

## TOPIC-6: BACK TRACKING

### Program 1: N-Queens Problem

#### Aim:

To place N queens on an  $N \times N$  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  $N \times N$  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.

```
1 - def is_safe(board, row, col, n):
2 -     for i in range(row):
3 -         if board[i] == col or abs(board[i] - col) == abs(i - row):
4 -             return False
5 -     return True
6
7 - def solve_nqueens(board, row, n):
8 -     if row == n:
9 -         print(board)
10    return
11   for col in range(n):
12     if is_safe(board, row, col, n):
13         board[row] = col
14         solve_nqueens(board, row + 1, n)
15
16 n = int(input("Enter number of queens: "))
17 board = [-1]*n
18 solve_nqueens(board, 0, n)
19 |
```

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 | Run | Share | Run | Output
1 def is_safe(board, row, col):
2     for i in range(row):
3         if board[i] == col or abs(board[i] - col) == abs(i - row):
4             return False
5     return True
6
7 def solve(board, row, n):
8     if row == n:
9         print(board)
10        return
11    for col in range(n):
12        if is_safe(board, row, col):
13            board[row] = col
14            solve(board, row + 1, n)
15
16 n = int(input("Enter board size: "))
17 board = [-1]*n
18 solve(board, 0, n)
19
```

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  $9 \times 9$  Sudoku puzzle using backtracking.

#### Algorithm:

1. Start the program.
2. Input the  $9 \times 9$  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.

```

Programiz Python Online Compiler
main.py | Run | Output | Clear
17     return True
18
19- def solve(grid):
20-     for row in range(9):
21-         for col in range(9):
22-             if grid[row][col] == 0:
23-                 for num in range(1, 10):
24-                     if is_valid(grid, row, col, num):
25-                         grid[row][col] = num
26-                         if solve(grid):
27-                             return True
28-                         grid[row][col] = 0
29-     return True
30
31
32 # ----- MAIN PROGRAM -----
33 grid = []
34 print("Enter Sudoku puzzle (9 rows, 9 numbers each, use 0 for blanks):")
35 for _ in range(9):
36     grid.append(list(map(int, input().split())))
37
38 if solve(grid):
39     print("\nSolved Sudoku:")
40     print_grid(grid)
41 else:
42     print("No solution exists.")

```

\* Enter Sudoku puzzle (9 rows, 9 numbers each, use 0 for blanks):  
5 3 0 0 7 0 0 0 0  
6 0 0 1 9 5 0 0 0  
0 9 8 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

Solved Sudoku:  
5 3 4 6 7 8 9 1 2  
6 7 2 1 9 5 3 4 8  
1 9 8 3 4 2 5 6 7  
8 5 9 7 6 1 4 2 3  
4 2 6 8 5 3 7 9 1  
7 1 3 9 2 4 8 5 6  
9 6 1 5 3 7 2 8 4  
2 8 7 4 1 9 6 3 5  
3 4 5 2 8 6 1 7 9

\*\*\* Code Execution Successful \*\*\*

## 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.

```

main.py | Run | Output | Clear
1- def solve(maze, x, y, path):
2-     n = len(maze)
3-     if x == n-1 and y == n-1:
4-         print(path)
5-         return
6-     if 0 <= x < n and 0 <= y < n and maze[x][y] == 1:
7-         maze[x][y] = 0
8-         solve(maze, x+1, y, path+'D')
9-         solve(maze, x, y+1, path+'R')
10        maze[x][y] = 1
11
12 n = int(input("Enter size: "))
13 maze = [list(map(int, input().split())) for _ in range(n)]
14 solve(maze, 0, 0, "")

```

Enter size: 4  
1 0 0 0  
1 1 0 1  
1 1 0 0  
0 1 1 1  
D R D R R  
D R D D R

\*\*\* Code Execution Successful \*\*\*

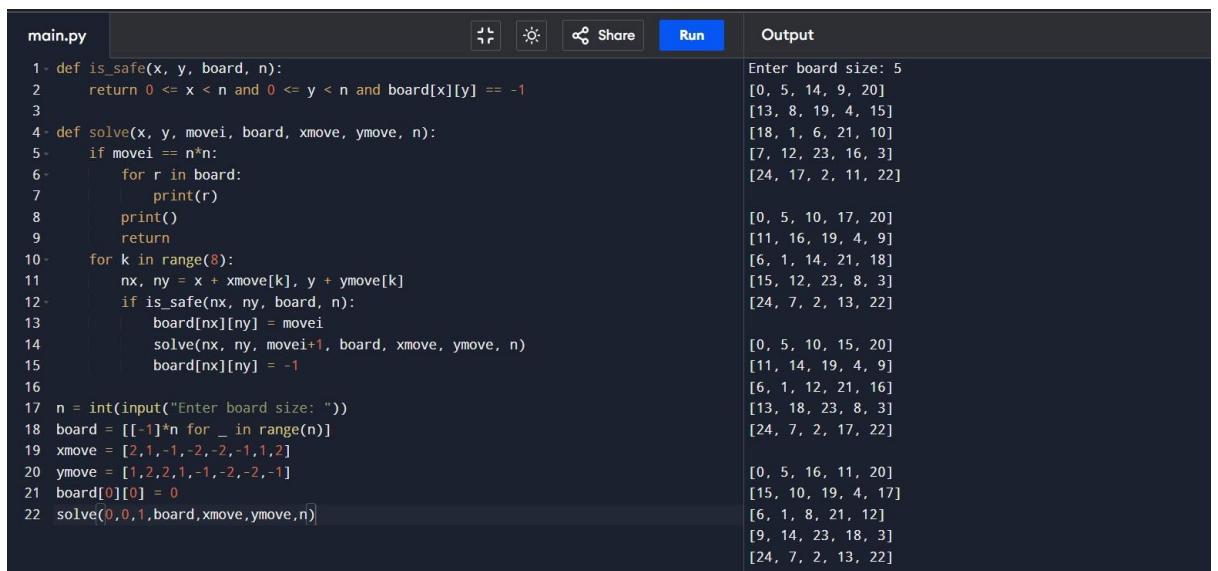
## Program 5: Knight's Tour Problem

### 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 ( $N \times N$ ).
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.



The screenshot shows a code editor interface with a tab labeled "main.py". The code is a Python script for solving the Knight's Tour problem. It includes a function to check if a move is safe and a recursive function to solve the tour. The output window shows the board size entered as 5 and a sequence of moves from (0,0) to (4,4) in a 5x5 grid.

```
main.py
1- def is_safe(x, y, board, n):
2-     return 0 <= x < n and 0 <= y < n and board[x][y] == -1
3
4- def solve(x, y, movei, board, xmove, ymove, n):
5-     if movei == n*n:
6-         for r in board:
7-             print(r)
8-         print()
9-         return
10-    for k in range(8):
11-        nx, ny = x + xmove[k], y + ymove[k]
12-        if is_safe(nx, ny, board, n):
13-            board[nx][ny] = movei
14-            solve(nx, ny, movei+1, board, xmove, ymove, n)
15-            board[nx][ny] = -1
16
17- n = int(input("Enter board size: "))
18- board = [[-1]*n for _ in range(n)]
19- xmove = [2, 1, -1, -2, -2, -1, 1, 2]
20- ymove = [1, 2, 2, 1, -1, -2, -2, -1]
21- board[0][0] = 0
22- 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]