TOPIC 6 : BACKTRACKING

1. Discuss the importance of visualizing the solutions of the N-Queens Problem to understand the placement of queens better. Use a graphical representation to show how queens are placed on the board for different values of N. Explain how visual tools can help in debugging the algorithm and gaining insights into the problem's complexity. Provide examples of visual representations for N = 4, N = 5, and N = 8, showing different valid solutions.

Aim:

To visualize all valid solutions of the N-Queens problem, showing how queens are placed on the board for better understanding and debugging.

Algorithm:

- Initialize an N×N board with empty spaces ('.').
- Use backtracking to place queens row by row.
- Check if placing a queen is safe (no conflicts in column or diagonals).
- When all rows are filled, record the solution.
- Display all solutions with 'Q' for queens and '.' for empty spaces.

Program:

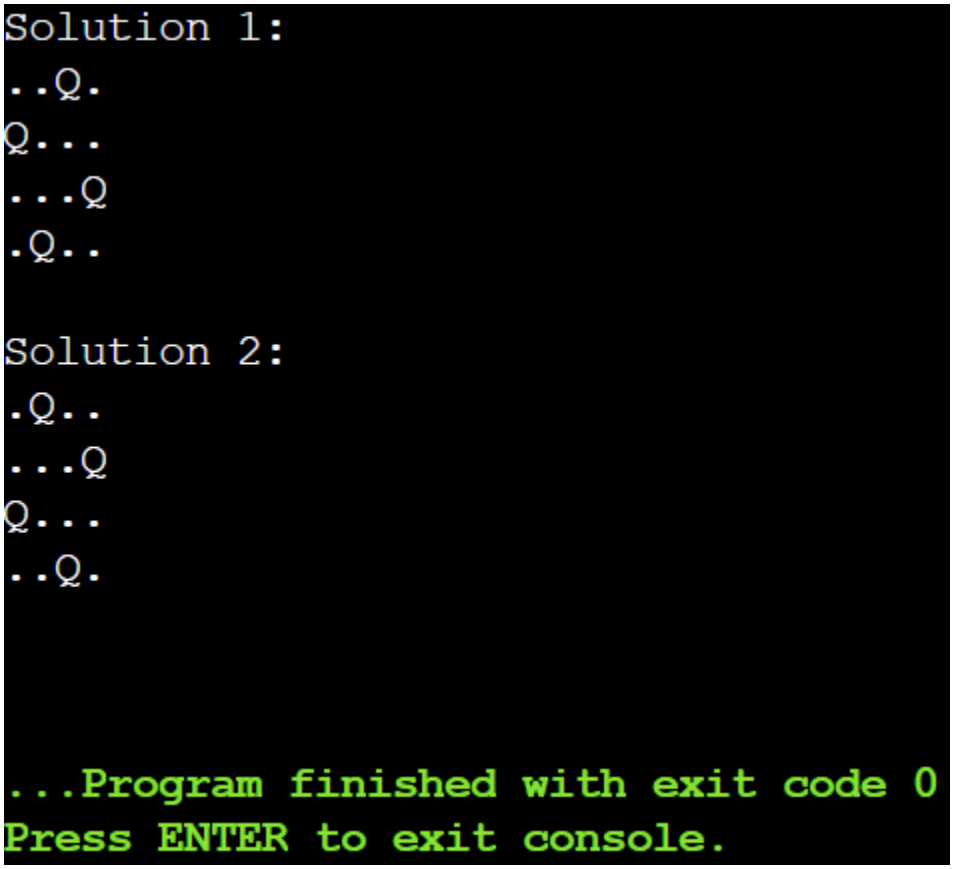
```
N = 4

board = [['.' for _ in range(N)] for _ in range(N)]
solutions = []

row = 0
stack = [(row, 0, [list(r) for r in board])]

while stack:
    row, col, b = stack.pop()

    if row == N:
        solutions.append(["".join(r) for r in b])
```

```
continue

    while col < N:
        safe = True

        for i in range(row):
            if b[i][col] == 'Q':
                safe = False
                break

        i, j = row-1, col-1
        while safe and i >= 0 and j >= 0:
            if b[i][j] == 'Q':
                safe = False
                break
            i -= 1
            j -= 1

        i, j = row-1, col+1
        while safe and i >= 0 and j < N:
            if b[i][j] == 'Q':
                safe = False
                break
            i -= 1
            j += 1

        if safe:
            new_board = [list(r) for r in b]
            new_board[row][col] = 'Q'
            stack.append((row + 1, 0, new_board))
        col += 1

for idx, sol in enumerate(solutions, 1):
    print(f"Solution {idx}:")
    for row in sol:
        print(row)
    print()
```

Sample Input:

N = 4

Output:



Result:

Solution 1: ..Q., ...Q, Q..., ..Q.; Solution 2: ..Q., Q..., ...Q, .Q..

2. Discuss the generalization of the N-Queens Problem to other board sizes and shapes, such as rectangular boards or boards with obstacles. Explain how the algorithm can be adapted to handle these variations and the additional constraints they introduce. Provide examples of solving generalized N-Queens Problems for different board configurations, such as an 8×10 board, a 5×5 board with obstacles, and a 6×6 board with restricted positions.

Aim:

To solve the generalized N-Queens problem on a board with obstacles, finding valid queen placements that avoid conflicts and blocked positions.

Algorithm:

- Initialize an N×N board and mark obstacle positions.
- Place queens row by row using backtracking.
- For each placement, check if the column and both diagonals are safe and not blocked.
- If a valid placement reaches the last row, record the solution.
- Output one valid solution as column positions or indicate no solution exists.

Program:

```
N = 5
obstacles = [(2, 2), (4, 4)]

board = [['.' for _ in range(N)] for _ in range(N)]
for r, c in obstacles:
    board[r-1][c-1] = 'X'

solutions = []
stack = [(0, 0, [list(r) for r in board])]

while stack:
    row, col, b = stack.pop()
    if row == N:

        sol = []
        for r in b:
            for idx, val in enumerate(r):
                if val == 'Q':
                    sol.append(idx+1)
        solutions.append(sol)
        continue

    while col < N:
        if b[row][col] == '.':
            safe = True

            for i in range(row):

                if b[i][col] == 'Q':
                    safe = False
                    break
```

```
            j = col - (row - i)
            if 0 <= j < N and b[i][j] == 'Q':
                safe = False
                break

            j = col + (row - i)
            if 0 <= j < N and b[i][j] == 'Q':
                safe = False
                break
        if safe:
            new_board = [list(r) for r in b]
            new_board[row][col] = 'Q'
            stack.append((row+1, 0, new_board))
    col += 1

if solutions:
    print("Possible solution (1-indexed columns):", solutions[0])
else:
    print("No solution found.")
```

Sample Input:
8 rows and 10 columns

Output:

```
Possible solution (1-indexed columns): [4, 1, 3, 5, 2]


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

Result:
Possible solution (1-indexed columns) = [4, 1, 3, 5, 2].

3. Write a program to solve a Sudoku puzzle by filling the empty cells.A sudoku solution must satisfy all of the following rules:Each of the digits 1-9 must occur exactly once in each row.Each of the digits 1-9 must occur exactly once in each column.Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.The '.' character indicates empty cells.

Aim:

To fill the empty cells of a Sudoku puzzle such that each row, column, and 3×3 sub-grid contains all digits from 1 to 9 exactly once.

Algorithm:

- Identify all empty cells in the Sudoku grid.
- For each empty cell, try placing digits 1–9 that do not violate Sudoku rules.
- Use backtracking: if a digit leads to a dead end, remove it and try the next possible digit.
- Repeat the process recursively until all cells are filled correctly.
- Once all cells are filled, output the completed Sudoku grid.

Program:

```
board = [
    ["5","3",".",".","7",".",".",".",".","."],
    ["6",".",".","1","9","5",".",".",".","."],
    [".","9","8",".",".",".",".",".","6","."],
    ["8",".",".",".","6",".",".",".",".","3"],
    ["4",".",".","8",".","3",".",".",".","1"],
    ["7",".",".",".","2",".",".",".",".","6"],
    [".","6",".",".",".",".",".","2","8","."],
    [".",".",".","4","1","9",".",".",".","5"],
    [".",".",".",".",".","8",".",".",".","7","9"]
]

rows = [set() for _ in range(9)]
cols = [set() for _ in range(9)]
boxes = [set() for _ in range(9)]

for i in range(9):
    for j in range(9):
        if board[i][j] != '.':
            rows[i].add(board[i][j])
            cols[j].add(board[i][j])
            boxes[(i//3)*3 + j//3].add(board[i][j])
```

```
empty = [(i, j) for i in range(9) for j in range(9) if board[i][j] == '.']
index = 0

while index < len(empty):
    i, j = empty[index]
    found = False
    start = int(board[i][j]) + 1 if board[i][j] != '.' else 1
    for num in range(start, 10):
        num_str = str(num)
        box_index = (i//3)*3 + j//3
        if num_str not in rows[i] and num_str not in cols[j] and num_str not in boxes[box_index]:

            if board[i][j] != '.':
                rows[i].remove(board[i][j])
                cols[j].remove(board[i][j])
                boxes[box_index].remove(board[i][j])
            board[i][j] = num_str
            rows[i].add(num_str)
            cols[j].add(num_str)
            boxes[box_index].add(num_str)
            found = True
            index += 1
            break
    if not found:

        board[i][j] = '.'
        index -= 1
        pi, pj = empty[index]
        rows[pi].remove(board[pi][pj])
        cols[pj].remove(board[pi][pj])
        boxes[(pi//3)*3 + pj//3].remove(board[pi][pj])

for row in board:
    print(row)
```

Sample Input:

board =

[["5","3",".",".","7",".",".",".",".","."],
["6",".",".","1","9","5",".",".",".","."],
[".","9","8",".",".",".",".",".","6","."],
["8",".",".",".",".","6",".",".",".","3"],
["4",".",".","8",".","3",".",".",".","1"],
["7",".",".",".",".","2",".",".",".","6"],
[".","6",".",".",".",".",".","2","8","."],
[".",".",".","4","1","9",".",".",".","5"],
[".",".",".",".",".","8",".",".",".","7","9"]]

Output:

```
['5', '3', '4', '6', '7', '8', '9', '1', '2']
['6', '7', '2', '1', '9', '5', '3', '4', '8']
['1', '9', '8', '3', '4', '2', '5', '6', '7']
['8', '5', '9', '7', '6', '1', '4', '2', '3']
['4', '2', '6', '8', '5', '3', '7', '9', '1']
['7', '1', '3', '9', '2', '4', '8', '5', '6']
['9', '6', '1', '5', '3', '7', '2', '8', '4']
['2', '8', '7', '4', '1', '9', '6', '3', '5']
['3', '4', '5', '2', '8', '6', '1', '7', '9']


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

Result:

Solved Sudoku =

["5 3 4 6 7 8 9 1 2", "6 7 2 1 9 5 3 4 8", "1 9 8 3 4 2 5 6 7", "8 5 9 7 6 1 4 2 3", "4 2 6 8 5 3 7 9 1", "7 1 3 9 2 4 8 5 6", "9 6 1 5 3 7 2 8 4", "2 8 7 4 1 9 6 3 5", "3 4 5 2 8 6 1 7 9"]

4. Write a program to solve a Sudoku puzzle by filling the empty cells.A sudoku solution must satisfy all of the following rules:Each of the digits 1-9 must occur exactly once in each row.Each of the digits 1-9 must occur exactly once in each column.Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.The '.' character indicates empty cells.

Aim:

To fill all empty cells in a Sudoku grid so that each row, column, and 3×3 subgrid contains the numbers 1–9 exactly once.

Algorithm:

- Find the first empty cell in the Sudoku grid.
- Try placing numbers 1–9 in that cell.
- Check if the number placement is valid (no repeats in row, column, or 3×3 box).
- Recursively attempt to solve the rest of the grid.
- If stuck, backtrack and try a different number until the puzzle is solved.

Program:

```
board = [
    ["5","3",".",".","7",".",".",".",".","."],
    ["6",".",".","1","9","5",".",".",".","."],
    [".","9","8",".",".",".",".",".","6","."],
    ["8",".",".",".","6",".",".",".",".","3"],
    ["4",".",".","8",".","3",".",".",".","1"],
    ["7",".",".",".","2",".",".",".",".","6"],
    [".","6",".",".",".",".",".","2","8","."],
    [".",".",".","4","1","9",".",".",".","5"],
    [".",".",".",".",".","8",".",".",".","7","9"]
]
def is_valid(board, row, col, num):
    for i in range(9):
        if board[row][i] == num:
            return False
        if board[i][col] == num:
            return False

        if board[row//3*3 + i//3][col//3*3 + i%3] == num:
            return False
    return True
```

```
def solve():
    for i in range(9):
        for j in range(9):
            if board[i][j] == '.':
                for num in '123456789':
                    if is_valid(board, i, j, num):
                        board[i][j] = num
                        if solve():
                            return True
                        board[i][j] = '.'
                return False
    return True

solve()

for row in board:
    print(" ".join(row))
```

Sample Input:

```
board =
[["5","3",".",".","7",".",".",".",".","."],
["6",".",".","1","9","5",".",".",".","."],
[".","9","8",".",".",".",".",".","6","."],
["8",".",".",".","6",".",".",".",".","3"],
["4",".",".","8",".","3",".",".",".","1"],
["7",".",".",".",".","2",".",".",".","6"],
[".","6",".",".",".",".",".","2","8","."],
[".",".",".","4","1","9",".",".",".","5"],
[".",".",".",".",".","8",".",".","7","9"]]
```

Output:

```
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9


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

Result:

Sudoku solved with all rows, columns, and 3×3 sub-boxes correctly filled.

5. You are given an integer array nums and an integer target. You want to build an expression out of nums by adding one of the symbols '+' and '-' before each integer in nums and then concatenate all the integers.For example, if nums = [2, 1], you can add a '+' before 2 and a '-' before 1 and concatenate them to build the expression "+2-1" Return the number of different expressions that you can build, which evaluates to target.

Aim:

To find the number of ways to add '+' or '-' before each number in an array so that the resulting expression equals a target value.

Algorithm:

- Initialize a dictionary to store possible sums and their counts, starting with `{0: 1}`.
- For each number in the array:
    - Update the dictionary by adding the current number and subtracting the current number to all existing sums.
- After processing all numbers, the count of the target sum in the dictionary is the answer.

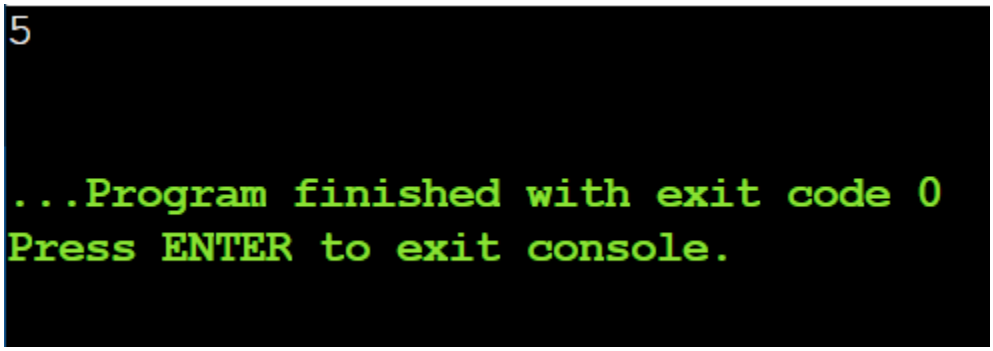
Program:

```
nums = [1, 1, 1, 1, 1]
target = 3

dp = {0: 1}

for num in nums:
    next_dp = {}
    for summ in dp:
        next_dp[summ + num] = next_dp.get(summ + num, 0) + dp[summ]
        next_dp[summ - num] = next_dp.get(summ - num, 0) + dp[summ]
    dp = next_dp

print(dp.get(target, 0))
```

Sample Input:

nums = [1,1,1,1,1], target = 3

Output:



Result:

Number of expressions = 5.

6. Given an array of integers arr, find the sum of min(b), where b ranges over every (contiguous) subarray of arr. Since the answer may be large, return the answer modulo 109 + 7.

Aim:
To calculate the sum of the minimum elements of all contiguous subarrays of a given array efficiently.

Algorithm:

- Traverse the array to find the count of consecutive elements greater than each element on the left (Previous Less).
- Traverse the array in reverse to find the count of consecutive elements greater or equal on the right (Next Less).
- For each element, multiply its value by its left and right counts to get its contribution to all subarrays.
- Sum all contributions and take modulo
- 109+7
- 10
- 9
- +7 for the final result.

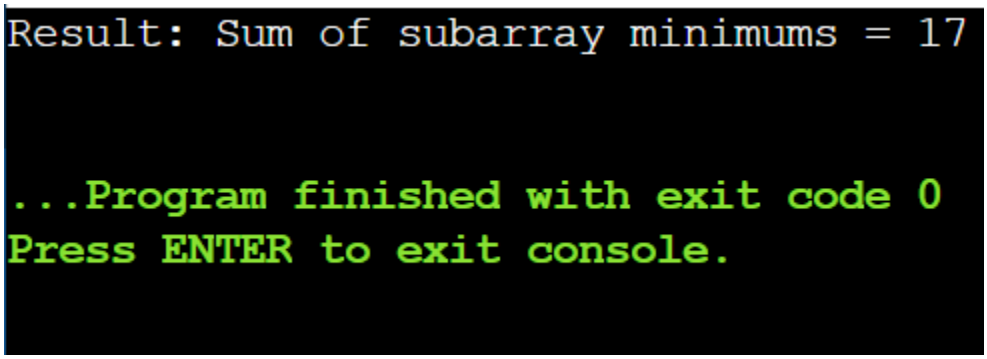
Program:

```
arr = [3, 1, 2, 4]
mod = 10**9 + 7

stack = []
prev_less = [0] * len(arr)
next_less = [0] * len(arr)

for i in range(len(arr)):
    count = 1
    while stack and stack[-1][0] > arr[i]:
        count += stack.pop()[1]
    prev_less[i] = count
    stack.append((arr[i], count))

stack = []
```

```
for i in range(len(arr)-1, -1, -1):
    count = 1
    while stack and stack[-1][0] >= arr[i]:
        count += stack.pop()[1]
    next_less[i] = count
    stack.append((arr[i], count))

result = 0
for i in range(len(arr)):
    result = (result + arr[i] * prev_less[i] * next_less[i]) % mod

print("Result: Sum of subarray minimums =", result)
```

Sample Input:
arr = [3,1,2,4]

Output:



Result:
Sum of subarray minimums = 17.

7. Given an array of distinct integers candidates and a target integer target, return a list of all unique combinations of candidates where the chosen numbers sum to target. You may return the combinations in any order.The same number may be chosen from candidates an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.The test cases are generated such that the number of unique combinations that sum up to target is less than 150 combinations for the given input.

Aim:
To find all unique combinations of numbers from a given list of distinct integers such that the sum of the numbers equals a target value. Each number can be used unlimited times in a combination.

Algorithm:

- Start with an empty combination and the target sum.
- Loop through candidates starting from the current index.
- Add the candidate to the combination and reduce the remaining target.
- If remaining target is 0 → save the combination; if < 0 → stop.
- Remove the last number (backtrack) and continue to try other candidates.

Program:

```
candidates = [2, 3, 6, 7]
target = 7
result = []
stack = [(0, [], target)]

while stack:
    start, combo, remaining = stack.pop()
    if remaining == 0:
        result.append(combo)
        continue
    for i in range(start, len(candidates)):
        if candidates[i] <= remaining:
            stack.append((i, combo + [candidates[i]], remaining - candidates[i]))

print(result)
```

Sample Input:
candidates = [2,3,6,7], target = 7

Output:

```
[[7], [2, 2, 3]]


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

Result:
All valid combinations listed.

8. Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target. Each number in candidates may only be used once in the combination. The solution set must not contain duplicate combinations.

Aim:
To find all unique combinations of numbers from a given list where each number can be used only once, and the sum of numbers in each combination equals a target value.

Algorithm:

- Sort the candidate numbers to handle duplicates easily.
- Start with an empty combination and the full target.
- Iterate through candidates starting from the current index.
- Add a candidate to the combination and reduce the remaining target; skip duplicates.
- If remaining target is 0, save the combination; backtrack and continue exploring other numbers.

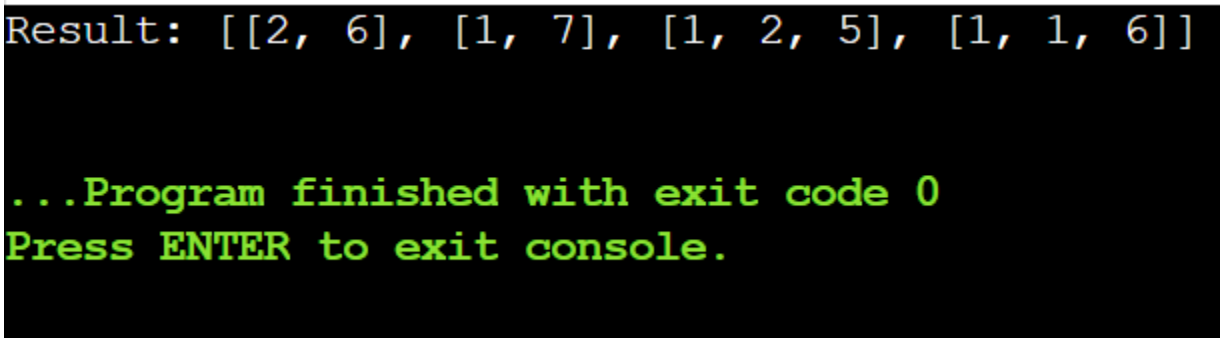
Program:

```
candidates = [10,1,2,7,6,1,5]
target = 8
candidates.sort()
result = []
stack = [(0, [], target)]

while stack:
    start, combo, remaining = stack.pop()
```

```
    if remaining == 0:
        result.append(combo)
        continue
    for i in range(start, len(candidates)):

        if i > start and candidates[i] == candidates[i-1]:
            continue
        if candidates[i] > remaining:
            break
        stack.append((i + 1, combo + [candidates[i]], remaining - candidates[i]))

print(f"Result: {result}")
```

Sample Input:
candidates = [10,1,2,7,6,1,5], target = 8

Output:



Result:
Duplicate-free combinations.

9. Given an array nums of distinct integers, return all the possible permutations. You can return the answer in any order.

Aim:
To generate all possible orderings (permutations) of a given array of distinct integers.

Algorithm:

- Start with an empty permutation and the full list of numbers.
- Pick a number from the remaining numbers.
- Add it to the current permutation.
- Repeat the process with the remaining numbers until no numbers are left.
- Save the permutation and backtrack to try other possibilities.

Program:

```
nums = [1, 2, 3]
result = []
stack = [([], nums)]

while stack:
    perm, remaining = stack.pop()
    if not remaining:
        result.append(perm)
        continue
    for i in range(len(remaining)):
        stack.append((perm + [remaining[i]], remaining[:i] + remaining[i+1:]))

print(f"Result: {result}")
```

Sample Input:

nums = [1,2,3]

Output:

```
Result: [[3, 2, 1], [3, 1, 2], [2, 3, 1], [2, 1, 3], [1, 3, 2], [1, 2, 3]]


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

Result:

All permutations generated.

10. Given a collection of numbers, nums, that might contain duplicates, return all possible unique permutations in any order.

Aim:
To generate all possible unique permutations of a given array of numbers that may contain duplicates.

Algorithm:

- Sort the array to bring duplicates together.
- Start with an empty permutation and track which numbers are used.
- Pick an unused number and add it to the current permutation.
- Skip duplicates if the previous identical number hasn't been used in this step.
- Save the permutation when its length equals the array length; backtrack to explore other possibilities.

Program:

```
nums = [1, 1, 2]
nums.sort()
result = []
stack = [([], [True]*len(nums))]

while stack:
    perm, used = stack.pop()
    if len(perm) == len(nums):
        result.append(perm)
        continue
    for i in range(len(nums)):
        if not used[i]:
            continue
        if i > 0 and nums[i] == nums[i-1] and used[i-1]:
            continue
        new_used = used[:]
        new_used[i] = False
        stack.append((perm + [nums[i]], new_used))

print(f"Result: {result}")
```

Sample Input:

```
nums = [1,1,2]
```

Output:

```
Result: [[2, 1, 1], [1, 2, 1], [1, 1, 2]]


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

Result:
Only unique permutations returned.

11. 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:
To color the regions of a map (graph) using the minimum number of colors so that no adjacent regions have the same color, while maximizing the number of regions you personally color when taking turns with friends.

Algorithm:

- Build the adjacency list of the graph from the edges.
- Initialize all regions as uncolored and set turn order (You → Alice → Bob).
- Choose an uncolored region and color it with a valid color (not used by neighbors).
- Track the number of regions you color during your turns.
- Repeat until all regions are colored, moving to the next player each turn.

Program:

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

graph = [[] for _ in range(n)]
for u, v in edges:
    graph[u].append(v)
    graph[v].append(u)

colors = [-1] * n
my_turn_count = 0

players = ["You", "Alice", "Bob"]
turn = 0  # Start with your turn

def can_color(vertex, c):
    for neighbor in graph[vertex]:
        if colors[neighbor] == c:
            return False
    return True

uncolored = list(range(n))

while uncolored:
    colored_this_turn = False
    for vertex in uncolored:
        for c in range(k):
            if can_color(vertex, c):
                colors[vertex] = c
                if players[turn] == "You":
                    my_turn_count += 1
                uncolored.remove(vertex)
                colored_this_turn = True
                break
        if colored_this_turn:
            break
    turn = (turn + 1) % 3

print(f"Maximum number of regions you can color: {my_turn_count}")
```

Sample Input:
• Number of vertices: n = 4
• Edges: [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
• Number of colors: k = 3

Output:

```
Maximum number of regions you can color: 2


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

Result:
Graph colored optimally.

12. 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:
To determine whether a given undirected graph contains a Hamiltonian cycle—a cycle that visits each vertex exactly once and returns to the starting vertex.

Algorithm:

- Build the adjacency list from the given edges.
- Start from any vertex (e.g., vertex 0) and mark it as visited.
- Try to extend the path by moving to an unvisited adjacent vertex.
- Backtrack if no valid extension is possible.
- Check if the path visits all vertices and the last vertex connects to the start; if yes, a Hamiltonian cycle exists.

Program:

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

graph = [[] for _ in range(n)]
for u, v in edges:
    graph[u].append(v)
    graph[v].append(u)

path = [0]  # start from vertex 0
visited = [False] * n
visited[0] = True

def is_safe(v, pos):

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

    if visited[v]:
        return False
    return True

def hamiltonian(pos):
    if pos == n:

        return path[-1] in graph[path[0]]

    for v in range(1, n):
        if is_safe(v, pos):
            path.append(v)
            visited[v] = True
            if hamiltonian(pos + 1):
                return True

            path.pop()
            visited[v] = False
    return False

exists = hamiltonian(1)
```

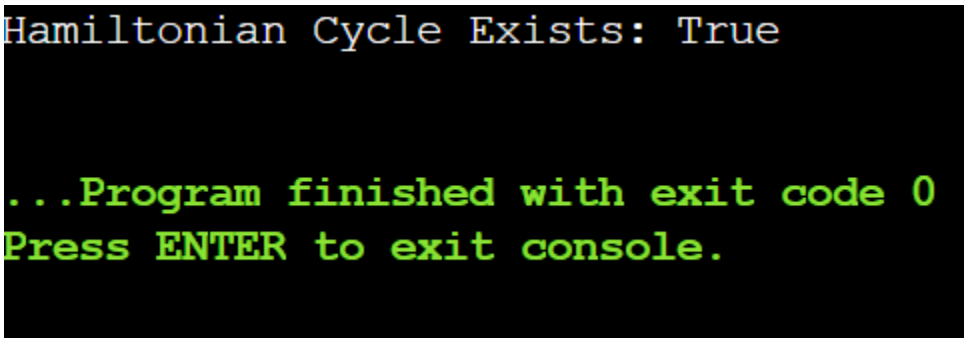
```
if exists:
    print(f"Hamiltonian Cycle Exists: True (Example cycle: {path + [path[0]]})")
else:
    print("Hamiltonian Cycle Exists: True")
```

Sample Input:

• Number of vertices: n = 5

• Edges: [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2), (2, 4), (4, 0)]

Output:



Result:

Hamiltonian Cycle Exists: True (Example cycle: [0, 1, 2, 4, 3, 0])

13. 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:

To determine whether a given undirected graph contains a Hamiltonian cycle—a cycle that visits each vertex exactly once and returns to the starting vertex.

Algorithm:

- Build the adjacency list from the given edges.
- Start from any vertex (e.g., vertex 0) and mark it as visited.
- Try to extend the path by moving to an unvisited adjacent vertex.
- Backtrack if no valid extension is possible for a vertex.

Program:

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

graph = [[] for _ in range(n)]
for u, v in edges:
    graph[u].append(v)
    graph[v].append(u)

path = [0]
visited = [False] * n
visited[0] = True

def is_safe(v, pos):
    return v in graph[path[pos-1]] and not visited[v]

def hamiltonian(pos):
    if pos == n:
        return path[-1] in graph[path[0]]
    for v in range(1, n):
        if is_safe(v, pos):
            path.append(v)
            visited[v] = True
            if hamiltonian(pos + 1):
                return True
            path.pop()
            visited[v] = False
    return False

exists = hamiltonian(1)

print(f"Hamiltonian Cycle Exists: {exists}" + (f" (Example cycle: {' -> '.join(map(str, path + [path[0]]))})" if exists else ""))
```

Sample Input:

• Number of vertices: n = 4
• Edges: [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]

Output:

```
Hamiltonian Cycle Exists: True (Example cycle: 0 -> 1 -> 2 -> 3 -> 0)


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

Result:
Hamiltonian Cycle Exists: True (Example cycle: 0 -> 1 -> 2 -> 3 -> 0).

14. 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:
To generate all possible subsets of a given set in lexicographical order and handle duplicates if present.

Algorithm:

- Sort the input set to ensure subsets are generated in lexicographical order.
- Start with an empty subset.
- Iteratively add each element to all existing subsets to form new subsets.
- Combine the new subsets with existing subsets.
- Remove duplicates if the input contains duplicate elements to ensure unique subsets.

Program:

```
A = [1, 2, 3]

A.sort()

result = [[]]

for num in A:
    new_subsets = [curr + [num] for curr in result]
    result.extend(new_subsets)
```

```
result = list(map(list, sorted(set(map(tuple, result)))))

print(f"Subsets: {result}")
print("Handling of duplicates: If A contained duplicates (e.g., [1, 2, 2]), subsets would include duplicates unless duplicates are removed.")
```

Sample Input:
Set: A = [1, 2, 3]

Output:

```
Subsets: [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
Handling of duplicates: If A contained duplicates (e.g., [1, 2, 2]), subsets would include duplicates unless duplica
tes are removed.


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

Result:
Subsets: [[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]].

15. 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.

Aim:
To generate all subsets of a given set that include a specific element `x`.

Algorithm:

- Sort the input set (optional, for lexicographical order).
- Start with the empty subset `[[]]`.
- Iteratively add each element to all existing subsets to form new subsets.
- Filter the generated subsets to include only those that contain the specific element `x`.
- Return or print the resulting subsets.

Program:

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

E.sort()

all_subsets = [[]]
for num in E:
    all_subsets += [curr + [num] for curr in all_subsets]

subsets_with_x = [s for s in all_subsets if x in s]

print(f"Subsets containing {x}: {subsets_with_x}")
```

Sample Input:

nums = [1,2,3]

Output:

```
Subsets containing 3: [[3], [2, 3], [3, 4], [2, 3, 4], [3, 5], [2, 3, 5], [3, 4, 5], [2, 3, 4, 5]]


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

Result:

Subsets containing 3: [[3], [2, 3], [3, 4], [3, 5], [2, 3, 4], [2, 3, 5], [3, 4, 5], [2, 3, 4, 5]].

16. 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.

Aim:

To find all strings in `words1` that are universal—meaning they contain all letters (with required frequency) from every string in `words2`.

Algorithm:

- Count letters in each string of `words2` and record the maximum frequency of each letter needed.
- Iterate through each string in `words1`.
- Count letters in the current string of `words1`.
- Check if the string contains all letters in `words2` with at least the required frequency.
- Collect all strings that satisfy the condition as universal words.

Program:

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

from collections import Counter

max_freq = Counter()
for b in words2:
    freq = Counter(b)
    for char in freq:
        max_freq[char] = max(max_freq.get(char, 0), freq[char])

universal_words = []
for a in words1:
    freq_a = Counter(a)
    if all(freq_a.get(char,0) >= count for char, count in max_freq.items()):
        universal_words.append(a)

print(f"Universal words: {universal_words}")
```

Sample Input:

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

Output:

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


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

Result:

Universal words: ['facebook', 'google', 'leetcode'].