N Queen Problem | Return all Distinct Solutions to the N-Queens Puzzle

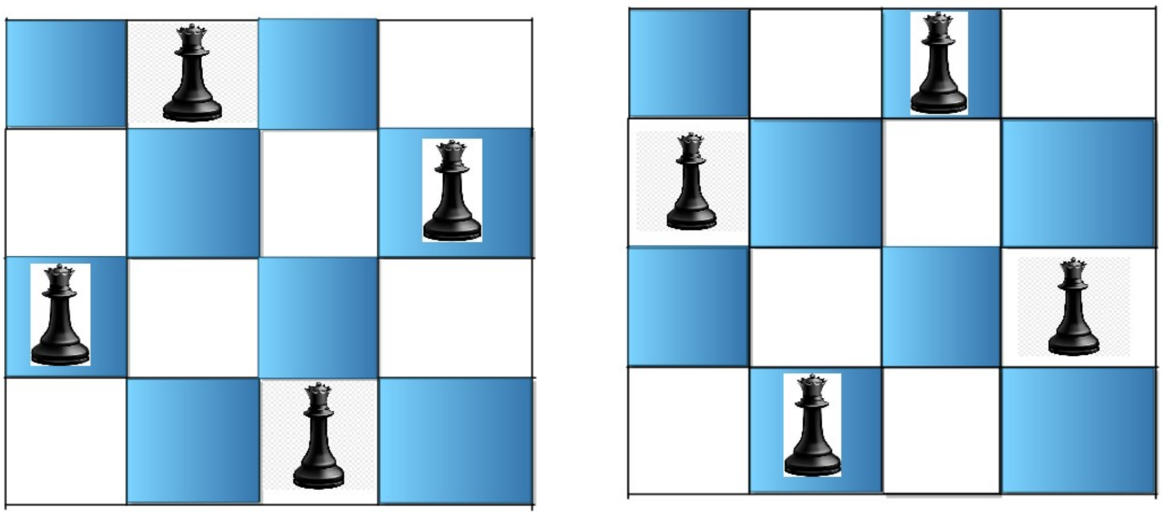
Problem Statement: The n-queens is the problem of placing n queens on n × n chessboard such that no two queens can attack each other. Given an integer n, return all distinct solutions to the n - queens puzzle. Each solution contains a distinct boards configuration of the queen's placement, where ‘Q’ and ‘.’ indicate queen and empty space respectively.

Examples:

Input: `n = 4`

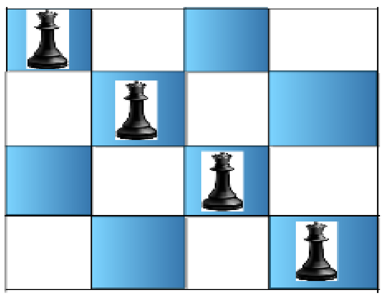
Output: `[[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]`

Explanation: There exist two distinct solutions to the 4-queens puzzle as shown below



Two arrangements possible for 4 queens

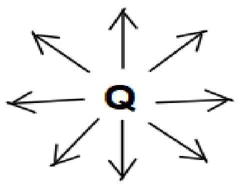
Let us first understand how can we place queens in a chessboard so that no attack on either of them can take place.



Rules for n-Queen in chessboard

1. Every row should have one Queen
2. Every column should have one Queen
3. No two queens can attack each other

Queen attack can take place in following way

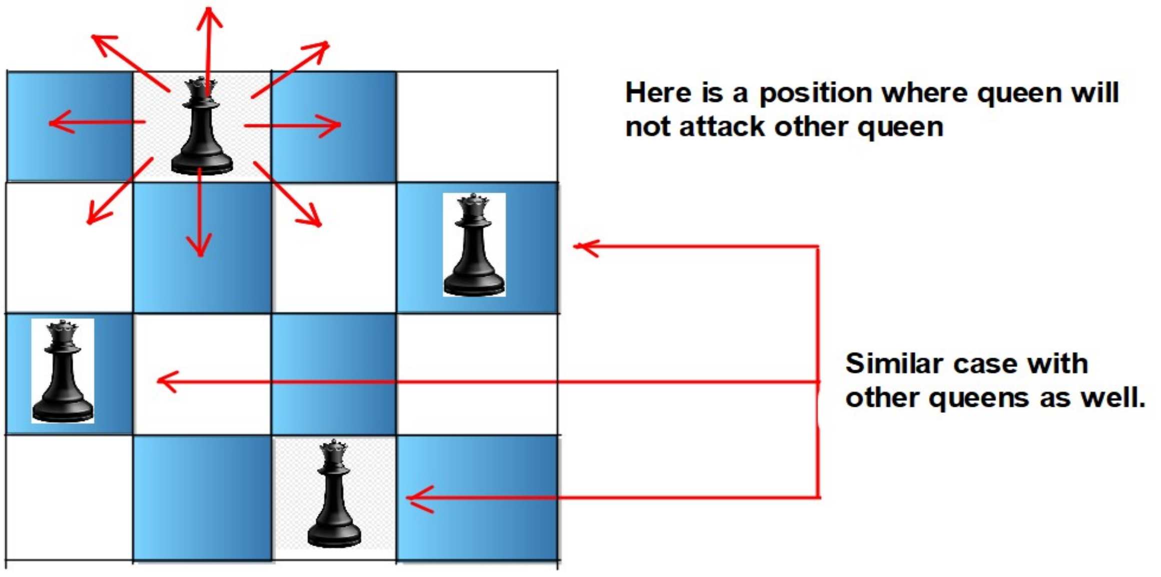


Solution

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

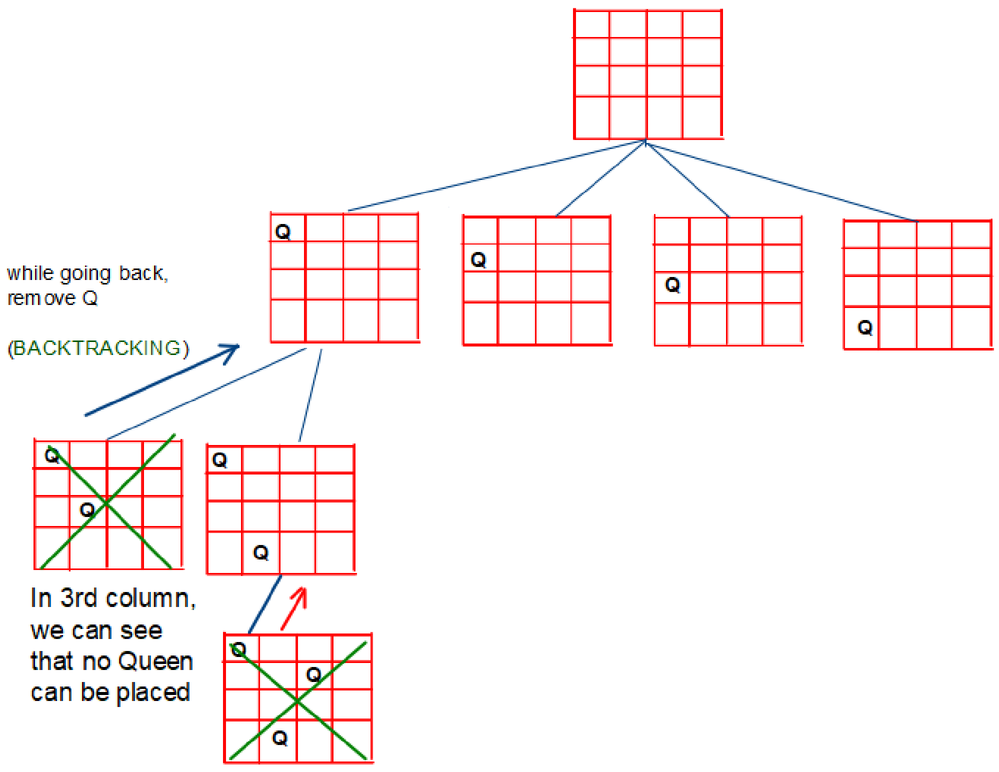
Solution 1:

Intuition: Using the concept of Backtracking, we will place Queen at different positions of the chessboard and find the right arrangement where all the n queens can be placed on the n*n grid.

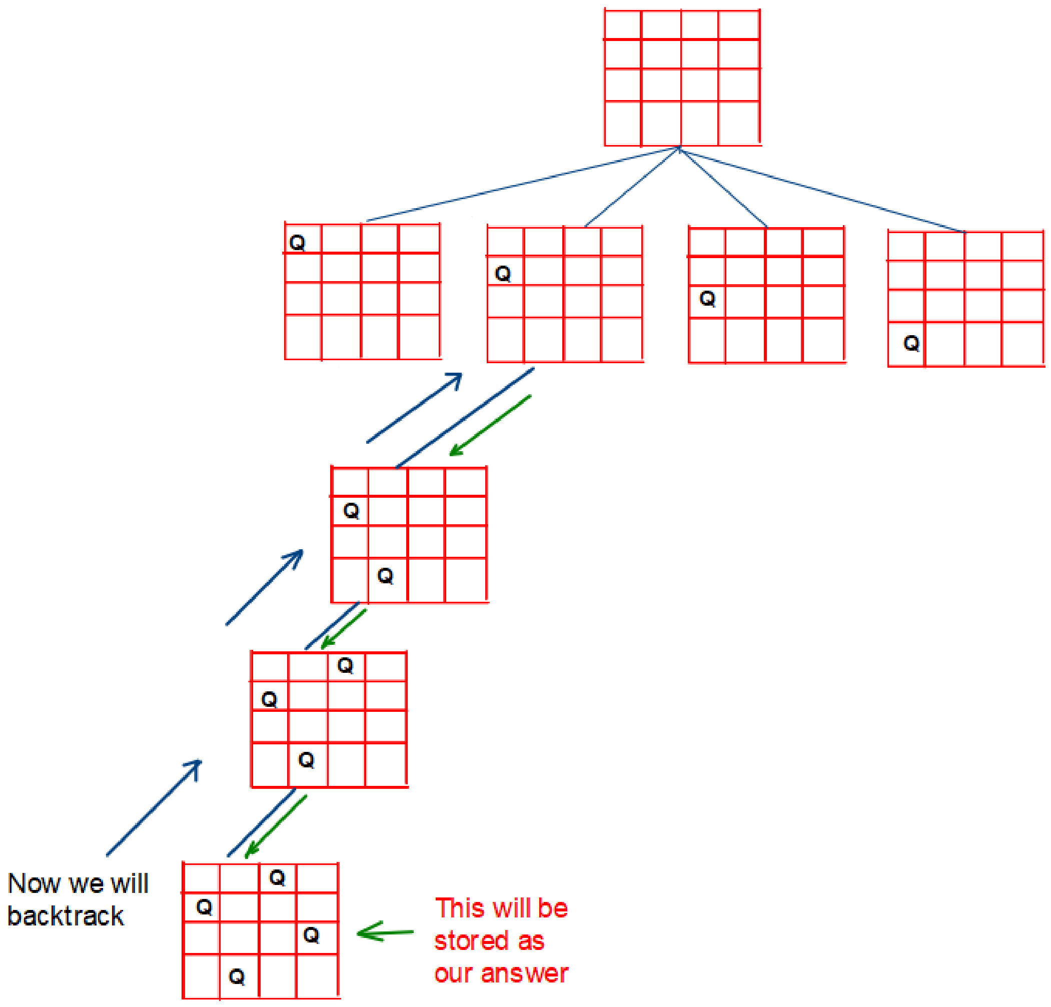

Here is a position where queen will not attack other queen
Similar case with other queens as well.

Approach:

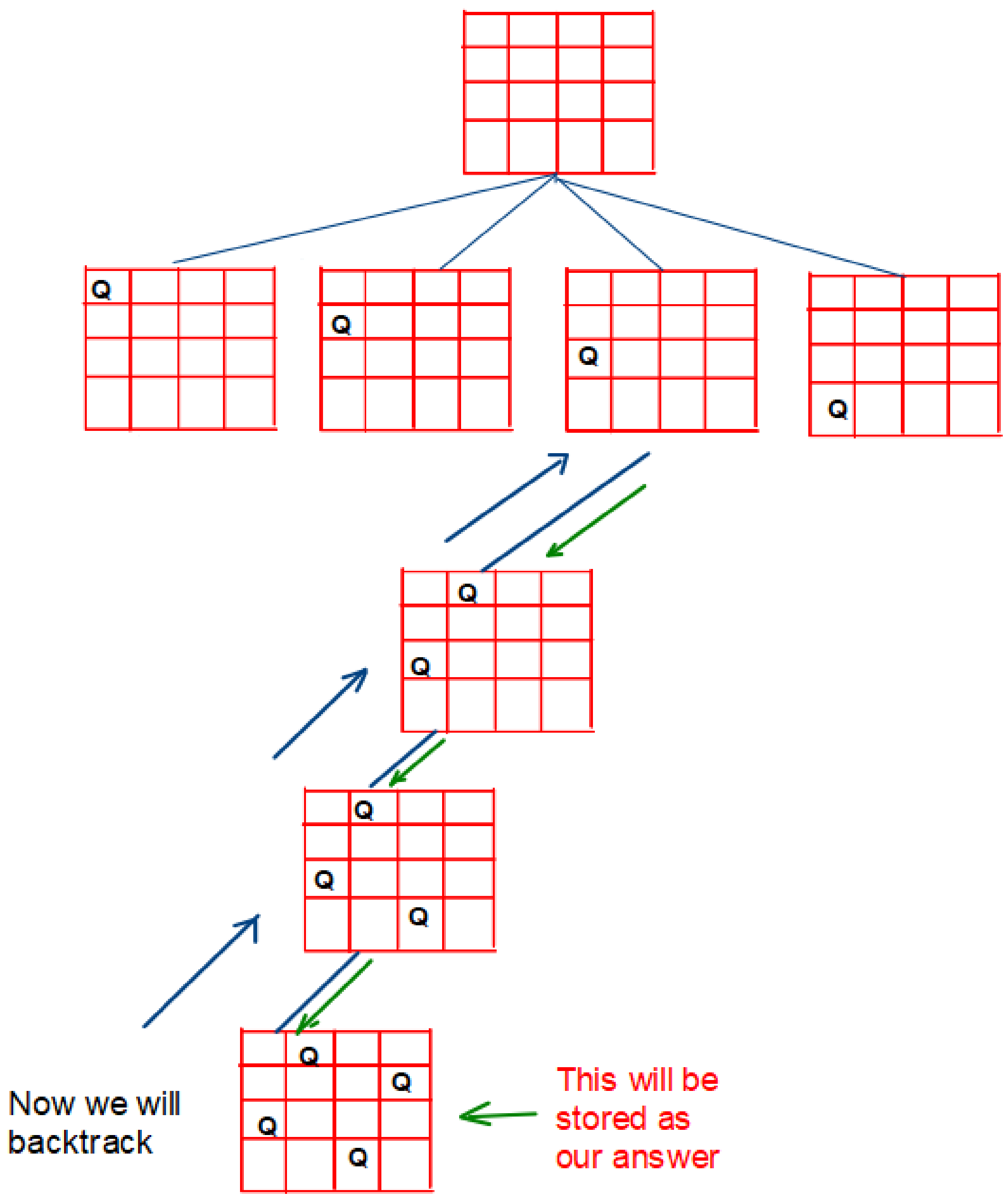
Ist position: This is the position where we can see no possible arrangement is found where all queens can be placed since, at the 3rd column, the Queen will be killed at all possible positions of row.


Q
Q
Q
Q
while going back, remove Q
(BACKTRACKING)
Q
Q
Q
Q
In 3rd column, we can see that no Queen can be placed
Q
Q
Q

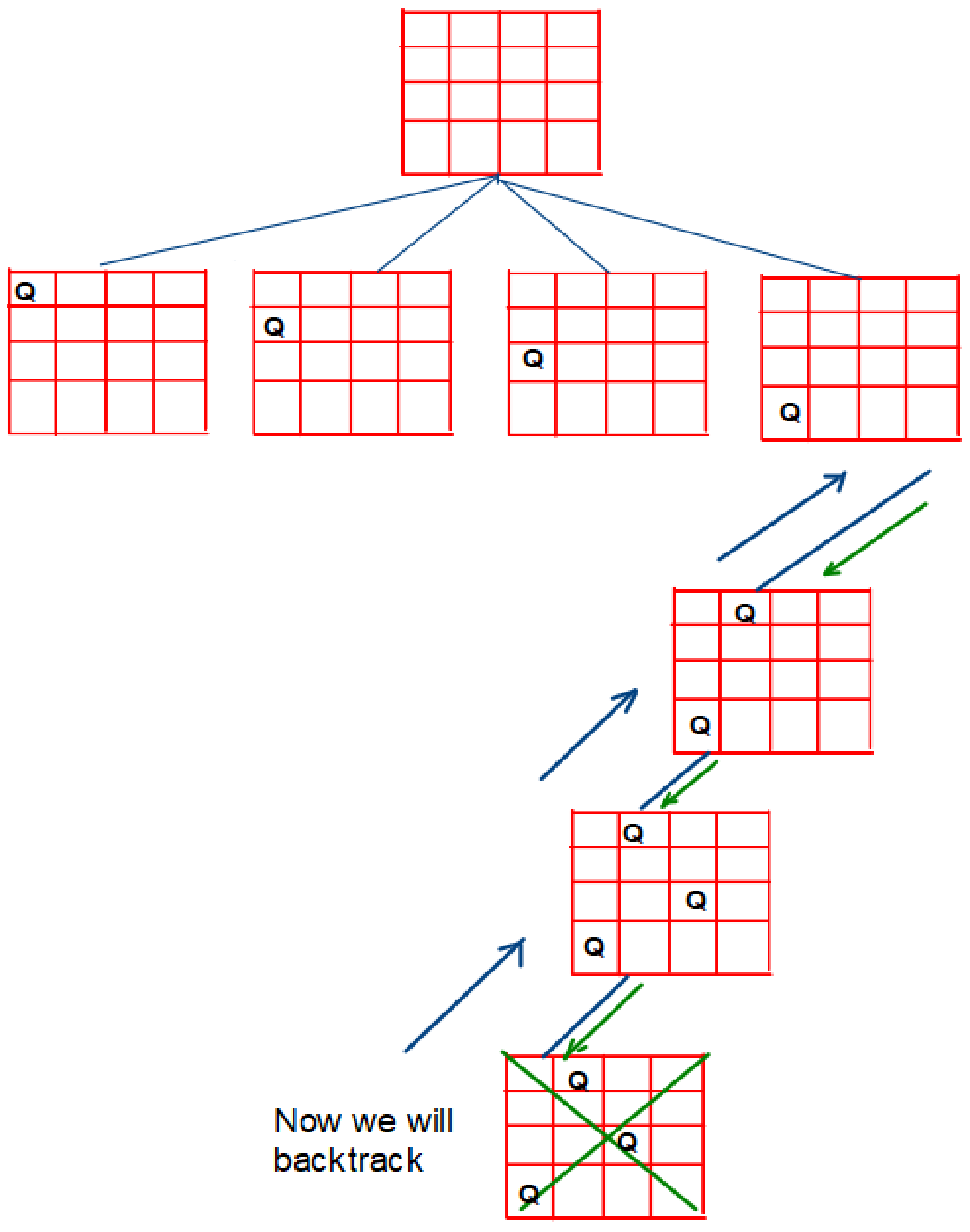
2nd position: One of the correct possible arrangements is found. So we will store it as our answer.



3rd position: One of the correct possible arrangements is found. So we will store it as our answer.



4th position: This is the position where we can see no possible arrangement is found where all queens can be placed since, at the 4th column, the Queen will be killed at all possible positions of row.



Code:

```
import java.util.*;
class Main {
    public static List < List < String >> solveNQueens(int n) {
        char[][] board = new char[n][n];
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                board[i][j] = '.';
        List < List < String >> res = new ArrayList < List < String >> ();
        dfs(0, board, res);
        return res;
    }

    static boolean validate(char[][] board, int row, int col) {
        int duprow = row;
        int dupcol = col;
        while (row >= 0 && col >= 0) {
            if (board[row][col] == 'Q') return false;
            row--;
            col--;
        }

        row = duprow;
        col = dupcol;
        while (col >= 0) {
            if (board[row][col] == 'Q') return false;
            col--;
        }

        row = duprow;
        col = dupcol;
        while (col >= 0 && row < board.length) {
            if (board[row][col] == 'Q') return false;
            col--;
            row++;
        }
        return true;
    }

    static void dfs(int col, char[][] board, List < List < String >> res) {
        if (col == board.length) {
            res.add(construct(board));
            return;
        }

        for (int row = 0; row < board.length; row++) {
            if (validate(board, row, col)) {
                board[row][col] = 'Q';
                dfs(col + 1, board, res);
                board[row][col] = '.';
            }
        }
    }



    static List < String > construct(char[][] board) {
        List < String > res = new LinkedList < String > ();
        for (int i = 0; i < board.length; i++) {
            String s = new String(board[i]);
            res.add(s);
        }
        return res;
    }
    public static void main(String args[]) {
        int N = 4;
        List < List < String >> queen = solveNQueens(N);
        int i = 1;
        for (List < String > it: queen) {
            System.out.println("Arrangement " + i);
            for (String s: it) {
                System.out.println(s);
            }
            System.out.println();
            i += 1;
        }

    }
}
```

Output:

Arrangement 1
..Q.
Q…
…Q
.Q..

Arrangement 2
.Q..
…Q
Q…
..Q.

Time Complexity: Exponential in nature since we are trying out all ways, to be precise its O(N! * N).

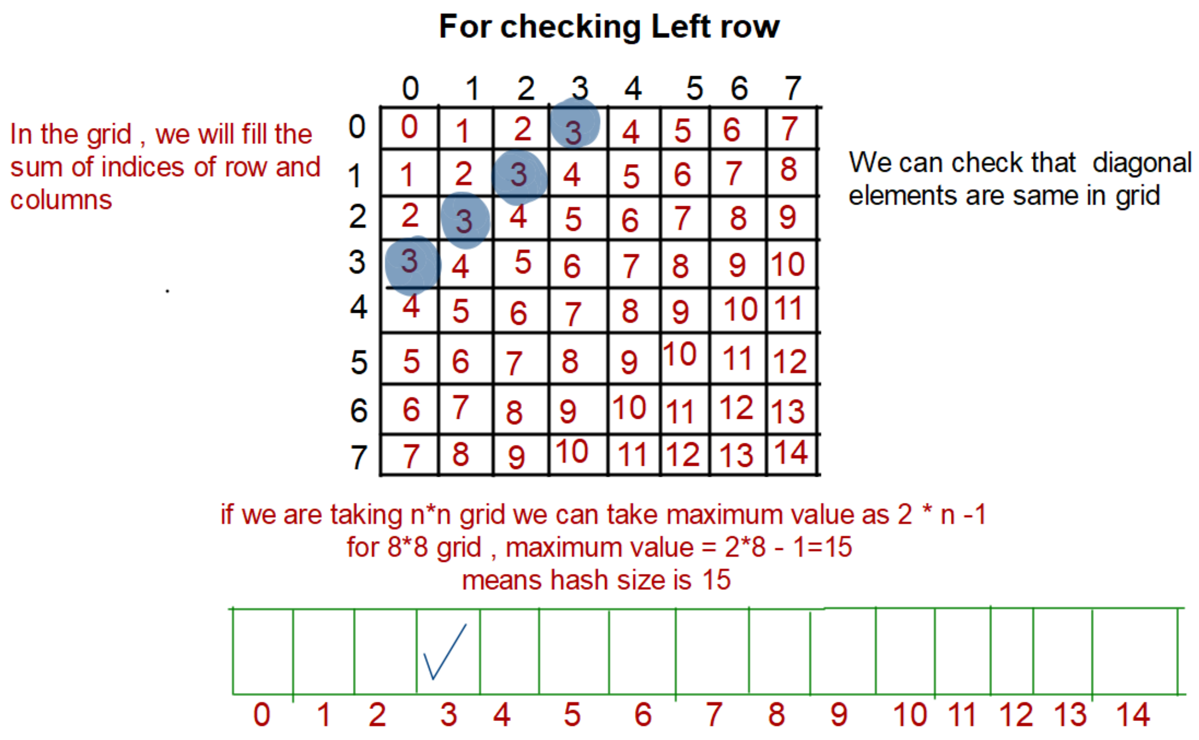
Space Complexity: O(N2)

Solution 2:

Intuition: This is the optimization of the issafe function. In the previous issafe function, we need o(N) for a row, o(N) for the column, and o(N) for the diagonal. Here, we will use hashing to maintain a list to check whether that position can be the right one or not.

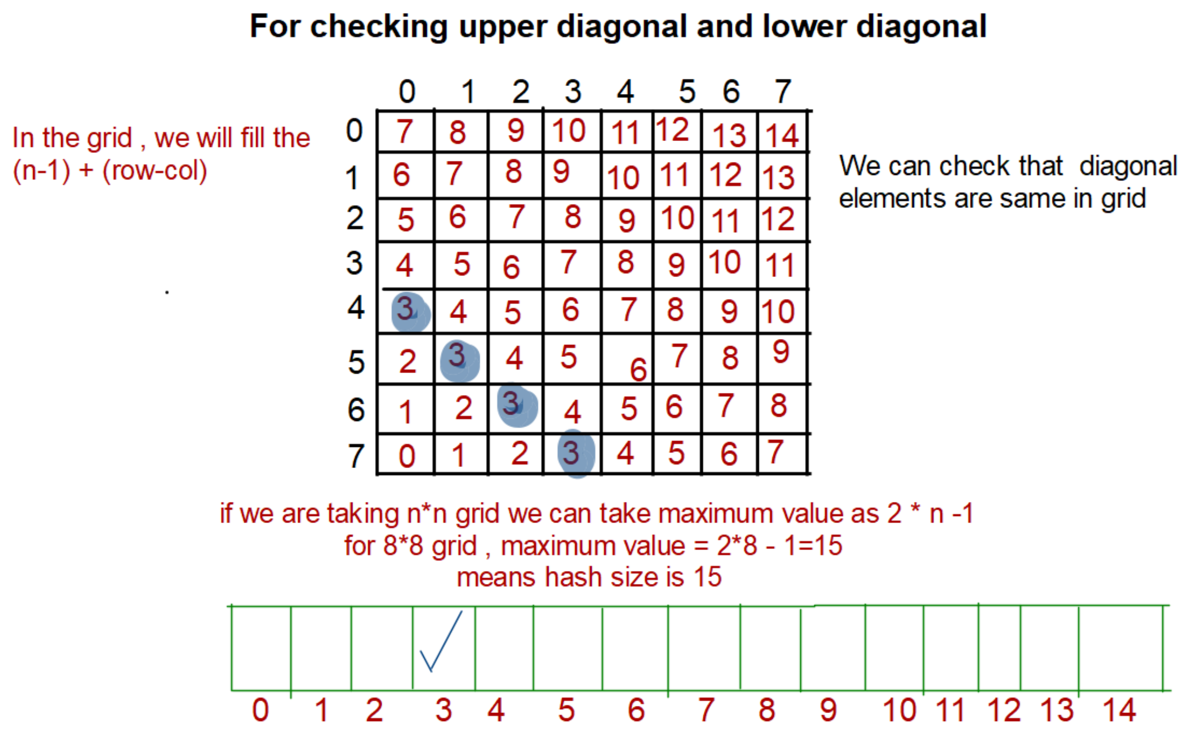
Approach:

For checking Left row elements


For checking Left row
In the grid , we will fill the sum of indices of row and columns
We can check that diagonal elements are same in grid
if we are taking n*n grid we can take maximum value as 2 * n -1
for 8*8 grid , maximum value = 2*8 - 1=15
means hash size is 15
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	8
2	2	3	4	5	6	7	8	9
3	3	4	5	6	7	8	9	10
4	4	5	6	7	8	9	10	11
5	5	6	7	8	9	10	11	12
6	6	7	8	9	10	11	12	13
7	7	8	9	10	11	12	13	14

For checking upper diagonal and lower diagonal



For checking upper diagonal and lower diagonal

In the grid , we will fill the (n-1) + (row-col)

	0	1	2	3	4	5	6	7
0	7	8	9	10	11	12	13	14
1	6	7	8	9	10	11	12	13
2	5	6	7	8	9	10	11	12
3	4	5	6	7	8	9	10	11
4	3	4	5	6	7	8	9	10
5	2	3	4	5	6	7	8	9
6	1	2	3	4	5	6	7	8
7	0	1	2	3	4	5	6	7

We can check that diagonal elements are same in grid

if we are taking n*n grid we can take maximum value as 2 * n -1
for 8*8 grid , maximum value = 2*8 - 1=15
means hash size is 15

			✓											
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Code:

```
import java.util.*;
class Main {
    public static List < List < String >> solveNQueens(int n) {
        char[][] board = new char[n][n];
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                board[i][j] = '.';
        List < List < String >> res = new ArrayList < List < String >> ();
        int leftRow[] = new int[n];
        int upperDiagonal[] = new int[2 * n - 1];
        int lowerDiagonal[] = new int[2 * n - 1];
        solve(0, board, res, leftRow, lowerDiagonal, upperDiagonal);
        return res;
    }



    static void solve(int col, char[][] board, List < List < String >> res, int
leftRow[], int lowerDiagonal[], int upperDiagonal[]) {
        if (col == board.length) {
            res.add(construct(board));
            return;
        }

        for (int row = 0; row < board.length; row++) {
            if (leftRow[row] == 0 && lowerDiagonal[row + col] == 0 &&
upperDiagonal[board.length - 1 + col - row] == 0) {
                board[row][col] = 'Q';
                leftRow[row] = 1;
                lowerDiagonal[row + col] = 1;
                upperDiagonal[board.length - 1 + col - row] = 1;
                solve(col + 1, board, res, leftRow, lowerDiagonal,
upperDiagonal);
                board[row][col] = '.';
                leftRow[row] = 0;
                lowerDiagonal[row + col] = 0;
                upperDiagonal[board.length - 1 + col - row] = 0;
            }
        }
    }


    static List < String > construct(char[][] board) {
        List < String > res = new LinkedList < String > ();
        for (int i = 0; i < board.length; i++) {
            String s = new String(board[i]);
            res.add(s);
        }
        return res;
    }
    public static void main(String args[]) {
        int N = 4;
        List < List < String >> queen = solveNQueens(N);
        int i = 1;
        for (List < String > it: queen) {
            System.out.println("Arrangement " + i);
            for (String s: it) {
                System.out.println(s);
            }
            System.out.println();
            i += 1;
        }

    }
}
```

Output:

```
Arrangement 1
..Q.
Q…
…Q
.Q..

Arrangement 2
.Q..
…Q
Q…
..Q.
```

Time Complexity: Exponential in nature since we are trying out all ways, to be precise its O(N! * N).

Space Complexity: O(N)