# Experiment-6

**Aim:**

To implement N Queens problem using backtracking

**Description:**

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return **false**.

**Algorithm:**

- **Initialize the Board**: Create an N×N chessboard initialized to 0, where 0 indicates an empty square.
- **Define the Backtracking Function**:
- **Input**: The current row index.
- **Base Case**: If the row index equals N, a valid arrangement has been found. Print or store the board configuration.
- **Iterate through Columns**:
    - For each column in the current row, check if placing a queen there is safe (i.e., check if it's not attacked by other queens).
    - If safe, place the queen (mark the board) and recursively call the backtracking function for the next row.
    - After returning from recursion, remove the queen (backtrack) and mark the board as empty.
- **Safety Check**: Implement a function to check if placing a queen at (row, col) is safe:
- Check the column for other queens.
- Check the upper-left diagonal.
- Check the upper-right diagonal.

**Program:**

```
global N
N = 4

def printSolution(board):
    for i in range(N):
        for j in range(N):
            if board[i][j] == 1:
                print("Q",end=" ")
            else:
                print(".",end=" ")
        print()
```

```python
def isSafe(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, N, 1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True

def solveNQUtil(board, col):
    if col >= N:
        return True
    for i in range(N):
        if isSafe(board, i, col):
            board[i][col] = 1
            if solveNQUtil(board, col + 1) == True:
                return True
            board[i][col] = 0
    return False

def solveNQ():
    board = [[0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0]]

    if solveNQUtil(board, 0) == False:
        print("Solution does not exist")
        return False

    printSolution(board)
    return True

# Driver Code
if __name__ == '__main__':
    solveNQ()

```

**Output:**

```
PS C:\Users\CBIT\Desktop\04> & C:/Users/CBIT/AppData/Local/Programs/Python/Python312/python.exe "c:/l
. . Q .
Q . . .
. . . Q
. Q . .
PS C:\Users\CBIT\Desktop\04>
```

## Experiment-7

**Aim:**

Implement graph coloring problem using backtracking.

**Description:**

**Graph coloring** refers to the problem of **coloring vertices** of a graph in such a way that **no two adjacent** vertices have the **same color**. This is also called the **vertex coloring** problem. If coloring is done using at most m colors, it is called m-coloring.

**Algorithm:**

- Create a recursive function that takes the graph, current index, number of vertices, and color array.
- If the current index is equal to the number of vertices. Print the color configuration in the color array.
- Assign a color to a vertex from the range (1 to m).
    - For every assigned color, check if the configuration is safe, (i.e. check if the adjacent vertices do not have the same color) and recursively call the function with the next index and number of vertices else return **false**
    - If any recursive function returns **true** then break the loop and return true
    - If no recursive function returns **true** then return **false**

**Program:**

```python
V = 4

def print_solution(color):
    print("Solution Exists: Following are the assigned colors")
    print(" ".join(map(str, color)))

def is_safe(v, graph, color, c):
    for i in range(V):
        if graph[v][i] and c == color[i]:
            return False
    return True

def graph_coloring_util(graph, m, color, v):
    if v == V:
        return True
    for c in range(1, m + 1):
        if is_safe(v, graph, color, c):
            color[v] = c
            if graph_coloring_util(graph, m, color, v + 1):
                return True
            color[v] = 0
    return False
```

```python
def graph_coloring(graph, m):
    color = [0] * V
    if not graph_coloring_util(graph, m, color, 0):
        print("Solution does not exist")
        return False
    print_solution(color)
    return True

if __name__ == "__main__":
    graph = [
        [0, 1, 1, 1],
        [1, 0, 1, 0],
        [1, 1, 0, 1],
        [1, 0, 1, 0],
    ]

    m = 3
    graph_coloring(graph, m)
```

**Output:**

```
PS C:\Users\CBIT\Desktop\04> & C:/Users/CBIT/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/C
Solution Exists: Following are the assigned colors
1 2 3 2
```

## Experiment – 8

**Aim:**

To implement Hamiltonian Cycle using backtracking.

**Description:**

Hamiltonian Path in a graph **G** is a path that visits every vertex of G exactly once and Hamiltonian Path doesn't have to return to the starting vertex. It's an open path.

**Algorithm:**

Create an empty path array and add vertex **0** to it. Add other vertices, starting from the vertex **1**. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return **false**.

**Program:**

```
class Graph():
    def __init__(self, vertices):
        self.graph = [[0 for column in range(vertices)]
                          for row in range(vertices)]
        self.V = vertices

    def isSafe(self, v, pos, path):
        if self.graph[ path[pos-1] ][v] == 0:
            return False
        for vertex in path:
            if vertex == v:
                return False
        return True

    def hamCycleUtil(self, path, pos):
        if pos == self.V:
            if self.graph[ path[pos-1] ][ path[0] ] == 1:
                return True
            else:
                return False
        for v in range(1,self.V):
            if self.isSafe(v, pos, path) == True:
                path[pos] = v
                if self.hamCycleUtil(path, pos+1) == True:
                    return True
                path[pos] = -1
        return False
```

```
    def hamCycle(self):
        path = [-1] * self.V
        path[0] = 0
        if self.hamCycleUtil(path,1) == False:
            print ("Solution does not exist\n")
            return False
        self.printSolution(path)
        return True

    def printSolution(self, path):
        print ("Solution Exists: Following",
                 "is one Hamiltonian Cycle")
        for vertex in path:
            print (vertex )

g1 = Graph(5)
g1.graph = [ [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], ]

g1.hamCycle();
g2 = Graph(5)
g2.graph = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],
        [0, 1, 0, 0, 1,], [1, 1, 0, 0, 0],
        [0, 1, 1, 0, 0], ]
g2.hamCycle();
```

**Output:**

```
Solution Exists: Following is one Hamiltonian Cycle
Solution Exists: Following is one Hamiltonian Cycle
0
0
1
2
4
4
3
Solution does not exist
```