NAME : TEJASWINI M

REG NO. : 192411079

COURSE CODE : CSA0613

COURSE NAME : DESIGN AND ANALYSIS OF ALGORITHMS FOR

OPTIMAL APPLICATIONS

SLOT : A

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 the solutions of the N-Queens problem to better understand queen placements on the board and gain insights into the problem's complexity.

ALGORITHM:

1. Start with an empty N×N board.
2. Place a queen in the first row, one column at a time.
3. Check if the placement is **safe**:
4. No queen in the same column
5. No queen on left or right diagonals
6. If safe, move to the next row.
7. If no safe column exists, **backtrack** and try another position.
8. Continue until all queens are placed.
9. Visualize the board for each valid solution.

CODE:

```python
def is_safe(board, row, col, n):
    # Check column
    for i in range(row):
        if board[i] == col:
            return False
    # Check left diagonal
    i, j = row - 1, col - 1
    while i >= 0 and j >= 0:
        if board[i] == j:
            return False
        i -= 1
        j -= 1
    # Check right diagonal
    i, j = row - 1, col + 1
    while i >= 0 and j < n:
        if board[i] == j:
            return False
        i -= 1
        j += 1
    return True
def solve_nqueens(board, row, n, solutions):
    if row == n:
        solutions.append(board.copy())
        return
    for col in range(n):
        if is_safe(board, row, col, n):
            board[row] = col
            solve_nqueens(board, row + 1, n, solutions)
            board[row] = -1  # backtrack
def print_board(solution, n):
    for i in range(n):
        for j in range(n):
            if solution[i] == j:
                print("Q", end=" ")
            else:
                print(".", end=" ")
        print()
def visualize_nqueens(n):
    board = [-1] * n
    solutions = []
    solve_nqueens(board, 0, n, solutions)
    print(f"Total solutions for N = {n}: {len(solutions)}\n")
    # Print only first 2 solutions for clarity
    for idx, sol in enumerate(solutions[:2]):
        print(f"Solution {idx + 1}:")
        print_board(sol, n)
```

INPUT:

```python
# Visualize for N = 4, 5, and 8
visualize_nqueens(4)
visualize_nqueens(5)
visualize_nqueens(8)
```

OUTPUT:

Total solutions for N = 4: 2

Total solutions for N = 5: 10

Total solutions for N = 8: 92

Solution 1:

```
. Q . .
. . . Q
Q . . .
. . Q .
```

Solution 1:

```
Q . . . .
. . Q . .
. . . . Q
. Q . . .
. . . Q .
```

Solution 1:

```
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
```

Solution 2:

```
. . Q .
Q . . .
. . . Q
. Q . .
```

Solution 2:

```
Q . . . .
. . . Q .
. Q . . .
. . . . Q
. . Q . .
```

Solution 2:

```
Q . . . . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
. . . . . . Q .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .
```

2

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 generalize the N-Queens Problem to boards of different sizes and shapes, including rectangular boards, boards with obstacles, and boards with restricted positions. The goal is to adapt the backtracking algorithm to handle additional constraints while ensuring no two queens threaten each other.

ALGORITHM:

1. **Board representation:** Represent obstacles or restricted positions using a matrix or a set of invalid coordinates.

2. **Backtracking constraints:** Before placing a queen:
   o Check if the current column is free.
   o Check if diagonals are safe.
   o Ensure the current cell is not an obstacle or restricted.

3. **Flexible column selection:** On rectangular boards, allow placement in any valid column for each row.

4. **Output:** Return one valid arrangement (or all possible arrangements) that satisfies the constraints.

CODE:

```python
def is_safe(board, row, col, placed):
    for r, c in placed:
        if c == col or abs(r - row) == abs(c - col):
            return False
    return True
def solve_general_nqueens(rows, cols, obstacles=set(), restricted=set()):
    solutions = []
    def backtrack(row, placed):
        if row == rows:
            solutions.append(placed.copy())
            return
        for col in range(cols):
            if (row, col) in obstacles or (row, col) in restricted:
                continue
            if is_safe(None, row, col, placed):
                placed.append((row, col))
                backtrack(row + 1, placed)
                placed.pop()
    backtrack(0, [])
    return solutions
```

```python
def print_board(rows, cols, solution, obstacles=set(), restricted=set()):
    for r in range(rows):
        for c in range(cols):
            if (r, c) in obstacles:
                print("X", end=" ")
            elif (r, c) in restricted:
                print("R", end=" ")
            elif (r, c) in solution:
                print("Q", end=" ")
            else:
                print(".", end=" ")
        print()
    print()
```

INPUT:

```python
solutions = solve_general_nqueens(8, 10)

print("One solution for 8x10 board:")

print_board(8, 10, solutions[0])

obstacles = {(0,2), (2,1), (4,3)}

solutions = solve_general_nqueens(5, 5, obstacles=obstacles)

print("5x5 board with obstacles:")

print_board(5, 5, solutions[0], obstacles=obstacles)

restricted = {(1,3), (4,0)}

solutions = solve_general_nqueens(6, 6, restricted=restricted)

print("6x6 board with restricted positions:")

print_board(6, 6, solutions[0], restricted=restricted)
```

OUTPUT:

```
One solution for 8x10 board:
Q . . . . . . . . .
. . Q . . . . . . .
. . . . Q . . . . .
. Q . . . . . . . .
. . . . . . . Q . .
. . . . . . . . . Q
. . . Q . . . . . .
. . . . . . Q . . .

5x5 board with obstacles:
. Q X . .
. . . Q .
Q X . . .
. . Q . .
. . . X Q

6x6 board with restricted positions:
. . . Q . .
Q . . R . .
. . . . Q .
. Q . . . .
R . . . Q
. . Q . . .
```

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 solve a Sudoku puzzle by filling all empty cells while satisfying the rules: each row, each column, and each 3×3 sub-box must contain digits 1–9 exactly once.

ALGORITHM:

1. Iterate through the board to find an empty cell (denoted by .).
2. Try filling numbers 1 through 9 in the empty cell.
3. For each number, check if it is safe to place it by verifying:
   o The number is not already in the same row.
   o The number is not already in the same column.

o   The number is not already in the 3×3 sub-box.

4. If the number is safe, place it and recursively attempt to fill the next empty cell.

5. If no number is valid for the current cell, backtrack by resetting the cell to empty and trying the previous placement again.

6. Continue until the board is fully filled.

CODE:

```python
def is_safe(board, row, col, num):
    # Check row
    for x in range(9):
        if board[row][x] == num:
            return False
    # Check column
    for x in range(9):
        if board[x][col] == num:
            return False
    # Check 3x3 sub-box
    start_row = row - row % 3
    start_col = col - col % 3
    for i in range(3):
        for j in range(3):
            if board[start_row + i][start_col + j] == num:
                return False
    return True
def solve_sudoku(board):
    for row in range(9):
        for col in range(9):
            if board[row][col] == '.':
                for num in map(str, range(1, 10)):
                    if is_safe(board, row, col, num):
                        board[row][col] = num
                        if solve_sudoku(board):
                            return True
                        board[row][col] = '.'  # Backtrack
                return False
    return True
def print_board(board):
    for row in board:
        print(" ".join(row))
```

INPUT:

```python
# Example Sudoku Puzzle
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:

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 solve a Sudoku puzzle by filling all empty cells while ensuring that each row, column, and 3×3 sub-box contains the digits 1–9 exactly once.

ALGORITHM:

1. Traverse the board to find an empty cell (.).
2. Try filling numbers 1 to 9 in the empty cell.
3. For each number, check if placing it is valid:
   - Not present in the same row.
   - Not present in the same column.
   - Not present in the corresponding 3×3 sub-box.
4. If valid, place the number and recursively solve for the next empty cell.
5. If no number works, backtrack by resetting the cell to . and try a previous placement.
6. Repeat until the board is completely filled.

CODE:

```python
def is_safe(board, row, col, num):
    # Check row
    for x in range(9):
        if board[row][x] == num:
            return False
    # Check column
    for x in range(9):
        if board[x][col] == num:
            return False
    # Check 3×3 sub-box
    start_row = row - row % 3
    start_col = col - col % 3
    for i in range(3):
        for j in range(3):
            if board[start_row + i][start_col + j] == num:
                return False
    return True
def solve_sudoku(board):
    for row in range(9):
        for col in range(9):
            if board[row][col] == '.':
                for num in map(str, range(1, 10)):
                    if is_safe(board, row, col, num):
                        board[row][col] = num
                        if solve_sudoku(board):
                            return True
                        board[row][col] = '.'  # backtrack
                return False
    return True
def print_board(board):
    for row in board:
        print(" ".join(row))
```

```
# Example input Sudoku board
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"]
]
```

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
```

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 determine the number of different expressions that can be formed by adding + or - before each element in an array nums such that the sum of the resulting expression equals a given target.

ALGORITHM:

1. Start from the first index of nums.

2. At each index, recursively try two choices:

   o Add the current number (+nums[i]).

   o Subtract the current number (-nums[i]).

3. Keep track of the running sum as you move through the array.

4. When you reach the end of the array, check if the running sum equals target.

   o If yes, count it as one valid expression.

5. Sum up all valid expressions and return the count.

This can also be implemented efficiently using **Dynamic Programming** (memoization) to avoid recalculating subproblems.

CODE:

```python
def findTargetSumWays(nums, target):
    count = 0
    def backtrack(index, current_sum):
        nonlocal count
        # If all numbers are used
        if index == len(nums):
            if current_sum == target:
                count += 1
            return
        # Add '+' before nums[index]
        backtrack(index + 1, current_sum + nums[index])
        # Add '-' before nums[index]
        backtrack(index + 1, current_sum - nums[index])
    backtrack(0, 0)
    return count
```

INPUT:

```python
# Example
nums = [1, 1, 1, 1, 1]
target = 3
```

OUTPUT:

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 $10^9 + 7$.**

AIM:

To compute the sum of the minimum elements of all contiguous subarrays of a given array arr. Since the number of subarrays can be large, the result should be returned modulo $10^9 + 7$.

ALGORITHM:

1. **Observation:** For each element arr[i], determine how many subarrays have arr[i] as the minimum.

2. Use **Next Smaller Element (NSE)** and **Previous Smaller Element (PSE)** approach:

    o **Left[i]**: Distance to the previous element smaller than arr[i] (number of subarrays ending at i where arr[i] is minimum).

8

- **Right[i]**: Distance to the next element smaller than arr[i] (number of subarrays starting at i where arr[i] is minimum).

3. The contribution of arr[i] to the total sum is:

4. arr[i] * Left[i] * Right[i]

5. Sum up contributions of all elements and take modulo $10^9 + 7$.

6. Using monotonic stacks ensures $O(n)$ time complexity.

CODE:

```python
def sumSubarrayMins(arr):
    MOD = 10**9 + 7
    n = len(arr)
    # Arrays to store distances
    left = [0] * n
    right = [0] * n
    stack = []
    # Previous Less Element (PLE)
    for i in range(n):
        count = 1
        while stack and stack[-1][0] > arr[i]:
            count += stack.pop()[1]
        left[i] = count
        stack.append((arr[i], count))
    stack.clear()
    # Next Less Element (NLE)
    for i in range(n - 1, -1, -1):
        count = 1
        while stack and stack[-1][0] >= arr[i]:
            count += stack.pop()[1]
        right[i] = count
        stack.append((arr[i], count))
    # Calculate result
    result = 0
    for i in range(n):
        result = (result + arr[i] * left[i] * right[i]) % MOD
    return result
```

INPUT: **OUTPUT:**

```python
# Example
arr = [3, 1, 2, 4]
print(sumSubarrayMins(arr))
```

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 given numbers that sum up to a target value, where each number can be used an unlimited number of times.

ALGORITHM:

1. Use a backtracking approach to explore all possible combinations.

2. Start from index 0 and try each candidate number.

3. Add the current number to the combination and reduce the remaining target.

4. If the remaining target becomes 0, store the current combination.

5. If the remaining target becomes negative, stop that path.

6. Repeat until all possibilities are explored.

CODE:

```python
def combinationSum(candidates, target):
    result = []
    def backtrack(start, current, total):
        if total == target:
            result.append(current.copy())
            return
        if total > target:
            return
        for i in range(start, len(candidates)):
            current.append(candidates[i])
            backtrack(i, current, total + candidates[i])  # reuse allowed
            current.pop()
    backtrack(0, [], 0)
    return result
```

INPUT:

```python
# Example
candidates = [2, 3, 6, 7]
target = 7
```

OUTPUT:

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

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 from the given array where each number is used only once and the sum of the selected numbers equals the target value, without repeating any combination.

ALGORITHM:

1. Sort the candidates array to handle duplicates easily.

2. Use backtracking to explore all possible combinations.

3. Start from a given index and try each number only once.

4. Skip duplicate numbers to avoid repeated combinations.

5. If the remaining target becomes 0, store the current combination.

6. If the remaining target becomes negative, stop exploring that path.

7. Continue until all valid combinations are found.

CODE:

```python
def combinationSum2(candidates, target):
    candidates.sort()   # sort to handle duplicates
    result = []
    def backtrack(start, current, total):
        if total == target:
            result.append(current.copy())
            return
        if total > target:
            return
        for i in range(start, len(candidates)):
            # Skip duplicates
            if i > start and candidates[i] == candidates[i - 1]:
                continue
            current.append(candidates[i])
            backtrack(i + 1, current, total + candidates[i])  # move forward
            current.pop()
    backtrack(0, [], 0)
    return result
```

INPUT: **OUTPUT:**

```
# Example
candidates = [10,1,2,7,6,1,5]
target = 8
```
                                    [[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]]

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 permutations of a given array of distinct integers. A permutation is a rearrangement of the elements in all possible orders.

ALGORITHM:

1. Use backtracking to generate all permutations.

2. Keep a temporary list (path) to store the current permutation.

3. Use a boolean array (used) to track which elements are already included.

4. If the length of path equals the length of nums, store it as a valid permutation.

5. Try all unused elements, add them to path, and recurse.

6. After recursion, remove the element (backtrack) and mark it as unused.

7. Continue until all permutations are generated.

CODE:

```python
def permute(nums):
    result = []

    def backtrack(current, used):
        if len(current) == len(nums):
            result.append(current.copy())
            return

        for i in range(len(nums)):
            if used[i]:
                continue
            used[i] = True
            current.append(nums[i])
            backtrack(current, used)
            current.pop()
            used[i] = False

    backtrack([], [False] * len(nums))
    return result
```

INPUT:

```
# Example
nums = [1, 2, 3]
print(permute(nums))
```

OUTPUT:

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

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 list of numbers that may contain duplicate values.

ALGORITHM:

1. Sort the input array to group duplicate elements together.

2. Use backtracking to build permutations step by step.

3. Maintain a boolean array used to mark whether an element is already included.

4. At each step, skip an element if it is already used.

5. To avoid duplicate permutations, skip the current element if it is the same as the previous one and the previous one was not used.

6. If the current permutation length equals the length of the array, store it as a valid permutation.

7. Continue until all unique permutations are generated.

CODE:

```python
def permuteUnique(nums):
    nums.sort()  # Sort to handle duplicates
    result = []
    used = [False] * len(nums)

    def backtrack(path):
        if len(path) == len(nums):
            result.append(path.copy())
            return

        for i in range(len(nums)):
            if used[i]:
                continue
            # Skip duplicates
            if i > 0 and nums[i] == nums[i - 1] and not used[i - 1]:
                continue

            used[i] = True
            path.append(nums[i])
            backtrack(path)
            path.pop()
            used[i] = False

    backtrack([])
    return result
```

INPUT:

```python
# Example
nums = [1, 1, 2]
print(permuteUnique(nums))
```

OUTPUT:

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

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 implement a graph coloring algorithm for an undirected graph using the minimum number of colors and to determine the maximum number of regions you can color when three players (You, Alice, Bob) take turns coloring one uncolored region at a time.

ALGORITHM:

1. Represent the graph using an adjacency list.

2. Use backtracking to color the graph such that no two adjacent vertices have the same color.

3. Try to color the graph using the given number of colors k.

4. Once all vertices are colored, simulate the turn-based coloring process:

    o   You color first

    o   Alice colors next

    o   Bob colors next

    o   Repeat until all regions are colored

5. Count how many times you get a turn.

6. Return the count as the maximum number of regions you can color.

CODE:

```python
def is_safe(vertex, graph, color, c):
    for i in range(len(graph)):
        if graph[vertex][i] == 1 and color[i] == c:
            return False
    return True
def graph_coloring_util(graph, m, color, vertex):
    if vertex == len(graph):
        return True
    for c in range(1, m + 1):
        if is_safe(vertex, graph, color, c):
            color[vertex] = c
            if graph_coloring_util(graph, m, color, vertex + 1):
                return True
            color[vertex] = 0  # backtrack
    return False
def graph_coloring(graph):
    n = len(graph)
    for m in range(1, n + 1):  # try minimum colors
        color = [0] * n
        if graph_coloring_util(graph, m, color, 0):
            print(f"Minimum number of colors required: {m}")
            print("Color assignment:", color)
            return
```

INPUT:

```
# Given graph
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
n = 4
# Create adjacency matrix
graph = [[0] * n for _ in range(n)]
for u, v in edges:
    graph[u][v] = 1
    graph[v][u] = 1
# Run graph coloring
graph_coloring(graph)
```

OUTPUT:

Minimum number of colors required: 3
Color assignment: [1, 2, 3, 2]

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, which is a cycle that visits every vertex exactly once and returns to the starting vertex.

ALGORITHM:

1. Represent the graph using an adjacency list.

2. Use backtracking to try all possible paths starting from a fixed vertex (say 0).

3. Maintain a path array to store the current path.

4. At each step, try to add an unvisited vertex that is adjacent to the last vertex in the path.

5. If all vertices are included and there is an edge from the last vertex back to the first, a Hamiltonian cycle exists.

6. If no such path is found after trying all possibilities, return false.

CODE:

```python
def hamiltonian_cycle(edges, n):
    # Step 1: Create adjacency matrix
    graph = [[0]*n for _ in range(n)]
    for u, v in edges:
        graph[u][v] = 1
        graph[v][u] = 1
    path = [-1] * n
    path[0] = 0  # Start from vertex 0
    # Step 2: Check if vertex v can be added at position pos
    def is_safe(v, pos):
        if graph[path[pos-1]][v] == 0:
            return False
        if v in path:
            return False
        return True
    # Step 3: Recursive utility function
    def ham_cycle_util(pos):
        if pos == n:
            return graph[path[pos-1]][path[0]] == 1
        for v in range(1, n):
            if is_safe(v, pos):
                path[pos] = v
                if ham_cycle_util(pos + 1):
                    return True
                path[pos] = -1
        return False
    return ham_cycle_util(1)
```

INPUT:

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

OUTPUT:

False

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, which is a cycle that visits every vertex exactly once and returns to the starting vertex.

ALGORITHM:

1. Represent the graph using an adjacency list.

2. Use backtracking to try all possible paths starting from a fixed vertex (say 0).

3. Maintain a path array to store the current path.

4. At each step, try to add an unvisited vertex that is adjacent to the last vertex in the path.

5. If all vertices are included and there is an edge from the last vertex back to the first, a Hamiltonian cycle exists.

6. If no such path is found after trying all possibilities, return false.

CODE:

```python
def hamiltonian_cycle(edges, n):
    # Create adjacency matrix
    graph = [[0]*n for _ in range(n)]
    for u, v in edges:
        graph[u][v] = 1
        graph[v][u] = 1
    path = [-1] * n
    path[0] = 0  # start from vertex 0

    def is_safe(v, pos):
        if graph[path[pos-1]][v] == 0:
            return False
        if v in path:
            return False
        return True
    def solve(pos):
        if pos == n:
            return graph[path[pos-1]][path[0]] == 1
        for v in range(1, n):
            if is_safe(v, pos):
                path[pos] = v
                if solve(pos + 1):
                    return True
                path[pos] = -1
        return False
    return solve(1)
```

INPUT:

OUTPUT:

True

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

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 determine whether a given undirected graph contains a Hamiltonian Cycle, which is a cycle that visits every vertex exactly once and returns to the starting vertex.

ALGORITHM:

1. Represent the graph using an adjacency list.

2. Use backtracking to try all possible paths starting from a fixed vertex (say 0).

3. Maintain a path array to store the current path.

4. At each step, try to add an unvisited vertex that is adjacent to the last vertex in the path.

5. If all vertices are included and there is an edge from the last vertex back to the first, a Hamiltonian cycle exists.

6. If no such path is found after trying all possibilities, return false.

CODE:

```python
def is_hamiltonian_cycle(edges, n):
    # Convert edge list to adjacency list
    graph = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    path = [0]  # start from vertex 0

    # Helper function for backtracking
    def backtrack(path):
        if len(path) == n:
            # If last vertex connects to first, cycle exists
            if path[0] in graph[path[-1]]:
                return True
            else:
                return False

        for neighbor in graph[path[-1]]:
            if neighbor not in path:  # safe to add
                path.append(neighbor)
                if backtrack(path):
                    return True
                path.pop()  # backtrack

        return False

    return backtrack(path)
```

INPUT:

```
# Example usage:
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2), (2, 4), (4, 0)]
n = 5
print(is_hamiltonian_cycle(edges, n))
```

OUTPUT:

False

15. 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 S containing n elements in lexicographical order and analyze how the algorithm handles duplicate elements.

ALGORITHM:

1. Sort the input array to ensure lexicographical order.

2. Use backtracking (recursion) to generate all subsets.

3. Start with an empty subset and at each step, decide whether to include the current element.

4. Add the current subset to the result list.

5. Recursively repeat for the remaining elements.

6. If duplicates exist and are not removed, duplicate subsets will appear. To avoid duplicates, skip repeated elements during recursion.

CODE:

```python
def generate_subsets(S):
    S.sort()  # Sort to ensure lexicographical order
    result = []

    def backtrack(start, path):
        result.append(path[:])  # Add a copy of current subset
        for i in range(start, len(S)):
            # Skip duplicates
            if i > start and S[i] == S[i-1]:
                continue
            path.append(S[i])
            backtrack(i + 1, path)
            path.pop()  # Backtrack

    backtrack(0, [])
    return result
```

INPUT:

```python
# Example usage:
A = [1, 2, 3]
subsets = generate_subsets(A)
print(subsets)
```

OUTPUT:

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

16. **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 of unique integers that **must contain a specific element (3)** and also to generate the **complete power set** (all possible subsets) of a given array without duplicates.

ALGORITHM:

Part 1: Subsets containing a specific element (3)

1. Generate all possible subsets using backtracking.

2. For each generated subset, check if it contains the element 3.

3. If yes, add it to the result list.

4. Return the filtered list.

Part 2: Power Set (All Subsets)

1. Start with an empty subset.

2. For each element, choose whether to include it or not.

3. Use recursion to generate all combinations.

4. Store every generated subset.

5. Return the final list.

CODE:

```python
# Subsets containing a specific element (3)
def subsets_with_element(arr, x):
    result = []
    def backtrack(start, subset):
        if x in subset:
            result.append(subset[:])
        for i in range(start, len(arr)):
            subset.append(arr[i])
            backtrack(i + 1, subset)
            subset.pop()
    backtrack(0, [])
    return result
# Power Set (All Subsets)
def power_set(nums):
    result = []
    def backtrack(start, subset):
        result.append(subset[:])
        for i in range(start, len(nums)):
            subset.append(nums[i])
            backtrack(i + 1, subset)
            subset.pop()
    backtrack(0, [])
    return result
```

INPUT:

```
# Example runs
E = [2, 3, 4, 5]
x = 3
print("Subsets containing 3:", subsets_with_element(E, x))
nums1 = [1, 2, 3]
nums2 = [0]
print("Power set of [1,2,3]:", power_set(nums1))
print("Power set of [0]:", power_set(nums2))
```

OUTPUT:

```
Subsets containing 3: [[2, 3], [2, 3, 4], [2, 3, 4, 5], [2, 3, 5], [3], [3, 4], [3, 4, 5], [3, 5]]
Power set of [1,2,3]: [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
Power set of [0]: [[], [0]]
```

17. **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 **universal strings** from words1 such that each string contains **all characters (with multiplicity)** of every word in words2.

ALGORITHM:

1. Create a frequency array for each word in words2.

2. Compute the **maximum frequency** required for each character across all words in words2.

3. For each word in words1:

    o Count its character frequencies.

    o Check if it satisfies all required character frequencies.

4. If it does, add it to the result list.

5. Return the final list.

CODE:

```python
from collections import Counter
def wordSubsets(words1, words2):
    max_freq = Counter()
    # Step 1: Find maximum frequency of each character in words2
    for word in words2:
        freq = Counter(word)
        for char in freq:
            max_freq[char] = max(max_freq[char], freq[char])
    result = []
    # Step 2: Check each word in words1
    for word in words1:
        freq_word = Counter(word)
        is_universal = True
        for char in max_freq:
            if freq_word[char] < max_freq[char]:
                is_universal = False
                break
        if is_universal:
            result.append(word)
    return result
```

INPUT:

```python
# Example Runs
words1 = ["amazon","apple","facebook","google","leetcode"]
words2 = ["e","o"]
print(wordSubsets(words1, words2))

words2 = ["l","e"]
print(wordSubsets(words1, words2))
```

OUTPUT:

['facebook', 'google', 'leetcode']

['apple', 'google', 'leetcode']