

Objective: **Implementation and analysis of 4-queen problem**

4-queen problem

It is based on two algorithm :

- 1) NAÏVE ALGORITHM
- 2) BACKTRACKING ALGORITHM

Naïve Algorithm:

```
while there are untried configurations
{
    generate the next configuration
    if queens don't attack in this configuration then
    {
        print this configuration;
    }
}
```

Backtracking Algorithm:

- 1) Start in the leftmost column
- 2) If all queens are placed
return true
- 3) Try all rows in the current column.
Do following for every tried row.
 - a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
 - b) If placing the queen in [row, column] leads to a solution then return true.

- c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
- 4) If all rows have been tried and nothing worked, return false to trigger backtracking.

Code:

```
global N
N = 4
def printSolution(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end=" ")
        print()
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 not solveNQUtil(board, 0):
        print("Solution does not exist")
        return False
    printSolution(board)
    return True
solveNQ()
```

The screenshot shows an IDE with the file '16_4-Queens Problem.py' open. The 'isSafe' function is defined, which checks if a queen can be placed at a given row and column without conflicting with previously placed queens. The function uses nested loops to check the column, upper-left diagonal, and lower-left diagonal. The global variable 'N' is set to 4.

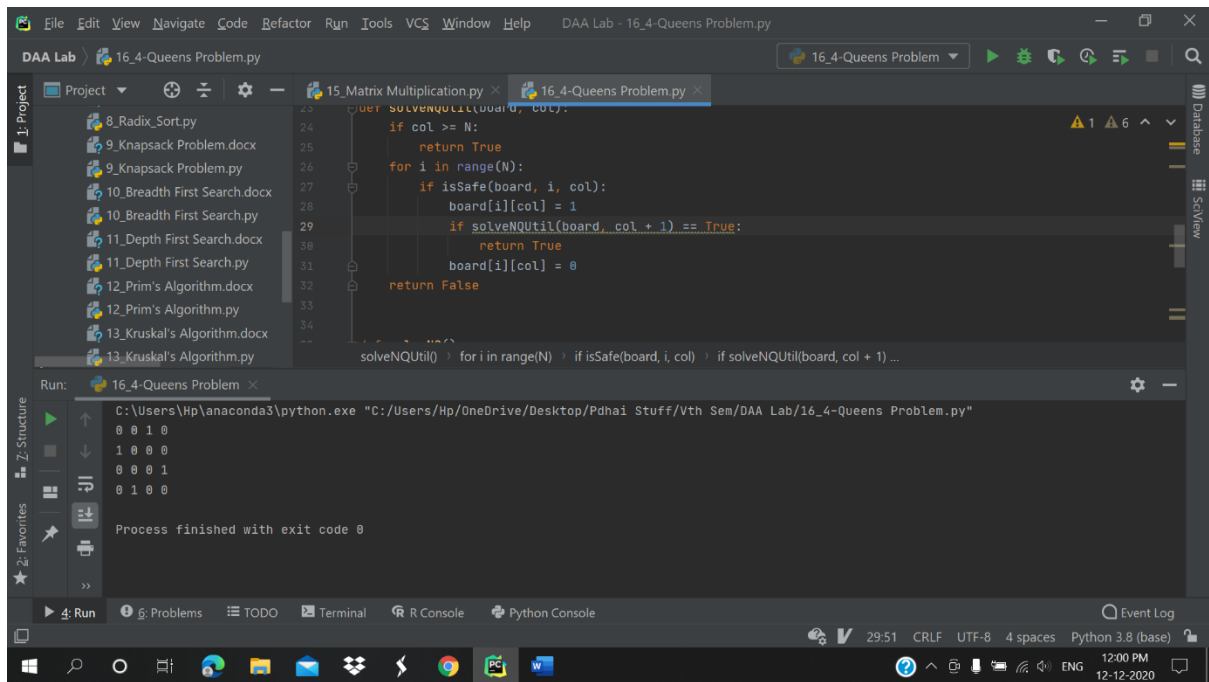
```
1 global N
2 N = 4
3
4
5 def printSolution(board):
6     for i in range(N):
7         for j in range(N):
8             print(board[i][j], end=" ")
9             print()
10
11
12 def isSafe(board, row, col):
13     for i in range(col):
14         if board[row][i] == 1:
15             return False
16     for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
17         if board[i][j] == 1:
18             return False
19     for i, j in zip(range(row, N, 1), range(col, -1, -1)):
20         if board[i][j] == 1:
21             return False
22     return True
23
24 isSafe() for i, j in zip(range(row, N, 1), range(col, -1, -1)):
```

The screenshot shows the same IDE with the 'solveNQUtil' and 'solveNQ' functions. 'solveNQUtil' is a recursive helper function that tries to place a queen in each row of a given column. 'solveNQ' is the main function that initializes the board and calls 'solveNQUtil' to find a solution. The board is represented as a 4x4 grid.

```
23 def solveNQUtil(board, col):
24     if col >= N:
25         return True
26     for i in range(N):
27         if isSafe(board, i, col):
28             board[i][col] = 1
29             if solveNQUtil(board, col + 1) == True:
30                 return True
31             board[i][col] = 0
32     return False
33
34
35 def solveNQ():
36     board = [[0, 0, 0, 0],
37             [0, 0, 0, 0],
38             [0, 0, 0, 0],
39             [0, 0, 0, 0]]
40     if not solveNQUtil(board, 0):
41         print("Solution does not exist")
42         return False
43     printSolution(board)
44     return True
45
46 solveNQ()
```

Output:

```
0010
1000
0001
0100
```



Time Complexity: $O(2^n)$