DAY 11 PROGRAM

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 asfollows: At each step, you can choose any uncolored region and color it with any availablecolor. Your friend Alice follows the same strategy immediately after you, and then yourfriend 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 maximumnumber of regions you can color before all regions are colored. Write a program toimplement the Graph coloring technique for an undirected graph. Implement an algorithmwith minimum number of colors. edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)] No. of vertices, n= 4

```python
def max_regions_colored(edges, n):
    graph = {i: [] for i in range(n)}
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    color = [-1] * n

    def min_available_color(vertex):
        available_colors = [True] * n
        for neighbor in graph[vertex]:
            if color[neighbor] != -1:
                available_colors[color[neighbor]] = False
        for c in range(n):
            if available_colors[c]:
                return c
        return -1

    your_turn = True
    regions_colored_by_you = 0
    uncolored_vertices = set(range(n))

    while uncolored_vertices:
        vertex = uncolored_vertices.pop()
        col = min_available_color(vertex)
        color[vertex] = col

        if your_turn:
            regions_colored_by_you += 1

        your_turn = not your_turn
        if uncolored_vertices:
            vertex = uncolored_vertices.pop()
            col = min_available_color(vertex)
            color[vertex] = col
            your_turn = not your_turn

        if uncolored_vertices:
            vertex = uncolored_vertices.pop()
            col = min_available_color(vertex)
            color[vertex] = col
            your_turn = not your_turn

    return regions_colored_by_you

edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
n = 4
print("Maximum regions you can color:", max_regions_colored(edges, n))
```

```
Maximum regions you can color: 1


...Program finished with exit code 0
Press ENTER to exit console.
```

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

```python
def max_regions_colored(edges, n, k):
    graph = {i: [] for i in range(n)}
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)
    color = [-1] * n
    def min_available_color(vertex):
        available_colors = [True] * k
        for neighbor in graph[vertex]:
            if color[neighbor] != -1:
                available_colors[color[neighbor]] = False
        for c in range(k):
            if available_colors[c]:
                return c
        return -1
    your_turn = True
    regions_colored_by_you = 0
    uncolored_vertices = set(range(n))
    while uncolored_vertices:
        vertex = uncolored_vertices.pop()
        col = min_available_color(vertex)
        color[vertex] = col
        if your_turn:
            regions_colored_by_you += 1
        your_turn = not your_turn
        if uncolored_vertices:
            vertex = uncolored_vertices.pop()
            col = min_available_color(vertex)
```

```python
        if uncolored_vertices:
            vertex = uncolored_vertices.pop()
            col = min_available_color(vertex)
            color[vertex] = col
            your_turn = not your_turn
        if uncolored_vertices:
            vertex = uncolored_vertices.pop()
            col = min_available_color(vertex)
            color[vertex] = col
            your_turn = not your_turn
    return regions_colored_by_you
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
n = 4
k = 3
print("Maximum regions you can color:", max_regions_colored(edges, n, k))
```

input

```
Maximum regions you can color: 1


...Program finished with exit code 0
Press ENTER to exit console.
```

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

```python
def is_hamiltonian_cycle(graph, path, pos, n):
    if pos == n:
        if path[pos - 1] in graph[path[0]]:
            return True
        else:
            return False
    for vertex in range(1, n):
        if vertex in graph[path[pos - 1]] and vertex not in path:
            path[pos] = vertex
            if is_hamiltonian_cycle(graph, path, pos + 1, n):
                return True
            path[pos] = -1
    return False
def hamiltonian_cycle(edges, n):
    graph = {i: [] for i in range(n)}
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)
    path = [-1] * n
    path[0] = 0
    if is_hamiltonian_cycle(graph, path, 1, n):
        return True
    else:
        return False
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2), (2, 4), (4, 0)]
n = 5
print(hamiltonian_cycle(edges, n))
```

input

```
False

...Program finished with exit code 0
```

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

```python
def is_hamiltonian_cycle(graph, path, pos, n):
    if pos == n:
        if path[pos - 1] in graph[path[0]]:
            return True
        else:
            return False
    for vertex in range(1, n):
        if vertex in graph[path[pos - 1]] and vertex not in path:
            path[pos] = vertex
            if is_hamiltonian_cycle(graph, path, pos + 1, n):
                return True
            path[pos] = -1
    return False
def hamiltonian_cycle(edges, n):
    graph = {i: [] for i in range(n)}
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)
    path = [-1] * n
    path[0] = 0
    if is_hamiltonian_cycle(graph, path, 1, n):
        return True
    else:
        return False
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
n = 4
print(hamiltonian_cycle(edges, n))
```

input

```
True

...Program finished with exit code 0
```

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]

```python
def generate_subsets(S):
    S.sort()
    subsets = []
    def backtrack(start, current_subset):
        # Add the current subset to the list of subsets
        subsets.append(current_subset[:])
        for i in range(start, len(S)):
            if i > start and S[i] == S[i-1]:
                continue
            current_subset.append(S[i])
            backtrack(i + 1, current_subset)
            current_subset.pop()
    backtrack(0, [])
    return subsets
S = [1, 2, 2]
subsets = generate_subsets(S)
for subset in subsets:
    print(subset)
```

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

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]]

```python
def generate_subsets_containing_element(nums, x):
    def generate_all_subsets(nums):
        subsets = []
        def backtrack(start, current_subset):
            subsets.append(current_subset[:])
            for i in range(start, len(nums)):
                current_subset.append(nums[i])
                backtrack(i + 1, current_subset)
                current_subset.pop()
        backtrack(0, [])
        return subsets
    all_subsets = generate_all_subsets(nums)
    subsets_containing_x = [subset for subset in all_subsets if x in subset]
    return subsets_containing_x
def generate_power_set(nums):
    def generate_all_subsets(nums):
        subsets = []
        def backtrack(start, current_subset):
            subsets.append(current_subset[:])
            for i in range(start, len(nums)):
                current_subset.append(nums[i])
                backtrack(i + 1, current_subset)
                current_subset.pop()
        backtrack(0, [])
        return subsets
```

input

```
Power set of [1, 2, 3]:
[[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]

Power set of [0]:
[[], [0]]
```

```python
        return subsets
    return generate_all_subsets(nums)
E = [2, 3, 4, 5]
x = 3
subsets_with_3 = generate_subsets_containing_element(E, x)
print("Subsets containing 3:")
for subset in subsets_with_3:
    print(subset)
nums1 = [1, 2, 3]
print("\nPower set of [1, 2, 3]:")
print(generate_power_set(nums1))
nums2 = [0]
print("\nPower set of [0]:")
print(generate_power_set(nums2))
```

DAY 11 PROGRAM

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"]

```python
from collections import Counter
def is_universal_word(word, combined_max_freq):
    word_freq = Counter(word)
    for char, freq in combined_max_freq.items():
        if word_freq[char] < freq:
            return False
    return True
def word_subsets(words1, words2):
    combined_max_freq = Counter()
    for word in words2:
        word_freq = Counter(word)
        for char, freq in word_freq.items():
            combined_max_freq[char] = max(combined_max_freq[char], freq)
    universal_words = []
    for word in words1:
        if is_universal_word(word, combined_max_freq):
            universal_words.append(word)
    return universal_words
words1 = ["amazon", "apple", "facebook", "google", "leetcode"]
words2 = ["e", "o"]
print(word_subsets(words1, words2))
words1 = ["amazon", "apple", "facebook", "google", "leetcode"]
words2 = ["l", "e"]
print(word_subsets(words1, words2))
```

input

```
Power set of [1, 2, 3]:
[[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]

Power set of [0]:
[[], [0]]
```