1.N queen problem

Program:

def solveNQueens(N):

def isSafe(board, row, col):

# Check this row on the left side

for i in range(col):

if board[row][i] == 1:

return False

# Check upper diagonal on the left side

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

# Check lower diagonal on the left side

for i, j in zip(range(row, N, 1), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solveNQUtil(board, col):

if col >= N:

return True

for i in range(N):

if isSafe(board, i, col):

board[i][col] = 1

if solveNQUtil(board, col + 1):

return True

board[i][col] = 0 # Backtrack

return False

board = [[0 for \_ in range(N)] for \_ in range(N)]

if not solveNQUtil(board, 0):

return None

else:

return board

# Example usage:

N = 8

solution = solveNQueens(N)

if solution:

for row in solution:

print(row)

else:

print("No solution exists")

2. subset sum

Program:

def isSubsetSum(set, n, sum):

# Initialize a 2D list to store the results of subproblems

subset = [[False for \_ in range(sum + 1)] for \_ in range(n + 1)]

# If sum is 0, then answer is true (0 sum can be achieved with an empty subset)

for i in range(n + 1):

subset[i][0] = True

# If sum is not 0 and set is empty, then answer is false

for i in range(1, sum + 1):

subset[0][i] = False

# Fill the subset table in a bottom-up manner

for i in range(1, n + 1):

for j in range(1, sum + 1):

if j < set[i - 1]:

subset[i][j] = subset[i - 1][j]

else:

subset[i][j] = subset[i - 1][j] or subset[i - 1][j - set[i - 1]]

return subset[n][sum]

# Example usage:

set = [3, 34, 4, 12, 5, 2]

sum = 9

n = len(set)

if isSubsetSum(set, n, sum):

print("Found a subset with given sum")

else:

print("No subset with given sum")

3.graph colouring

Program:

def isSafe(graph, color, v, c):

for i in range(len(graph)):

if graph[v][i] == 1 and color[i] == c:

return False

return True

def graphColoringUtil(graph, m, color, v):

if v == len(graph):

return True

for c in range(1, m + 1):

if isSafe(graph, color, v, c):

color[v] = c

if graphColoringUtil(graph, m, color, v + 1):

return True

color[v] = 0

return False

def graphColoring(graph, m):

color = [0] \* len(graph)

if graphColoringUtil(graph, m, color, 0):

return color

else:

return None

# Example usage:

graph = [

[0, 1, 1, 1],

[1, 0, 1, 0],

[1, 1, 0, 1],

[1, 0, 1, 0]

]

m = 3 # Number of colors

solution = graphColoring(graph, m)

if solution:

print("Solution exists: ", solution)

else:

print("No solution exists")

4. Hamiltoniam circuit problem

Program:

def isSafe(v, pos, path, graph):

# Check if this vertex is an adjacent vertex of the previously added vertex

if graph[path[pos - 1]][v] == 0:

return False

# Check if the vertex has already been included in the path

if v in path:

return False

return True

def hamiltonianCycleUtil(graph, path, pos):

# Base case: If all vertices are included in the path

if pos == len(graph):

# And if there is an edge from the last included vertex to the first vertex

if graph[path[pos - 1]][path[0]] == 1:

return True

else:

return False

# Try different vertices as the next candidate in Hamiltonian Cycle

for v in range(1, len(graph)):

if isSafe(v, pos, path, graph):

path[pos] = v

if hamiltonianCycleUtil(graph, path, pos + 1):

return True

# If adding vertex v doesn't lead to a solution, remove it

path[pos] = -1

return False

def hamiltonianCycle(graph):

path = [-1] \* len(graph)

path[0] = 0 # Start from the first vertex

if not hamiltonianCycleUtil(graph, path, 1):

return None

else:

path.append(path[0]) # Add starting point to complete the cycle

return path

# Example usage:

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]

]

solution = hamiltonianCycle(graph)

if solution:

print("Hamiltonian cycle exists: ", solution)

else:

print("No Hamiltonian cycle exists")

5. permutation and comibination

Program:’

import math

def permutations(n, r):

return math.factorial(n) // math.factorial(n - r)

def combinations(n, r):

return math.factorial(n) // (math.factorial(r) \* math.factorial(n - r))

# Example usage:

n = 5

r = 3

print(f"Permutations P({n}, {r}) = {permutations(n, r)}")

print(f"Combinations C({n}, {r}) = {combinations(n, r)}")

5. Sudoku solver

Program:’

def is\_safe(board, row, col, num):

# Check if the number is not repeated in the current row, column, and 3x3 subgrid

for i in range(9):

if board[row][i] == num or board[i][col] == num or board[row - row % 3 + i // 3][col - col % 3 + i % 3] == num:

return False

return True

def solve\_sudoku(board):

for row in range(9):

for col in range(9):

if board[row][col] == 0:

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 # Backtrack

return False

return True

def print\_board(board):

for row in board:

print(" ".join(str(num) for num in row))

# Example usage:

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("Sudoku puzzle solved:")

print\_board(board)

else:

print("No solution exists")