92. 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

Aim: implementation of the graph coloring technique to maximize the number of regions you can color.

Algorithm:

Step-1: Initialize an empty color assignment and a queue with all uncolored regions.
Step-2: While there are uncolored regions, choose the region with the minimum degree and color it with the minimum available color.
Step-3: Update the color assignment and the queue.
Step-4: Return the maximum number of regions you can color.

Program:

main.py

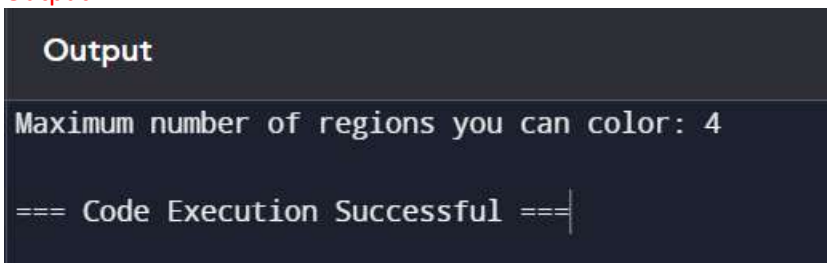
Share Run

```
from collections import defaultdict, deque
def max_regions_to_color(edges, n):
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)
    color_assignment = [-1] * n
    queue = deque(range(n))
    max_regions = 0
    while queue:
        region = min(queue, key=lambda x: len(graph[x]))
        queue.remove(region)
        available_colors = set(range(n))
        for neighbor in graph[region]:
            if color_assignment[neighbor] != -1:
                available_colors.discard(color_assignment[neighbor])
        color_assignment[region] = min(available_colors)
        max_regions += 1
    return max_regions
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
n = 4
print("Maximum number of regions you can color:", max_regions_to_color(edges, n))
```

Input:

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

Output:



Time Complexity: T(n) = O(n^2)
Result: The program outputs the maximum number of regions you can color.

93. 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
Aim: implementation of the graph coloring technique to maximize the number of regions you can color:
Algorithm:
Step-1: Initialize an empty color assignment and a queue with all uncolored regions.
Step-2: While there are uncolored regions, choose the region with the minimum degree and color it with the minimum available color.
Step-3: Update the color assignment and the queue.
Step-4: Return the maximum number of regions you can color.
Program:

main.py

Share Run

```python
from collections import defaultdict, deque
def max_regions_to_color(edges, n, k):
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)
    color_assignment = [-1] * n
    queue = deque(range(n))
    max_regions = 0
    while queue:
        region = min(queue, key=lambda x: len(graph[x]))
        queue.remove(region)
        available_colors = set(range(k))
        for neighbor in graph[region]:
            if color_assignment[neighbor] != -1:
                available_colors.discard(color_assignment[neighbor])
        color_assignment[region] = min(available_colors)
        max_regions += 1
    return max_regions
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
n = 4
k = 3
print("Maximum number of regions you can color:", max_regions_to_color(edges, n,
    k))
```

Input:

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

Output:

```
Output
Maximum number of regions you can color: 4

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

Time Complexity: T(n) = O(n^2)
Result: The program outputs the maximum number of regions you can color.

94.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

Aim: implementation of the algorithm to determine if there exists a Hamiltonian cycle in the graph.

Algorithm:

Step-1: Create an adjacency matrix representation of the graph.
Step-2: Use a backtracking approach to find a Hamiltonian cycle.
Step-3: If a Hamiltonian cycle is found, return true.
Step-4: If no Hamiltonian cycle is found after exploring all possibilities, return false.

Program:

main.py

```python
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
    def backtrack(path, visited):
        if len(path) == n:
            if graph[path[0]][path[-1]] == 1:
                return True
            return False
        for i in range(n):
            if not visited[i] and graph[path[-1]][i] == 1:
                visited[i] = True
                path.append(i)
                if backtrack(path, visited):
                    return True
                path.pop()
                visited[i] = False
        return False
    visited = [False] * n
    visited[0] = True
    path = [0]
    return backtrack(path, visited)
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2), (2, 4), (4, 0)]
n = 5
print("Has Hamiltonian cycle:", has_hamiltonian_cycle(edges, n))
```

Input:

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

Output:

```
Output

Has Hamiltonian cycle: False

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

Time Complexity: T(n) = O(n!)

Result: The program outputs whether there exists a Hamiltonian cycle in the graph.

95.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.

Aim: implementation of the algorithm to determine if there exists a Hamiltonian cycle in the graph:

Algorithm:

Step-1: Create an adjacency matrix representation of the graph.
Step-2: Use a backtracking approach to find a Hamiltonian cycle, starting from each vertex.
Step-3: If a Hamiltonian cycle is found, return true.
Step-4: If no Hamiltonian cycle is found after exploring all possibilities, return false.

Program:

main.py

Share Run

```
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
    def backtrack(path, visited):
        if len(path) == n:
            if graph[path[0]][path[-1]] == 1:
                return True
            return False
        for i in range(n):
            if not visited[i] and graph[path[-1]][i] == 1:
                visited[i] = True
                path.append(i)
                if backtrack(path, visited):
                    return True
                path.pop()
                visited[i] = False
        return False
    for i in range(n):
        visited = [False] * n
        visited[i] = True
        path = [i]
        if backtrack(path, visited):
            return True
    return False
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
n = 4
print("Has Hamiltonian cycle:", has_hamiltonian_cycle(edges, n))
```

Input:

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

Output:

```
Output

Has Hamiltonian cycle: True

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

Time Complexity: T(n) = O(n!)
Result: The program outputs whether there exists a Hamiltonian cycle in the graph.

96.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].
Aim: implementation of the algorithm to generate all subsets of a given set S containing n elements:
Algorithm:
Step-1: Use a recursive approach to generate all subsets.
Step-2: In each recursive call, add the current element to the subset or not.
Step-3: Return the list of all generated subsets.
Step-4: Sort the list of subsets in lexicographical order.
Program:

main.py Share Run

```
def generate_subsets(S):
    def recursive_generate_subsets(S, current_subset, subsets):
        if len(current_subset) == len(S):
            subsets.append(current_subset[:])
            return
        recursive_generate_subsets(S, current_subset + [S[0]], subsets)
        recursive_generate_subsets(S[1:], current_subset, subsets)

    subsets = []
    recursive_generate_subsets(S, [], subsets)
    subsets.sort()
    return subsets
A = [1, 2, 3]
print("Subsets of", A, ":", generate_subsets(A))
```

Input:

```
A = [1, 2, 3]
print("Subsets of", A, ":", generate_subsets(A))
```

Output:

```
Output                                                                    Clea

Subsets of [1, 2, 3] : [[], [1], [1, 1], [1, 1, 1], [1, 2], [2], [2, 2], [3]]

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

Time Complexity: T(n) = O(2^n)
Result: The program outputs the list of all subsets of the given set S in lexicographical order.

97.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]]
Aim: implementation of the algorithm to generate all subsets that contain a specific element.
Algorithm:
Step-1: Use a recursive approach to generate all subsets of the given set.
Step-2: In each recursive call, add the current element to the subset or not.
Step-3: Filter the generated subsets to only include those that contain the specific element.
Step-4: Return the list of filtered subsets.
Program:

main.py Share Run

```python
def generate_subsets_with_element(nums, x):
    def recursive_generate_subsets(nums, current_subset, subsets):
        if len(current_subset) == len(nums):
            subsets.append(current_subset[:])
            return
        recursive_generate_subsets(nums, current_subset + [nums[0]], subsets)
        recursive_generate_subsets(nums[1:], current_subset, subsets)

    subsets = []
    recursive_generate_subsets(nums, [], subsets)
    subsets = [subset for subset in subsets if x in subset]
    return subsets
E = [2, 3, 4, 5]
x = 3
print("Subsets containing", x, ":", generate_subsets_with_element(E, x))
```

Input:

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

Output:

```
Output                                                    Clear
Subsets containing 3 : [[2, 2, 3], [2, 3, 3], [2, 3], [3, 3, 3], [3, 3], [3, 4], [3]]

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

Time Copmlexity: T(n) = O(2^n)

Result: The program outputs the list of all possible subsets (power set) of the given set.

98.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"]

Aim: implementation of the algorithm to find all universal strings in words1:

Algorithm:

Step-1: Create a frequency count dictionary for each string in words2.

Step-2: For each string in words1, check if it is a universal string by verifying that every letter in the frequency count dictionary is a subset of the string.

Step-3: If the string is universal, add it to the result list.

Step-4: Return the list of universal strings.

Program:

main.py

Share Run

```
from collections import Counter
def universal_strings(words1, words2):
    freq_count_dict = [Counter(word) for word in words2]
    universal_strings = []
    for word in words1:
        word_count = Counter(word)
        is_universal = all(all(char in word_count and word_count[char] >=
            freq_count[char] for char in freq_count) for freq_count in
            freq_count_dict)
        if is_universal:
            universal_strings.append(word)
    return universal_strings
words1 = ["amazon","apple","facebook","google","leetcode"]
words2 = ["e","o"]
print("Universal strings:", universal_strings(words1, words2))
```

Input:

```
words1 = ["amazon","apple","facebook","google","leetcode"]
words2 = ["e","o"]
```

Output:

Output

```
Universal strings: ['facebook', 'google', 'leetcode']

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

Time Complexity: The time complexity of this implementation is O(m * n * k), where m is the number of strings in words1, n is the number of strings in words2, and k is the average length of a string in words1 and words2. This is because we iterate over each string in words1 and words2, and for each string, we create a frequency count dictionary and verify if it is a subset of the other strings.

Result: The program outputs the list of all universal strings in words1.