1.N-QUEEN

```
def print_board(board):
    for row in board:
        print(" ".join("Q" if col else "." for col in row))
def is_safe(board, row, col, n):
    for i in range(col):
        if board[row][i]:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j]:
            return False
    for i, j in zip(range(row, n, 1), range(col, -1, -1)):
        if board[i][j]:
            return False
    return True

def solve_n_queens(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(board, col + 1, n):
                return True
            board[i][col] = 0
    return False

def n_queen(n):
    board = [[0 for _ in range(n)] for _ in range(n)]
    if solve_n_queens(board, 0, n):
        print_board(board)
    else:
        print("No solution exists")
n_queen(8)
```

2.SUBSET SUM PROBLEM

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

arr = [3, 34, 4, 12, 5, 2]
sum = 9
n = len(arr)
print(is_subset_sum(arr, n, sum))
```

3.GRAPH COLOURING

```
def is_safe(graph, color, c, v):
    for i in range(len(graph)):
        if graph[v][i] == 1 and color[i] == c:
            return False
    return True


def graph_coloring_util(graph, m, color, v):
    if v == len(graph):
        return True
    for c in range(1, m+1):
        if is_safe(graph, color, c, v):
            color[v] = c
            if graph_coloring_util(graph, m, color, v+1):
                return True
            color[v] = 0
    return False

def graph_coloring(graph, m):
    color = [0] * len(graph)
    if graph_coloring_util(graph, m, color, 0):
        return color
    else:
        return "No solution"

graph = [[0, 1, 1, 1],
         [1, 0, 1, 0],
         [1, 1, 0, 1],
         [1, 0, 1, 0]]
m = 3
print(graph_coloring(graph, m))
```

4. Hamiltonian Circuit Problem

```
def is_valid(v, pos, path, graph):
    if graph[path[pos-1]][v] == 0:
        return False
    for vertex in path:
        if vertex == v:
            return False
    return True

def hamiltonian_util(graph, path, pos):
    if pos == len(graph):
        if graph[path[pos-1]][path[0]] == 1:
            return True
        else:
            return False
    for v in range(1, len(graph)):
        if is_valid(v, pos, path, graph):
            path[pos] = v
            if hamiltonian_util(graph, path, pos+1):
                return True
            path[pos] = -1
    return False

def hamiltonian_cycle(graph):
    path = [-1] * len(graph)
    path[0] = 0
    if hamiltonian_util(graph, path, 1):
        return path + [path[0]]
    else:
        return "No solution"
graph = [[0, 1, 0, 1, 0],
         [1, 0, 1, 1, 1],
         [0, 1, 0, 0, 1],
         [1, 1, 0, 0, 1],
         [0, 1, 1, 1, 0]]
print(hamiltonian_cycle(graph))
```

5. Permutation and Combination

```
from itertools import permutations, combinations
def print_permutations(elements):
    perms = list(permutations(elements))
    for perm in perms:
        print(perm)

def print_combinations(elements, r):
    combs = list(combinations(elements, r))
    for comb in combs:
        print(comb)
elements = [1, 2, 3]
print("Permutations:")
print_permutations(elements)
print("Combinations (r=2):")
print_combinations(elements, 2)
```

6. Sudoku Solver

```
def print_board(board):
    for row in board:
        print(" ".join(str(num) if num != 0 else '.' for num in row))

def find_empty_location(board, l):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                l[0], l[1] = i, j
                return True
    return False

def used_in_row(board, row, num):
    return any(board[row][i] == num for i in range(9))

def used_in_col(board, col, num):
    return any(board[i][col] == num for i in range(9))

def used_in_box(board, row, col, num):
    return any(board[i][j] == num for i in range(row, row + 3) for j in range(col, col + 3))

def is_safe(board, row, col, num):
    return not used_in_row(board, row, num) and \
           not used_in_col(board, col, num) and \
           not used_in_box(board, row - row % 3, col - col % 3, num)

def solve_sudoku(board):
    l = [0, 0]
    if not find_empty_location(board, l):
        return True
    row, col = l[0], l[1]
    for num in range(1, 10):
        if is_safe(board, row, col, num):
            board[row][col] = num
            if solve_sudoku(board):
                return True
            board[row][col] = 0
    return False
board = [[5, 3, 0, 0, 7, 0, 0, 0, 0],
         [6, 0, 0, 1, 9, 5, 0, 0, 0],
         [0, 9, 8, 0, 0, 0, 0, 6, 0],
         [8, 0, 0, 0, 6, 0, 0, 0, 3],
         [4, 0, 0, 8, 0, 3, 0, 0, 1],
         [7, 0, 0, 0, 2, 0, 0, 0, 6],
         [0, 6, 0, 0, 0, 0, 2, 8, 0],
         [0, 0, 0, 4, 1, 9, 0, 0, 5],
         [0, 0, 0, 0, 8, 0, 0, 7, 9]]

if solve_sudoku(board):
    print_board(board)
else:
    print("No solution exists")
```