Dear reader, welcome to the next problem in the Recursion & Backtracking section named ‘[**NKnight Combinations - 2d as 1d - Knight Chooses**](https://www.pepcoding.com/resources/data-structures-and-algorithms-in-java-levelup/recursion-and-backtracking/nknights-combinations-2das1d-knight-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 [‘**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). 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 knights can be placed on the n \* n chess-board.
* Note, in this problem, you must take into consideration the fact that **no knight shall be able to kill each other**.
* A knight can kill any other knight if it is one knight’s move from it.
* 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 knights (n) = 3

*Output*:



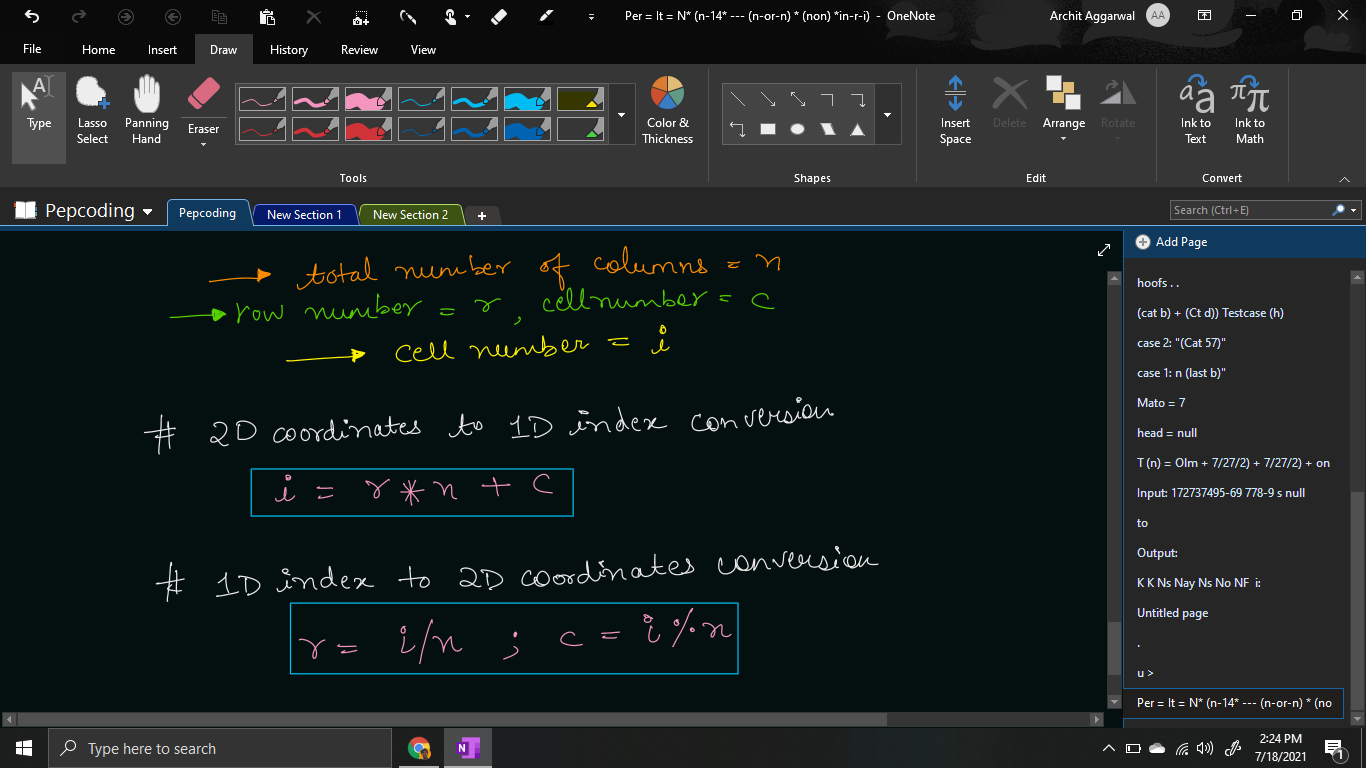
***Solution***

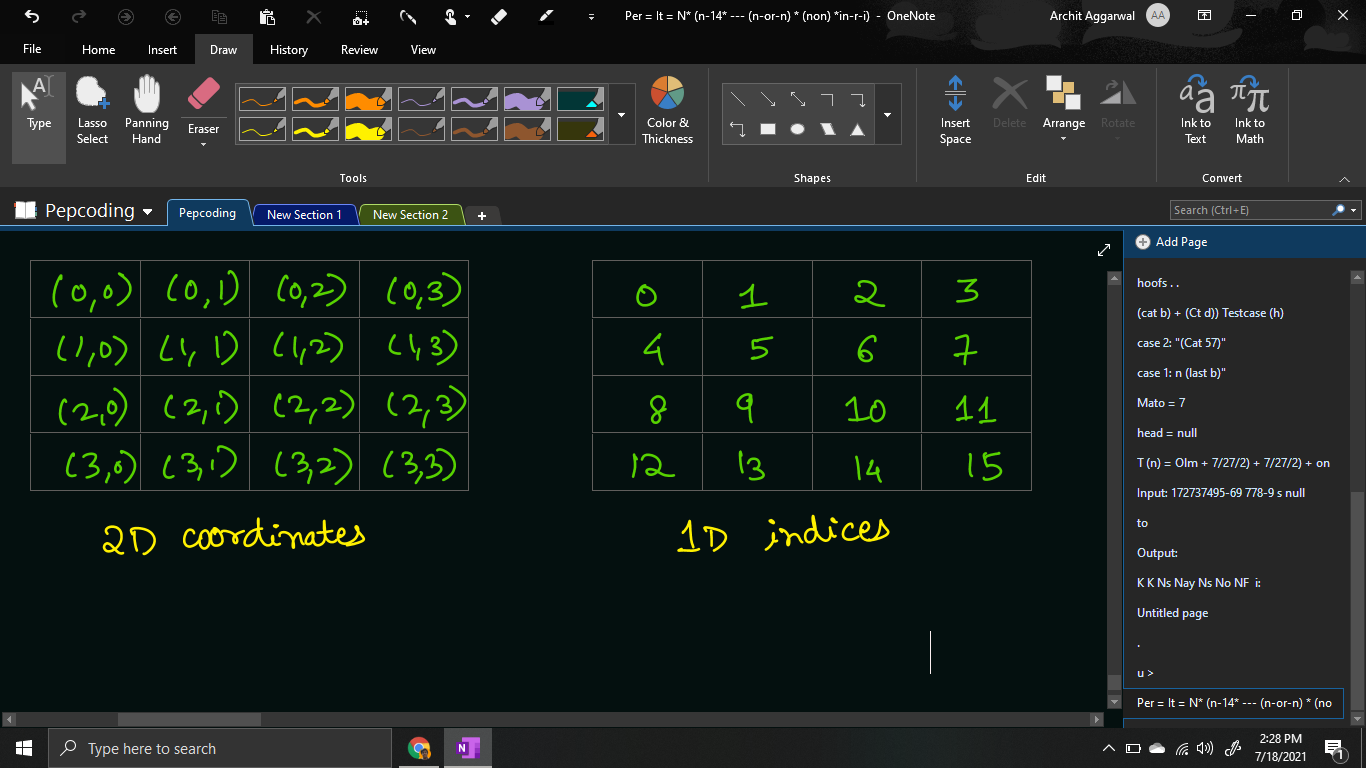
We have already learnt how to generate [**combinations**](https://www.pepcoding.com/resources/data-structures-and-algorithms-in-java-levelup/recursion-and-backtracking/combinations-2-official/ojquestion) of identical items in a 1d array by taking levels as items (in increasing order only) and choices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

In this problem, we are given the **knights 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 **knights (in increasing order only) as the levels** in the recursion tree, and the **choice/edge will be selecting an empty cell (after the last knight’s cell)**.

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

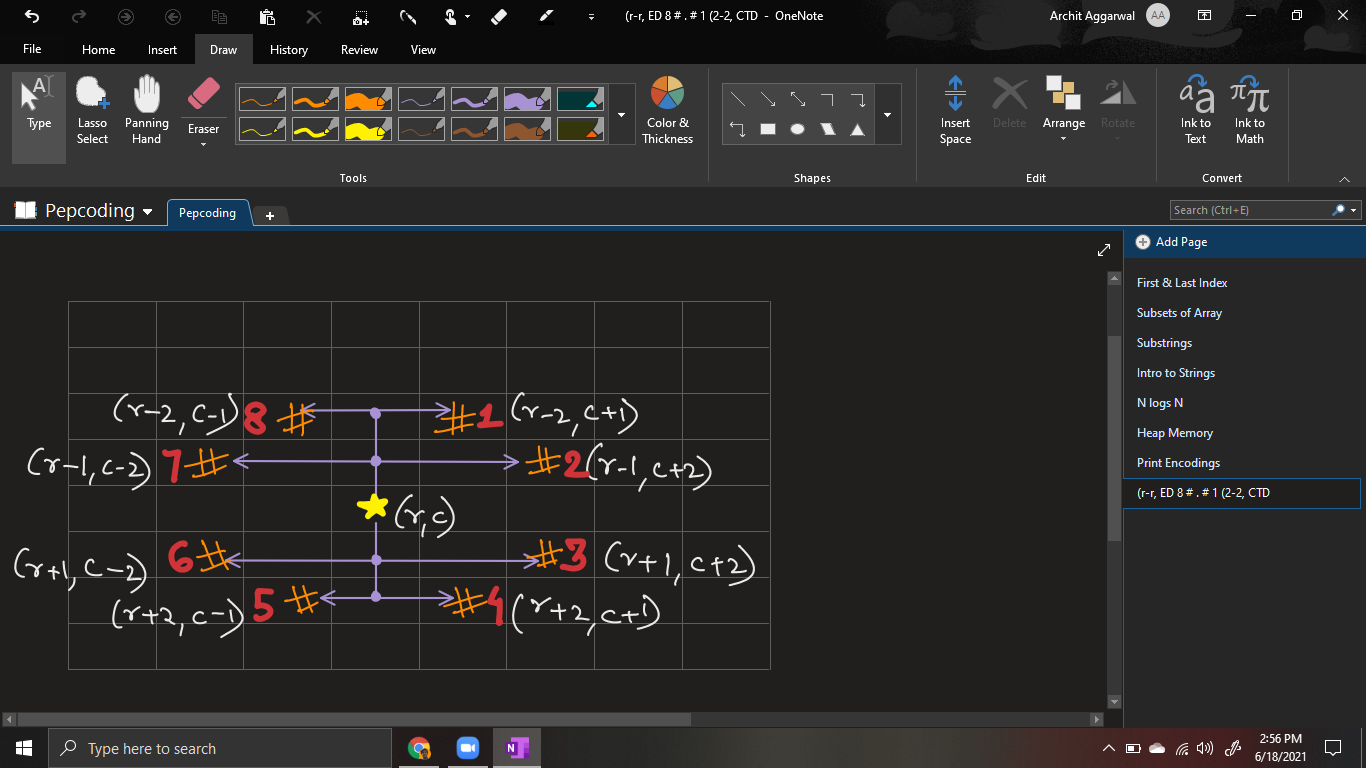




Then, what is the difference in this problem?

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

Hence, **no two knights can be at one knight’s move from each other**.

****

If the previous cell i which knight 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 knight 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

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

chess[row][col] = true;

nknights(kpsf + 1, tk, 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 knights, i.e. the knights placed so far (qpsf) is equal to the total number of knights available (n or tq). When we hit the base case, we will print the grid, by writing the ‘k’ for filled cells, else print ‘-’ followed by tab space for the empty cell.

if (kpsf == tk) {

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

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

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

}

System.out.println();

}

System.out.println();

return;

}

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

We will check for cells **(i-1, j-2) , (i-2, j-1) , (i - 2, j+1) , (i-1, j+2)** whether there is any knight in at least one of the four cells. (Also, before checking for chess[r][c], check whether (r, c) are valid coordinates or not). If there is no knight present in all 4 cells, then return true, else return false.

*Note*: We are only checking 4 out of 8 possible knight’s moves because we are sure that there will be no knight in the cells after the current row and current column. This is because we are filling cells in increasing order only.

public static boolean IsKnightSafe(boolean[][] chess, int i, int j) {

if(i - 1 >= 0 && j - 2 >= 0 && chess[i - 1][j - 2]){

return false;

}

if(i - 2 >= 0 && j - 1 >= 0 && chess[i - 2][j - 1]){

return false;

}

if(i - 2 >= 0 && j + 1 < chess.length && chess[i - 2][j + 1]){

return false;

}

if(i - 1 >= 0 && j + 2 < chess.length && chess[i - 1][j + 2]){

return false;

}

return true;

}

**Java Code**

import java.io.\*;

import java.util.\*;

public class Main {

public static boolean IsKnightSafe(boolean[][] chess, int i, int j) {

if(i - 1 >= 0 && j - 2 >= 0 && chess[i - 1][j - 2]){

return false;

}

if(i - 2 >= 0 && j - 1 >= 0 && chess[i - 2][j - 1]){

return false;

}

if(i - 2 >= 0 && j + 1 < chess.length && chess[i - 2][j + 1]){

return false;

}

if(i - 1 >= 0 && j + 2 < chess.length && chess[i - 1][j + 2]){

return false;

}

return true;

}

public static void nknights(int kpsf, int tk,

boolean[][] chess, int lcno) {

if (kpsf == tk) {

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

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

System.out.print(chess[row][col] ? "k\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

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

chess[row][col] = true;

nknights(kpsf + 1, tk, 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];

nknights(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 knights as levels, and the choices as selecting an empty box. Since levels are n (knights) 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 knights, we are checking whether placing it will result in valid configuration or not. Checking for validity requires checking 4 cells. Hence, the time complexity will be O( (4 \* n^2) \* (4 \* n^2) \* …. n times) = O((4n)n) = **O(nn)**.

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

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

**Follow Up:** We have solved another problem based on Knights and Chessboard, in the recursion & backtracking section of level 1. The problem name was [**‘Knight’s Tour**](https://www.pepcoding.com/resources/online-java-foundation/recursion-backtracking/knights-tour-official/ojquestion)’ and it was a very classic interview problem. So, please solve that problem as well.

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/)