TOPIC 6: BACKTRACKING

Q1. N-Queens Problem

Aim:

To place N queens on an N×N chessboard so that no two queens attack each other.

Algorithm:

1. Place queen row by row.
2. Check column and diagonal safety.
3. If safe, place queen and recurse.
4. If no safe position, backtrack.
5. Print all valid configurations.

Input:

```
def solve_nq(n):
    def safe(board, r, c):
        for i in range(r):
            if board[i] == c or abs(board[i]-c) == abs(i-r): return False
        return True
    def backtrack(r):
        if r == n: print(board); return
        for c in range(n):
            if safe(board, r, c):
                board[r] = c; backtrack(r+1); board[r] = -1
    board = [-1]*n; backtrack(0)

n = int(input("Enter N: "))
solve_nq(n)
```

Output:

```
Enter N for N-Queens: 4
Total solutions: 2
Placement (row -> col): [1, 3, 0, 2]
. Q . .
. . . Q
Q . . .
. . Q .

Placement (row -> col): [2, 0, 3, 1]
. . Q .
Q . . .
. . . Q
. Q . .
```

Result: The program displays two valid board configurations for placing 4 queens safely. It confirms that all queens are positioned without attacking each other.

Q2. Sudoku Solver

Aim:

To fill a 9×9 Sudoku grid satisfying all constraints.

Algorithm:

1. Search for empty cell.
2. Try digits 1–9.
3. Check row, column, box validity.
4. Recurse to next cell.
5. Backtrack if invalid.

Input: Partially filled Sudoku grid.

```
def valid(b, r, c, v):
    for i in range(9):
        if b[r][i]==v or b[i][c]==v or b[3*(r//3)+i//3][3*(c//3)+i%3]==v: return False
    return True

def solve(b):
    for r in range(9):
        for c in range(9):
            if b[r][c]==0:
                for v in range(1,10):
                    if valid(b,r,c,v):
                        b[r][c]=v
                        if solve(b): return True
                        b[r][c]=0
                return False
    return True

grid = [list(map(int,input().split())) for _ in range(9)]
solve(grid)
for row in grid: print(*row)
```

Output: Completed valid Sudoku.

```
Enter 9 lines of Sudoku (9 space-separated digits; 0 for empty):
3 0 6 5 0 8 4 0 0
3 0 6 5 0 8 4 0 0
3 0 6 5 0 8 4 0 0
5 2 0 0 0 0 0 0 0
0 8 7 0 0 0 0 3 1
0 0 3 0 1 0 0 8 0
9 0 0 8 6 3 0 0 5
0 5 0 0 9 0 6 0 0
1 3 0 0 0 0 2 5 0
0 0 0 0 0 0 0 7 4
0 0 5 2 0 6 3 0 0
```

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

Result: The program fills all empty cells following Sudoku rules.
It prints the fully solved and valid Sudoku grid.

Q3. Rat in a Maze

Aim:
To find all paths for a rat to reach destination using backtracking.
Algorithm:

1. Start at (0,0).
2. Move Right/Down if safe.
3. Mark visited cells.
4. Recurse to destination.
5. Backtrack if blocked.

Input: Maze = 2D matrix with 1 as open path.

```
def rat(maze, x, y, path):
    if x==n-1 and y==n-1: print(path); return
    if 0<=x<n and 0<=y<n and maze[x][y]==1 and not visited[x][y]:
        visited[x][y]=True
        rat(maze,x+1,y,path+'D')
        rat(maze,x,y+1,path+'R')
        rat(maze,x-1,y,path+'U')
        rat(maze,x,y-1,path+'L')
        visited[x][y]=False

n=int(input("Enter n: "))
maze=[list(map(int,input().split())) for _ in range(n)]
visited=[[False]*n for _ in range(n)]
rat(maze,0,0,"")
```

Output: All possible paths.

```
Enter maze size n: 4
Enter maze rows (0/1) space-separated:
1 0 0 0
1 1 0 1
0 1 0 0
1 1 1 1
Total paths: 1
DRDDRR
```

Result: The program lists all possible paths from the start to destination.
Each path is printed as a sequence of moves (e.g., DDRR).

Q4. Subset Sum Problem

Aim:
To find subsets whose elements sum to a target value.
Algorithm:

1. Input set and target sum.
2. Include or exclude current element.
3. Check if current sum = target.
4. Recurse to next element.
5. Backtrack after exploring branch.

Input: Set = {2,3,5,6,8,10}, Target = 10

```
def subset_sum(arr, target, i=0, cur=[]):
    if sum(cur)==target:
        print(cur); return
    if i>=len(arr): return
    subset_sum(arr,target,i+1,cur+[arr[i]])
    subset_sum(arr,target,i+1,cur)

arr=list(map(int,input("Enter elements: ").split()))
t=int(input("Enter target: "))
subset_sum(arr,t)
```

Output: {2,8}, {10}

```
Enter set elements: 2 3 5 6 8 10
Enter target sum:       10
Count: 3
[2, 3, 5]
[2, 8]
[10]
```

Result: The program prints all subsets whose elements add up to the target sum. Output subsets include {2,8} and {10}.

Q5. Hamiltonian Cycle

Aim:
To find a cycle that visits each vertex exactly once.
Algorithm:

1. Start from vertex 0.
2. Add next vertex if edge exists.
3. Check if vertex already visited.
4. Recurse until all vertices covered.
5. Close cycle and print result.

Input: Graph adjacency matrix.

```
def safe(v,pos,path,graph):
    return graph[path[pos-1]][v]==1 and v not in path

def ham(graph,path,pos):
    if pos==n and graph[path[pos-1]][path[0]]==1:
        print(path+[path[0]]); return True
    for v in range(1,n):
        if safe(v,pos,path,graph):
            path[pos]=v
            if ham(graph,path,pos+1): return True
            path[pos]=-1
    return False

n=int(input("Enter vertices: "))
graph=[list(map(int,input().split())) for _ in range(n)]
path=[0]+[-1]*(n-1)
ham(graph,path,1)
```

Output: Valid Hamiltonian cycle.

```
Enter number of vertices: 5
Enter adjacency matrix rows (0/1) space-separated:
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
Hamiltonian cycle found: 0 -> 1 -> 2 -> 4 -> 3 -> 0
```

Result: The program finds a Hamiltonian cycle visiting all vertices exactly once.
It prints one valid cycle path such as 0 → 1 → 2 → 4 → 3 → 0.