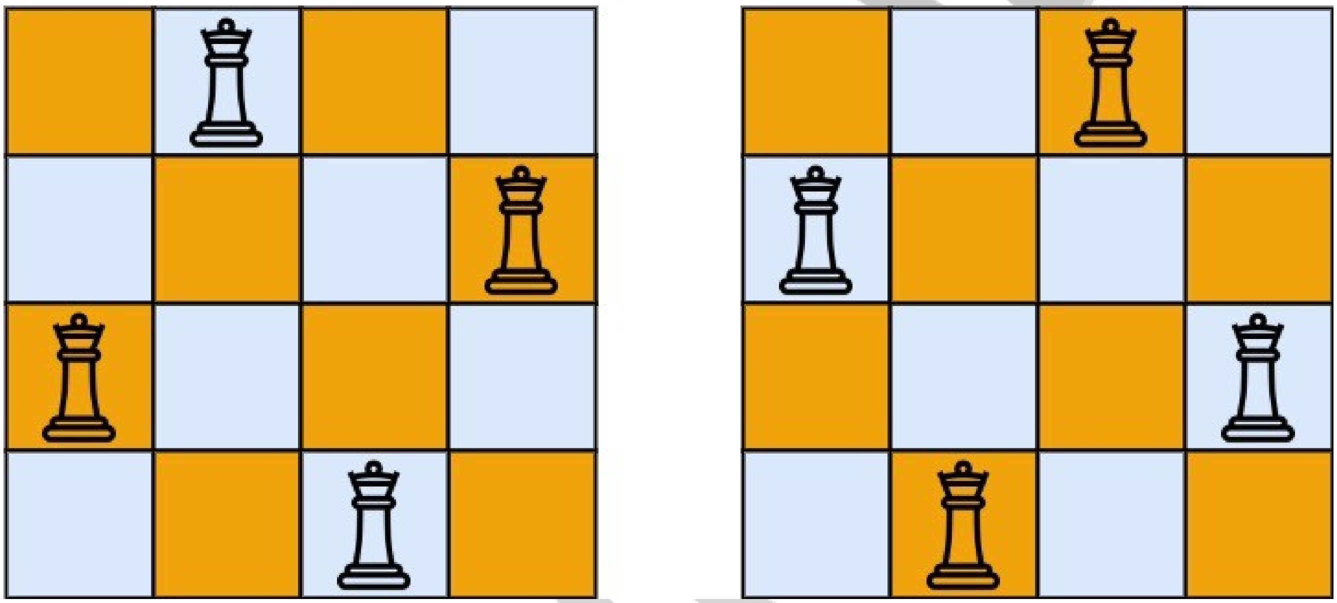
leetcode: 52.N-Queens II

Problem statement

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.
```

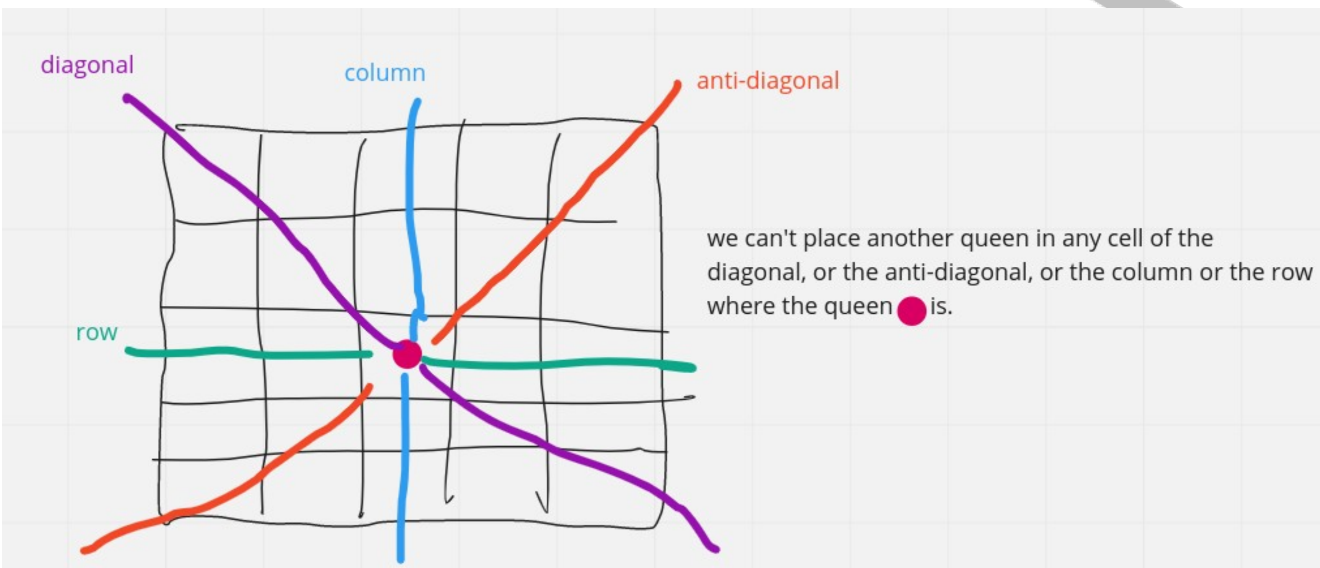
Example 2:

```
Input: n = 1
Output: 1
```

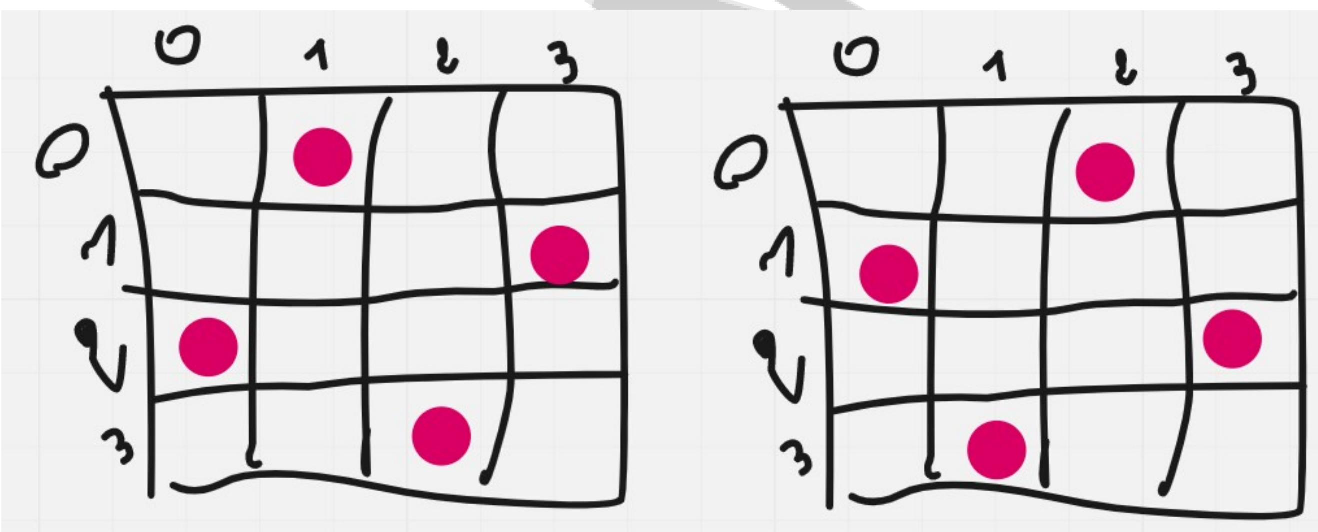
Constraints:

- `1 <= n <= 9`

Understand the problem



For a board of size 4x4, we have two solutions:



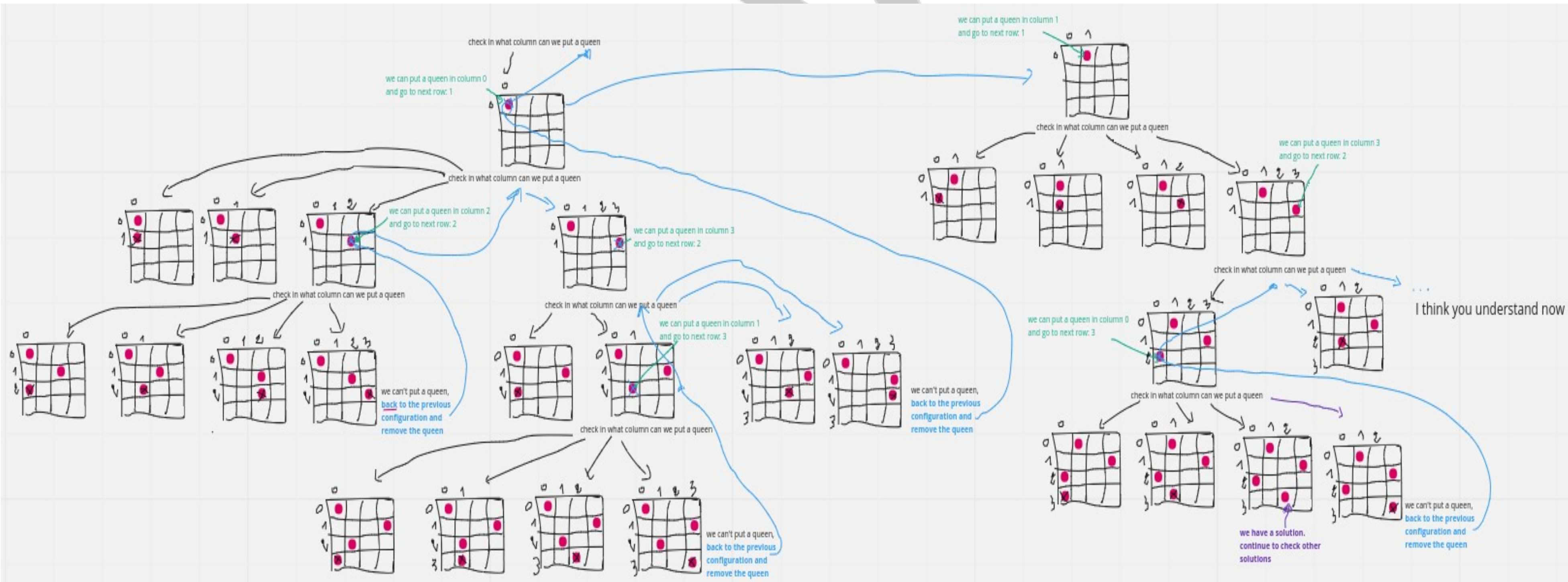
Main idea

See (***Fig.1***)

for a queen Q in row i , for all columns from 0 to $n-1$, check if we can not place another queen Q' in any column at row $i+1$, <u>**back to the previous row**</u> i and check if we can place the queen Q at another column.

We find a solution when we place the last queen in a valid column at row $n-1$.

Repeat all the above process until we check all configurations and every time we found a solution, display it.



<u>Fig.1</u>
Part of the hand execution of the process

Pseudo code

We're going to use the power of non-terminal recursive function to do the backtracking. (**F*ig.2***)

At row row , we're going to check if we can put a queen in any column from 0 to $n-1$.

If yes: check if we can put a queen in any column from 0 to $n-1$ for the next row $row+1$,

if no: it means the *for loop* is terminated, back to the previous call of the function $solve()$ and terminate it.

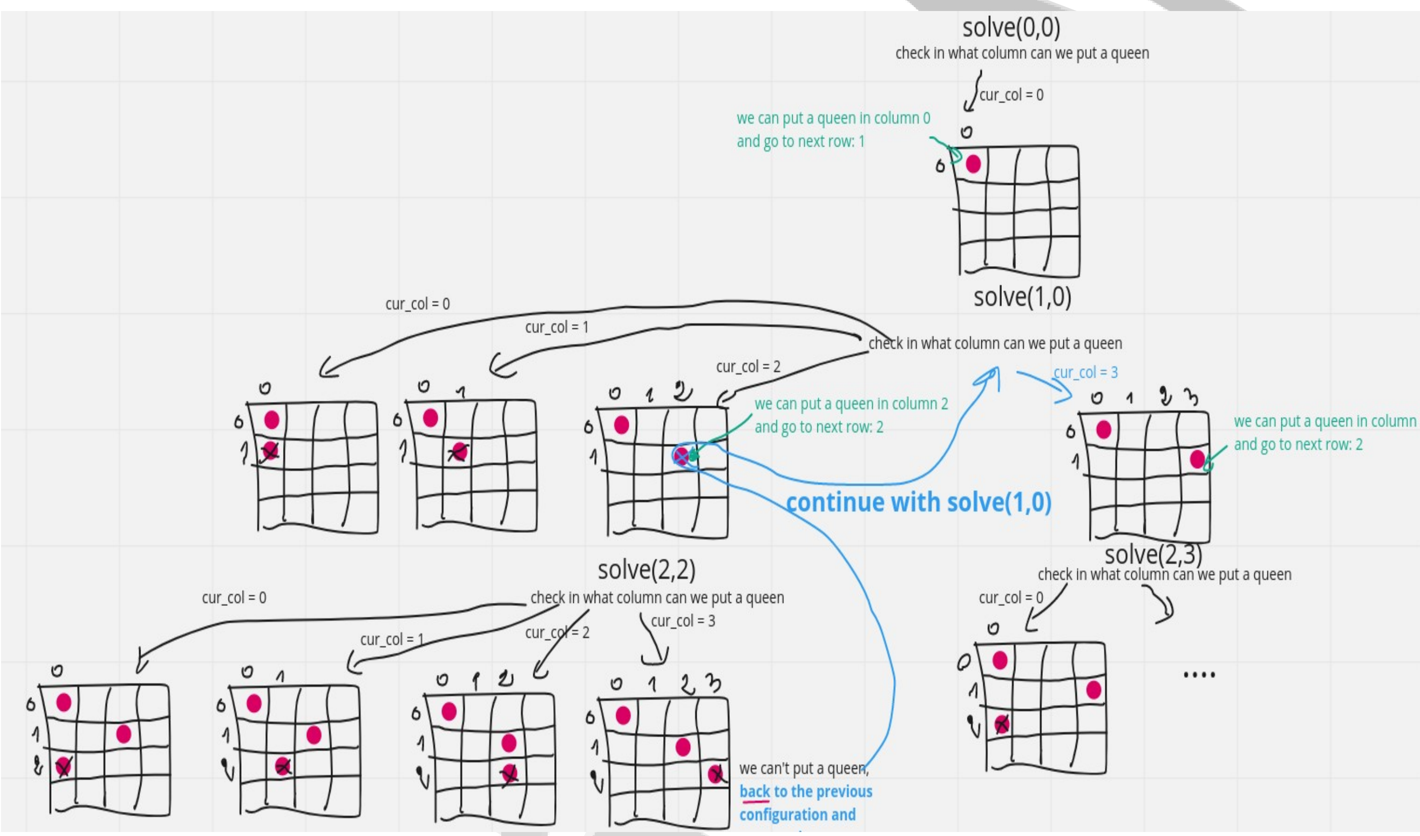


Fig.2
Part of the recursion tree

Same for if we find a solution, continue with the previous call.

At each base case and end of the *for loop* , the program will pop a call from the calls stack until all calls are terminated.

```
void solve(string board[][], int row, int col, int n) {
    // Base case
    // Remember that the last row is at  n−1 , if row reach n that means we have a valid queen
    at //row  n−1 .
    if (row == n) {
        ans++;
        // Backtracking: Trigger the last call of the function solve, goes to the previous configuration
        // No need to a return, but I don't know if it will works will all C++ compilers
        return;
    }
    // At the row  row , check in which column, we can put a queen
    for cur_col ∈ [0,n-1]{
        if we can not put a queen in cell (row,cur_col), then go the the next column
        otherwise:
        put a queen in cell (row,cur_col)
        solve the next row: solve(board,row+1,cur_col)  // go to next row
        remove the queen in cell (row,cur_col)
    }
    // Backtracking: Trigger the last call of the function solve, goes to the previous configuration
    // No need to a return, but I don't know if it will works will all C++ compilers
    return;
}
```

Now let's focus on how to know if we can put a queen at $cell(row, col)$?

Can we place a queen?: Naive approach

The obvious approach is to check:

- check the column col cells above the row row :
  $board[i][col]$, where $0 \le i \le row-1$ (***Fig.3***)
- check the diagonal cells above the row row :
  $board[i][j]$, where $0 \le i \le row-1$ and $0 \le j \le col-1$ (***Fig.4***)
- check the anti-diagonal cells above the row row :
  $board[i][j]$, where $0 \le i \le row-1$ and $col+1 \le j \le n-1$ (***Fig.5***)

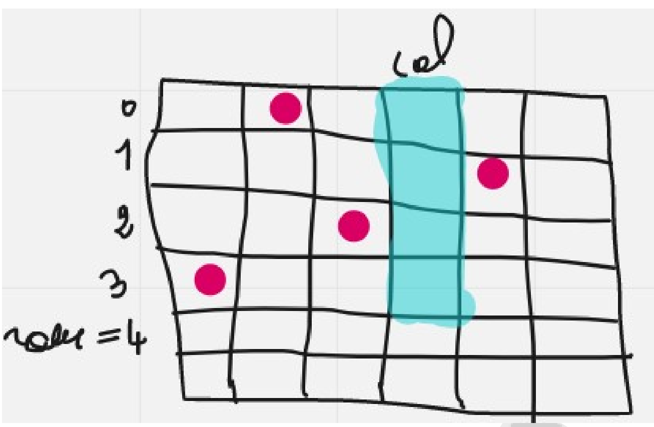


Fig.3
Check if a queen exists in the column of the cell(row,col)

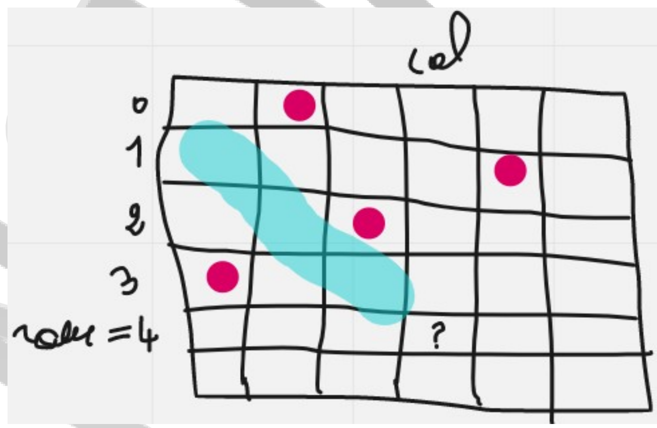


Fig.4
Check if a queen exists in the diagonal of the cell(row,col)

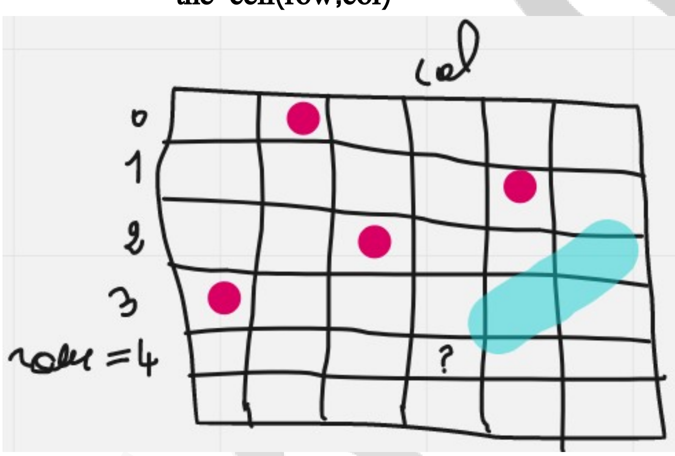


Fig.5
Check if a queen exists in the anti-diagonal of the cell(row,col)

this is the pseudo code of the function:

$$\text{we\_can\_put\_a\_queen(board,row,col)} = \begin{cases} true, if\ there\ is\ no\ other\ queen \in column\ col \text{ and } diagonal \text{ and } anti-diagonal \\ false, otherwise \end{cases}$$

```
we_can_put_a_queen(string board[][], int row, int col, int n){
  for i ∈[0,row-1]{
     if board[i][col] == "Q" return false;
  }
  for i ∈[row-1,0,-1] and j ∈[col-1,0,-1] {
     if board[i][j] == "Q" return false;
  }
  for i ∈[row-1,0,-1] and j ∈[col+1,n-1] {
     if board[i][j] == "Q" return false;
  }
  return true;
}
```

Time complexity is $O(n)$

space complexity is $O(n^2)$

Whole code: C++

```
#include <bits/stdc++.h>
/*
 * @lc app=leetcode id=52 lang=cpp
 *
 * [52] N-Queens II
 */

// @lc code=start
class Solution {
  private:
    int ans = 0;

  public:
    bool we_can_put_a_queen_in_column(int row, int col, int n,std::vector<std::vector<std::string>> board){
        // Check if a queen exists in same column
       // No need to check below the row
        for (int i = 0 ; i < row ; ++i)
          if (board[i][col] == "Q") return false;

        // Check if a queen exists in the diagonal of the given cell
        // No need to check the bottom side of the diagonal
        for(int i = row-1 , j = col-1 ; i >= 0 && j >= 0 ; --i,--j)
          if (board[i][j] == "Q") return false;

        // Chech if a queen exists in the anti-diagonal
        // No need to check the bottom side of the diagonal
        for(int i = row-1 , j = col+1 ; i >=0 && j < n ; --i,++j)
          if (board[i][j] == "Q") return false;

        return true;
      }

      void solve (std::vector<std::vector<std::string>> board, int row, int col, int n){
        if (row == n){
          ans++;
          return;
        }

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

          if (!we_can_put_a_queen_in_column(row, cur_col, n, board)) continue;

          board[row][cur_col] = "Q";

          solve(board,row + 1, cur_col, n);

          board[row][cur_col] = ".";
        }
        return;
      }
```

```
    int totalNQueens(int n) {
        std::vector<std::vector<std::string>> board(n, std::vector<std::string>(n, "."));
        solve(board,0,0,n);
        return ans;
    }
};
// @lc code=end
```

https://github.com/Mourad-NOUAILI/leet-code/blob/main/52.%20N-Queens-II/52.n-queens-ii-bf.cpp

The time complexity: $O(nxn!)$

Space complexity is $O(n^2)$

We can improve the time complexity of the function we_can_put_a_queen_in_column() to $O(1)$

Let's see how.

Can we place a queen?: efficient approach

To be honest, I didn't figure out this solution, I found it in YouTube.

The principle is easy, if a queen exists in a cell, set 1 to a specific position in the ***three* bitmask** of the columns and/or diagonals and/or anti-diagonals, otherwise set it to 0 .

First let's remind some bit-wise manipulations.

- To know if the i^{th} bit of a number x is set or not, shift the digit 1 by i places to the left, then make a $bitwise\,\mathrm{AND}$ with x . $x\,AND(1 \ll i)$

examples:

$$x=(12)_{10}=(0000\mathbf{1}100)_2$$

Is the bit #3 set in x ?

$$1 \ll 3=00000001 \ll 1=00001000$$

$$12\,AND\,00001000=00001100\,AND\,0001000=00010000=0001000=(8)_{10}$$

result ≠ 0, so the answer is yes.

Is the bit #5 set in x ?

$$x=(12)_{10}=(00\mathbf{0}01100)_2$$

$$1 \ll 5=00000001 \ll 1=00100000$$

$$12\,AND\,00100000=00001100\,AND\,00100000=00000000=0=(0)_{10}$$

result = 0 , so the answer is No.

- To set the i^{th} bit, shift the digit 1 by i places to the left, then make a $bitwise\,\mathrm{OR}$ with x . $x\,OR(1 \ll i)$

Example:

Set the bit #5 in x ?

$$x=(12)_{10}=(00\mathbf{0}01100)_2$$

$$1 \ll 5=00000001 \ll 1=00100000$$

$$12\,OR\,00100000=00\mathbf{0}01100\,OR\,00100000=00\mathbf{1}01100=(44)_{10}$$

- To flip the i^{th} bit, shift the digit 1 by i places to the left, then make a $bitwise\,\mathrm{XOR}$ with x . $x\,XOR(1 \ll i)$

Example:

Flip the bit #5 in $x=(12)_{10}$?

$x=(12)_{10}=(00\mathbf{0}01100)_2$

$1 \ll 5 = 00000001 \ll 1 = 00100000$

$12\,XOR\,00100000 = 00\mathbf{0}01100\,XOR\,00100000 = 00\mathbf{1}01100 = (44)_{10}$

How can this be helpful?

For a $n \times n$ board:

- column n refer to LSB and column 0 refer to MSB,
- diagonal $2n-1$ refer to LSB and diagonal $2(n-1)$ refers to MSB
- anti-diagonal 0 refers to LSB and anti-diagonal $2(n-1)$ refers to MSB

Example: n=4

The figures (***Fig.6***, ***Fig.7*** and ***Fig.8***) show respectively how to attribute the bits of the bitmasks of the columns, the diagonals and the anti-diagonals.

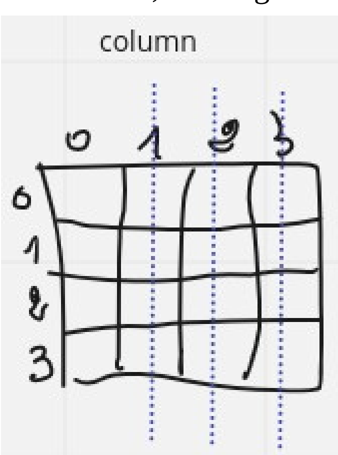


Fig.6
Column #0 refers to LSB
Column #3 refers to MSB

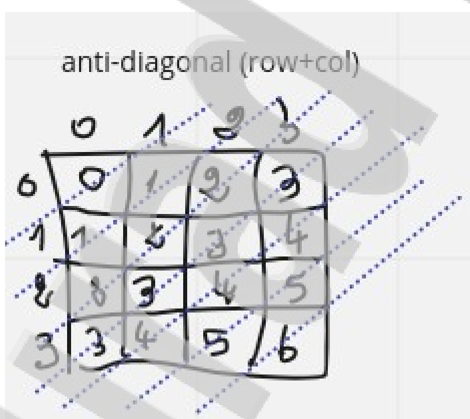


Fig.7
Column #7 refers to LSB
Column #6 refers to MSB

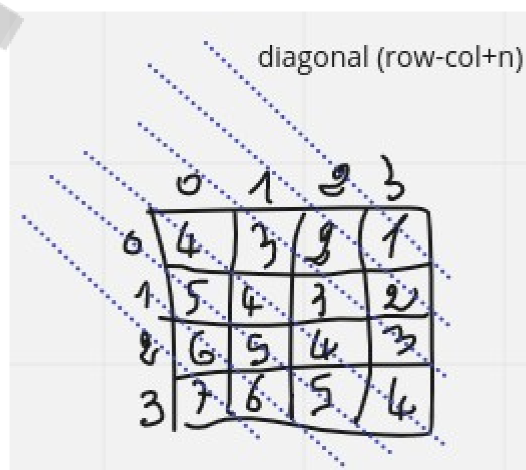
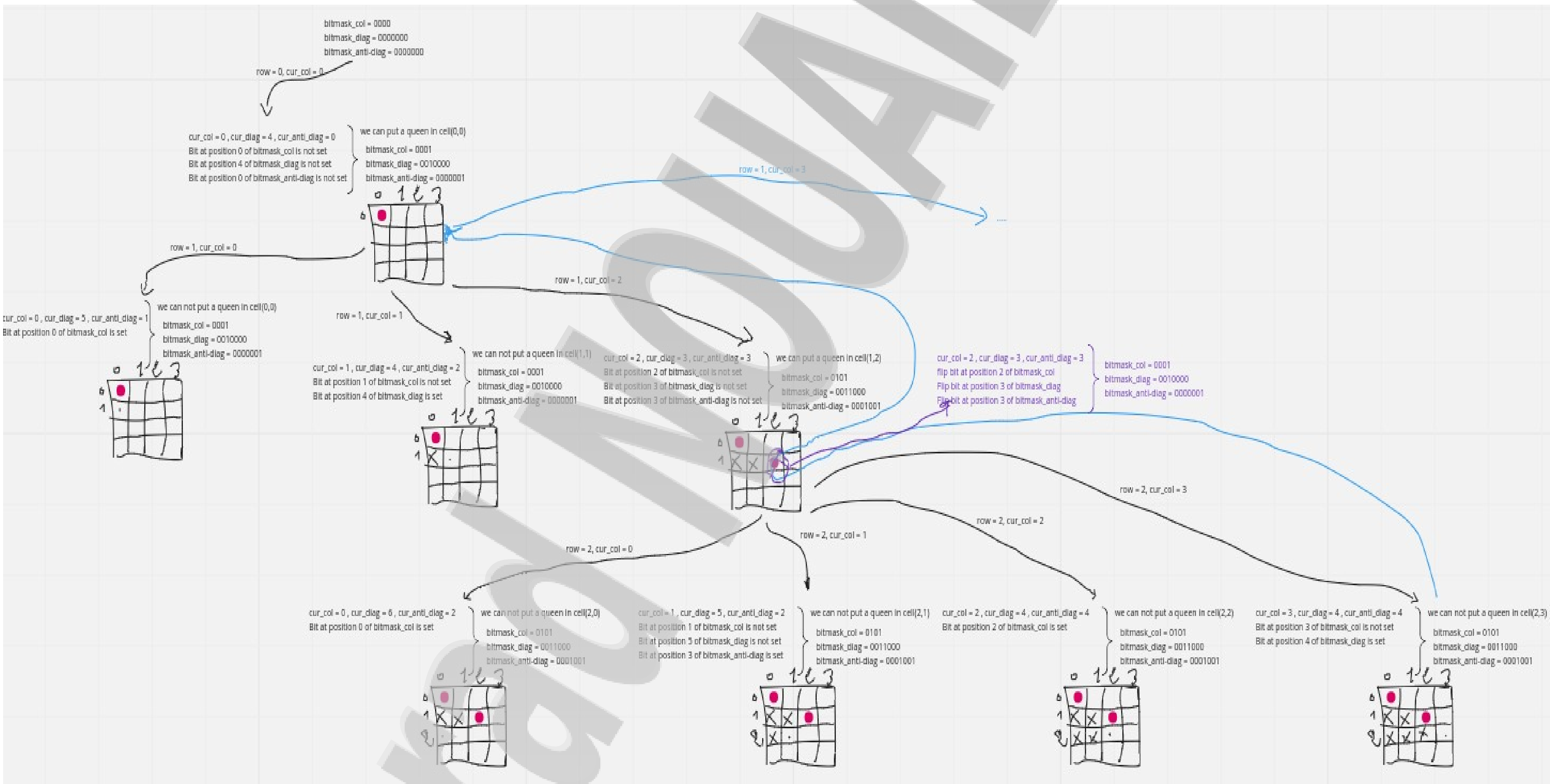


Fig.6
Column #0 refers to LSB
Column #6 refers to MSB

https://github.com/Mourad-NOUAILI/leet-code/tree/main/52.%20N-Queens-II

To understand more, let’s draw a part of the recursion tree for a 4×4 board:



Whole code: C++

```cpp
#include <bits/stdc++.h>
/*
 * @lc app=leetcode id=52 lang=cpp
 *
 * [52] N-Queens II
 */
// @lc code=start
class Solution {
  private:
      int ans = 0;

   public:
      void solve (int row, int bitmask_col, int bitmask_diag, int bitmask_anti_diag, int n){
         if (row == n){
            ans++;
            return;
         }
         for (int cur_col = 0 ; cur_col < n ; ++cur_col){
            int cur_diag = row-cur_col+n;
            int cur_anti_diag = row+cur_col;
            if ( (bitmask_col & (1 << cur_col)) != 0  || (bitmask_diag & (1 << cur_diag)) != 0 || (bitmask_anti_diag & (1 << cur_anti_diag)) != 0 ) continue;
            bitmask_col |= (1 << cur_col);
            bitmask_diag |= (1 << cur_diag);
            bitmask_anti_diag |= (1 << cur_anti_diag);
            solve(row + 1, bitmask_col, bitmask_diag, bitmask_anti_diag, n);
            bitmask_col ^= (1 << cur_col);
            bitmask_diag ^= (1 << cur_diag);
            bitmask_anti_diag ^= (1 << cur_anti_diag);
         }
         return;
      }
      int totalNQueens(int n) {
         solve(0,0,0,0,n);
         return ans;
      }
};
// @lc code=end
```

Time complexity: $O(n!)$, Space complexity: $O(1)$

https://github.com/Mourad-NOUAILI/leet-code/tree/main/52.%20N-Queens-II


52.n-queens-ii.cpp
52-n-queen-ii-backtracking > 52.n-queens-ii.cpp > ...
#include <bits/stdc++.h>
/*
* @lc app=leetcode id=52 lang=cpp
*
* [52] N-Queens II
*/
// @lc code=start
class Solution {
private:
int ans = 0;
public:
void solve (int row, int bitmask_col, int bitmask_diag, int bitmask_anti_diag,
if (row == n){
ans++;
return;
}
for (int cur_col = 0 ; cur_col < n ; ++cur_col){
int cur_diag = row-cur_col+n;
int cur_anti_diag = row+cur_col;
if ((bitmask_col & (1 << cur_col)) != 0 || (bitmask_diag & (1 << cu
bitmask_col |= (1 << cur_col);
bitmask_diag |= (1 << cur_diag);
bitmask_anti_diag |= (1 << cur_anti_diag);
solve(row + 1, bitmask_col, bitmask_diag, bitmask_anti_diag, n);
bitmask_col ^= (1 << cur_col);
bitmask_diag ^= (1 << cur_diag);
bitmask_anti_diag ^= (1 << cur_anti_diag);
}
return;
}
int totalNQueens(int n) {
solve(0,0,0,0,n);
return ans;
}
};
Submit | Test | Debug | Debug Input
// @lc code=end
Submission
Accepted
9/9 cases passed (0 ms)
Your runtime beats 100 % of cpp submissions
Your memory usage beats 76.91 % of cpp submissions (6 MB)