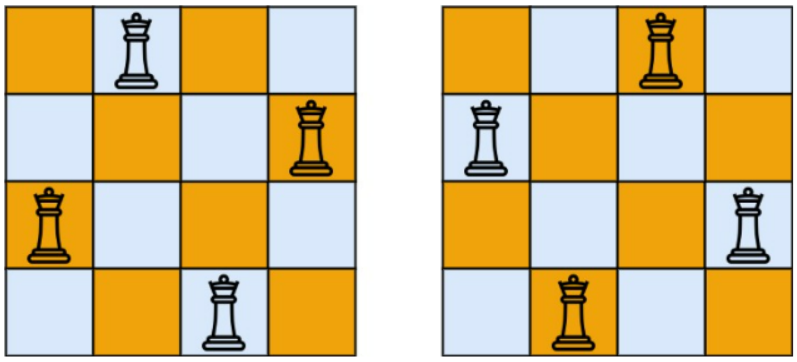
<https://leetcode.com/problems/n-queens-ii/>

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 *the number of distinct solutions to the****n-queens puzzle***.

**Example 1:**



Input: n = 4

Output: 2

Explanation: There are two distinct solutions to the 4-queens puzzle as shown.

**Example 2:**

Input: n = 1

Output: 1

**Constraints:**

* 1 <= n <= 9

**Attempt 1: 2022-12-04**

**Solution 1:  Backtracking (10 min)**

**Style 1: Void helper with global variable**

class Solution {

int count = 0;

public int totalNQueens(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] = '.';

}

}

// Start from row = 0 and recursively assign each 'Q' to next row

helper(board, 0);

return count;

}

private void helper(char[][] board, int rowIndex) {

if(rowIndex == board.length) {

count++;

return;

}

for(int i = 0; i < board[0].length; i++) {

// Backtrack

board[rowIndex][i] = 'Q';

if(isValid(board, rowIndex, i)) {

helper(board, rowIndex + 1);

}

board[rowIndex][i] = '.';

}

}

private boolean isValid(char[][] board, int rowIndex, int colIndex) {

// only check rows above current row

for(int i = 0; i < rowIndex; i++) {

for(int j = 0; j < board[0].length; j++) {

// if 'Q' in the same col or the diagonal line, return false

if((j == colIndex || Math.abs(i - rowIndex) == Math.abs(j - colIndex)) && board[i][j] == 'Q') {

return false;

}

}

}

return true;

}

}

Time Complexity : O(N!), Since we have N choices in the first row, then N-1 choices in the second row and so on so the overall complexity become O(N!)

Another saying for Time Complexity is O(N! \* N), the additional N is coming from in isValid call the inner for loop consumes as N

Space Complexity: O(N\*N), Just the board and recursive stack space

**Style 2: Return type helper without global variable**

class Solution {

public int totalNQueens(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] = '.';

}

}

// Start from row = 0 and recursively assign each 'Q' to next row

return helper(board, 0);

}

private int helper(char[][] board, int rowIndex) {

if(rowIndex == board.length) {

return 1;

}

int count = 0;

for(int i = 0; i < board[0].length; i++) {

// Backtrack

board[rowIndex][i] = 'Q';

if(isValid(board, rowIndex, i)) {

count += helper(board, rowIndex + 1);

}

board[rowIndex][i] = '.';

}

return count;

}

private boolean isValid(char[][] board, int rowIndex, int colIndex) {

// only check rows above current row

for(int i = 0; i < rowIndex; i++) {

for(int j = 0; j < board[0].length; j++) {

// if 'Q' in the same col or the diagonal line, return false

if((j == colIndex || Math.abs(i - rowIndex) == Math.abs(j - colIndex)) && board[i][j] == 'Q') {

return false;

}

}

}

return true;

}

}

Time Complexity : O(N!), Since we have N choices in the first row, then N-1 choices in the second row and so on so the overall complexity become O(N!)

Another saying for Time Complexity is O(N! \* N), the additional N is coming from in isValid call the inner for loop consumes as N

Space Complexity: O(N\*N), Just the board and recursive stack space

**Refer to**

<https://leetcode.com/problems/n-queens-ii/solutions/1237811/short-easy-w-explanation-visualization-backtracking-explained/>

int totalNQueens(int n) {

vector<vector<bool>> board(n, vector<bool>(n, false));

return solve(board, 0);

}

bool check(vector<vector<bool>>& board, int row, int col) {

int n = size(board);

for(int i = 0; i <= row; i++) {

if(board[i][col]) return false; // checking if any queen already placed on same column previously

// checking if all diagonals are safe -

if(row - i >= 0 && col - i >= 0 && board[row - i][col - i]) return false;

if(row - i >= 0 && col + i < n && board[row - i][col + i]) return false;

}

return true;

}

int solve(vector<vector<bool>>& board, int row) {

if(row == size(board)) return 1;

int count = 0;

for(int col = 0; col < size(board); col++)

if(check(board, row, col)){ // check if we can place at (row, col)

board[row][col] = true; // place the queen at (row, col)

count += solve(board, row + 1); // explore for the next row. The function will return 1 if all N queens get placed for current combination

board[row][col] = false; // backtrack - remove previously placed queen and try for different columns

}

return count;

}