(self):
    for vertex in self.graph:
        if len(self.graph[vertex]) % 2 != 0:
            return False
    return True

# Check if the graph is connected
def is_connected(self):
    visited = [False] * self.V

    # Find a vertex with a non-zero degree to start DFS
    start_vertex = -1
    for i in range(self.V):
        if len(self.graph[i]) > 0:
            start_vertex = i
            break

    if start_vertex == -1:  # No edges in the graph
        return True

    # Perform DFS to check connectivity
    self.dfs(start_vertex, visited)

    # If any vertex with non-zero degree is not visited, return False
    for i in range(self.V):
        if len(self.graph[i]) > 0 and not visited[i]:
            return False

    return True

# DFS to traverse the graph
def dfs(self, vertex, visited):
```

```
    visited[vertex] = True
    for neighbor in self.graph[vertex]:
        if not visited[neighbor]:
            self.dfs(neighbor, visited)

# Function to find the Euler circuit using Hierholzer's algorithm
def find_euler_circuit(self):
    # Step 1: Check if the graph has an Eulerian circuit
    if not self.has_even_degree():
        print("No Euler Circuit exists")
        return

    if not self.is_connected():
        print("No Euler Circuit exists")
        return

    # Step 2: Initialize stack to hold the Eulerian circuit
    circuit = []

    # Find a vertex with a non-zero degree to start the circuit
    current_vertex = 0
    stack = [current_vertex]

    while stack:
        if self.graph[current_vertex]:
            stack.append(current_vertex)
            next_vertex = self.graph[current_vertex].pop()
            self.graph[next_vertex].remove(current_vertex)
            current_vertex = next_vertex
        else:
            circuit.append(current_vertex)
            current_vertex = stack.pop()

    # Print the Euler circuit
```

```
        print("Euler Circuit:", circuit[::-1])

# Example usage:
g = Graph(5)
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(1, 3)
g.add_edge(2, 3)
g.add_edge(3, 4)
g.add_edge(4, 0)

g.find_euler_circuit()
```

Output:

Given the graph:

0 - 1 - 2 - 3 - 4 – 0

Euler Circuit: [0, 4, 3, 2, 1, 0]

b) Write a Python program to implement Dijkstra's algorithm.

```
import heapq
# Function to implement Dijkstra's algorithm
def dijkstra(graph, start):
    # Initialize the priority queue (min-heap)
    pq = [(0, start)]  # (distance, vertex)

    # Initialize distance dictionary with infinity for all vertices
    distances = {vertex: float('inf') for vertex in graph}
    distances[start] = 0

    # Initialize a dictionary to keep track of the shortest path
    previous_vertices = {vertex: None for vertex in graph}
```

```
    while pq:
        # Get the vertex with the smallest distance from the priority queue
        current_distance, current_vertex = heapq.heappop(pq)

        # If the current distance is greater than the stored distance, skip processing
        if current_distance > distances[current_vertex]:
            continue

        # Explore each neighbor of the current vertex
        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            # If a shorter path to the neighbor is found
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous_vertices[neighbor] = current_vertex
                heapq.heappush(pq, (distance, neighbor))  # Add neighbor to the priority queue

    # Return the shortest distance dictionary and the path reconstruction
    return distances, previous_vertices

# Function to reconstruct the shortest path from the start to the target
def reconstruct_path(previous_vertices, target):
    path = []
    while target is not None:
        path.append(target)
        target = previous_vertices[target]
    return path[::-1]  # Return the reversed path

# Example usage:
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
```

```
    'D': {'B': 5, 'C': 1}
}

# Run Dijkstra's algorithm starting from node 'A'
start_node = 'A'
distances, previous_vertices = dijkstra(graph, start_node)

# Output the shortest distance from 'A' to all nodes
print("Shortest distances:", distances)

# Reconstruct and print the shortest path from 'A' to 'D'
path = reconstruct_path(previous_vertices, 'D')
print("Shortest path from A to D:", path)
```

Output:

Shortest distances: {'A': 0, 'B': 1, 'C': 3, 'D': 4}
Shortest path from A to D: ['A', 'B', 'C', 'D']

c) Given a connected graph G with N nodes and M edges (edges are bi-directional). Every node is assigned a value A[i]. We define a value of a simple path as:
Value of path = Maximum of (absolute difference between values of adjacent nodes in a path). A path consists of a sequence of nodes starting with start node S and end node E.
S-u1-u2-…-E is a simple path if all nodes on the path are distinct and S,u1,u2,..E are nodes in G.
Given start node S and end node E, find the minimum possible "value of path" which starts with node S and ends with node E.

```
from collections import deque

def bfs(graph, A, start, end, max_diff):
    # BFS to check if a path exists from start to end with max_diff constraint
    visited = [False] * len(A)
    queue = deque([start])
    visited[start] = True
```

```
    while queue:
        node = queue.popleft()

        if node == end:
            return True

        for neighbor in graph[node]:
            if not visited[neighbor] and abs(A[node] - A[neighbor]) <= max_diff:
                visited[neighbor] = True
                queue.append(neighbor)

    return False

def min_path_value(graph, A, start, end, N):
    # Binary search on the value of the path (max absolute difference)
    left, right = 0, max(max(A) - min(A), max(A) - min(A))

    while left < right:
        mid = (left + right) // 2
        if bfs(graph, A, start, end, mid):
            right = mid  # try to reduce the maximum difference
        else:
            left = mid + 1  # increase the maximum difference

    return left

# Example Usage:
N = 5  # Number of nodes
M = 6  # Number of edges
graph = {0: [1, 2], 1: [0, 3], 2: [0, 3, 4], 3: [1, 2], 4: [2]}
A = [10, 20, 15, 25, 30]  # Values assigned to nodes
start = 0  # Start node
end = 4  # End node
```

```
print(min_path_value(graph, A, start, end, N))
```

Output:

15

d) Yatin created an interesting problem for his college juniors.

Can you solve it?

Given N rooms, where each room has a one-way door to a room denoted by room[i]. where 1<=i<=N) and continuously moves to the room it is connected to i.e. room[i]), the person should end up in room i after K Steps,

Note: The condition should hold for each room. If there are multiple possible values of K modulo (109+7), find the smallest one. If there is no valid value of K, output -1.

```
import math

MOD = 10**9 + 7

# Function to compute LCM of two numbers
def lcm(a, b):
    return (a * b) // math.gcd(a, b)

# Function to find the cycle length starting from a room
def find_cycle_length(start, rooms, visited, in_recursion):
    length = 0
    current = start

    while not visited[current]:
        visited[current] = True
        in_recursion[current] = True
        current = rooms[current] - 1  # rooms are 1-indexed, adjust to 0-indexed
        length += 1

    # Check if we encountered a cycle
```

```
    if in_recursion[current]:
        return length
    else:
        return -1  # no valid cycle, return -1

# Function to solve the problem and find the smallest K
def find_smallest_k(N, rooms):
    visited = [False] * N
    in_recursion = [False] * N  # To track if we are in the current recursion (cycle detection)
    cycle_lengths = []

    # Traverse all rooms and find the cycle lengths
    for i in range(N):
        if not visited[i]:
            # Mark the rooms in the current recursion stack
            cycle_length = find_cycle_length(i, rooms, visited, in_recursion)
            if cycle_length == -1:
                return -1  # If we detect no valid cycle, return -1
            cycle_lengths.append(cycle_length)

        # Reset the recursion stack after finishing this cycle check
        in_recursion = [False] * N

    # Find the LCM of all cycle lengths
    result = 1
    for length in cycle_lengths:
        result = lcm(result, length)
        if result >= MOD:
            result %= MOD

    return result

# Main function to execute the code
if __name__ == "__main__":
```

```
N = int(input("Enter the number of rooms: "))
rooms = list(map(int, input("Enter the room connections (space-separated): ").split()))

# Find the smallest K
K = find_smallest_k(N, rooms)
print(f"The smallest integer K is: {K}")
```

Output:

Enter the number of rooms: 6

Enter the room connections (space-separated): 2 3 4 2 6 5

The smallest integer K is: 4

Week 9 - Viva questions

- **What is a graph in data structures?**
- **What are the main components of a graph?**
- **What is the difference between a directed and an undirected graph?**
- **What are weighted and unweighted graphs?**
- **How can you represent a graph in Python?**
- **What is an adjacency list?**
- **What is an adjacency matrix?**
- **Which is more memory efficient: adjacency list or matrix?**
- **How do you detect if a graph has a cycle?**
- **What is a path and a connected component in a graph?**

Week 10:

Implement the following using python

a) M-coloring

b) Vertex coloring

c) Edge coloring

a) M-coloring

```
# Function to check if the current color assignment is safe for vertex v
def is_safe(v, graph, color, c):
    for i in range(len(graph)):
        if graph[v][i] == 1 and color[i] == c:
            return False
    return True

# Function to solve the M-coloring problem using backtracking
def m_coloring(graph, m, color, v):
    # If all vertices are assigned a color then return True
    if v == len(graph):
        return True

    # Try different colors for vertex v
    for c in range(1, m + 1):
        # Check if assigning color c to vertex v is safe
        if is_safe(v, graph, color, c):
            color[v] = c  # Assign color c to vertex v

            # Recur to assign colors to the next vertices
            if m_coloring(graph, m, color, v + 1):
                return True

            # If assigning color c doesn't lead to a solution, backtrack
            color[v] = 0  # Reset the color assignment

    # If no color can be assigned to this vertex, return False
    return False

# Function to check if the graph can be colored with at most m colors
def graph_coloring(graph, m):
```

```
    color = [0] * len(graph)  # Initialize all vertices as uncolored
    if m_coloring(graph, m, color, 0):
        print("Solution exists: Colors assigned to vertices are:")
        print(color)
    else:
        print("Solution does not exist")

# Example usage
if __name__ == "__main__":
    # Graph represented as an adjacency matrix
    # 0 indicates no edge, 1 indicates an edge between vertices
    graph = [
        [0, 1, 1, 1],  # Vertex 0 is connected to vertex 1, 2, 3
        [1, 0, 1, 0],  # Vertex 1 is connected to vertex 0, 2
        [1, 1, 0, 1],  # Vertex 2 is connected to vertex 0, 1, 3
        [1, 0, 1, 0]   # Vertex 3 is connected to vertex 0, 2
    ]

    m = 3  # Number of colors

    graph_coloring(graph, m)
```

Output:

```
Solution exists: Colors assigned to vertices are:
[1, 2, 3, 2]
```

b) Vertex coloring

```
# Function to check if it's safe to color a vertex with a color
def is_safe(v, graph, color, c):
    for i in range(len(graph)):
        if graph[v][i] == 1 and color[i] == c:
            return False
```

```
    return True

# Function to assign colors to vertices of the graph
def vertex_coloring(graph):
    n = len(graph)  # Number of vertices
    color = [-1] * n  # Initialize all vertices as uncolored

    # Assign colors to all vertices one by one
    color[0] = 0  # Assign the first color to the first vertex

    # Assign colors to the rest of the vertices
    for v in range(1, n):
        # Try different colors for vertex v
        for c in range(0, n):
            if is_safe(v, graph, color, c):
                color[v] = c
                break

    return color

# Function to print the color assignments
def print_coloring(color):
    print("Coloring of vertices:")
    for i, c in enumerate(color):
        print(f"Vertex {i} is colored with color {c}")

# Example usage
if __name__ == "__main__":
    # Graph represented as an adjacency matrix
    # 0 means no edge, 1 means an edge between vertices
    graph = [
        [0, 1, 1, 1],  # Vertex 0 is connected to vertex 1, 2, 3
        [1, 0, 1, 0],  # Vertex 1 is connected to vertex 0, 2
        [1, 1, 0, 1],  # Vertex 2 is connected to vertex 0, 1, 3
```

```
    [1, 0, 1, 0]   # Vertex 3 is connected to vertex 0, 2
]

# Find the vertex coloring
color = vertex_coloring(graph)

# Print the coloring result
print_coloring(color)
```

Output:

Coloring of vertices:
Vertex 0 is colored with color 0
Vertex 1 is colored with color 1
Vertex 2 is colored with color 2
Vertex 3 is colored with color 1

c) Edge coloring

```
# Function to check if it is safe to color an edge with a given color
def is_safe(u, v, graph, color, c):
    # Check if any edge incident to u or v has the same color
    for i in range(len(graph)):
        if graph[u][i] == 1 and color[(min(u, i), max(u, i))] == c:  # check u's edges
            return False
        if graph[v][i] == 1 and color[(min(v, i), max(v, i))] == c:  # check v's edges
            return False
    return True

# Function to assign colors to the edges of the graph
def edge_coloring(graph):
    n = len(graph)
    edges = []

    # Create list of edges (u, v) for all pairs of connected vertices
```

```
    for u in range(n):
        for v in range(u + 1, n):
            if graph[u][v] == 1:
                edges.append((u, v))

    # Dictionary to store color assignment for each edge
    color = {}
    edge_color = 1  # Start coloring from color 1

    # Assign colors to edges one by one
    for u, v in edges:
        # Find the smallest available color for this edge
        while not is_safe(u, v, graph, color, edge_color):
            edge_color += 1  # Try the next color

        color[(min(u, v), max(u, v))] = edge_color  # Assign the color to the edge

    return color

# Function to print the color assignment for the edges
def print_edge_coloring(color):
    print("Edge Coloring:")
    for edge, c in color.items():
        u, v = edge
        print(f"Edge ({u}, {v}) is colored with color {c}")

# Example usage
if __name__ == "__main__":
    # Graph represented as an adjacency matrix
    # 0 means no edge, 1 means an edge between vertices
    graph = [
        [0, 1, 1, 0],  # Vertex 0 is connected to vertex 1, 2
        [1, 0, 1, 1],  # Vertex 1 is connected to vertex 0, 2, 3
        [1, 1, 0, 1],  # Vertex 2 is connected to vertex 0, 1, 3
```

```
    [0, 1, 1, 0]   # Vertex 3 is connected to vertex 1, 2
]

# Find the edge coloring
color = edge_coloring(graph)

# Print the edge coloring result
print_edge_coloring(color)
```

Output:

Input:

```
graph = [
    [0, 1, 1, 0],  # Vertex 0 is connected to vertex 1, 2
    [1, 0, 1, 1],  # Vertex 1 is connected to vertex 0, 2, 3
    [1, 1, 0, 1],  # Vertex 2 is connected to vertex 0, 1, 3
    [0, 1, 1, 0]   # Vertex 3 is connected to vertex 1, 2
]
```

Output:

```
Edge Coloring:
Edge (0, 1) is colored with color 1
Edge (0, 2) is colored with color 2
Edge (1, 2) is colored with color 3
Edge (1, 3) is colored with color 2
Edge (2, 3) is colored with color 1
```

Week 11:

Implement the following graph traversal methods.

a) Depth-First Search

```
 # Iterative Depth-First Search function using a stack
def dfs_iterative(graph, start_vertex):
    visited = set()  # Set to track visited vertices
```

```
    stack = [start_vertex]  # Stack for DFS

    while stack:
        vertex = stack.pop()  # Get the last element from the stack
        if vertex not in visited:
            visited.add(vertex)  # Mark the vertex as visited
            print(vertex, end=' ')  # Process the vertex (printing in this case)

            # Add all unvisited neighbors to the stack
            for neighbor in graph[vertex]:
                if neighbor not in visited:
                    stack.append(neighbor)

# Example usage
if __name__ == "__main__":
    # Graph represented as an adjacency list
    graph = {
        0: [1, 2],
        1: [0, 3, 4],
        2: [0],
        3: [1],
        4: [1]
    }

    print("\nDFS traversal (iterative):")
    dfs_iterative(graph, 0)  # Start DFS from vertex 0
```

Output:

DFS traversal (iterative):
0 2 1 4 3

b) Breadth-First Search

```
from collections import deque
```

```
def bfs(graph, start):
    visited = set()  # to track visited nodes
    queue = deque([start])  # queue for BFS

    while queue:
        node = queue.popleft()  # dequeue node from the queue

        if node not in visited:
            print(node, end=" ")  # process the node (here, we just print it)
            visited.add(node)  # mark the node as visited

            # enqueue all unvisited neighbors
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)

# Example graph as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

# Start BFS from node 'A'
bfs(graph, 'A')
```

Output:

A B C D E F

c) You are presented with a network comprising N computers and M wired connections between them. Your Objective is to optimize the network's connectivity using precisely K wires from your inventory. The aim is to maximize the number of computers that can be linked together within the given constraints. Your task is to determine and report the size of the largest network that can be formed by establishing these connections.
In the context of this problem, computers are considered connected if they share either a direct or indirect wired connection. It is worth noting that the value of K will always be less than the number of isolated (standalone) networks in the given configuration, and it may even be zero.

```
from collections import defaultdict
# Function to perform DFS and find the size of a connected component
def dfs(graph, node, visited):
    stack = [node]
    visited.add(node)
    size = 0
    while stack:
        current = stack.pop()
        size += 1
        for neighbor in graph[current]:
            if neighbor not in visited:
                visited.add(neighbor)
                stack.append(neighbor)
    return size

# Main function to optimize the network
def optimize_network(N, M, edges, K):
    # Step 1: Build the graph from the edges
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    # Step 2: Find the connected components
```

```
    visited = set()
    component_sizes = []

    for node in range(1, N + 1):
        if node not in visited:
            size = dfs(graph, node, visited)
            component_sizes.append(size)

    # Step 3: Sort the components by size in descending order
    component_sizes.sort(reverse=True)

    # Step 4: Use K wires to connect the largest components
    # We'll start by combining the K smallest components
    if K > 0:
        for i in range(K):
            if i < len(component_sizes) - 1:
                component_sizes[i + 1] += component_sizes[i]
                component_sizes[i] = 0  # This component is now merged

    # Step 5: Return the size of the largest connected component after optimization
    return max(component_sizes)

# Example usage
if __name__ == "__main__":
    N = 6  # Number of computers
    M = 4  # Number of wired connections
    edges = [
        (1, 2),
        (2, 3),
        (4, 5),
        (5, 6)
    ]
    K = 2  # Number of wires to add
```

```
largest_network_size = optimize_network(N, M, edges, K)
print(f"The size of the largest network after optimization: {largest_network_size}")
```

Output:

The size of the largest network after optimization: 6

d) A country consists of N cities. These cities are connected with each other using N-1 bidirectional roads that are in the form of a tree. Each city is numbered from 1 to N. You want to safeguard all the roads in the country from any danger, and therefore, you decide to place cameras in certain cities. A camera in a city can safeguard all the roads directly connected to it. Your task is to determine the minimum number of cameras that are required to safeguard the entire country.

```
import sys
sys.setrecursionlimit(10**6)

# Function to perform DFS and compute the DP values
def dfs(u, graph, dp, parent):
    dp[u][0] = 0  # If no camera at u
    dp[u][1] = 1  # If camera at u

    for v in graph[u]:
        if v == parent:  # Don't go back to the parent
            continue
        dfs(v, graph, dp, u)

        # If there's no camera at u, all children must have cameras
        dp[u][0] += dp[v][1]

        # If there's a camera at u, the children can either have a camera or not
        dp[u][1] += min(dp[v][0], dp[v][1])

# Function to find the minimum number of cameras
def min_cameras(N, roads):
    # Create an adjacency list for the tree
```

```
    graph = [[] for _ in range(N + 1)]
    for u, v in roads:
        graph[u].append(v)
        graph[v].append(u)

    # DP table: dp[u][0] = min cameras to cover subtree rooted at u without camera at u
    #           dp[u][1] = min cameras to cover subtree rooted at u with camera at u
    dp = [[0, 0] for _ in range(N + 1)]

    # Start DFS from node 1 (or any node)
    dfs(1, graph, dp, -1)

    # The minimum number of cameras is the minimum of placing a camera at the root or not
    return min(dp[1][0], dp[1][1])

# Example usage
if __name__ == "__main__":
    N = 7  # Number of cities
    roads = [
        (1, 2),
        (1, 3),
        (2, 4),
        (2, 5),
        (3, 6),
        (3, 7)
    ]

    result = min_cameras(N, roads)
    print(f"The minimum number of cameras required: {result}")
```

Output:

Input:

N = 7

roads = [

(1, 2),
(1, 3),
(2, 4),
(2, 5),
(3, 6),
(3, 7)
]

Output:

The minimum number of cameras required: 2

Viva Questions

1.What is the M-Coloring problem in graphs?

2. What does 'M' represent in the M-coloring problem?

3.What is the goal of the M-coloring algorithm?

4. Is M-coloring an NP-complete problem?

5. What are the common methods to solve M-coloring?

6. How do you represent a graph in Python for M-coloring?

7. Can we use the networkx library for coloring?

8. What happens if a vertex cannot be colored with any of the M colors?

9. Which algorithm is used in Python to solve M-coloring?

Week 12: Travelling Salesman problem.

a) You are working in a salesmen company as a programmer.

There are n towns in your country and m directed roads between them. Each road has a cost person should spend on fuel. The company wants to sell goods in all n towns. There

are infinitely many salesmen in the company. We can choose some positive number of salesmen and give a non-empty list of towns to each of them. Towns from the list are the towns to sell goods in. Each salesman will visit all the towns in his list in this particular order in cycle (after the last town he will return to the first town and so on). Salesman can visit other towns on his way but he will not sell goods in these towns. Two Salesmnen cannot sell goods in one town because it will attract unnecessary attention to your company. But for every town there must be a salesman who sell goods in this town. If salesman's list of towns consists of exactly one town then he should pay fee to stay in this town each month (each town has its own fee) or he should go for a round trip and spend money on fuel.
Your task is to calculate the minimal amount of money company must spend monthiy to achieve its goals. We will assume that every salesman will spend a month to make one cycle.

```
import heapq
import sys
from collections import defaultdict

# Helper function for finding SCCs using Kosaraju's Algorithm
def kosaraju_scc(n, graph):
    def dfs(v, graph, visited, stack):
        visited[v] = True
        for neighbor in graph[v]:
            if not visited[neighbor]:
                dfs(neighbor, graph, visited, stack)
        stack.append(v)

    def reverse_dfs(v, rev_graph, visited, component):
        visited[v] = True
        component.append(v)
        for neighbor in rev_graph[v]:
            if not visited[neighbor]:
                reverse_dfs(neighbor, rev_graph, visited, component)
```

```
    visited = [False] * n
    stack = []

    # Step 1: Perform DFS to get the order of nodes to process in reverse graph
    for i in range(n):
        if not visited[i]:
            dfs(i, graph, visited, stack)

    rev_graph = defaultdict(list)
    for u in range(n):
        for v in graph[u]:
            rev_graph[v].append(u)

    visited = [False] * n
    sccs = []

    # Step 2: Reverse DFS based on stack to get the SCCs
    while stack:
        node = stack.pop()
        if not visited[node]:
            component = []
            reverse_dfs(node, rev_graph, visited, component)
            sccs.append(component)

    return sccs

# Dijkstra's Algorithm to find the shortest path
def dijkstra(n, graph, start):
    dist = [float('inf')] * n
    dist[start] = 0
    pq = [(0, start)]  # (cost, node)

    while pq:
        curr_dist, u = heapq.heappop(pq)
```

```
        if curr_dist > dist[u]:
            continue
        for v, cost in graph[u]:
            if dist[u] + cost < dist[v]:
                dist[v] = dist[u] + cost
                heapq.heappush(pq, (dist[v], v))

    return dist

# Main function to calculate the minimal monthly cost
def minimum_cost(n, roads, stay_fees):
    graph = defaultdict(list)
    rev_graph = defaultdict(list)

    # Build the graph
    for u, v, cost in roads:
        graph[u].append((v, cost))
        rev_graph[v].append((u, cost))

    # Step 1: Find SCCs
    sccs = kosaraju_scc(n, graph)
    total_cost = 0

    # Step 2: Calculate minimum cost for each SCC
    for scc in sccs:
        if len(scc) == 1:
            # Only one town in this SCC, consider stay fee vs round trip cost
            town = scc[0]
            round_trip_cost = float('inf')
            for v, cost in graph[town]:
                round_trip_cost = min(round_trip_cost, cost)
            total_cost += min(stay_fees[town], round_trip_cost)
        else:
            # More than one town, find the minimum cycle cost
```

```
            # Compute shortest paths between all pairs of towns in this SCC
            min_cycle_cost = float('inf')
            for u in scc:
                dist = dijkstra(n, graph, u)
                min_cycle_cost = min(min_cycle_cost, dist[u])
            total_cost += min_cycle_cost

    return total_cost

# Example usage
if __name__ == "__main__":
    n = 5  # Number of towns
    roads = [
        (0, 1, 10),
        (1, 2, 20),
        (2, 0, 30),
        (2, 3, 10),
        (3, 4, 10)
    ]
    stay_fees = [5, 10, 8, 7, 6]

    result = minimum_cost(n, roads, stay_fees)
    print(f"The minimal monthly cost: {result}")
```

Output:

Example Input:

- **Number of towns**: n = 5
- **Roads between towns**:
    - (0, 1, 10) (Town 0 to Town 1 with a cost of 10)
    - (1, 2, 20) (Town 1 to Town 2 with a cost of 20)
    - (2, 0, 30) (Town 2 to Town 0 with a cost of 30)
    - (2, 3, 10) (Town 2 to Town 3 with a cost of 10)
    - (3, 4, 10) (Town 3 to Town 4 with a cost of 10)
- **Stay fees for each town**: [5, 10, 8, 7, 6]

Output:

To determine the minimal cost, the solution needs to:

1. **Find SCCs** (Strongly Connected Components).
2. **Calculate the minimum cost for each SCC**:
    - If a single town forms an SCC, choose between the stay fee or round-trip cost.
    - If multiple towns are in an SCC, find the minimal cost cycle using the roads connecting them.

Expected Output Walkthrough:

Let's break down the process:

1. **Finding SCCs**:
    - SCC 1: {0, 1, 2} (Towns 0, 1, and 2 form a strongly connected component since they can all reach each other).
    - SCC 2: {3, 4} (Towns 3 and 4 form another SCC since they are connected directly by roads).
2. **Cycle cost for SCC 1** ({0, 1, 2}):
    - For SCC 1, we compute the minimum cost cycle. You can use Dijkstra's algorithm or simply observe that there's a cycle: (0 → 1 → 2 → 0) with total fuel costs 10 (0 → 1), 20 (1 → 2), and 30 (2 → 0). The total cost is 60.
3. **Cycle cost for SCC 2** ({3, 4}):
    - For SCC 2, there's only one road from 3 to 4 with a fuel cost of 10, and the total round-trip cost would be 10 + 10 = 20.
4. **Final minimal cost**:
    - For SCC 1, use the cycle cost of 60.
    - For SCC 2, use the round-trip cost of 20.
    - Total minimal cost = 60 (SCC 1) + 20 (SCC 2) = 80.

b) It is the final leg of the most famous amazing race. The top 'n' competitors have made it to the final. The final race has just begun. The race has 'm' checkpoints. Each team can reach any of the 'm' checkpoint but after a team reaches a particular checkpoint that checkpoint gets closed and is not open to any other team. The race ends when 'k' teams finish the race. Each team travel at a constant speed throughout the race which might be different for different teams. Given the coordinates of n teams and m checkpoints and speed of individual team return the value of minimum time needed to end the race.

```
import math
import heapq

def min_time_to_finish_race(n, m, k, coordinates_teams, coordinates_checkpoints, speeds):
    # List to store the time for each team to each checkpoint
    time_to_reach = []

    # Calculate time for each team to each checkpoint
    for i in range(n):
        team_x, team_y = coordinates_teams[i]
        speed = speeds[i]
        for j in range(m):
            checkpoint_x, checkpoint_y = coordinates_checkpoints[j]
            distance = math.sqrt((checkpoint_x - team_x) ** 2 + (checkpoint_y - team_y) ** 2)
            time = distance / speed
            time_to_reach.append((time, i, j))  # Store (time, team_id, checkpoint_id)

    # Sort the times in increasing order
    time_to_reach.sort()

    # Min-heap to track the minimum times for finishing the race
    checkpoint_used = set()  # To track which checkpoints are used
    teams_finished = 0
    total_time = 0

    # Process the sorted times
    for time, team_id, checkpoint_id in time_to_reach:
        # If the team hasn't finished and the checkpoint is not already used
        if checkpoint_id not in checkpoint_used:
            checkpoint_used.add(checkpoint_id)
            teams_finished += 1
            total_time = time
            if teams_finished == k:
```

```
            break

    return total_time

# Example inputs
n = 3  # 3 teams
m = 4  # 4 checkpoints
k = 2  # We need 2 teams to finish
coordinates_teams = [(0, 0), (1, 2), (3, 4)]  # Coordinates of the teams
coordinates_checkpoints = [(0, 1), (2, 3), (4, 5), (6, 7)]  # Coordinates of the checkpoints
speeds = [1, 2, 3]  # Speeds of the teams

# Call the function
result = min_time_to_finish_race(n, m, k, coordinates_teams, coordinates_checkpoints,
speeds)
print(f"The minimum time needed to finish the race is: {result}")
```

Output:

The minimum time needed to finish the race is: 0.47140452079103173

c) Little Jhool of is a very lenient teaching assistant in his college. He doesn't like cutting the marks of students, so obviously, every student in his tutorial loves him. But anyway, the teacher has got to know about the leniency of Jhool while giving marks, so this time in well exam, he decides to give a different exam paper to every single student to check how well have the students been taught by Jhool. Now, Little Jhool knows the strong and weak topics of every single student, so he wants to maximize the total marks obtained by students in his tutorial. You are given the number of students in Jhool's tutorial, denoted by n -n also being the number of different exam papers - that is, one for every student. Every student will get only one exam paper to solve. You are further given a matrix (n x n) denoting the marks every student will get if he attempts a particular exam paper. You've to help Jhool figure out a way by which he could maximize the total score obtained by his entire class.

```
import numpy as np
```

```
from scipy.optimize import linear_sum_assignment

def maximize_marks(matrix):
    # Convert the matrix to a numpy array
    cost_matrix = np.array(matrix)

    # Since we are maximizing the marks, we need to minimize the cost, so we negate the matrix
    cost_matrix = -cost_matrix

    # Apply the Hungarian algorithm to find the optimal assignment
    row_ind, col_ind = linear_sum_assignment(cost_matrix)

    # The result is the total of the original marks for the optimal assignment
    total_marks = cost_matrix[row_ind, col_ind].sum() * -1  # Negate again to get original marks
    return total_marks

# Example input
n = 4  # Number of students and papers
marks_matrix = [
    [10, 20, 30, 40],
    [40, 50, 60, 70],
    [70, 80, 90, 100],
    [100, 110, 120, 130]
]

# Call the function
result = maximize_marks(marks_matrix)
print(f"Maximum total marks: {result}")
```

Output:

```
Given the matrix:
10  20  30  40
```

40 50 60 70

70 80 90 100

100 110 120 130

Output:

Maximum total marks: 360

Viva Questions

1. **What is the Travelling Salesman Problem (TSP)?**
2. **Is TSP a decision problem or an optimization problem?**
3. **What is the time complexity of the brute-force TSP algorithm?**
4. **Is the Travelling Salesman Problem NP-complete?**
5. **What are the real-life applications of TSP?**
6. **What are the common algorithms used to solve TSP?**
7. **What is the Held-Karp algorithm?**
8. **What is the difference between exact and approximate algorithms for TSP?**
9. **Why is brute-force not practical for large numbers of cities?**
10. **What is the Greedy approach for TSP?**

Week 13: Construct minimal spanning tree using the following

a) Prim's Algorithm

```
import heapq
def prim(graph, start_node):
    # Number of nodes
    n = len(graph)

    # Initialize MST related structures
    mst_set = [False] * n  # Keeps track of nodes included in MST
    min_edge = [(float('inf'), -1)] * n  # Stores the minimum edge to each node (weight, from node)
    min_edge[start_node] = (0, -1)  # Start node has no parent and zero weight
```

```
    pq = [(0, start_node)]  # Priority queue (min-heap), storing (weight, node)
    mst_edges = []  # This will store the edges of the MST

    total_cost = 0  # To store the total cost of MST

    while pq:
        # Select the edge with the smallest weight
        weight, u = heapq.heappop(pq)

        # If this node is already in the MST, skip it
        if mst_set[u]:
            continue

        # Include this node in the MST
        mst_set[u] = True
        total_cost += weight

        # If u has a valid parent, record the edge
        if min_edge[u][1] != -1:
            mst_edges.append((min_edge[u][1], u, weight))  # (from, to, weight)

        # Update the priority queue with the adjacent edges to the unvisited nodes
        for v, w in graph[u]:
            if not mst_set[v] and w < min_edge[v][0]:
                min_edge[v] = (w, u)
                heapq.heappush(pq, (w, v))

    return mst_edges, total_cost

# Example graph represented as an adjacency list
# Each node is connected to other nodes with respective edge weights
graph = {
    0: [(1, 10), (2, 6), (3, 5)],
    1: [(0, 10), (3, 15)],
```

```
    2: [(0, 6), (3, 4)],
    3: [(0, 5), (1, 15), (2, 4)]
}

# Call the Prim's algorithm
start_node = 0  # Starting from node 0
mst_edges, total_cost = prim(graph, start_node)

# Display the result
print("Edges in the MST:", mst_edges)
print("Total cost of MST:", total_cost)
```

Output:

```
Edges in the MST: [(0, 3, 5), (3, 2, 4), (0, 1, 10)]
Total cost of MST: 19
```

b) Kruskal's Algorithm

```
class UnionFind:
    def __init__(self, n):
        # Initialize the parent and rank arrays
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, u):
        # Path compression optimization
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v):
        # Union by rank optimization
        root_u = self.find(u)
        root_v = self.find(v)
```

```
        if root_u != root_v:
            # Union the two sets
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1
            return True
        return False

def kruskal(n, edges):
    # Sort edges by weight
    edges.sort(key=lambda x: x[2])

    uf = UnionFind(n)
    mst = []
    total_cost = 0

    for u, v, weight in edges:
        # If u and v are not connected, add the edge to the MST
        if uf.union(u, v):
            mst.append((u, v, weight))
            total_cost += weight

    return mst, total_cost

# Example graph represented as a list of edges
# Each edge is a tuple (u, v, weight)
edges = [
    (0, 1, 10),
    (0, 2, 6),
```

```
    (0, 3, 5),
    (1, 3, 15),
    (2, 3, 4)
]

n = 4  # Number of nodes

# Call the Kruskal's algorithm
mst_edges, total_cost = kruskal(n, edges)

# Display the result
print("Edges in the MST:", mst_edges)
print("Total cost of MST:", total_cost)
```

Output:

```
Edges in the MST: [(2, 3, 4), (0, 3, 5), (0, 1, 10)]
Total cost of MST: 19
```

c) There are total N Hacker-cities in plane. Each city is located on coordinates (X[i],Y[i]) and there can be any number of cities on the same coordinates. You have to make these cities connected by constructing some roads in such a way that it is possible to travel between every pair of cities by traversing the roads. The cost of constructing one road between any two cities is the minimum of the absolute difference between their X and Y coordinates.
As you want to earn more and more, you decided to do this in the most optimal way Possible, such that the total cost of constructing these roads is minimal. You have to return the minimum money you need to spend on connecting all the cities.

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, u):
```

```
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])  # Path compression
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)

        if root_u != root_v:
            # Union by rank
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1
            return True
        return False

def minimum_cost_to_connect_cities(X, Y):
    n = len(X)  # Number of cities
    edges = []

    # Create edges based on sorting by X and Y
    cities = list(range(n))

    # Sort cities by X coordinate
    X_sorted = sorted(cities, key=lambda i: X[i])
    for i in range(1, n):
        u, v = X_sorted[i - 1], X_sorted[i]
        cost = min(abs(X[u] - X[v]), abs(Y[u] - Y[v]))
        edges.append((cost, u, v))
```

```
    # Sort cities by Y coordinate
    Y_sorted = sorted(cities, key=lambda i: Y[i])
    for i in range(1, n):
        u, v = Y_sorted[i - 1], Y_sorted[i]
        cost = min(abs(X[u] - X[v]), abs(Y[u] - Y[v]))
        edges.append((cost, u, v))

    # Sort edges by cost
    edges.sort(key=lambda x: x[0])

    # Apply Kruskal's algorithm to find MST
    uf = UnionFind(n)
    mst_cost = 0
    edges_used = 0

    for cost, u, v in edges:
        if uf.union(u, v):
            mst_cost += cost
            edges_used += 1
            if edges_used == n - 1:
                break

    return mst_cost

# Example Input
X = [0, 2, 3, 5]
Y = [0, 1, 2, 3]

# Call the function
result = minimum_cost_to_connect_cities(X, Y)
print(f"Minimum cost to connect all cities: {result}")
```

Output:

Minimum cost to connect all cities: 3

c) Tom is visiting the country Hackerland., Hackerland has n cities and m bi-directional roads. There are k types of tokens. Token *i* costs *ci.* The costs of the tokens are such that for all 2≤i≤k, ci≥2ci-1. For each road, you need to have a particular set of tokens, if you want to travel it. Note that you don't have to give the tokens, you just need to show them. Thus, one token can be used at any number of roads, where it is required. Tom wants to select a set of tokens, such that using them, he can go from any city to any other city. You have to help him minimize the total cost of tokens he buys.

```
import heapq
def find_mst_with_tokens(n, m, roads, token_costs):
    # Create the graph from roads
    graph = [[] for _ in range(n)]

    for u, v, tokens_required in roads:
        graph[u].append((v, tokens_required))
        graph[v].append((u, tokens_required))

    # Prim's algorithm to find MST
    mst_cost = 0
    visited = [False] * n
    pq = [(0, 0)]  # (cost, node)

    while pq:
        cost, node = heapq.heappop(pq)

        if visited[node]:
            continue

        visited[node] = True
        mst_cost += cost

        # For the current node, choose the least token cost for roads to other cities
        for neighbor, tokens_required in graph[node]:
```

```
            if not visited[neighbor]:
                # Find the minimum token cost required for this road
                min_token_cost = min([token_costs[token] for token in tokens_required])
                heapq.heappush(pq, (min_token_cost, neighbor))

    return mst_cost

# Example Input
n = 5  # Number of cities
m = 6  # Number of roads
roads = [
    (0, 1, [1, 2]),  # Road between city 0 and city 1, tokens 1 and 2 required
    (0, 2, [2]),      # Road between city 0 and city 2, token 2 required
    (1, 2, [1, 3]),   # Road between city 1 and city 2, tokens 1 and 3 required
    (1, 3, [3]),      # Road between city 1 and city 3, token 3 required
    (2, 3, [2, 4]),   # Road between city 2 and city 3, tokens 2 and 4 required
    (3, 4, [1, 4])    # Road between city 3 and city 4, tokens 1 and 4 required
]

token_costs = [1, 2, 4, 8, 16]  # Costs of the tokens: token 0 costs 1, token 1 costs 2, etc.

# Call the function
result = find_mst_with_tokens(n, m, roads, token_costs)
print(f"Minimum cost to connect all cities: {result}")
```

Output:

Minimum cost to connect all cities: 10

Viva Questions

1. What is a Minimum Spanning Tree (MST)?
2. What is the difference between a spanning tree and a minimum spanning tree?
3. Can a graph have more than one MST?

4. What are the conditions for a graph to have a spanning tree?
5. Where are MSTs used in real life?
6. What algorithms are commonly used to find MSTs?
7. How does Kruskal's algorithm work?
8. How does Prim's algorithm work?
9. Which algorithm is better for sparse graphs?
10. What data structure is used in Prim’s algorithm?