TOPIC-6: BACK TRACKING

Program 1: N-Queens Problem

Aim:

To place N queens on an N×N chessboard such that no two queens attack each other using backtracking.

Algorithm:

1. Start the program.
2. Input the number of queens N.
3. Create an N×N board initialized with zeros.
4. Define a function is_safe() to check column and diagonal safety.
5. Place a queen in a column if safe, and move to the next row.
6. If all queens are placed, print the solution.
7. If placement not possible, backtrack.
8. Stop the program.

```python
def is_safe(board, row, col, n):
    for i in range(row):
        if board[i] == col or abs(board[i] - col) == abs(i - row):
            return False
    return True

def solve_nqueens(board, row, n):
    if row == n:
        print(board)
        return
    for col in range(n):
        if is_safe(board, row, col, n):
            board[row] = col
            solve_nqueens(board, row + 1, n)

n = int(input("Enter number of queens: "))
board = [-1]*n
solve_nqueens(board, 0, n)
```

```
Enter number of queens: 4
[1, 3, 0, 2]
[2, 0, 3, 1]

=== Code Execution Successful ===
```

Program 2: Generalized N-Queens

Aim:

To solve the N-Queens problem for any N value and different board sizes or restrictions.

Algorithm:

1. Start the program.
2. Input board dimensions and obstacles (if any).
3. Initialize the board with zeros or ‘X’ for blocked cells.
4. Use a recursive function to place queens safely.
5. Check rows, columns, and diagonals before placing.

6. If valid configuration found, display it.
7. Stop the program.

main.py

```python
def is_safe(board, row, col):
    for i in range(row):
        if board[i] == col or abs(board[i] - col) == abs(i - row):
            return False
    return True

def solve(board, row, n):
    if row == n:
        print(board)
        return
    for col in range(n):
        if is_safe(board, row, col):
            board[row] = col
            solve(board, row + 1, n)

n = int(input("Enter board size: "))
board = [-1]*n
solve(board, 0, n)
```

Output

```
Enter board size: 5
[0, 2, 4, 1, 3]
[0, 3, 1, 4, 2]
[1, 3, 0, 2, 4]
[1, 4, 2, 0, 3]
[2, 0, 3, 1, 4]
[2, 4, 1, 3, 0]
[3, 0, 2, 4, 1]
[3, 1, 4, 2, 0]
[4, 1, 3, 0, 2]
[4, 2, 0, 3, 1]

=== Code Execution Successful ===
```

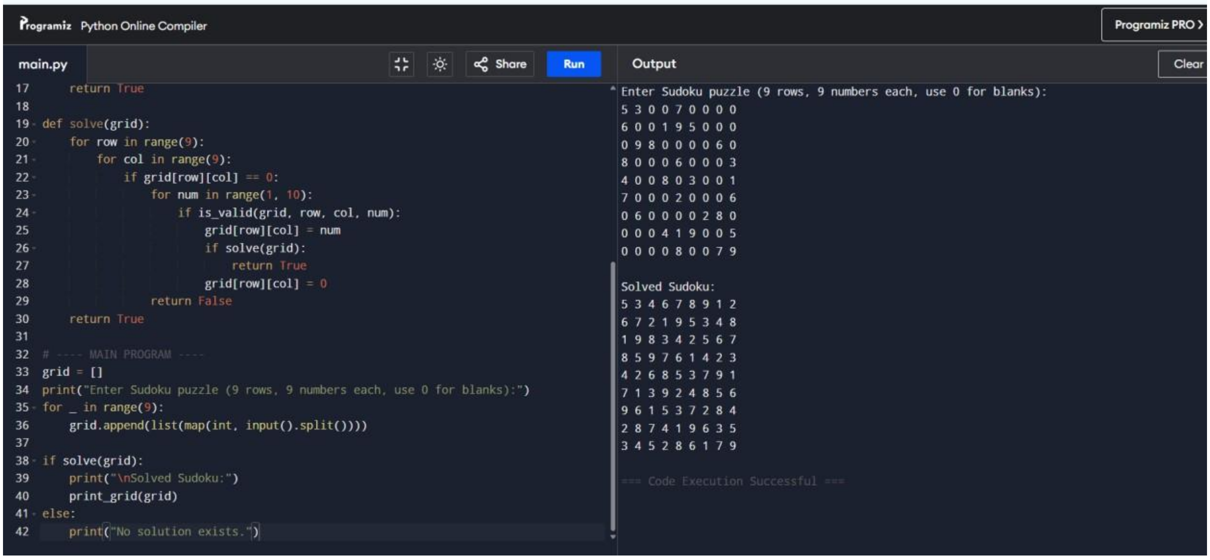
Program 3: Sudoku Solver

Aim:

To solve a 9×9 Sudoku puzzle using backtracking.

Algorithm:

1. Start the program.
2. Input the 9×9 Sudoku grid.
3. Find the first empty cell.
4. Try numbers 1–9 sequentially.
5. If the number is valid in row, column, and subgrid, place it.
6. Recursively solve for the next cell.
7. If no valid number, backtrack.
8. Display the solved Sudoku.
9. Stop the program.



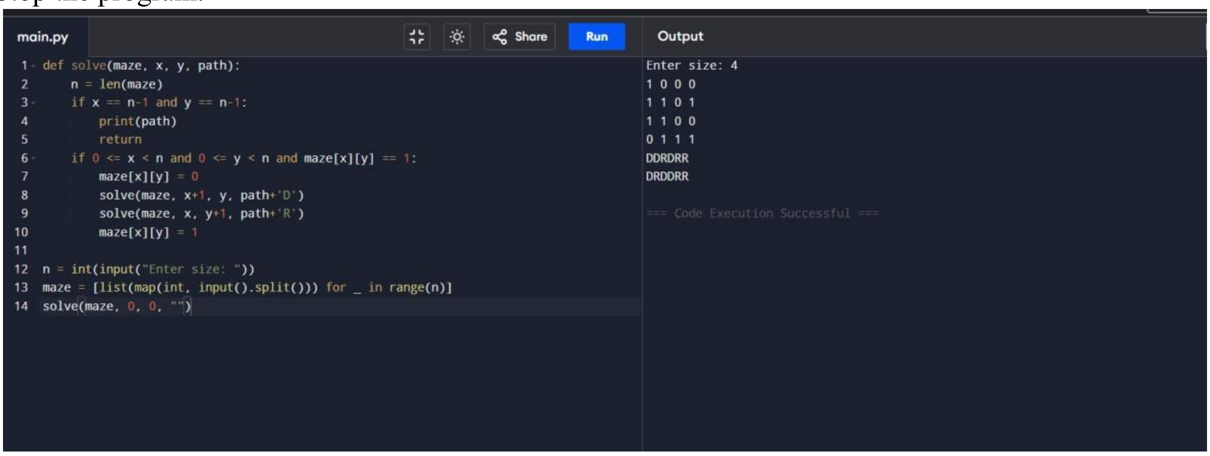
Program 4: Rat in a Maze

Aim:

To find all possible paths for a rat to reach the destination using backtracking.

Algorithm:

1. Start the program.
2. Input the maze as a matrix (1 for open path, 0 for blocked).
3. Start from (0,0) position.
4. Move in allowed directions (down, right, up, left).
5. Mark visited cells to avoid repetition.
6. If destination reached, record the path.
7. Backtrack to explore new paths.
8. Stop the program.



Program 5: Knight's Tour Problem

Aim:

To find a sequence of moves for a knight to visit every cell on a chessboard exactly once.

Algorithm:

1. Start the program.
2. Input the size of the chessboard (N×N).
3. Initialize the board with -1.
4. Define all possible knight moves.
5. Place the knight at (0,0).
6. Recursively try all valid moves.
7. If all cells are visited, print the solution.
8. If not, backtrack and try another path.
9. Stop the program.

main.py

```python
def is_safe(x, y, board, n):
    return 0 <= x < n and 0 <= y < n and board[x][y] == -1

def solve(x, y, movei, board, xmove, ymove, n):
    if movei == n*n:
        for r in board:
            print(r)
        print()
        return
    for k in range(8):
        nx, ny = x + xmove[k], y + ymove[k]
        if is_safe(nx, ny, board, n):
            board[nx][ny] = movei
            solve(nx, ny, movei+1, board, xmove, ymove, n)
            board[nx][ny] = -1

n = int(input("Enter board size: "))
board = [[-1]*n for _ in range(n)]
xmove = [2,1,-1,-2,-2,-1,1,2]
ymove = [1,2,2,1,-1,-2,-2,-1]
board[0][0] = 0
solve(0,0,1,board,xmove,ymove,n)
```

Output

```
Enter board size: 5
[0, 5, 14, 9, 20]
[13, 8, 19, 4, 15]
[18, 1, 6, 21, 10]
[7, 12, 23, 16, 3]
[24, 17, 2, 11, 22]

[0, 5, 10, 17, 20]
[11, 16, 19, 4, 9]
[6, 1, 14, 21, 18]
[15, 12, 23, 8, 3]
[24, 7, 2, 13, 22]

[0, 5, 10, 15, 20]
[11, 14, 19, 4, 9]
[6, 1, 12, 21, 16]
[13, 18, 23, 8, 3]
[24, 7, 2, 17, 22]

[0, 5, 16, 11, 20]
[15, 10, 19, 4, 17]
[6, 1, 8, 21, 12]
[9, 14, 23, 18, 3]
[24, 7, 2, 13, 22]
```