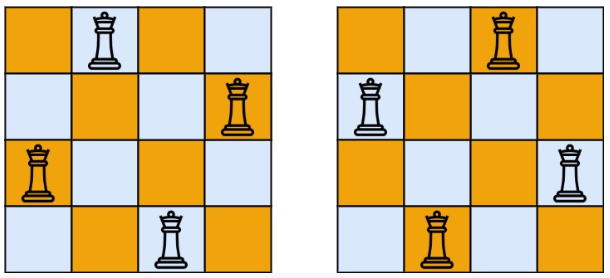
<https://leetcode.com/problems/n-queens/>

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:**



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

**Example 2:**

Input: n = 1

Output: [["Q"]]

**Constraints:**

* 1 <= n <= 9

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

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

class Solution {

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

List<List<String>> result = new ArrayList<List<String>>();

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(result, board, 0);

return result;

}

private void helper(List<List<String>> result, char[][] board, int rowIndex) {

if(rowIndex == board.length) {

List<String> list = new ArrayList<String>();

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

list.add(new String(board[i]));

}

result.add(list);

return;

}

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

// Backtrack

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

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

helper(result, 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

**Refer to**

**Start from row = 0**

<https://leetcode.com/problems/n-queens/discuss/19805/My-easy-understanding-Java-Solution/150112>

public class Solution {

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

List<List<String>> res = new LinkedList<>();

if (n <= 0) { return res; }

// build chessboard @mat(=matrix)

char[][] mat = new char[n][n];

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

for (int j = 0; j < n; j++) {

mat[i][j] = '.';

}

}

helper(mat, 0, res);

return res;

}

private void helper(char[][] mat, int row, List<List<String>> res) {

// reach solution

if (row == mat.length) {

res.add(builder(mat));

return;

}

for (int i = 0; i < mat.length; i++) {

// try

mat[row][i] = 'Q';

// if possible

if (isValid(mat, row, i)) {

helper(mat, row + 1, res);

}

// un-try

mat[row][i] = '.';

}

return;

}

private boolean isValid(char[][] mat, int x, int y) {

// only check rows above current one

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

for (int j = 0; j < mat.length; j++) {

// not need to check current position

if (i == x && j == y) {

continue;

}

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

if ((j == y || Math.abs(x - i) == Math.abs(y - j)) && mat[i][j] == 'Q') {

return false;

}

}

}

return true;

}

// build solution from temporary chessboard

private List<String> builder(char[][] mat) {

List<String> tmp = new LinkedList<>();

for (int i = 0; i < mat.length; i++) {

String t = new String(mat[i]);

tmp.add(t);

}

return tmp;

}

}

**Refer to**

**Start from column = 0**

<https://leetcode.com/problems/n-queens/discuss/19805/My-easy-understanding-Java-Solution>

public class Solution {

public 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(board, 0, res);

return res;

}

private void dfs(char[][] board, int colIndex, List<List<String>> res) {

if(colIndex == board.length) {

res.add(construct(board));

return;

}

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

if(validate(board, i, colIndex)) {

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

dfs(board, colIndex + 1, res);

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

}

}

}

private boolean validate(char[][] board, int x, int y) {

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

for(int j = 0; j < y; j++) {

if(board[i][j] == 'Q' && (x + j == y + i || x + y == i + j || x == i))

return false;

}

}

return true;

}

private 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;

}

}

**Solution 2:  Backtracking with Bit-manipulation (60 min)**

class Solution {

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

List<List<String>> result = new ArrayList<List<String>>();

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(result, board, 0, 0, 0, 0);

return result;

}

private void helper(List<List<String>> result, char[][] board, int row, int cols, int diags, int antiDiags) {

if(row == board.length) {

List<String> list = new ArrayList<String>();

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

list.add(new String(board[i]));

}

result.add(list);

return;

}

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

int curDiag = row - col + board.length;

int curAntiDiag = row + col;

// Check if the current Queen placement is valid with "&"

if((cols & (1 << col)) != 0 || (diags & (1 << curDiag)) != 0 || (antiDiags & (1 << curAntiDiag)) != 0) {

continue;

}

// Backtracking

board[row][col] = 'Q';

// Update given 'Q' at [row, col]'s other attribute as cols, diags and antiDiags with "|="

cols |= (1 << col);

diags |= (1 << curDiag);

antiDiags |= (1 << curAntiDiag);

helper(result, board, row + 1, cols, diags, antiDiags);

board[row][col] = '.';

// Rollback given 'Q' at [row, col]'s other attribute as cols, diags and antiDiags with "^="

cols ^= (1 << col);

diags ^= (1 << curDiag);

antiDiags ^= (1 << curAntiDiag);

}

}

}

Time Complexity: O(N!) since we look for every valid board state.

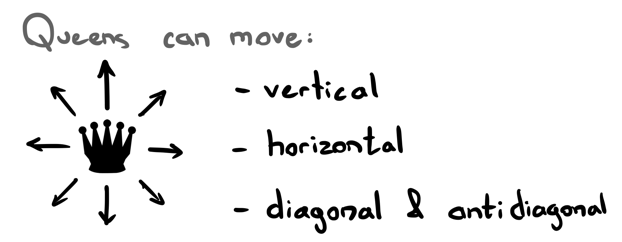
Space Complexity: O(N^2) to build our board.

**Refer to**

<https://leetcode.com/problems/n-queens/discuss/2107776/Explained-with-Diagrams-or-Backtracking-and-Bit-manipulation>

**Logic:**

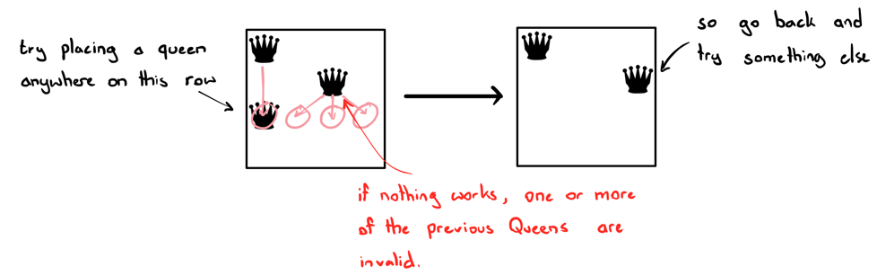
The best way to generate all valid Queen positions is through backtracking. First, we need to understand how a Queen moves and what constitutes a valid position. Queen's can move in literally any straight-lined direction.



It's possible to place a Queen down if and only if:

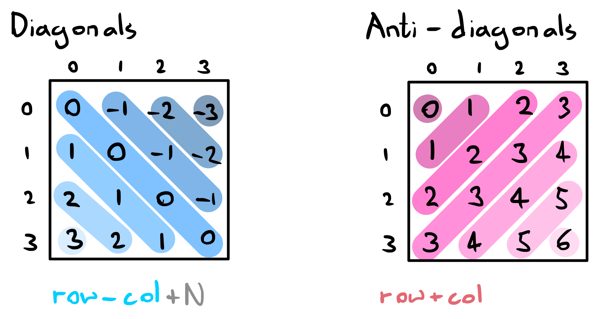
* There exists no Queen on the current row.
* There exists no Queen on the current column.
* There exists no Queen on the current diagonal.
* There exists no Queen on the current anti-diagonal.

Backtracking: Try a promising Queen position, see how it goes. If it fails, undo that Queen and try again somewhere else.



Hopefully it's now clear that we need some way to keep track of Queens on previous rows, columns and diagonals. We can take care of rows automatically by incrementing each queen placement by row. In other words, after each successful Queen placement, we move to the next row (since no two Queens can share the same row).

We can keep track of previous columns just by their column indexes. What about the diagonals? Well here's an interesting observation:



(the "+N" is to offset negative values. You will see why we do this in the next section). Now we have everything we need to keep track of previous Queens and start thinking about our algorithm! But just one more thing; let's keep track of "used" columns and diagonals using bit masks.

**Why Integer Bit Masks?**

We prefer integer bitmasks in this question for 2 main reasons:

1. Sets in Java are a slow data structure. So keeping track of visited columns and diagonals using integers is much quicker through bit manipulation!
2. It's slightly more space efficient since we're only storing three integers instead of arrays or other data structures.

**Bit Manipulation Tricks**

1. **Check the ith bit:** x & (1 << i), where 1 << i is shifting 1 to the left i number of times.

For example, if we want to check if a Queen exists at column 5, we can check if (cols & (1 << 5) != 0). Basically, if we haven't seen a Queen at this column before, then the bitwise AND operation at that specific bit will be 0. Otherwise, it will be a non-zero value.

1. **Set the ith bit:** x |= (1 << i). This performs a bitwise OR operation on the ith bit. This will always set the ith bit to 1. For example, if the ith bit in cols is currently 1, then 1 | 1 = 1. If it's set to 0, then 0 | 0 is still = 1.
2. **Flip the ith bit:** x ^= (1 << i) To reverse the previous action, we use the inverse logic of an OR; a bitwise XOR (exclusive-OR). The basic idea of XOR is that if two bits are the same (0,0) and (1,1), you will get 0. However if the two bits are different (1,0) or (0,1), you get 1. This is the exact opposite of OR, hence the name.

**Algorithm:**

Awesome! Now we have everything we need. Here's how the backtracking algorithm will work:

1. Check if we've reached the end:
   * If row == N, we've filled in all our rows successfully which implies the current board state is a valid combination. Let's add it to our output list.
2. Loop through each column in the current row.
   1. If we can't add a Queen at this position, skip this col value.
   2. If we can, add a Queen at this position and adjust our bitmasks respectively.
   3. Continue to the next row (call backtrack for row+1).
   4. Undo our changes so we can try other col values.

**Code:**

If you have any questions, suggestions or improvements, feel free to let me know. Thank you for reading!

class Solution {

private List<List<String>> res;

private int N;

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

res = new ArrayList<>();

N = n;

char[][] emptyBoard = new char[N][N];

for (char[] row: emptyBoard) Arrays.fill(row, '.');

backtrack(emptyBoard, 0, 0, 0, 0);

return res;

}

private void backtrack(char[][] board, int row, int cols, int diags, int antiDiags) {

// if we've successfuly placed a Queen at all rows, we have a valid board state

if (row == N) {

res.add(toBoard(board));

return;

}

for (int col=0; col<N; col++) {

int currDiag = row-col+N;

int currAntiDiag = row+col;

// check if the current Queen placement is valid

if ((cols & (1 << col)) != 0 || (diags & (1 << currDiag)) != 0 || (antiDiags & (1 << currAntiDiag)) != 0) continue;

// if so, add changes

board[row][col] = 'Q';

cols |= (1 << col);

diags |= (1 << currDiag);

antiDiags |= (1 << currAntiDiag);

// continue to the next row

backtrack(board, row + 1, cols, diags, antiDiags);

// undo changes and continue

board[row][col] = '.';

cols ^= (1 << col);

diags ^= (1 << currDiag);

antiDiags ^= (1 << currAntiDiag);

}

}

private List<String> toBoard(char[][] board) {

List<String> newBoard = new ArrayList<>();

for (char[] row: board) newBoard.add(new String(row));

return newBoard;

}

}

**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!)

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