1) **Write the python program to solve 8-Puzzle problem.**

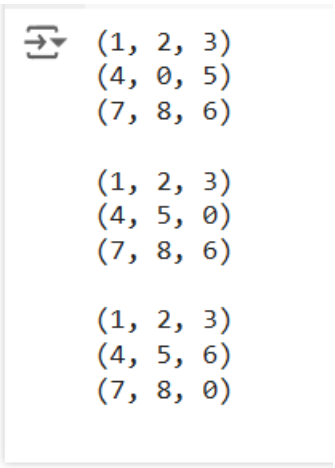
PROGRAM:

```
import heapq
import itertools
class PuzzleState:
    def __init__(self, board, moves=0, parent=None):
        self.board = board
        self.blank_pos = board.index(0)
        self.moves = moves
        self.parent = parent
        self.h = self.heuristic()
    def heuristic(self):
        goal_pos = {n: (i, j) for i, row in enumerate([(1, 2, 3), (4, 5, 6), (7, 8, 0)]) for j, n in
enumerate(row)}
        h = 0
        for idx, val in enumerate(self.board):
            if val != 0:
                goal_x, goal_y = goal_pos[val]
                current_x, current_y = idx // 3, idx % 3
                h += abs(goal_x - current_x) + abs(goal_y - current_y)
        return h
    def __lt__(self, other):
        return (self.moves + self.h) < (other.moves + other.h)
def generate_moves(state):
    def swap_and_create(board, pos1, pos2):
        new_board = list(board)
        new_board[pos1], new_board[pos2] = new_board[pos2], new_board[pos1]
        return tuple(new_board)
    moves = []
    blank_pos = state.blank_pos
    blank_x, blank_y = divmod(blank_pos, 3)
    for dx, dy in [(0, -1), (0, 1), (-1, 0), (1, 0)]:
        new_x, new_y = blank_x + dx, blank_y + dy
```

```
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_blank_pos = new_x * 3 + new_y
            new_board = swap_and_create(state.board, blank_pos, new_blank_pos)
            moves.append(PuzzleState(new_board, state.moves + 1, state))
    return moves
def a_star_search(start_state):
    frontier = []
    heapq.heappush(frontier, start_state)
    explored = set()
    while frontier:
        current_state = heapq.heappop(frontier)
        if current_state.board == (1, 2, 3, 4, 5, 6, 7, 8, 0):
            return current_state
        explored.add(current_state.board)
        for next_state in generate_moves(current_state):
            if next_state.board not in explored:
                heapq.heappush(frontier, next_state)
    return None
def print_solution(state):
    if state is None:
        print("No solution found.")
        return
    path = []
    while state:
        path.append(state.board)
        state = state.parent
    path.reverse()
    for step in path:
        print(f"{step[0:3]}\n{step[3:6]}\n{step[6:9]}\n")
if __name__ == "__main__":
    start_board = (1, 2, 3, 4, 0, 5, 7, 8, 6)
    start_state = PuzzleState(start_board)
```

```
solution = a_star_search(start_state)
print_solution(solution)
```

OUTPUT:



```
(1, 2, 3)
(4, 0, 5)
(7, 8, 6)

(1, 2, 3)
(4, 5, 0)
(7, 8, 6)

(1, 2, 3)
(4, 5, 6)
(7, 8, 0)
```

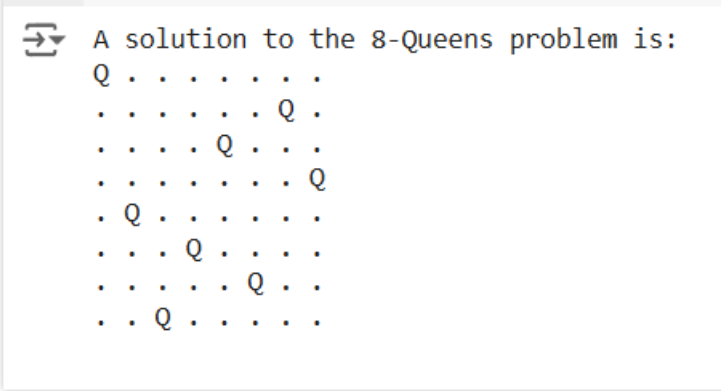
2) Write the python program to solve 8-Queen problem.

PROGRAM:

```
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, len(board)), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True
def solve_nqueens(board, col):
    if col >= len(board):
        return True
    for i in range(len(board)):
        if is_safe(board, i, col):
            board[i][col] = 1
            if solve_nqueens(board, col + 1):
                return True
```

```
            board[i][col] = 0  # Backtrack
    return False
def print_board(board):
    for row in board:
        print(" ".join("Q" if x == 1 else "." for x in row))
    print()
def solve_8queens():
    board = [[0] * 8 for _ in range(8)]
    if solve_nqueens(board, 0):
        print("A solution to the 8-Queens problem is:")
        print_board(board)
    else:
        print("No solution exists.")
if __name__ == "__main__":
    solve_8queens()
```

OUTPUT:



3) Write the python program for Water Jug Problem.

PROGRAM:

```
from collections import deque
def water_jug_problem(capacity_a, capacity_b, target):
    initial_state = (0, 0)
    queue = deque([(initial_state, [])])
    visited = set()
    visited.add(initial_state)
    while queue:
```

```
        (current_a, current_b), path = queue.popleft()
        if current_a == target or current_b == target:
            return path + [(current_a, current_b)]
        actions = [
            ("Fill A", (capacity_a, current_b)),
            ("Fill B", (current_a, capacity_b)),
            ("Empty A", (0, current_b)),
            ("Empty B", (current_a, 0)),
            ("Pour A to B", (max(0, current_a - (capacity_b - current_b)), min(capacity_b, current_b + current_a))),
            ("Pour B to A", (min(capacity_a, current_a + current_b), max(0, current_b - (capacity_a - current_a))))
        ]
        for action, (next_a, next_b) in actions:
            next_state = (next_a, next_b)
            if next_state not in visited:
                visited.add(next_state)
                queue.append((next_state, path + [action]))
    return None
def print_solution(actions):
    if actions is None:
        print("No solution exists.")
    else:
        print("Solution steps:")
        for step in actions:
            print(step)
if __name__ == "__main__":
    capacity_a = 4
    capacity_b = 3
    target = 2

    actions = water_jug_problem(capacity_a, capacity_b, target)
    print_solution(actions)
```

OUTPUT:

```
Solution steps:
Fill B
Pour B to A
Fill B
Pour B to A
(4, 2)
```

4) **Write the python program for Cript-Arithmetic problem.**

PROGRAM:

```
from itertools import permutations
def is_valid_assignment(send, more, money, assignment):
    letters = "SENDMOREMONEY"
    unique_letters = set(letters)
    letter_to_digit = dict(zip(unique_letters, assignment)) # Create dictionary with all unique letters
    send_value = int(''.join(str(letter_to_digit[char]) for char in send))
    more_value = int(''.join(str(letter_to_digit[char]) for char in more))
    money_value = int(''.join(str(letter_to_digit[char]) for char in money))
    return send_value + more_value == money_value
def solve_cryptarithmetic():
    letters = "SENDMOREMONEY"
    unique_letters = set(letters)
    if len(unique_letters) > 10:
        print("Too many unique letters for digits 0-9.")
        return
    for perm in permutations(range(10), len(unique_letters)): # Generate permutations with the correct length
        if is_valid_assignment("SEND", "MORE", "MONEY", perm):
            letter_to_digit = dict(zip(unique_letters, perm))
            print("Solution found:")
            print(" ".join(f"{letter}: {digit}" for letter, digit in letter_to_digit.items()))
            print(f"SEND = {int(''.join(str(letter_to_digit[char]) for char in 'SEND'))}")
            print(f"MORE = {int(''.join(str(letter_to_digit[char]) for char in 'MORE'))}")
            print(f"MONEY = {int(''.join(str(letter_to_digit[char]) for char in 'MONEY'))}")
            return
```

```
    print("No solution exists.")
if __name__ == "__main__":
    solve_cryptarithmetic()
```

OUTPUT:

```
Solution found:
Y: 1 O: 7 S: 6 M: 0 E: 8 D: 3 R: 2 N: 5
SEND = 6853
MORE = 728
MONEY = 7581
```

5) Write the python program for Missionaries Cannibal problem.

PROGRAM:

```
from collections import deque
def is_valid(m_left, c_left, m_right, c_right):
    return (0 <= m_left <= 3 and 0 <= c_left <= 3 and
            0 <= m_right <= 3 and 0 <= c_right <= 3 and
            (m_left == 0 or m_left >= c_left) and
            (m_right == 0 or m_right >= c_right))
def get_neighbors(state):
    m_left, c_left, m_right, c_right, boat = state
    moves = [(2, 0), (1, 1), (0, 2), (1, 0), (0, 1)]
    neighbors = []
    if boat:
        for m, c in moves:
            if is_valid(m_left - m, c_left - c, m_right + m, c_right + c):
                neighbors.append((m_left - m, c_left - c, m_right + m, c_right + c, False))
    else:
        for m, c in moves:
            if is_valid(m_left + m, c_left + c, m_right - m, c_right - c):
                neighbors.append((m_left + m, c_left + c, m_right - m, c_right - c, True))
    return neighbors
def bfs_solver():
    start = (3, 3, 0, 0, True)
    goal = (0, 0, 3, 3, False)
```

```
    queue = deque([(start, [])])
    visited = set([start])
    while queue:
        (state, path) = queue.popleft()
        if state == goal:
            return path
        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [neighbor]))
    return None
def print_solution(path):
    if path is None:
        print("No solution exists.")
        return
    for step in path:
        m_left, c_left, m_right, c_right, boat = step
        print(f"Left: {m_left}M, {c_left}C | Right: {m_right}M, {c_right}C | Boat: {'Left' if boat else
'Right'}")
if __name__ == "__main__":
    path = bfs_solver()
    print_solution(path)
```

OUTPUT:

```
Left: 2M, 2C | Right: 1M, 1C | Boat: Right
Left: 3M, 2C | Right: 0M, 1C | Boat: Left
Left: 3M, 0C | Right: 0M, 3C | Boat: Right
Left: 3M, 1C | Right: 0M, 2C | Boat: Left
Left: 1M, 1C | Right: 2M, 2C | Boat: Right
Left: 2M, 2C | Right: 1M, 1C | Boat: Left
Left: 0M, 2C | Right: 3M, 1C | Boat: Right
Left: 0M, 3C | Right: 3M, 0C | Boat: Left
Left: 0M, 1C | Right: 3M, 2C | Boat: Right
Left: 1M, 1C | Right: 2M, 2C | Boat: Left
Left: 0M, 0C | Right: 3M, 3C | Boat: Right
```