TOPIC 6:BACK TRACKING PROBLEM

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.

```
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 ===
```

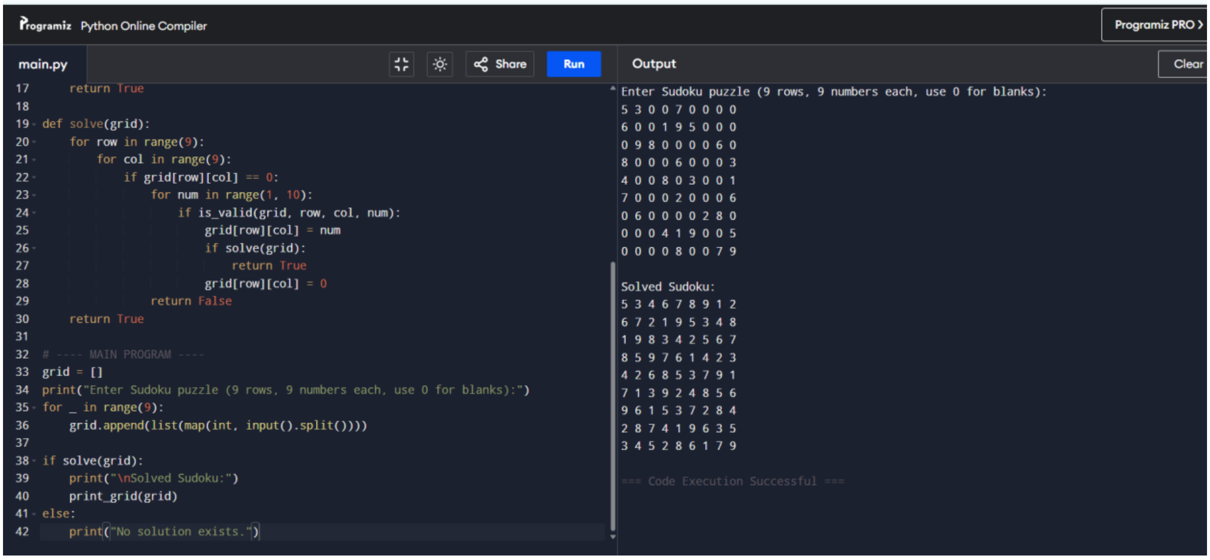
Program 3: Sudoku Solver

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.



Programiz Python Online Compiler

main.py

```python
    return True

def solve(grid):
    for row in range(9):
        for col in range(9):
            if grid[row][col] == 0:
                for num in range(1, 10):
                    if is_valid(grid, row, col, num):
                        grid[row][col] = num
                        if solve(grid):
                            return True
                        grid[row][col] = 0
                return False
    return True

# ---- MAIN PROGRAM ----
grid = []
print("Enter Sudoku puzzle (9 rows, 9 numbers each, use 0 for blanks):")
for _ in range(9):
    grid.append(list(map(int, input().split())))

if solve(grid):
    print("\nSolved Sudoku:")
    print_grid(grid)
else:
    print("No solution exists.")
```

Output

```
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 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 ===
```

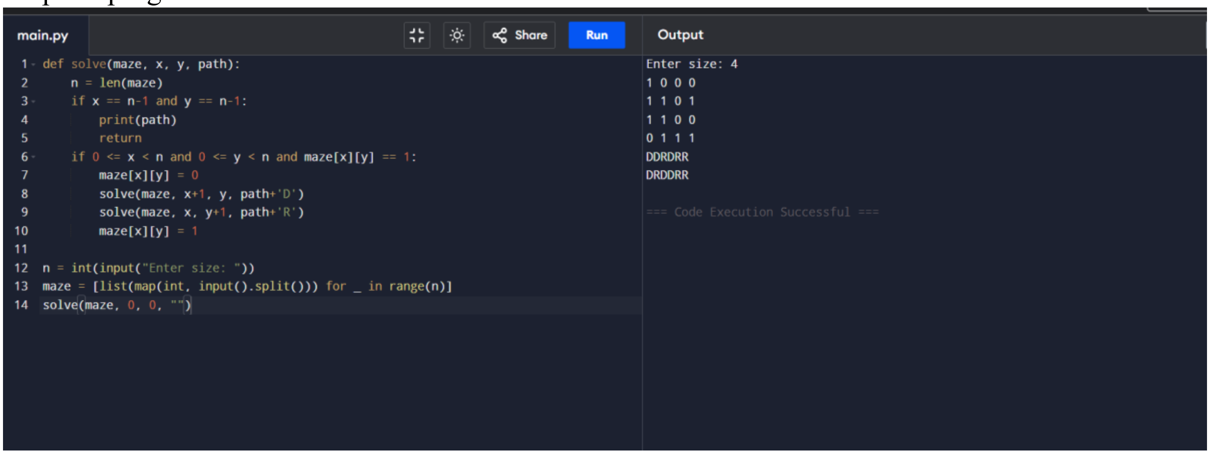
Program 4: Rat in a Maze

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

```python
def solve(maze, x, y, path):
    n = len(maze)
    if x == n-1 and y == n-1:
        print(path)
        return
    if 0 <= x < n and 0 <= y < n and maze[x][y] == 1:
        maze[x][y] = 0
        solve(maze, x+1, y, path+'D')
        solve(maze, x, y+1, path+'R')
        maze[x][y] = 1

n = int(input("Enter size: "))
maze = [list(map(int, input().split())) for _ in range(n)]
solve(maze, 0, 0, "")
```

Output

```
Enter size: 4
1 0 0 0
1 1 0 1
1 1 0 0
0 1 1 1
DDRDRR
DRDDRR

=== 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×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]
```

Program 6: Peak Element (Divide and Conquer)

Aim:

To find a peak element in an array using the divide and conquer technique.

Algorithm:

1. Start the program.
2. Input the size and elements of the array.
3. Find the middle index.

4. If the middle element is greater than or equal to both neighbors, print it as peak.
5. If the left neighbor is greater, search left subarray.
6. Otherwise, search right subarray.
7. Stop the program.

main.py

```python
def find_peak(arr, low, high, n):
    mid = (low + high)//2
    if (mid==0 or arr[mid-1]<=arr[mid]) and (mid==n-1 or arr[mid+1]<=arr[mid]):
        return arr[mid]
    elif mid>0 and arr[mid-1]>arr[mid]:
        return find_peak(arr, low, mid-1, n)
    else:
        return find_peak(arr, mid+1, high, n)

arr = list(map(int, input("Enter array: ").split()))
print("Peak element:", find_peak(arr, 0, len(arr)-1, len(arr)))
```

Output

```
Enter array: 1 3 2 4 1 0
Peak element: 3

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

Program 7: Merge Sort

Aim:

To sort elements of an array using the merge sort algorithm.

Algorithm:

1. Start the program.
2. Input array elements.
3. Divide the array into two halves.
4. Recursively sort both halves.
5. Merge the sorted halves into a single sorted list.
6. Display the sorted array.
7. Stop the program.

```
def merge_sort(arr):
    if len(arr)>1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]
        merge_sort(L)
        merge_sort(R)
        i=j=k=0
        while i<len(L) and j<len(R):
            if L[i]<R[j]:
                arr[k]=L[i]; i+=1
            else:
                arr[k]=R[j]; j+=1
            k+=1
        arr[k:]=L[i:]+R[j:]
arr = list(map(int, input("Enter numbers: ").split()))
merge_sort(arr)
print("Sorted:", arr)
```

```
Enter numbers: 3 9 6 4 2
Sorted: [2, 3, 4, 6, 9]

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

Program 8: Quick Sort

Aim:

To sort elements of an array using the quick sort algorithm.

Algorithm:

1. Start the program.
2. Input array elements.
3. Select a pivot element.
4. Partition the array into elements smaller and greater than pivot.
5. Recursively apply quick sort to each subarray.
6. Combine the results.
7. Display the sorted array.
8. Stop the program.

main.py Share Run

```
def quick_sort(arr):
    if len(arr)<=1:
        return arr
    pivot = arr[0]
    left = [x for x in arr[1:] if x<=pivot]
    right = [x for x in arr[1:] if x>pivot]
    return quick_sort(left)+[pivot]+quick_sort(right)

arr = list(map(int, input("Enter numbers: ").split()))
print("Sorted:", quick_sort(arr))
```

Output Clear

```
Enter numbers: 4 2 9 7 1
Sorted: [1, 2, 4, 7, 9]

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

Program 9: Tower of Hanoi

Aim:

To move N disks from source peg to destination peg using an auxiliary peg.

Algorithm:

1. Start the program.
2. Input the number of disks N.
3. If only one disk, move it directly.
4. Move N–1 disks to auxiliary peg.
5. Move last disk to destination.
6. Move N–1 disks from auxiliary to destination.
7. Stop the program.

```python
def hanoi(n, src, aux, dest):
    if n==1:
        print(f"Move disk 1 from {src} to {dest}")
        return
    hanoi(n-1, src, dest, aux)
    print(f"Move disk {n} from {src} to {dest}")
    hanoi(n-1, aux, src, dest)

n = int(input("Enter number of disks: "))
hanoi(n, 'A', 'B', 'C')
```

```
Enter number of disks: 3
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C

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

Program 10: Subset Sum Problem

Aim:

To find subsets that sum to a given target using recursion and backtracking.

Algorithm:

1. Start the program.
2. Input array elements and target sum.
3. Recursively include or exclude each element.
4. Keep track of current sum.
5. If sum equals target, print the subset.
6. Backtrack and try other combinations.

7. Stop the program

main.py

```python
def subset_sum(arr, target, subset=[], index=0):
    if sum(subset) == target:
        print(subset)
        return
    if sum(subset) > target or index == len(arr):
        return
    subset_sum(arr, target, subset + [arr[index]], index + 1)
    subset_sum(arr, target, subset, index + 1)

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

Output

```
Enter numbers: 3 4 5 2
Enter target: 7
[3, 4]
[5, 2]

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

.

Program 11: Permutations of a String

Aim:

To generate all possible permutations of a given string using recursion.

Algorithm:

1. Start the program.
2. Input the string.
3. Fix one character and recursively find permutations of remaining characters.
4. Swap characters to explore all positions.
5. Print all permutations.
6. Stop the program.

main.py

```python
def permute(s, l, r):
    if l == r:
        print(''.join(s))
    else:
        for i in range(l, r+1):
            s[l], s[i] = s[i], s[l]
            permute(s, l+1, r)
            s[l], s[i] = s[i], s[l]

s = list(input("Enter string: "))
permute(s, 0, len(s)-1)
```

Output

```
Enter string: abc
abc
acb
bac
bca
cba
cab

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

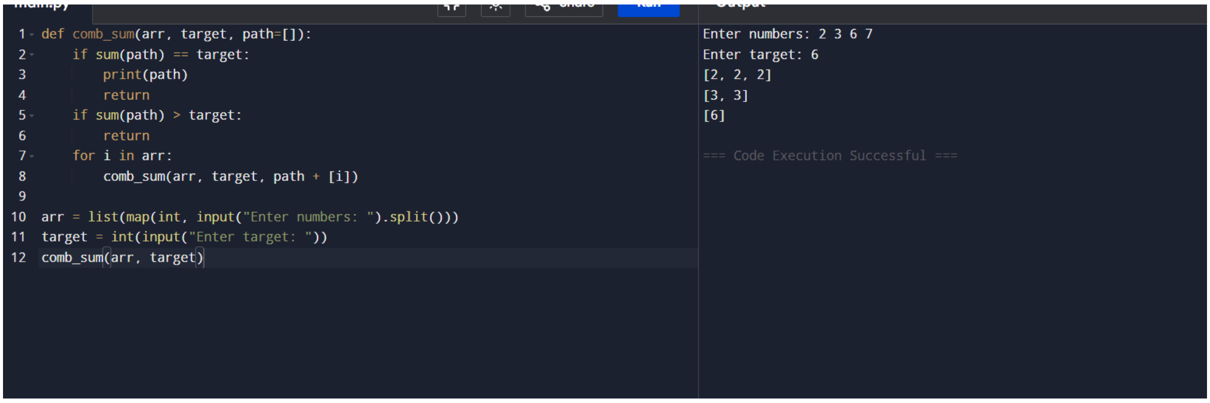
Program 12: Combination Sum

Aim:

To find all combinations of numbers that add up to a target using recursion.

Algorithm:

1. Start the program.
2. Input list of numbers and target sum.
3. Use recursion to include or exclude each number.
4. If sum equals target, print the combination.
5. Backtrack to explore new possibilities.
6. Stop the program.



Program 13: Hamiltonian Cycle

Aim:

To find a Hamiltonian cycle in a given graph using backtracking.

Algorithm:

1. Start the program.
2. Input number of vertices and adjacency matrix.
3. Choose a starting vertex.
4. Try adding next vertex if it is connected and not already visited.
5. If all vertices are included and last vertex connects to first, print cycle.
6. Else, backtrack.
7. Stop the program.

main.py

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

def ham_cycle(graph, path, pos):
    n = len(graph)
    if pos==n:
        if graph[path[pos-1]][path[0]]==1:
            print(path+[path[0]])
        return
    for v in range(1,n):
        if is_valid(v,pos,path,graph):
            path[pos]=v
            ham_cycle(graph,path,pos+1)
            path[pos]=-1

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

Output

```
Enter number of vertices: 4
0 1 1 1
1 0 1 0
1 1 0 1
1 0 1 0

[0, 1, 2, 3, 0]
[0, 3, 2, 1, 0]

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

Program 14: Traveling Salesman Problem (TSP)

Aim:

To find the shortest possible route visiting all cities exactly once and returning to the start using backtracking.

Algorithm:

1. Start the program.
2. Input number of cities and cost matrix.
3. Start from the first city.
4. Visit each unvisited city recursively and calculate path cost.
5. Keep track of the minimum cost path.
6. Display the shortest route and cost.
7. Stop the program.

main.py

```python
from itertools import permutations

n = int(input("Enter number of cities: "))
graph = [list(map(int,input().split())) for _ in range(n)]
min_cost = float('inf')
cities = range(n)

for perm in permutations(cities[1:]):
    cost = 0
    k = 0
    for j in perm:
        cost += graph[k][j]
        k = j
    cost += graph[k][0]
    min_cost = min(min_cost, cost)
print("Minimum cost:", min_cost)
```

Output

```
Enter number of cities: 4
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0

Minimum cost: 80

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

Program 15: Subset Generation

Aim:

To generate all possible subsets of a given set using recursion.

Algorithm:

1. Start the program.
2. Input the set elements.
3. For each element, choose to include or exclude it.
4. Recursively generate all combinations.
5. Print each subset.
6. Stop the program.

```python
def subsets(arr, curr=[]):
    if arr == []:
        print(curr)
        return
    subsets(arr[1:], curr + [arr[0]])
    subsets(arr[1:], curr)

arr = list(map(int, input("Enter numbers: ").split()))
subsets(arr)
```

```
Enter numbers: 1 2 3 7
[1, 2, 3, 7]
[1, 2, 3]
[1, 2, 7]
[1, 2]
[1, 3, 7]
[1, 3]
[1, 7]
[1]
[2, 3, 7]
[2, 3]
[2, 7]
[2]
[3, 7]
[3]
[7]
[]

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

Program 16: String Permutation (Backtracking)

Aim:

To generate all permutations of a given string using backtracking.

Algorithm:

1. Start the program.
2. Input the string.
3. Define a recursive function that swaps each character.
4. Recurse for next positions.
5. Print the permutation when all characters are fixed.
6. Stop the program.

```python
def permute(s, l, r):
    if l==r:
        print(''.join(s))
    else:
        for i in range(l,r+1):
            s[l],s[i]=s[i],s[l]
            permute(s,l+1,r)
            s[l],s[i]=s[i],s[l]

s = list(input("Enter string: "))
permute(s,0,len(s)-1)
```

```
Enter string: sad
sad
sda
asd
ads
das
dsa

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

Program 17: Universal String Problem

Aim:

To find if a string contains all binary codes of length k.

Algorithm:

1. Start the program.
2. Input the binary string and integer k.
3. Generate all binary codes of length k.
4. Check if each binary code exists as a substring.
5. If all exist, return True; else False.
6. Stop the program.

main.py | Share | Run | Output

```python
def has_all_codes(s, k):
    codes = {s[i:i+k] for i in range(len(s)-k+1)}
    return len(codes) == 2**k

s = input("Enter binary string: ")
k = int(input("Enter k: "))
print("Contains all codes:", has_all_codes(s, k))
```

```
Enter binary string: 4
Enter k: 2
Contains all codes: False

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