1.N Queen problem:-

Program:-

```
def is_safe(board, row, col, N):
    for i in range(col):
        if board[row][i] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True

def solve_n_queens_util(board, col, N):
    if col >= N:
        return True
    for i in range(N):
        if is_safe(board, i, col, N):
            board[i][col] = 1
            if solve_n_queens_util(board, col + 1, N) == True:
                return True
            board[i][col] = 0
    return False

def solve_n_queens(N):
```

```
    board = [[0 for _ in range(N)] for _ in range(N)]
    if solve_n_queens_util(board, 0, N) == False:
        return False
    return board

def print_solution(board):
    for row in board:
        print(row)

# Example usage
N = 4
solution = solve_n_queens(N)
if solution:
    print_solution(solution)
else:
    print("No solution exists for N =", N)
```

2.Subset sum:-

Program:-

```
def isSubsetSum(arr, n, sum):
    if sum == 0:
        return True
    if n == 0 and sum != 0:
        return False
    if arr[n-1] > sum:
```

```
        return isSubsetSum(arr, n-1, sum)
    return isSubsetSum(arr, n-1, sum) or isSubsetSum(arr, n-1, sum-arr[n-1])

arr = [3, 34, 4, 12, 5, 2]
sum = 9
n = len(arr)
if isSubsetSum(arr, n, sum) == True:
    print("Found a subset with the given sum")
else:
    print("No subset with the given sum")
```

3.Graph Colouring:-

Program:-

```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.Graph()
G.add_edges_from([(1, 2), (1, 3), (2, 3), (3, 4), (4, 5), (4, 6), (5, 6)])

colors = nx.greedy_color(G, strategy='largest_first')

pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_color=[colors[node] for node in
G.nodes], cmap=plt.cm.rainbow)
plt.show()
```

4.Hamiltonian Circuit:-

Program:-

```
def hamiltonian(graph, start_v):
    def hamiltonian_util(v, visited):
        if len(visited) == len(graph):
            return start_v in graph[v]

        for u in graph[v]:
            if u not in visited:
                visited.add(u)
                if hamiltonian_util(u, visited):
                    return True
                visited.remove(u)

        return False

    visited = set([start_v])
    return hamiltonian_util(start_v, visited)

# Example Usage
graph = {
    'A': ['B', 'C', 'D'],
    'B': ['A', 'C', 'D'],
    'C': ['A', 'B', 'D'],
    'D': ['A', 'B', 'C']
}
```

```
start_vertex = 'A'
print(hamiltonian(graph, start_vertex))
```

5.Permutation n combination:-

Program:-

```
from itertools import permutations, combinations

# Permutations
n = 4
perm = permutations(range(n))
print("Permutations:")
for i in list(perm):
    print(i)

# Combinations
r = 2
comb = combinations(range(n), r)
print("\nCombinations:")
for i in list(comb):
    print(i)
```

6.Sudoku solver:-

Program:-

```
def find_empty(bo):
    for i in range(9):
        for j in range(9):
            if bo[i][j] == 0:
                return i, j  # row, col
    return None, None
```

```
def valid(bo, num, pos):
    # Check row
    for i in range(9):
        if bo[pos[0]][i] == num and pos[1] != i:
            return False

    # Check column
    for i in range(9):
        if bo[i][pos[1]] == num and pos[0] != i:
            return False

    # Check box
    box_x = pos[1] // 3
    box_y = pos[0] // 3

    for i in range(box_y*3, box_y*3 + 3):
        for j in range(box_x * 3, box_x*3 + 3):
            if bo[i][j] == num and (i,j) != pos:
                return False

    return True

def solve_sudoku(bo):
    find = find_empty(bo)
    if find[0] is None:
        return True

    row, col = find

    for i in range(1,10):
        if valid(bo, i, (row, col)):
            bo[row][col] = i

            if solve_sudoku(bo):
                return True

            bo[row][col] = 0

    return False

def print_board(bo):
    for i in range(9):
```

```
        if i % 3 == 0 and i != 0:
            print("- - - - - - - - - - -")

        for j in range(9):
            if j % 3 == 0 and j != 0:
                print(" | ", end="")

            if j == 8:
                print(bo[i][j])
            else:
                print(str(bo[i][j]) + " ", end="")

# Test the solver
board = [
    [7, 8, 0, 4, 0, 0, 1, 2, 0],
    [6, 0, 0, 0, 7, 5, 0, 0, 9],
    [0, 0, 0, 6, 0, 1, 0, 7, 8],
    [0, 0, 7, 0, 4, 0, 2, 6, 0],
    [0, 0, 1, 0, 5, 0, 9, 3, 0],
    [9, 0, 4, 0, 6, 0, 0, 0, 5],
    [0, 7, 0, 3, 0, 0, 0, 1, 2],
    [1, 2, 0, 0, 0, 7, 4, 0, 0],
    [0, 4, 9, 2, 0, 6, 0, 0, 7]
]

print_board(board)
solve_sudoku(board)
print("--------------------")
print_board(board)
```