Dear reader, welcome to the next problem in the Recursion & Backtracking section named [‘**NQueens Combinations - 2d as 1d - Queen Chooses**’](https://www.pepcoding.com/resources/data-structures-and-algorithms-in-java-levelup/recursion-and-backtracking/nqueens-combinations-2das1d-queen-chooses-official/ojquestion).

If somehow you have landed on this problem directly, then I must tell you that the train is on an intermediate station.

The station of origin of our train was ‘[**Queens Combinations - 2d As 2d - Box Chooses**](https://www.pepcoding.com/resources/data-structures-and-algorithms-in-java-levelup/recursion-and-backtracking/queens-combinations-2das2d-box-chooses-official/ojquestion)’ and the previous station was ‘[**Queens Combinations - 2d as 1d - Queen Chooses**](https://www.pepcoding.com/resources/data-structures-and-algorithms-in-java-levelup/recursion-and-backtracking/queens-combinations-2das1d-queen-chooses-official/ojquestion)’. Please join the journey from the beginning to experience the full joy.

Also, there is a strong prerequisite for this set of problems on permutations & combinations in 2D grid, which is ***permutations & combinations in 1D***.

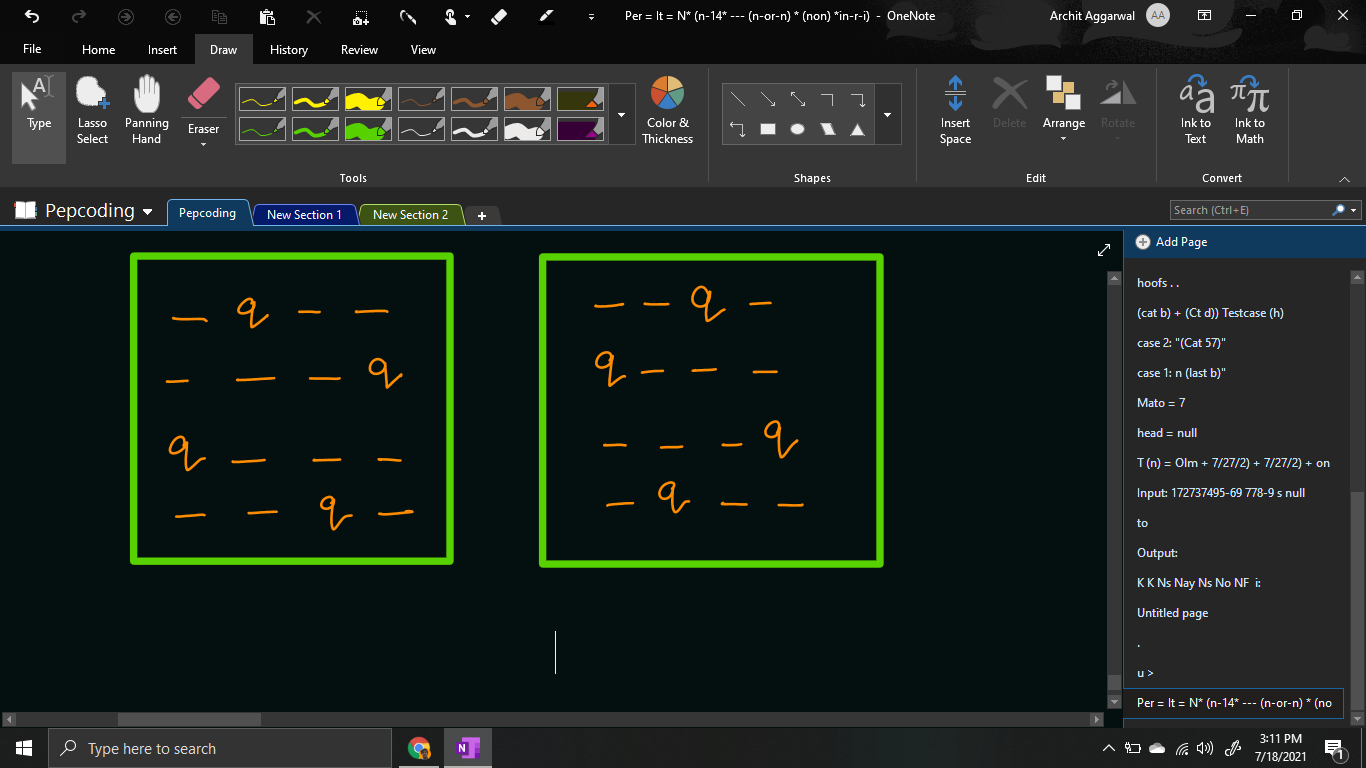
***Problem Statement:***

* You are given a number n, representing the size of a n \* n chess board.
* You are required to calculate and print the **combinations** in which n queens can be placed on the n \* n chess-board.
* Note, in this problem, you must take into consideration the fact that **no queen shall be able to kill each other**.
* A queen can kill any other queen if both are placed in the same row, or same column, or same left diagonal or same right diagonal.
* Note: Use the code snippet and follow the input/output format. The judge can't force you but the intention is to teach a concept. Play in the spirit of the question.

***Example:***

*Input*: Number of queens (n) = 4

*Output*:



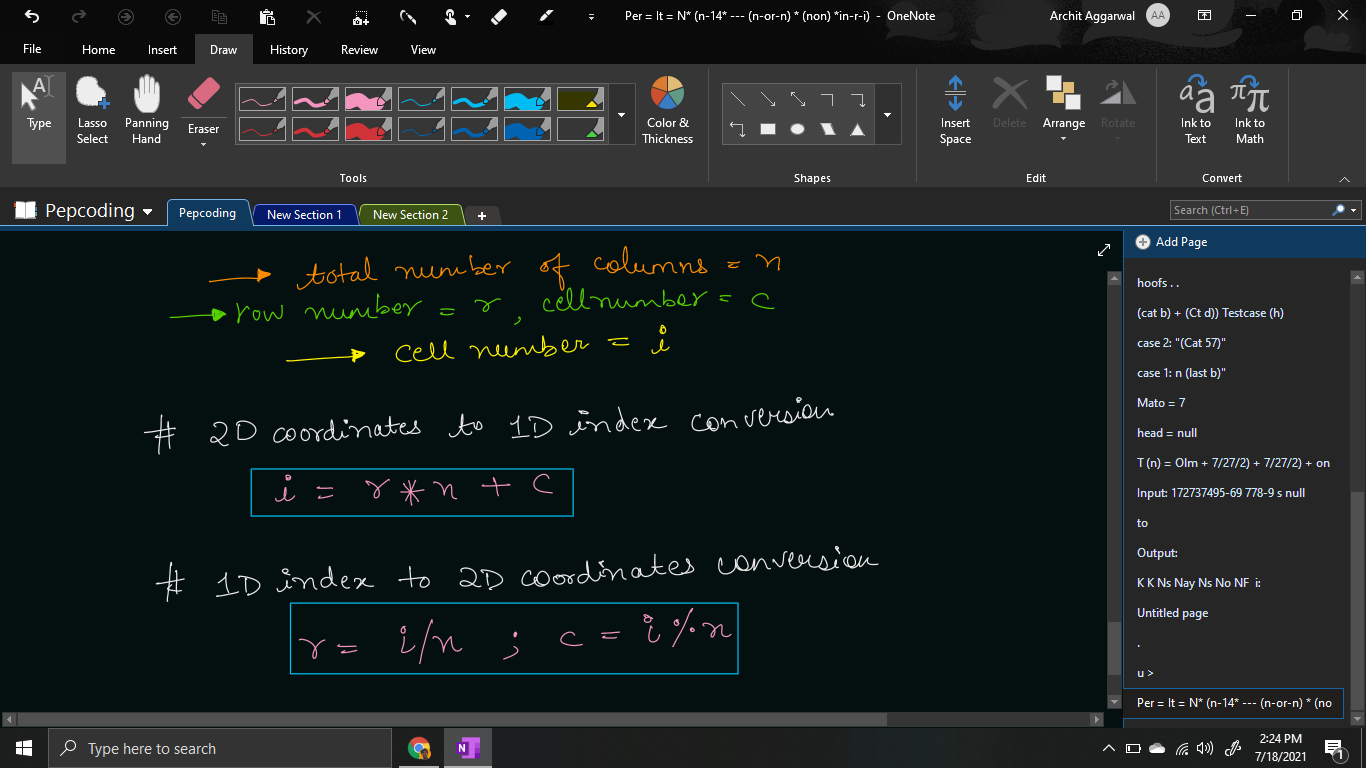
***Solution***

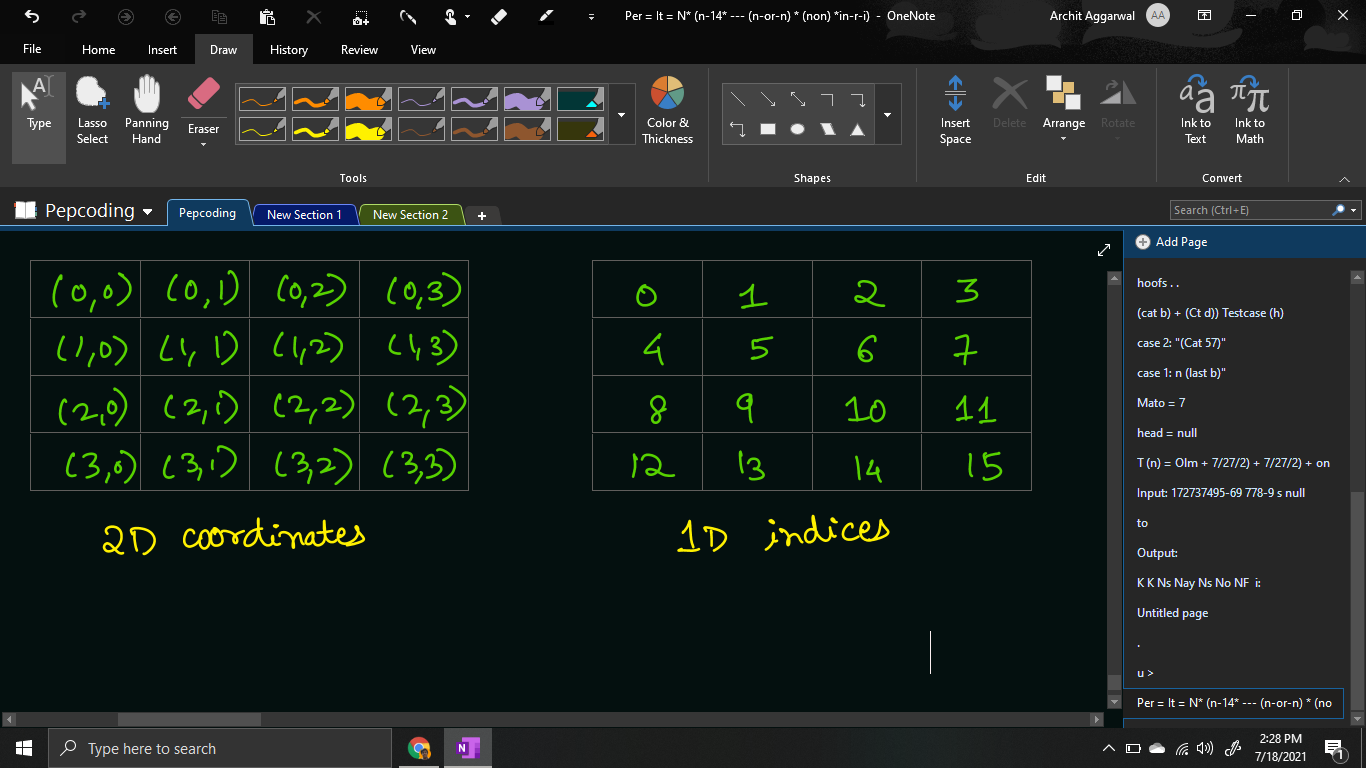
We have already learnt how to generate [**combinations**](https://www.pepcoding.com/resources/data-structures-and-algorithms-in-java-levelup/recursion-and-backtracking/combinations-2-official/ojquestion) of identical items in a 1d array by taking levels as items (in increasing order only) and choices as selecting an empty box.

In this problem, we are given the **queens as identical items**, and there is a slight variation that instead of 1d array of boxes, we are given a 2d array/grid of the chessboard.

So, we will take the **queens (in increasing order only) as the levels** in the recursion tree, and the **choice/edge will be selecting an empty cell (after the last queen’s cell)**.

Also, please remember the conversion from 2d coordinates to 1d indices of the cells, which we saw in the previous solution.





Then, what is the difference in this problem?

As mentioned in the problem statement, queens should be placed such that no two queens can kill each other.

Hence, no two queens can be in the same **row**, or in the same **column**, or in the same **left diagonal** or in the same **right diagonal**.

Hence, if the previous cell i which queen was placed had 1d index lcno, then we will traverse from all the cells in range [lcno + 1, n^2 - 1], find the 2d coordinates accordingly, and check whether placing a queen in that cell will lead to valid configuration or not.

for (int i = lcno + 1; i < chess.length \* chess.length; i++) {

int row = i / chess.length;

int col = i % chess.length;

if (chess[row][col] == false

&& IsQueenSafe(chess, row, col)){

chess[row][col] = true;

nqueens(qpsf + 1, tq, chess, row \* chess.length + col);

chess[row][col] = false;

}

}

Please note what should be the **base case** of this problem?

Base case will remain the same. It can be considered when we have made a decision for all of the queens, i.e. the queens placed so far (qpsf) is equal to the total number of queens available (n or tq). When we hit the base case, we will print the grid, by writing the ‘q’ for filled cells, else print ‘-’ followed by tab space for the empty cell.

if (qpsf == tq) {

for (int row = 0; row < chess.length; row++) {

for (int col = 0; col < chess.length; col++) {

System.out.print(chess[row][col] ? "q\t" : "-\t");

}

System.out.println();

}

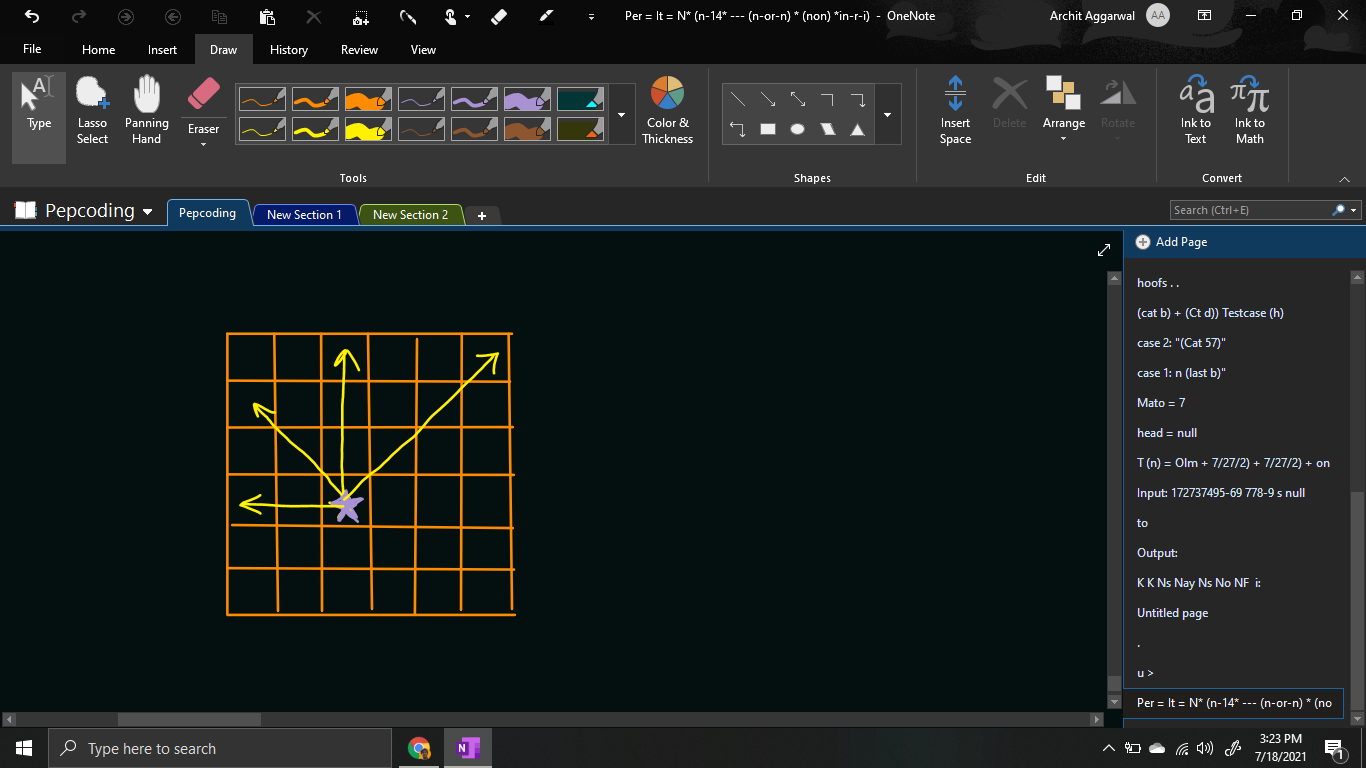
System.out.println();

return;

}

Now, the only point of difference is checking whether we can place a queen in the given cell or not. We should complete the function IsQueenSafe(chess, row, col).

We will check for any queen in the row, column, left or right diagonal, and if we find any cell filled with queen, then both can kill each other. Hence, we will return false in such a case. Else, if all the cells in 4 directions are empty, then we can return true, i.e. we can place a queen at (row, col).



**Java Code**

import java.io.\*;

import java.util.\*;

public class Main {

public static boolean IsQueenSafe(boolean[][] chess, int row, int col) {

// vertical

for (int i = row, j = col; i >= 0; i--) {

if(chess[i][j]){

return false;

}

}

// horizontal

for (int i = row, j = col; j >= 0; j--) {

if(chess[i][j]){

return false;

}

}

// diagonal

for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {

if(chess[i][j]){

return false;

}

}

// anti-diagonal

for (int i = row, j = col; i >= 0 && j < chess.length; i--, j++) {

if(chess[i][j]){

return false;

}

}

return true;

}

public static void nqueens(int qpsf, int tq, boolean[][] chess, int lcno) {

if (qpsf == tq) {

for (int row = 0; row < chess.length; row++) {

for (int col = 0; col < chess.length; col++) {

System.out.print(chess[row][col] ? "q\t" : "-\t");

}

System.out.println();

}

System.out.println();

return;

}

for (int i = lcno + 1; i < chess.length \* chess.length; i++) {

int row = i / chess.length;

int col = i % chess.length;

if (chess[row][col] == false

&& IsQueenSafe(chess, row, col)){

chess[row][col] = true;

nqueens(qpsf + 1, tq, chess, row \* chess.length + col);

chess[row][col] = false;

}

}

}

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int n = Integer.parseInt(br.readLine());

boolean[][] chess = new boolean[n][n];

nqueens(0, n, chess, -1);

}

}

Java Code is written and explained by our team in the [solution video](https://www.youtube.com/watch?v=zOmrEPnrrJQ&list=TLGGhyfYkSFJq1IxODA3MjAyMQ). Please refer to it for a better understanding of the algorithm and the implementation.

* What is the ***time complexity*** of the above code?

In the recursion tree, we are having queens as levels, and the choices as selecting an empty box. Since levels are n (queens) and the total cells (choices) are n^2, hence the total time complexity should be O(n^2 \* n^2 \* …. n times) = O(n2n).

But before placing any queen, we are checking whether placing it will result in valid configuration or not. Checking for validity requires traversal over O(n) cells in 4 directions. Hence, the time complexity will be O(n^3 \* n^3 \* …. n times) = **O(n3n)**.

* What is the ***space complexity*** of the above code?

Since, the maximum depth of recursion is equal to the number of queens = n, hence the space complexity will be **O(n)**, as recursion takes function call stack space.

**Follow Up:** Please note that we have already solved the [Nqueen problem](https://www.pepcoding.com/resources/online-java-foundation/recursion-backtracking/n-queens-official/ojquestion) in the level 1 section of recursion & backtracking. Here, the time complexity is poor as we are exploring all the cells as possible options. So, please go through the optimized algorithm discussed there.

Hope that you liked the article. Subscribe to Pepcoding’s youtube channel for more such amazing video content on Data Structures & Algorithms. You can suggest any improvements to the article on our telegram channel, or on the youtube channel’s comment section.

Article Contributed by: [Archit Aggarwal](https://www.linkedin.com/in/archit-aggarwal-6a7716189/)