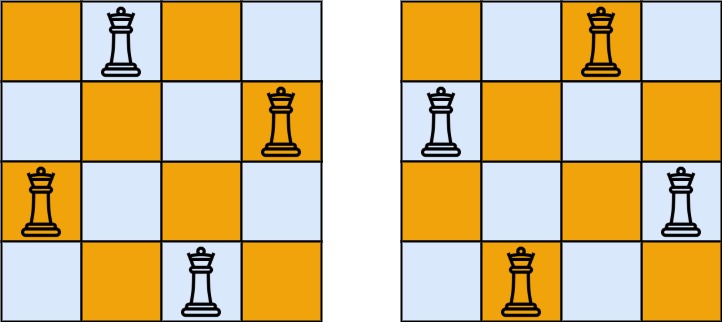
# Question

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

# Solution

#### **Intuition**

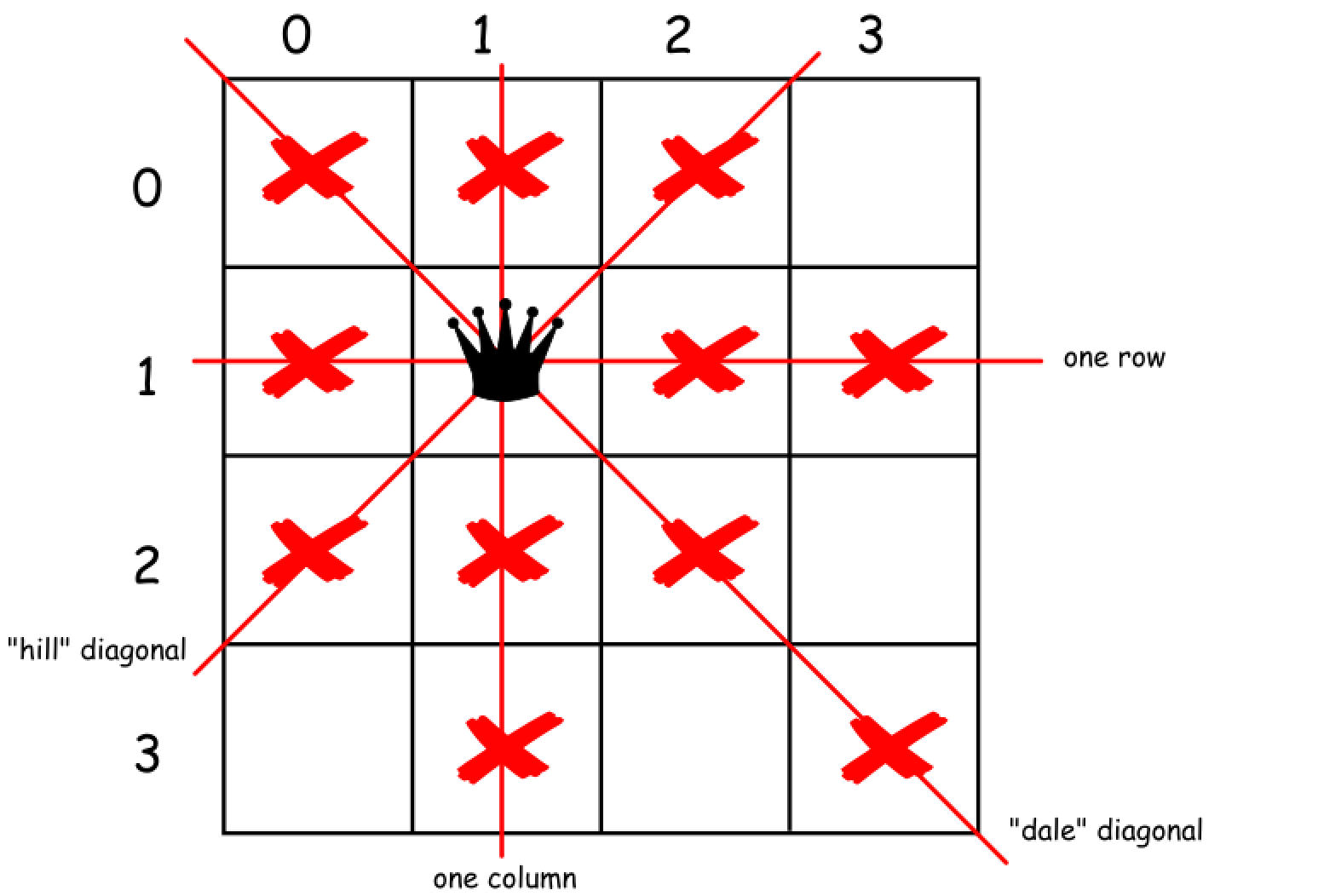
This problem is a classical one and it's important to know the solution to feel classy.

The first idea is to use brute-force that means to generate all possible ways to put N queens on the board, and then check them to keep only the combinations with no queen under attack. That means O(N^n) time complexity and hence we're forced to think further how to optimize.

There are two programming conceptions here which could help.

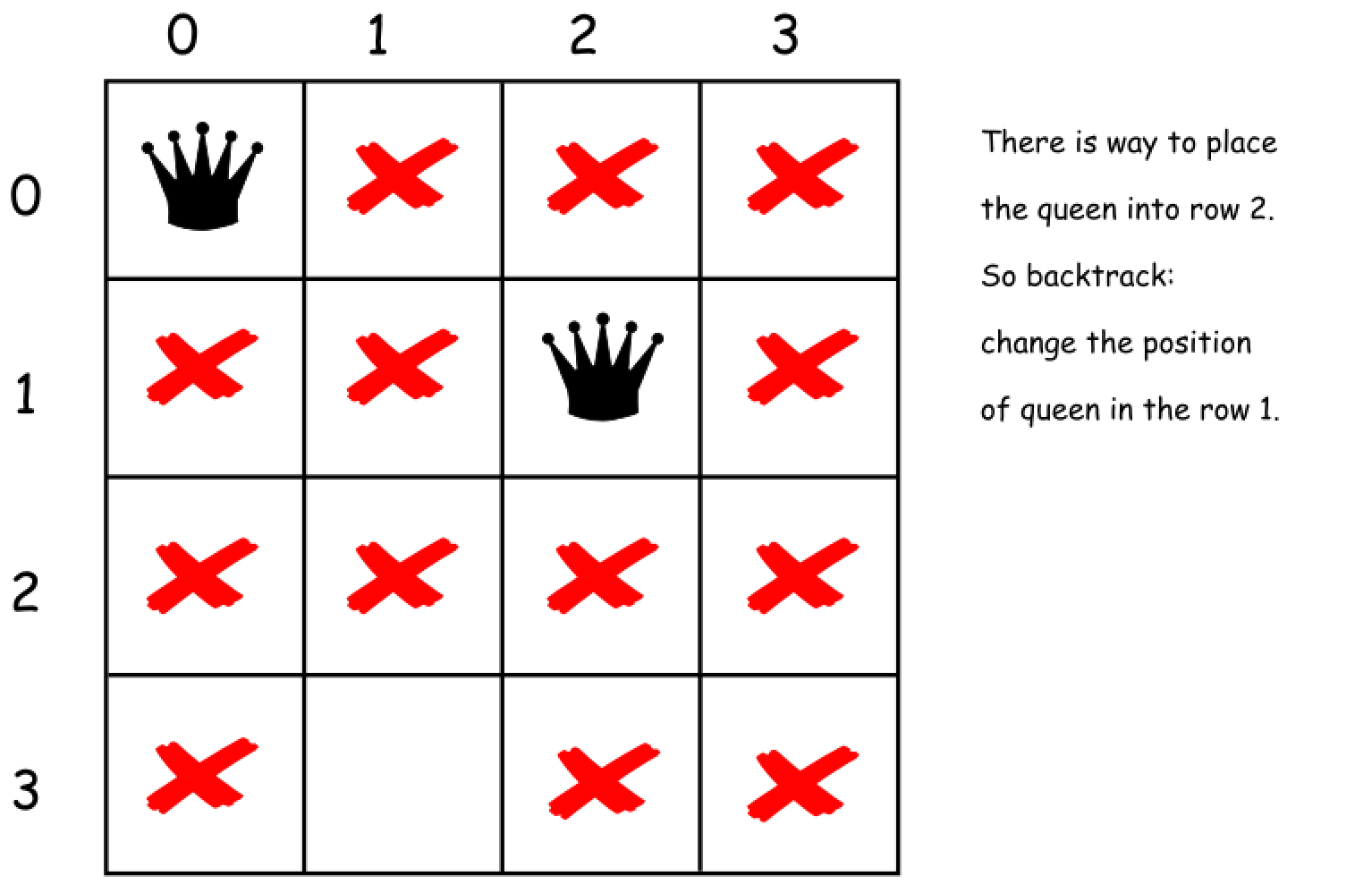
The first one is called constrained programming.

That basically means to put restrictions after each queen placement. One puts a queen on the board and that immediately excludes one column, one row and two diagonals for the further queens placement. That propagates constraints and helps to reduce the number of combinations to consider.



The second one called backtracking.

Let's imagine that one puts several queens on the board so that they don't attack each other. But the combination chosen is not the optimal one and there is no place for the next queen. What to do? To backtrack. That means to come back, to change the position of the previously placed queen and try to proceed again. If that would not work either, backtrack again.



#### **Approach 1: Backtracking**

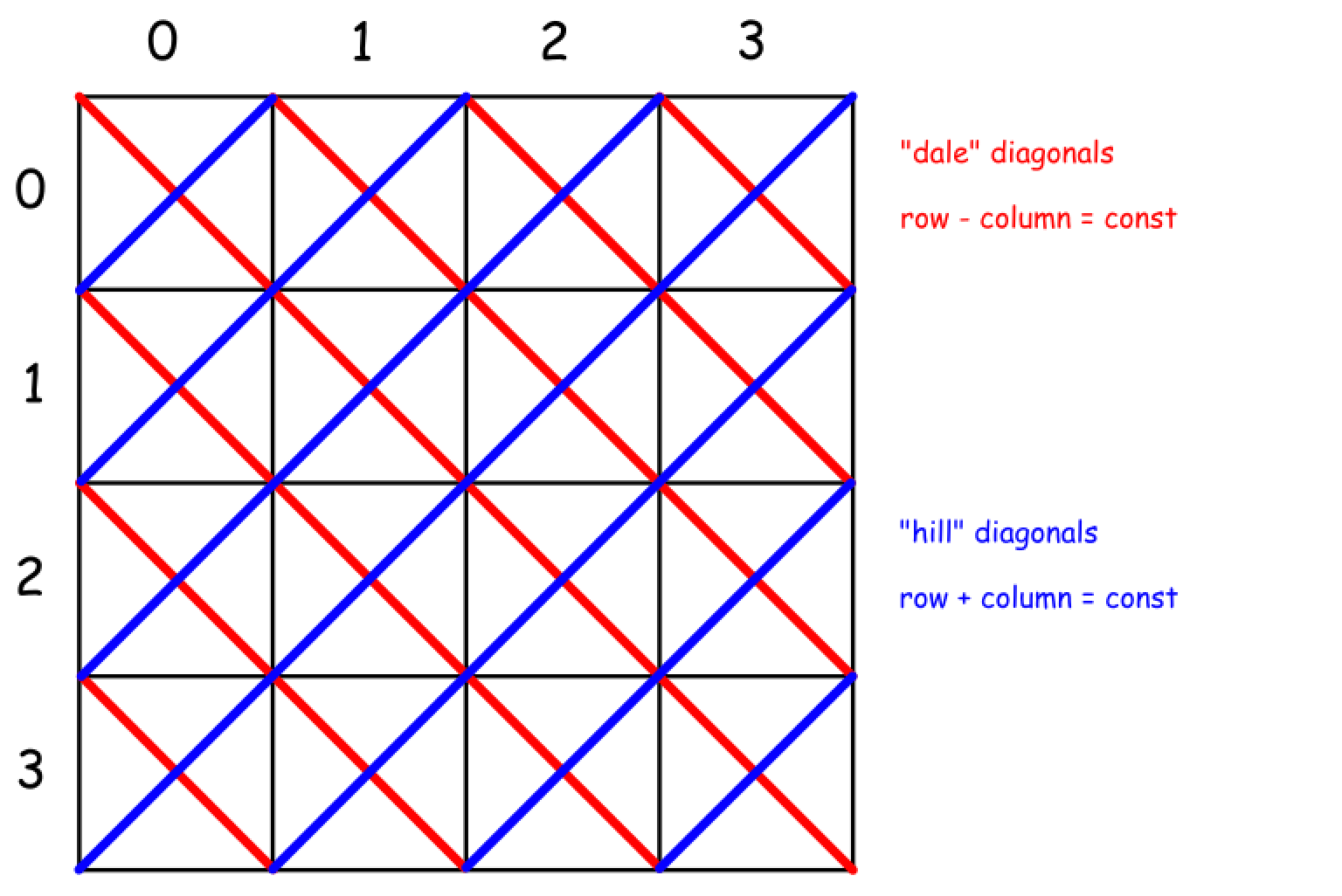
Before to construct the algorithm, let's figure out two tips that could help.

There could be the only one queen in a row and the only one queen in a column.

That means that there is no need to consider all squares on the board. One could just iterate over the columns.

For all "hill" diagonals row + column = const, and for all "dale" diagonals row - column = const.

That would allow us to mark the diagonals which are already under attack and to check if a given square (row, column) is under attack.



Now everything is ready to write down the backtrack function backtrack(row = 0, count = 0).

* Start from the first row = 0.
* Iterate over the columns and try to put a queen in each column.
  + If square (row, column) is not under attack
    - Place the queen in (row, column) square.
    - Exclude one row, one column and two diagonals from further consideration.
    - If all rows are filled up row == N
      * That means that we find out one more solution count++.
    - Else
      * Proceed to place further queens backtrack(row + 1, count).
    - Now backtrack : remove the queen from (row, column) square.

Here is a straightforward implementation of the above algorithm.

1 / 34

|  |
| --- |
| **class Solution {**  **public boolean is\_not\_under\_attack(int row, int col, int n,**  **int [] rows,**  **int [] hills,**  **int [] dales) {**  **int res = rows[col] + hills[row - col + 2 \* n] + dales[row + col];**  **return (res == 0) ? true : false;**  **}**  **public int backtrack(int row, int count, int n,**  **int [] rows,**  **int [] hills,**  **int [] dales) {**  **for (int col = 0; col < n; col++) {**  **if (is\_not\_under\_attack(row, col, n, rows, hills, dales)) {**  **// place\_queen**  **rows[col] = 1;**  **hills[row - col + 2 \* n] = 1; // "hill" diagonals**  **dales[row + col] = 1; //"dale" diagonals**  **// if n queens are already placed**  **if (row + 1 == n) count++;**  **// if not proceed to place the rest**  **else count = backtrack(row + 1, count, n,**  **rows, hills, dales);**  **// remove queen**  **rows[col] = 0;**  **hills[row - col + 2 \* n] = 0;**  **dales[row + col] = 0;**  **}**  **}**  **return count;**  **}**  **public int totalNQueens(int n) {**  **int rows[] = new int[n];**  **// "hill" diagonals**  **int hills[] = new int[4 \* n - 1];**  **// "dale" diagonals**  **int dales[] = new int[2 \* n - 1];**  **return backtrack(0, 0, n, rows, hills, dales);**  **}**  **}** |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N!)O(*N*!). There is N possibilities to put the first queen, not more than N (N - 2) to put the second one, not more than N(N - 2)(N - 4) for the third one etc. In total that results in \mathcal{O}(N!)O(*N*!) time complexity.
* Space complexity : \mathcal{O}(N)O(*N*) to keep an information about diagonals and rows.

#### **Approach 2: Backtracking via bitmap**

If you're on the interview - use the approach 1.

The next algorithm has the same time complexity \mathcal{O}(N!)O(*N*!) but works the way faster because of [bitwise operators usage](https://wiki.python.org/moin/BitwiseOperators). Kudos for this algorithm go to [takaken](http://www.ic-net.or.jp/home/takaken/e/queen/).

To facilitate the understanding of the algorithm, here is the code with step by step explanations.

|  |
| --- |
| class Solution {  public int backtrack(int row, int hills, int next\_row, int dales, int count, int n) {  /\*\*  row: current row to place the queen  hills: "hill" diagonals occupation [1 = taken, 0 = free]  next\_row: free and taken slots for the next row [1 = taken, 0 = free]  dales: "dale" diagonals occupation [1 = taken, 0 = free]  count: number of all possible solutions  \*/  // all columns available for this board,  // i.e. n times '1' in binary representation  // bin(cols) = 0b1111 for n = 4, bin(cols) = 0b111 for n = 3  // [1 = available]  int columns = (1 << n) - 1;  if (row == n) // if all n queens are already placed  count++; // we found one more solution  else {  // free columns in the current row  // ! 0 and 1 are inversed with respect to hills, next\_row and dales  // [0 = taken, 1 = free]  int free\_columns = columns & ~(hills | next\_row | dales);  // while there's still a column to place next queen  while (free\_columns != 0) {  // the first bit '1' in a binary form of free\_columns  // on this column we will place the current queen  int curr\_column = - free\_columns & free\_columns;  // place the queen  // and exclude the column where the queen is placed  free\_columns ^= curr\_column;  count = backtrack(row + 1,  (hills | curr\_column) << 1,  next\_row | curr\_column,  (dales | curr\_column) >> 1,  count, n);  }  }  return count;  }  public int totalNQueens(int n) {  return backtrack(0, 0, 0, 0, 0, n);  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N!)O(*N*!).
* Space complexity : \mathcal{O}(N)O(*N*).