STUDENT NAME: T.NIKHIL KUMAR REDDY

REG.NO: 192372024.

SUB:DESIGN AND ANALYSIS OF ALGORITHM.

SUB.CODE:CSA0689.

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.

```python
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.adj_list = defaultdict(list)

    def add_edge(self, u, v):
        self.adj_list[u].append(v)
        self.adj_list[v].append(u)

    def is_safe(self, v, color, colors):
        for neighbor in self.adj_list[v]:
            if colors[neighbor] == color:
                return False
        return True

    def graph_coloring_util(self, m, colors, v):
        if v == self.vertices:
            return True

        for color in range(1, m + 1):
            if self.is_safe(v, color, colors):
                colors[v] = color
                if self.graph_coloring_util(m, colors, v + 1):
                    return True
                colors[v] = 0

        return False

    def find_min_colors(self):
        m = 1
        while True:
            colors = [0] * self.vertices
            if self.graph_coloring_util(m, colors, 0):
                return m
            m += 1

def max_colored_regions(n, edges):
    g = Graph(n)
    for u, v in edges:
        g.add_edge(u, v)
    min_colors = g.find_min_colors()
    colors = [0] * n

    def turn_based_coloring():
        turn = 0
        regions_colored = 0
        while True:
            all_colored = all(color != 0 for color in colors)
            if all_colored:
                break
            available_colors = set(range(1, min_colors + 1))
            for i in range(n):
                if colors[i] == 0:
                    safe_colors = {color for color in available_colors if g.is_safe(i, color, colors)}
                    if safe_colors:
                        chosen_color = safe_colors.pop()
                        colors[i] = chosen_color
                        regions_colored += 1
                        turn = (turn + 1) % 3
                        break
        return regions_colored
    return turn_based_coloring()
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
n = 4
print(max_colored_regions(n, edges))
```

```
4

=== Code Execution Successful ===
```

1. 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

```python
from collections import defaultdict
class Graph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.adj_list = defaultdict(list)
    def add_edge(self, u, v):
        self.adj_list[u].append(v)
        self.adj_list[v].append(u)
    def is_safe(self, v, color, colors):
        for neighbor in self.adj_list[v]:
            if colors[neighbor] == color:
                return False
        return True
    def graph_coloring_util(self, m, colors, v):
        if v == self.vertices:
            return True
        for color in range(1, m + 1):
            if self.is_safe(v, color, colors):
                colors[v] = color
                if self.graph_coloring_util(m, colors, v + 1):
                    return True
                colors[v] = 0

        return False
    def find_min_colors(self):
        m = 1
        while True:
            colors = [0] * self.vertices
            if self.graph_coloring_util(m, colors, 0):
                return m
            m += 1

def max_colored_regions(n, edges, k):
    g = Graph(n)
    for u, v in edges:
        g.add_edge(u, v)

    min_colors = g.find_min_colors()

    if min_colors > k:
        raise ValueError("The number of colors required exceeds the available colors.")
    colors = [0] * n
    turn = 0
    regions_colored_by_you = 0

    while True:
        all_colored = all(color != 0 for color in colors)
        if all_colored:
            break

        available_colors = set(range(1, k + 1))
        for i in range(n):
            if colors[i] == 0:
                safe_colors = {color for color in available_colors if g.is_safe(i, color, colors)}
                if safe_colors:
                    chosen_color = safe_colors.pop()
                    colors[i] = chosen_color
                    if turn == 0:
                        regions_colored_by_you += 1
                    turn = (turn + 1) % 3
                    break

    return regions_colored_by_you

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

```
2

=== Code Execution Successful ===
```

2. 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.

```python
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.adj_list = defaultdict(list)

    def add_edge(self, u, v):
        self.adj_list[u].append(v)
        self.adj_list[v].append(u)

    def is_valid(self, v, pos, path):
        if v not in self.adj_list[path[pos - 1]]:
            return False
        if v in path:
            return False
        return True

    def ham_cycle_util(self, path, pos):
        if pos == self.vertices:
            return path[0] in self.adj_list[path[-1]]

        for v in range(1, self.vertices):
            if self.is_valid(v, pos, path):
                path[pos] = v
                if self.ham_cycle_util(path, pos + 1):
                    return True
                path[pos] = -1

        return False

    def has_hamiltonian_cycle(self):
        path = [-1] * self.vertices
        path[0] = 0
        return self.ham_cycle_util(path, 1)

def has_hamiltonian_cycle(n, edges):
    g = Graph(n)
    for u, v in edges:
        g.add_edge(u, v)

    return g.has_hamiltonian_cycle()

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

```
False

=== Code Execution Successful ===
```

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

```python
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.adj_list = defaultdict(list)

    def add_edge(self, u, v):
        self.adj_list[u].append(v)
        self.adj_list[v].append(u)
    def is_valid(self, v, pos, path):
        if v not in self.adj_list[path[pos - 1]]:
            return False

        if v in path:
            return False

        return True

    def ham_cycle_util(self, path, pos):
        if pos == self.vertices:
            return path[0] in self.adj_list[path[-1]]
        for v in range(1, self.vertices):
            if self.is_valid(v, pos, path):
                path[pos] = v
                if self.ham_cycle_util(path, pos + 1):
                    return True
                path[pos] = -1

        return False

    def has_hamiltonian_cycle(self):
        path = [-1] * self.vertices
        path[0] = 0
        return self.ham_cycle_util(path, 1)

def has_hamiltonian_cycle(n, edges):
    g = Graph(n)
    for u, v in edges:
        g.add_edge(u, v)

    return g.has_hamiltonian_cycle()

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

Output

```
True

=== Code Execution Successful ===
```

4. 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].

```python
def subsets(nums):
    def backtrack(start, path):
        results.append(path[:])
        for i in range(start, len(nums)):
            path.append(nums[i])
            backtrack(i + 1, path)
            path.pop()

    results = []
    nums.sort()
    backtrack(0, [])
    return results

A = [1, 2, 3]
print(subsets(A))
```

```
[[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]

=== Code Execution Successful ===
```

```python
def subsets_with_duplicates(nums):
    def backtrack(start, path):
        results.append(path[:])
        for i in range(start, len(nums)):
            if i > start and nums[i] == nums[i - 1]:
                continue
            path.append(nums[i])
            backtrack(i + 1, path)
            path.pop()

    results = []
    nums.sort()
    backtrack(0, [])
    return results

A = [1, 2, 2]
print(subsets_with_duplicates(A))
```

```
[[], [1], [1, 2], [1, 2, 2], [2], [2, 2]]

=== Code Execution Successful ===
```

5. 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]]

```python
def generate_all_subsets(nums):
    def backtrack(start, path):
        results.append(path[:])

        for i in range(start, len(nums)):
            path.append(nums[i])
            backtrack(i + 1, path)
            path.pop()

    results = []
    nums.sort()
    backtrack(0, [])
    return results

nums1 = [1, 2, 3]
print(generate_all_subsets(nums1))

nums2 = [0]
print(generate_all_subsets(nums2))
```

```
[[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
[[], [0]]

=== Code Execution Successful ===
```

6. 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"].

main.py

Share Run

```python
from collections import Counter
def wordSubsets(words1, words2):
    def get_max_freq(words):
        max_freq = Counter()
        for word in words:
            word_freq = Counter(word)
            for char, count in word_freq.items():
                max_freq[char] = max(max_freq[char], count)
        return max_freq
    def is_universal(word, max_freq):
        word_freq = Counter(word)
        for char, count in max_freq.items():
            if word_freq[char] < count:
                return False
        return True
    max_freq = get_max_freq(words2)
    return [word for word in words1 if is_universal(word, max_freq)]

words1_1 = ["amazon","apple","facebook","google","leetcode"]
words2_1 = ["e","o"]
print(wordSubsets(words1_1, words2_1))

words1_2 = ["amazon","apple","facebook","google","leetcode"]
words2_2 = ["l","e"]
print(wordSubsets(words1_2, words2_2))
```

Output

```
['facebook', 'google', 'leetcode']
['apple', 'google', 'leetcode']

=== Code Execution Successful ===
```