# N queen problem

```
def is_safe(board, row, col, n):
    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on 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 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 solve_n_queens_util(board, col, n):
    # base case: If all queens are placed, return True
    if col >= n:
        return True

    # Consider this column and try placing this queen in all rows one by one
    for i in range(n):
        if is_safe(board, i, col, n):
            # Place this queen in board[i][col]
            board[i][col] = 1

            # recur to place rest of the queens
```

```
        if solve_n_queens_util(board, col + 1, n):
            return True

        # If placing queen in board[i][col] doesn't lead to a solution
        # then remove queen from board[i][col]
        board[i][col] = 0

    # if the queen cannot be placed in any row in this column col then return False
    return False

def solve_n_queens(n):
    board = [[0 for _ in range(n)] for _ in range(n)]

    if not solve_n_queens_util(board, 0, n):
        print("Solution does not exist")
        return False

    print_board(board)
    return True

def print_board(board):
    for row in board:
        print(" ".join(str(x) for x in row))

# Solve for 8 queens
solve_n_queens(8)
```

## subset sum

```
def is_subset_sum(set, n, sum):
    # Initialize the dp array
    dp = [[False for x in range(sum + 1)] for y in range(n + 1)]
```

```
    # If sum is 0, then answer is true
    for i in range(n + 1):
        dp[i][0] = True

    # 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]:
                dp[i][j] = dp[i - 1][j]
            else:
                dp[i][j] = dp[i - 1][j] or dp[i - 1][j - set[i - 1]]

    return dp[n][sum]

# Example usage
set = [3, 34, 4, 12, 5, 2]
sum = 9
n = len(set)
if is_subset_sum(set, n, sum):
    print("Found a subset with given sum")
else:
    print("No subset with given sum")
```

# GRAPH COLOURING

```
def is_safe(graph, color, v, c):
    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, v, c):
         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 not graph_coloring_util(graph, m, color, 0):
      print("Solution does not exist")
      return False

   print("Solution exists: Following are the assigned colors")
   print(color)
   return True

# Example usage
graph = [
   [0, 1, 1, 1],
   [1, 0, 1, 0],
   [1, 1, 0, 1],
   [1, 0, 1, 0]
]
m = 3
```

```
graph_coloring(graph, m)
```

## Hamiltoniam circuit problem

```
class Graph:
    def _init_(self, vertices):
        self.graph = [[0 for column in range(vertices)] for row in range(vertices)]
        self.V = vertices

    def is_safe(self, v, pos, path):
        # Check if this vertex is an adjacent vertex of the previously added vertex.
        if self.graph[path[pos - 1]][v] == 0:
            return False

        # Check if the vertex has already been included.
        if v in path:
            return False

        return True

    def ham_cycle_util(self, path, pos):
        # Base case: If all vertices are included in the path
        if pos == self.V:
            # And if there is an edge from the last included vertex to the first vertex
            if self.graph[path[pos - 1]][path[0]] == 1:
                return True
            else:
                return False

        # Try different vertices as the next candidate in the Hamiltonian Cycle.
        for v in range(1, self.V):
            if self.is_safe(v, pos, path):
```

```
            path[pos] = v

            if self.ham_cycle_util(path, pos + 1) == True:
                return True

            # Remove current vertex if it doesn't lead to a solution
            path[pos] = -1

        return False

    def ham_cycle(self):
        path = [-1] * self.V
        # Let the first vertex in the path be 0
        path[0] = 0

        if self.ham_cycle_util(path, 1) == False:
            print("Solution does not exist")
            return False

        self.print_solution(path)
        return True

    def print_solution(self, path):
        print("Solution Exists: Following is one Hamiltonian Cycle")
        for vertex in path:
            print(vertex, end=" ")
        print(path[0], "\n")

# Example usage
g = Graph(5)
g.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]]

g.ham_cycle()
```

## Permutation n computation

```
def permute(elements):
    if len(elements) == 0:
        return []

    if len(elements) == 1:
        return [elements]

    perms = []  # List to store current permutations
    for i in range(len(elements)):
        m = elements[i]

        # Remaining elements
        rem_elements = elements[:i] + elements[i+1:]

        # Generate all permutations where m is the first element
        for p in permute(rem_elements):
            perms.append([m] + p)
    return perms

# Example usage
elements = [1, 2, 3]
permutations = permute(elements)
for perm in permutations:
```

```
    print(perm)
```

## sudoku slover

```
def is_safe(board, row, col, num):
    # Check if 'num' is not in the given row
    for x in range(9):
        if board[row][x] == num:
            return False

    # Check if 'num' is not in the given column
    for x in range(9):
        if board[x][col] == num:
            return False

    # Check if 'num' is not in the particular 3x3 box
    start_row = row - row % 3
    start_col = col - col % 3
    for i in range(3):
        for j in range(3):
            if board[i + start_row][j + start_col] == num:
                return False

    return True

def solve_sudoku(board):
    empty = find_empty_location(board)
    if not empty:
        return True  # No empty space left, puzzle solved

    row, col = empty
```

```
    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  # Reset if num doesn't lead to a solution

    return False

def find_empty_location(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                return (i, j)
    return None

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 solved successfully:")
    print_board(board)
else:
    print("No solution exists.")
```