DAA DAY 11

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 is_safe(graph, color, v, c):
    for i in graph[v]:
        if color[i] == c:
            return False
    return True
def graph_coloring_util(graph, m, color, v):
    if v == len(graph):
        return True
    for c in range(1, m+1):
        if is_safe(graph, color, v, c):
            color[v] = c
            if graph_coloring_util(graph, m, color, v + 1):
                return True
            color[v] = 0
    return False
def find_min_colors(graph):
    for m in range(1, len(graph) + 1):
        color = [0] * len(graph)
        if graph_coloring_util(graph, m, color, 0):
            return m
    return len(graph)
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
n = 4
graph = [[] for _ in range(n)]
```

```
for u, v in edges:
    graph[u].append(v)
    graph[v].append(u)
min_colors = find_min_colors(graph)
print(f"Minimum number of colors required: {min_colors}")
```

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 is_safe(graph, color, v, c):
    for i in graph[v]:
        if color[i] == c:
            return False
    return True
def graph_coloring_util(graph, m, color, v):
    if v == len(graph):
        return True
    for c in range(1, m + 1):
        if is_safe(graph, color, v, c):
            color[v] = c
            if graph_coloring_util(graph, m, color, v + 1):
                return True
            color[v] = 0
    return False
def find_min_colors(graph):
    for m in range(1, len(graph) + 1):
        color = [0] * len(graph)
        if graph_coloring_util(graph, m, color, 0):
            return m
```

```
    return len(graph)
```

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_valid(v, pos, path, graph):
    if graph[path[pos - 1]][v] == 0:
        return False
    if v in path:
        return False
    return True
def hamiltonian_cycle_util(graph, path, pos):
    if pos == len(graph):
        return graph[path[pos - 1]][path[0]] == 1
    for v in range(1, len(graph)):
        if is_valid(v, pos, path, graph):
            path[pos] = v
            if hamiltonian_cycle_util(graph, path, pos + 1):
                return True
            path[pos] = -1
    return False
def has_hamiltonian_cycle(edges, n):
    graph = [[0] * n for _ in range(n)]
    for u, v in edges:
        graph[u][v] = 1
        graph[v][u] = 1
    path = [-1] * n
    path[0] = 0
    return hamiltonian_cycle_util(graph, path, 1)
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2), (2, 4), (4, 0)]
n = 5
```

print("Hamiltonian cycle exists:" if has_hamiltonian_cycle(edges, n) else "Hamiltonian cycle does not exist.")

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

```
def is_valid(v, pos, path, graph):
    if graph[path[pos - 1]][v] == 0:
        return False
    if v in path:
        return False
    return True
def hamiltonian_cycle_util(graph, path, pos):
    if pos == len(graph):
        return graph[path[pos - 1]][path[0]] == 1
    for v in range(len(graph)):
        if is_valid(v, pos, path, graph):
            path[pos] = v
            if hamiltonian_cycle_util(graph, path, pos + 1):
                return True
            path[pos] = -1
    return False
def has_hamiltonian_cycle(edges, n):
    graph = [[0] * n for _ in range(n)]
    for u, v in edges:
        graph[u][v] = 1
        graph[v][u] = 1
    path = [-1] * n
    path[0] = 0
    return hamiltonian_cycle_util(graph, path, 1)
```

```
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]

n = 4

print("Hamiltonian cycle exists:" if has_hamiltonian_cycle(edges, n) else "Hamiltonian cycle does not exist.")
```

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]

```
def subsets_helper(nums, start, path, result):
    result.append(path[:])
    for i in range(start, len(nums)):
        # Skip duplicate elements
        if i > start and nums[i] == nums[i - 1]:
            continue
        path.append(nums[i])
        subsets_helper(nums, i + 1, path, result)
        path.pop()
def generate_subsets(nums):
    nums.sort()
    result = []
    subsets_helper(nums, 0, [], result)
    return result
A = [1, 2, 3]
print("Subsets of [1, 2, 3] are:", generate_subsets(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.

```
def subsets_with_element(nums, x):
    def backtrack(start, path):
        if x in path:
```

```
            result.append(path[:])
        for i in range(start, len(nums)):
            path.append(nums[i])
            backtrack(i + 1, path)
            path.pop()
    result = []
    nums.sort()
    backtrack(0, [])
    return result
E = [2, 3, 4, 5]
x = 3
print("Subsets containing 3 are:", 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.

```
from collections import Counter
def is_universal(word, required_count):
    word_count = Counter(word)
    for char, count in required_count.items():
        if word_count[char] < count:
            return False
    return True
def find_universal_words(words1, words2):
    required_count = Counter()
    for word in words2:
        word_count = Counter(word)
        for char, count in word_count.items():
            required_count[char] = max(required_count[char], count)
    result = [word for word in words1 if is_universal(word, required_count)]
    return result
```

```
words1 = ["amazon", "apple", "facebook", "google", "leetcode"]
words2 = ["e", "o"]
print("Universal words:", find_universal_words(words1, words2))
```