1. N-Queens Problem

```
python
Copy code
def solveNQueens(n):
    def is_safe(board, row, col):
        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(board, col):
        if col >= n:
            return True
        for i in range(n):
            if is_safe(board, i, col):
                board[i][col] = 1
                if solve(board, col + 1):
                    return True
                board[i][col] = 0
        return False

    board = [[0] * n for _ in range(n)]
    if not solve(board, 0):
        return "No solution exists"
    return board

print(solveNQueens(4))
```

2. Subset Sum Problem

```
python
Copy code
def isSubsetSum(set, n, sum):
    if sum == 0:
        return True
    if n == 0 and sum != 0:
        return False
    if set[n - 1] > sum:
        return isSubsetSum(set, n - 1, sum)
    return isSubsetSum(set, n - 1, sum) or isSubsetSum(set, n - 1, sum -
set[n - 1])

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

3. Graph Coloring Problem

```
python
Copy code
def graphColoring(graph, m):
    def isSafe(v, color, c):
        for i in range(len(graph)):
            if graph[v][i] == 1 and color[i] == c:
                return False
        return True

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

    color = [0] * len(graph)
    if not graphColoringUtil(m, color, 0):
        return "Solution does not exist"
    return color

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

4. Hamiltonian Circuit Problem

```
python
Copy code
def isSafe(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 hamCycleUtil(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 isSafe(v, pos, path, graph):
            path[pos] = v
            if hamCycleUtil(graph, path, pos + 1):
                return True
            path[pos] = -1
    return False

def hamCycle(graph):
    path = [-1] * len(graph)
    path[0] = 0
    if not hamCycleUtil(graph, path, 1):
        return "Solution does not exist"
    return path
```

```
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(hamCycle(graph))
```

5. Permutation

```
python
Copy code
def permute(s, answer):
    if len(s) == 0:
        print(answer, end=" ")
        return
    for i in range(len(s)):
        ch = s[i]
        left_substr = s[0:i]
        right_substr = s[i + 1:]
        rest = left_substr + right_substr
        permute(rest, answer + ch)

string = "ABC"
answer = ""
permute(string, answer)
```

6. Sudoku Solver

```
python
Copy code
def solveSudoku(board):
    def isSafe(board, row, col, num):
        for x in range(9):
            if board[row][x] == num or board[x][col] == num:
                return False
        startRow, startCol = 3 * (row // 3), 3 * (col // 3)
        for i in range(3):
            for j in range(3):
                if board[i + startRow][j + startCol] == num:
                    return False
        return True

    def solve(board):
        l = [0, 0]
        if not findEmptyLocation(board, l):
            return True
        row, col = l[0], l[1]
        for num in range(1, 10):
            if isSafe(board, row, col, num):
                board[row][col] = num
                if solve(board):
                    return True
                board[row][col] = 0
        return False

    def findEmptyLocation(board, l):
        for row in range(9):
            for col in range(9):
                if board[row][col] == 0:
                    l[0], l[1] = row, col
                    return True
        return False
    if not solve(board):
        return "No solution exists"
    return board

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]
]
print(solveSudoku(board))
```