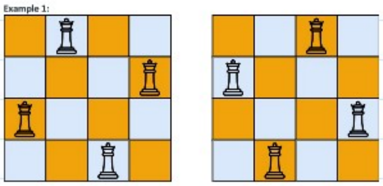
L16 N-Queens I

Tuesday, January 3, 2023 1:48 AM

The **n-queens** puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other.

Given an integer n, return *all distinct solutions to the **n-queens puzzle***. You may return the answer in **any order**.
Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space, respectively.

Example 1:

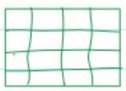


↳ One of the most important problem based on Recursion and backtracking

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 above

→ Basically in this problem we have NXN Chessboard and our task is finding all the possible chess board where we placed our Queen

Example N=4
↳ that means we have 4X4 Chessboard and we placed 4 Queen into this Chessboard

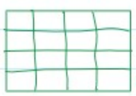


(nxn) Matrix

#Thought Process

→ Now in the Interview if this question comes then always remember if we talk about the Possible Way type of problem that means here used 100% Recursion Approach and go with Recursion Approach

N=4

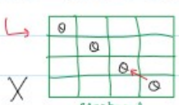


(NXN) Matrix

Rules of the Placing Queen Into the Chessboard

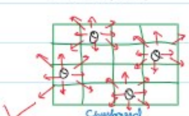
→ Every Row have 1 Queen
→ Column have 1 Queen
→ None of the Queen Attack to each other
↳ In the eight direction

Example N=4
→ that means we have 4X4 Chessboard and placed 4 Queen into the Chessboard



↳ Not a Valid Chessboard

→ Every Row and Column have 1 Queen
→ But Not follow the 3rd Rules
↳ Attack to each other direction

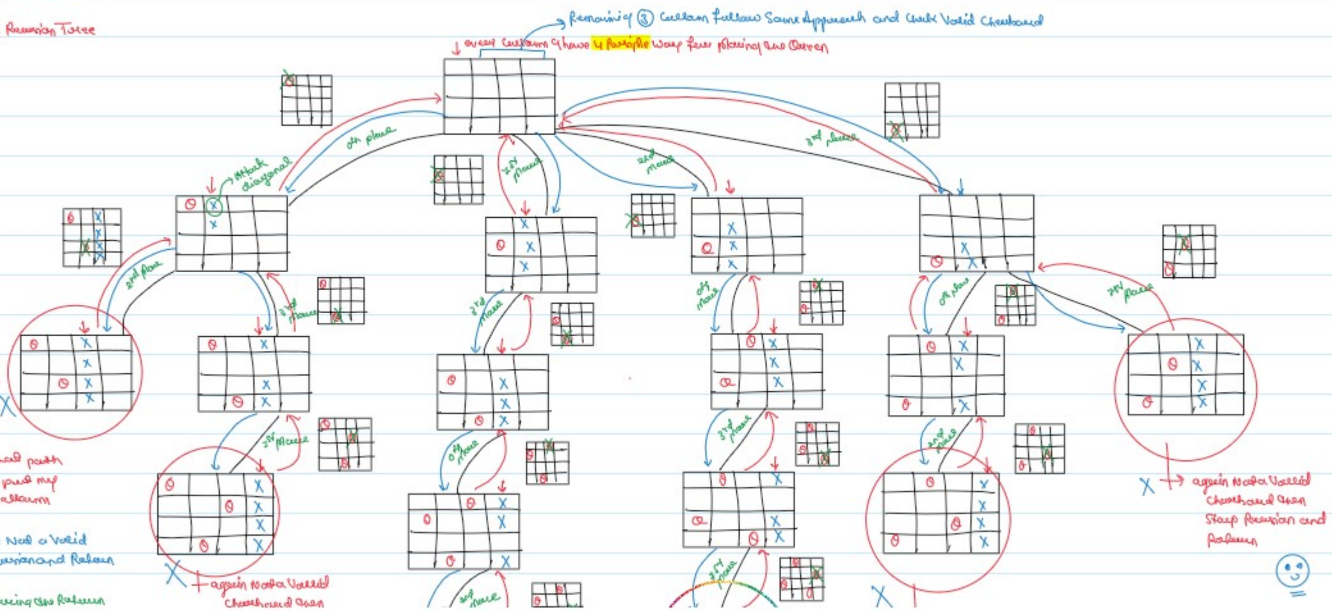


→ Every Row and Column have 1 Queen
→ And Not Attack to each other direction

↳ That means and is the one of the answer So that is way we finding All the Valid Chessboard

→ Lets understand with the Help of Recursion Tree

N=4



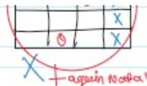
→ Now if I follow that path So I can Not able to put my Queen into the 3rd Column

↳ that Means that is Not a Valid Chessboard Stop Recursion and Return

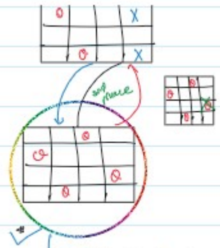
↳ And Remember during the Return

→ that means that is not a valid chessboard, stop recursion and return

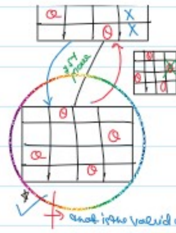
→ And remember during the return time remove the queen just previously column, that is known as backtracking



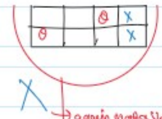
→ again not a valid chessboard, then stop recursion and return



→ that is the valid one chessboard, so store that chessboard, return and removed the queen



→ that is the valid one chessboard, so store that chessboard, return and removed the queen



→ again not a valid chessboard, then stop recursion and return

stop recursion and return

→ that is the way how to the recursion work

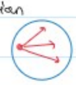
Important Point

→ We know that we search into 8 different direction

→ But logically we required 3 direction to check

because if we standing any particular point then

→ Required to check because we know that in a row/column only one queen possible and upcoming all row and column are empty, then no need to check that direction



X No Need

```python
class Solution:

    # GfG one
    def indexAnswer(self, ans):
        # for store index value
        finalans = []
        for s in ans:
            temp = []
            for j in s:
                ind = j.index('Q') + 1
                temp.append(ind)
            finalans.append(temp)

        return finalans

    # function to check if the queens threaten each other or not
    def isSafe(self, chessBord, row, coll, n):

        # return false if two queens share the same column
        for i in range(row):
            if chessBord[i][coll] == 'Q':
                return False

        # return false if two queens share the same `upper` diagonal
        dupcrow = row
        dupcoll = coll

        while duprow >= 0 and dupcoll >= 0:
            if chessBord[duprow][dupcoll] == 'Q':
                return False
            duprow -= 1
            dupcoll -= 1

        # return false if two queens share the same `lower` diagonal
        dupcrow = row
        dupcoll = coll

        while duprow >= 0 and dupcoll < n:
            if chessBord[duprow][dupcoll] == 'Q':
                return False
            duprow -= 1
            dupcoll += 1

        return True

    def FormatedAnswer(self, cheesBord):
        ans = []
        for i in cheesBord:
            temp = ''
            for j in i:
                if j == '.' or j == 'Q':
                    temp += j
            ans.append(temp)

        return ans

    def nQueen(self, cheesBord, coll, ans, n):

        # if `N` queens are placed successfully, print the solution
        if coll == n:
            ans.append(self.FormatedAnswer(cheesBord))
            return

        # place queen at every square in the current row `r`
        # and recur for each valid movement
        for row in range(n):

            # if no two queens attack each other
            if self.isSafe(cheesBord, coll, row, n):
                # place queen on the current square
                cheesBord[coll][row] = 'Q'

                # recur for the next row
                self.nQueen(cheesBord, coll + 1, ans, n)

                # backtrack and remove the queen from the current square
                cheesBord[coll][row] = '.'

    def solveNQueens(self, n):

        ans = []
        cheesBord = []
        for i in range(n):
            temp = ['.'] * n
            cheesBord.append(temp)

        print(cheesBord)

        self.nQueen(cheesBord, 0, ans, n)

        print(ans)
        return ans
```

GfG searching for finding index of the queen, next queen

Base case

```java
import java.util.*;

public class LC N Queens {
    public static void main(String[] args) {
        System.out.println("LC N Queens");
    }
}

class Solution {

    static List<String> construct(char[][] chessBoard) {
        List<String> res = new LinkedList<String>();
        for (int i = 0; i < chessBoard.length; i++) {
            String s = new String(chessBoard[i]);
            res.add(s);
        }
        System.out.println(res);
        return res;
    }

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

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

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

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

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

    public List<List<String>> solveNQueens(int n) {

        char[][] chessBoard = new char[n][n];
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                chessBoard[i][j] = '.';

        List<List<String>> res = new ArrayList<List<String>>();
        nQueen(0, chessBoard, res);
        return res;
    }
}
```

Time Complexity $= O(N!) \times N$

→ for all the possible combination recursion

→ for all the other searching

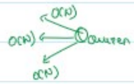
Space Complexity $= O(N^2)$

→ because we used a nested array of the size (N) for storing the answer

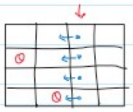
```
print(ans)
return ans
```

→ But this is Not a Good Approach because We used **3O(N)** extra time try to reduced this time

→ using **Hashing** We removed that 3 Extra O(N) time



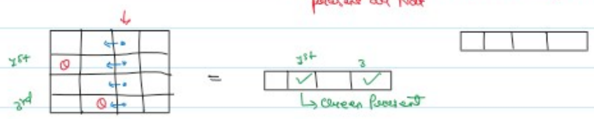
Lets Understand



↳ Now if I placed a queen into the 2nd Column of any block then only check

↳ any another Queen present or not for that particular Row or Not in ← direction

↳ And Maintaining the list for Marking which Row queen present or Not



→ that's way We finding the position into the ← direction without searching and loop using hashing

→ How to find the diagonal Position lets understand ↙ direction

↳ lets understand using **8X8 Matrix**

	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

→ Adding Row and column Value to each other and put into the Matrix

if I want to placed my Queen Here then Check and ↗ are Not present any Queen ↙

→ Now after placing all the element we Clearly find out the pattern

→ So what can do is we Maintain the hashing Array that size (2n-1) and tick Mark the Queen position

Example $n=8$

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

↳ So if i want to placed Queen in **5th Row and 6th column** then **add Column + Row Value** and Mark into the hashing Array that Index

$5+6=11$ → that Means **Index = 11 Marked**

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

↳ Now if I try to placed Queen in the **7th Row and 4th** Column that Means

$7+4=11$ → at Index of **11 already** Queen Present So we can not placed Queen Here X

→ that is the way How to deal with **↙ direction**

→ Now How to find the Position into the **↖ direction**

→ lets understand again using **nxn Matrix**

→ In this Approach we Just **Reverse the all Row** of Matrix that we learned Previously

→ Apply formula $(N-1)+(Row-Col)$

	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

If I want to placed my Queen Here then check ↖ direction queen present or Not

→ Now After placing all the element into the Matrix we Clearly find out the pattern

→ Again follow same Approach Creating **(2N-1) hashing** Array for Marking our Queen

→ So what can do is we Maintain the hashing Array that size (2n-1) and tick Mark the Queen position

Example $n=8$

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

→ So what can do is we maintain the hashing Array that size (2n-1) and tick mark the Queen position

Example   n=8

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

↳ So if i want to placed Queen in 5th Row and 6th collum then Apply the formula that we learned and mark into the hashing Array that index

$\boxed{\text{Apply formula } (N-1) + (Row - Col)}$

n=8
Row=5
Collum=6 } → Apply formula
= (8-1)+(5-6)
= 7+1
= 8 → that means index 8 Marked

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

↳ Now if i want to keep placed my Queen on 4th Row and 5th Collum that means

(n-1) + (Row-Coll)
= 7+(4-5)
= 8 → that means we placed Queen at index 8 but index 8 all ready filled so we cannot placed queen X

↳ that is the way Have to deal with ↖ direction

Space Complexity = O(N!) → we Removed extra (N) for Hash → for All the Combination

Space Complexity = O(N²) → for using Nested data Structure for Store the our answer