```java
import java.io.*;
import java.util.Arrays;

class NQueens {
   public static void main(String args[]) {
       int N = 8;

       NQBranchAndBond NQBaB = new NQBranchAndBond(N);
       NQBaB.solveNQ();

       NQBacktracking NQBt = new NQBacktracking(N);
       NQBt.solveNQ();
   }
}

class NQBranchAndBond {
   private int N;

   NQBranchAndBond(int N) {
       this.N = N;
   }

   void printSolution(int board[][]) {
       System.out.println("N Queen Branch And Bound Solution:");
       for(int i = 0; i < N; i++) {
           for(int j = 0; j < N; j++)
               System.out.printf("%2d ", board[i][j]);
           System.out.printf("\n");
       }
   }

   static boolean isSafe (
       int row, int col,
       int slashCode[][],
       int backslashCode[][],
       boolean rowLookup[],
       boolean slashCodeLookup[],
       boolean backslashCodeLookup[]
   ) {
       return !(
           slashCodeLookup[slashCode[row][col]] ||
           backslashCodeLookup[backslashCode[row][col]] ||
           rowLookup[row]
       );
   }

   // A recursive utility function to solve N Queen problem
   boolean solveNQUtil(
       int board[][], int col, int slashCode[][],
       int backslashCode[][], boolean rowLookup[],
```

```
    boolean slashCodeLookup[], boolean backslashCodeLookup[]
) {
    // base case: If all queens are placed then return True
    if (col >= N)
        return true;

    for(int i = 0; i < N; i++) {
        if (isSafe(
            i, col, slashCode,
            backslashCode, rowLookup,
            slashCodeLookup, backslashCodeLookup
        )) {

            // Place this queen in board[i][col]
            board[i][col] = 1;
            rowLookup[i] = true;
            slashCodeLookup[slashCode[i][col]] = true;
            backslashCodeLookup[backslashCode[i][col]] = true;

            // recur to place rest of the queens
            if (solveNQUtil(
                board, col + 1, slashCode,
                backslashCode, rowLookup,
                slashCodeLookup,
                backslashCodeLookup
            ))
                return true;

            // If placing queen in board[i][col] doesn't
            // lead to a solution, then backtrack

            // Remove queen from board[i][col]
            board[i][col] = 0;
            rowLookup[i] = false;
            slashCodeLookup[slashCode[i][col]] = false;
            backslashCodeLookup[backslashCode[i][col]] = false;
        }
    }

    // If queen can not be place in any row
    // in this column col then return false
    return false;
}

/*
* This function solves the N Queen problem using Branch and Bound.
* 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.
* This function prints one of the feasible solutions.
```

```
    */
    boolean solveNQ() {
        int board[][] = new int[N][N];

        // Helper matrices
        int slashCode[][] = new int[N][N];
        int backslashCode[][] = new int[N][N];

        // Arrays to tell us which rows are occupied
        boolean[] rowLookup = new boolean[N];

        // Keep two arrays to tell us which diagonals are occupied
        boolean slashCodeLookup[] = new boolean[2 * N - 1];
        boolean backslashCodeLookup[] = new boolean[2 * N - 1];

        // Initialize helper matrices
        for(int r = 0; r < N; r++)
            for(int c = 0; c < N; c++) {
                slashCode[r][c] = r + c;
                backslashCode[r][c] = r - c + N - 1;
            }

        if (solveNQUtil(
            board, 0, slashCode,
            backslashCode, rowLookup,
            slashCodeLookup,
            backslashCodeLookup
        ) == false) {
            System.out.printf("Solution does not exist");
            return false;
        }

        // Solution found
        printSolution(board);
        return true;
    }
}


class NQBacktracking {
    private int N;

    NQBacktracking(int N){
        this.N = N;
    }

    /* ld is an array where its indices indicate row-col+N-1 (N-1)
    is for shifting the difference to store negative indices */
    static int []ld = new int[30];
```

```
/* 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 */
static int []rd = new int[30];

/*column array where its indices indicates column and used
to check whether a queen can be placed in that row or not*/
static int []cl = new int[30];

/* A utility function to print solution */
void printSolution(int board[][]) {
    System.out.println("\n\nN Queen Backtracking Solution:");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            System.out.printf("%2d ", board[i][j]);
        System.out.printf("\n");
    }
}

/* A recursive utility function to solve N Queen problem */
boolean solveNQUtil(int board[][], int col) {
    /* base case: If all queens are placed then return true */
    if (col >= N)
        return true;

    /* Consider this column and try placing
    this queen in all rows one by one */
    for (int i = 0; i < N; i++) {

        /* 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 ld[row-col+n-1] and rd[row+coln]
        where ld and rd are for left and right diagonal respectively */
        if ((ld[i - col + N - 1] != 1 &&
            rd[i + col] != 1) && cl[i] != 1) {

            /* Place this queen in board[i][col] */
            board[i][col] = 1;
            ld[i - col + N - 1] =
            rd[i + col] = cl[i] = 1;

            /* recur to place rest of the queens */
            if (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
            ld[i - col + N - 1] =
            rd[i + col] = cl[i] = 0;
```

```
            }
        }

        /* If the queen cannot be placed in any row in
            this column col then return false */
        return false;
    }

    /* 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. This function prints one of the feasible solutions.
    */
    boolean solveNQ() {
        int board[][] = new int[N][N];

        if (solveNQUtil(board, 0) == false) {
            System.out.printf("Solution does not exist");
            return false;
        }

        printSolution(board);
        return true;
    }
}
```

OUTPUT:

```
N Queen Branch And Bound Solution:
1  0  0  0  0  0  0  0
0  0  0  0  0  0  1  0
0  0  0  0  1  0  0  0
0  0  0  0  0  0  0  1
0  1  0  0  0  0  0  0
0  0  0  1  0  0  0  0
0  0  0  0  0  1  0  0
0  0  1  0  0  0  0  0


N Queen Backtracking Solution:
1  0  0  0  0  0  0  0
0  0  0  0  0  0  1  0
0  0  0  0  1  0  0  0
0  0  0  0  0  0  0  1
0  1  0  0  0  0  0  0
0  0  0  1  0  0  0  0
0  0  0  0  0  1  0  0
0  0  1  0  0  0  0  0
```