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

**CODE:**

def color\_graph(edges, n):

color = [-1] \* n

available = [True] \* n

def find\_color(available):

for clr in range(len(available)):

if available[clr]:

return clr

return -1

def assign\_color(vertex):

for u, v in edges:

if u == vertex and color[v] != -1:

available[color[v]] = False

if v == vertex and color[u] != -1:

available[color[u]] = False

clr = find\_color(available)

color[vertex] = clr

for i in range(len(available)):

available[i] = True

your\_turn = True

your\_count = 0

for vertex in range(n):

if your\_turn:

assign\_color(vertex)

your\_count += 1

else:

assign\_color(vertex)

your\_turn = not your\_turn

return your\_count

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

n = 4

result = color\_graph(edges, n)

print("Maximum number of regions you can color:", result)

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

**CODE:**

def color\_graph(edges, n, k):

color = [-1] \* n

available\_colors = [True] \* k

adj\_list = {i: [] for i in range(n)}

for u, v in edges:

adj\_list[u].append(v)

adj\_list[v].append(u)

def assign\_color(vertex):

for neighbor in adj\_list[vertex]:

if color[neighbor] != -1:

available\_colors[color[neighbor]] = False

for clr in range(k):

if available\_colors[clr]:

color[vertex] = clr

break

for neighbor in adj\_list[vertex]:

if color[neighbor] != -1:

available\_colors[color[neighbor]] = True

your\_turn = True

your\_count = 0

for vertex in range(n):

if your\_turn:

assign\_color(vertex)

your\_count += 1

else:

assign\_color(vertex)

your\_turn = not your\_turn

return your\_count

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

n = 4

k = 3

result = color\_graph(edges, n, k)

print("Maximum number of regions you can color:", result)

1. 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

**CODE:**

from collections import defaultdict

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

def is\_safe(v, pos, path):

if v not in graph[path[pos - 1]]:

return False

if v in path:

return False

return True

def hamiltonian\_cycle\_util(path, pos):

if pos == n:

if path[pos - 1] in graph[path[0]]:

return True

else:

return False

for v in range(1, n):

if is\_safe(v, pos, path):

path[pos] = v

if hamiltonian\_cycle\_util(path, pos + 1):

return True

path[pos] = -1

return False

path = [-1] \* n

path[0] = 0

if not hamiltonian\_cycle\_util(path, 1):

return False

else:

return True

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

n = 5

graph = defaultdict(list)

for u, v in edges:

graph[u].append(v)

graph[v].append(u)

result = is\_hamiltonian\_cycle(graph, n)

print(f"Does the graph contain a Hamiltonian cycle? {result}")

1. 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

CODE:

from collections import defaultdict

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

def is\_safe(v, pos, path):

if v not in graph[path[pos - 1]]:

return False

if v in path:

return False

return True

def hamiltonian\_cycle\_util(path, pos):

if pos == n:

if path[pos - 1] in graph[path[0]]:

return True

else:

return False

for v in range(1, n):

if is\_safe(v, pos, path):

path[pos] = v

if hamiltonian\_cycle\_util(path, pos + 1):

return True

path[pos] = -1

return False

path = [-1] \* n

path[0] = 0

if not hamiltonian\_cycle\_util(path, 1):

return False

else:

return True

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

n = 4

graph = defaultdict(list)

for u, v in edges:

graph[u].append(v)

graph[v].append(u)

result = is\_hamiltonian\_cycle(graph, n)

print(f"Does the graph contain a Hamiltonian cycle? {result}")

1. 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]

**CODE:**

def generate\_subsets(S):

def backtrack(start, path):

result.append(path[:])

for i in range(start, len(S)):

if i > start and S[i] == S[i - 1]:

continue

path.append(S[i])

backtrack(i + 1, path)

path.pop()

S.sort()

result = []

backtrack(0, [])

return result

A = [1, 2, 3]

subsets = generate\_subsets(A)

print("The subsets are:")

for subset in subsets:

print(subset)

1. 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]]

**CODE:**

from typing import List

def subsets\_with\_element(nums: List[int], x: int) -> List[List[int]]:

def backtrack(start: int, path: List[int]):

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 = []

backtrack(0, [])

return result

def all\_subsets(nums: List[int]) -> List[List[int]]:

def backtrack(start: int, path: List[int]):

result.append(path[:])

for i in range(start, len(nums)):

path.append(nums[i])

backtrack(i + 1, path)

path.pop()

result = []

backtrack(0, [])

return result

E = [2, 3, 4, 5]

x = 3

subsets\_with\_3 = subsets\_with\_element(E, x)

print("The subsets containing 3 are:")

for subset in subsets\_with\_3:

print(subset)

nums = [1, 2, 3]

all\_subsets\_result = all\_subsets(nums)

print("\nAll possible subsets are:")

for subset in all\_subsets\_result:

print(subset)

1. 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"]

**CODE:**

from collections import Counter

from typing import List

def wordSubsets(words1: List[str], words2: List[str]) -> List[str]:

max\_count = Counter()

result = []

for word in words1:

word\_count = Counter(word)

if all(word\_count[char] >= max\_count[char] for char in max\_count):

result.append(word)

return result

words1\_1 = ["amazon", "apple", "facebook", "google", "leetcode"]

words2\_1 = ["e", "o"]

output\_1 = wordSubsets(words1\_1, words2\_1)

print("Output for example 1:", output\_1)