```python
class NQBacktracking:
    def __init__(self, x_, y_):
        """self.ld is an array where its indices indicate row-col+N-1
        (N-1) is for shifting the difference to store negative indices"""
        self.ld = [0] * 30

        """ self.rd is an array where its indices indicate row+col and used
        to check whether a queen can be placed on right diagonal or not"""
        self.rd = [0] * 30

        """column array where its indices indicates column and
        used to check whether a queen can be placed in that row or not"""
        self.cl = [0] * 30

        """Initial position of 1st queen"""
        self.x = x_
        self.y = y_

    def printSolution(self, board):
        """A utility function to print solution"""
        print(
            "N Queen Backtracking Solution:\nGiven initial position of 1st queen at row:",
            self.x,
            "column:",
            self.y,
            "\n",
        )
        for line in board:
            print(" ".join(map(str, line)))

    def solveNQUtil(self, board, col):
        """A recursive utility function to solve N
        Queen problem"""

        # base case: If all queens are placed then return True
        if col >= N:
            return True

        # Overlook the column where 1st queen is placed
        if col == self.y:
            return self.solveNQUtil(board, col + 1)

        for i in range(N):
            # Overlook the row where 1st queen is placed
            if i == self.x:
                continue
            # Consider this column and try placing
            # this queen in all rows one by one

            # Check if the queen can be placed on board[i][col]
            # A check if a queen can be placed on board[row][col].
            # We just need to check self.ld[row-col+n-1] and self.rd[row+coln]
            # where self.ld and self.rd are for left and right diagonal respectively
            if (self.ld[i - col + N - 1] != 1 and self.rd[i + col] != 1) and self.cl[
                i
            ] != 1:

                # lace this queen in board[i][col]
                board[i][col] = 1
                self.ld[i - col + N - 1] = self.rd[i + col] = self.cl[i] = 1

                # recur to place rest of the queens
                if self.solveNQUtil(board, col + 1):
                    return True

                # If placing queen in board[i][col]
                # doesn't lead to a solution,
                # then remove queen from board[i][col]
                board[i][col] = 0  # BACKTRACK
                self.ld[i - col + N - 1] = self.rd[i + col] = self.cl[i] = 0

                # If the queen cannot be placed in
                # any row in this column col then return False
                # print("col:", col, "i:", i, board)
        return False

    def solveNQ(self):
        """This function solves the N Queen problem using
        Backtracking. It mainly uses solveNQUtil() to
        solve the problem. It returns False if queens
        cannot be placed, otherwise, return True and
        prints placement of queens in the form of 1s.
        Please note that there may be more than one
        solutions, this function prints one of the
        feasible solutions."""
        board = [[0 for _ in range(N)] for _ in range(N)]
        board[self.x][self.y] = 1

        self.ld[self.x - self.y + N - 1] = self.rd[self.x + self.y] = self.cl[
            self.x
        ] = 1

        if not self.solveNQUtil(board, 0):
            print("Solution does not exist")
            return False
        self.printSolution(board)
        return True


if __name__ == "__main__":
    N = 8
    x, y = 3, 2

    NQBt = NQBacktracking(x, y)
    NQBt.solveNQ()


"""
OUTPUT:

N Queen Backtracking Solution:
Given initial position of 1st queen at row: 3 column: 2

1 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0
"""
```