1. You and your friends are assigned the task of coloring a map with a limited number of colors. The map is represented as a list of regions and their adjacency relationships. The rules are as follows: At each step, you can choose any uncolored region and color it with any available color. Your friend Alice follows the same strategy immediately after you, and then your friend Bob follows suit. You want to maximize the number of regions you personally color. Write a function that takes the map's adjacency list representation and returns the maximum number of regions you can color before all regions are colored. Write a program to implement the Graph coloring technique for an undirected graph. Implement an algorithm with minimum number of colors. edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)] No. of vertices, n = 4

def graph\_coloring(n, edges):

colors = [-1] \* n

adj\_list = [[] for \_ in range(n)]

for u, v in edges:

adj\_list[u].append(v)

adj\_list[v].append(u)

def color\_node(node):

available\_colors = set(range(n))

for neighbor in adj\_list[node]:

if colors[neighbor] in available\_colors:

available\_colors.remove(colors[neighbor])

colors[node] = min(available\_colors)

for node in range(n):

if colors[node] == -1:

color\_node(node)

return colors

n, edges = 4, [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]

print("Coloring of the graph:", graph\_coloring(n, edges))

2. You and your friends are tasked with coloring a map using a limited set of colors, with the following rules: At each step, you can choose any region of the map that hasn't been colored yet and color it with any available color. Your friend Alice will then color the next region using the same strategy, followed by your friend Bob. You aim to maximize the number of regions you color. Given a map represented as a list of regions and their adjacency relationships, write a function to determine the maximum number of regions you can color. Write a program to implement the Graph coloring technique for an undirected graph. Implement an algorithm with minimum number of colors. edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)] No. of vertices, n = 4, k = 3

def max\_regions\_colored(n, edges, k):

colors = [-1] \* n

adj\_list = [[] for \_ in range(n)]

for u, v in edges:

adj\_list[u].append(v)

adj\_list[v].append(u)

def color\_node(node, color):

colors[node] = color

for neighbor in adj\_list[node]:

if colors[neighbor] == -1:

colors[neighbor] = (color + 1) % k

for node in range(0, n, k):

color\_node(node, node % k)

return sum(1 for color in colors if color == 0)

n, k, edges = 4, 3, [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]

print("Maximum regions you can color:", max\_regions\_colored(n, edges, k))

3. You are given an undirected graph represented by a list of edges and the number of vertices n. Your task is to determine if there exists a Hamiltonian cycle in the graph. A Hamiltonian cycle is a cycle that visits each vertex exactly once and returns to the starting vertex. Write a function that takes the list of edges and the number of vertices as input and returns true if there exists a Hamiltonian cycle in the graph, otherwise return false. Example: Given edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2), (2, 4), (4, 0)] and n = 5

def is\_hamiltonian\_cycle(n, edges):

adj\_list = [[] for \_ in range(n)]

for u, v in edges:

adj\_list[u].append(v)

adj\_list[v].append(u)

def dfs(vertex, visited, start, count):

visited[vertex] = True

if count == n and start in adj\_list[vertex]:

return True

for neighbor in adj\_list[vertex]:

if not visited[neighbor]:

if dfs(neighbor, visited, start, count + 1):

return True

visited[vertex] = False

return False

for i in range(n):

if dfs(i, [False] \* n, i, 1):

return True

return False

n, edges = 5, [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2), (2, 4), (4, 0)]

print("Does a Hamiltonian cycle exist?", is\_hamiltonian\_cycle(n, edges))

4. You are given an undirected graph represented by a list of edges and the number of vertices n. Your task is to determine if there exists a Hamiltonian cycle in the graph. A Hamiltonian cycle is a cycle that visits each vertex exactly once and returns to the starting vertex. Write a function that takes the list of edges and the number of vertices as input and returns true if there exists a Hamiltonian cycle in the graph, otherwise return false. Example:edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)] and n = 4

n, edges = 4, [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]

print("Does a Hamiltonian cycle exist?", is\_hamiltonian\_cycle(n, edges))

5. You are tasked with designing an efficient coading to generate all subsets of a given set S containing n elements. Each subset should be outputted in lexicographical order. Return a list of lists where each inner list is a subset of the given set. Additionally, find out how your coading handles duplicate elements in S. A = [1, 2, 3] The subsets of [1, 2, 3] are: [], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]

from itertools import combinations

def all\_subsets\_lexicographical(S):

S.sort()

subsets = []

for i in range(len(S) + 1):

for combo in combinations(S, i):

subsets.append(list(combo))

return subsets

A = [1, 2, 3]

print("All subsets:", all\_subsets\_lexicographical(A))

6. Write a program to implement the concept of subset generation. Given a set of unique integers and a specific integer 3, generate all subsets that contain the element 3. Return a list of lists where each inner list is a subset containing the element 3 E = [2, 3, 4, 5], x = 3, The subsets containing 3 : [3], [2, 3], [3, 4], [3,5], [2, 3, 4], [2, 3, 5], [3, 4, 5], [2, 3, 4, 5] Given an integer array nums of unique elements, return all possible subsets(the power set). The solution set must not contain duplicate subsets. Return the solution in any order. Example 1: Input: nums = [1,2,3] Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]] Example 2: Input: nums = [0] Output: [[],[0]]

def subsets\_with\_element(S, x):

subsets = all\_subsets\_lexicographical(S)

return [subset for subset in subsets if x in subset]

E, x = [2, 3, 4, 5], 3

print("Subsets containing", x, ":", subsets\_with\_element(E, x))

7. You are given two string arrays words1 and words2. A string b is a subset of string a if every letter in b occurs in a including multiplicity. For example, "wrr" is a subset of "warrior" but is not a subset of "world". A string a from words1 is universal if for every string b in words2, b is a subset of a. Return an array of all the universal strings in words1. You may return the answer in any order. Example 1: Input: words1 = ["amazon","apple","facebook","google","leetcode"], words2 = ["e","o"] Output: ["facebook","google","leetcode"] Example 2: Input: words1 = ["amazon","apple","facebook","google","leetcode"], words2 = ["l","e"] Output: ["apple","google","leetcode"]

from collections import Counter

def is\_subset(word, target\_count):

word\_count = Counter(word)

for char, count in target\_count.items():

if word\_count[char] < count:

return False

return True

def universal\_strings(words1, words2):

max\_counts = Counter()

for word in words2:

word\_count = Counter(word)

for char in word\_count:

max\_counts[char] = max(max\_counts[char], word\_count[char])

result = [word for word in words1 if is\_subset(word, max\_counts)]

return result

words1 = ["amazon", "apple", "facebook", "google", "leetcode"]

words2 = ["e", "o"]

print("Universal strings:", universal\_strings(words1, words2))