Dear reader, welcome to the combined articles on the problems named **‘**[**Knights Tour**](https://www.pepcoding.com/resources/online-java-foundation/recursion-backtracking/knights-tour-official/ojquestion)**’.** Another problem based on ***Chess,*** I think we programmers like playing chess a lot!

***Problem Statement:***

Given a number n which represents the size of a chess board, and a row and a column, as a starting point for a knight piece, you are required to generate the all moves of a knight starting in (row, col) such that knight visits all cells of the board exactly once.

A knight should move according to the rules in chess. Please explore the next moves in the ***clockwise direction*** to get the same result as ours.

One of the possible valid configuration for N = 8 can be:



You can also watch the [question video](https://www.youtube.com/watch?v=EgoKDqfpbMg) to understand the problem statement in more detail.

*Note*: If you have not tried enough to come up with logic, then we recommend you to first spend an hour or so doing it, else read only the logic used, take it as a hint and try the problem again with the same logic.

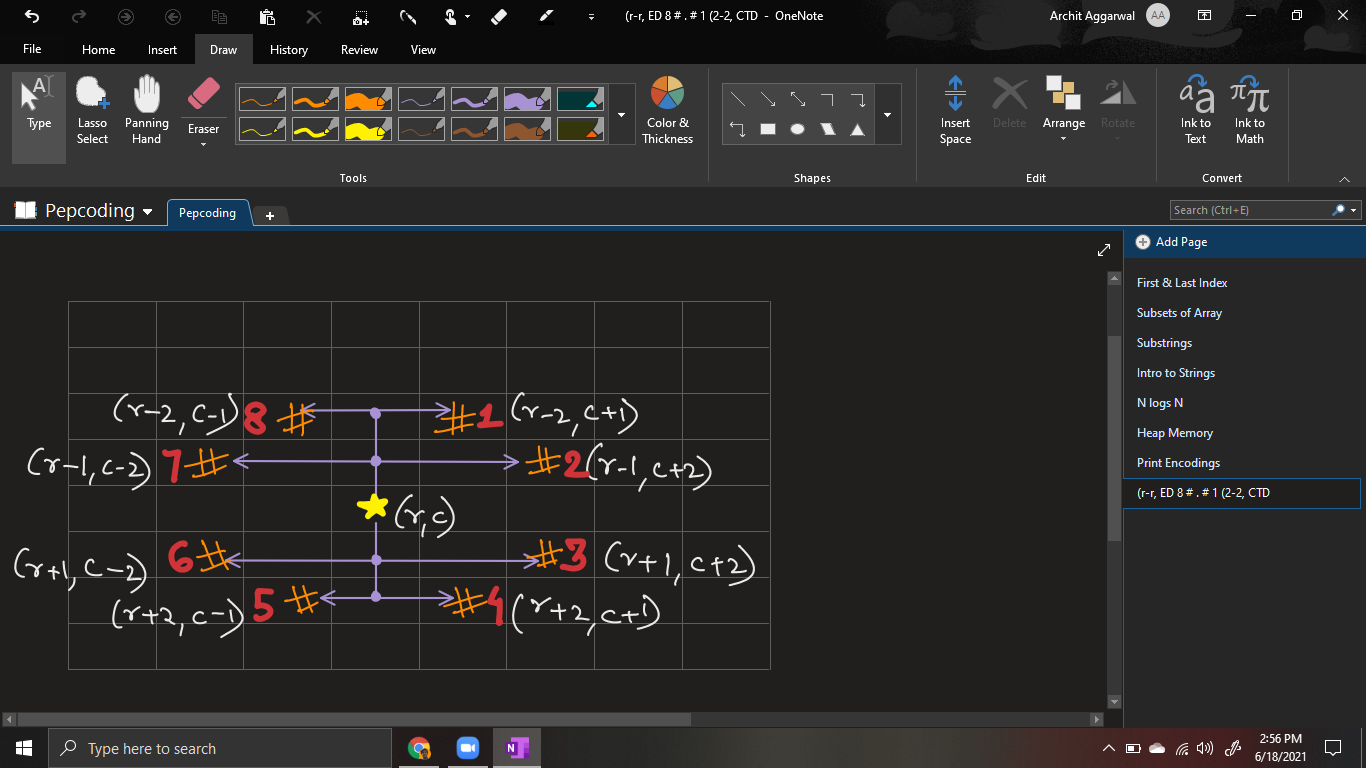
***(Naive) Solution:*** Naive solution will be to generate all possible permutations of numbers, and then check if every moving knight from cell with value v to cell with value v + 1 is possible or not.

***(Backtracking) Solution:***

We are starting with the chess grid with all cells filled with 0 (empty cells). Now, we need to fill these cells so that the configuration becomes valid.

For every cell, we will try to place the knight (give the cell number with value = current move), and then call the recursive function for all the possible 8 moves as the next cell and move number greater by 1.

We should try to visualize how a knight moves on a chessboard.

****

If the current cell is (r, c), then we will try for all possible 8 moves (in clockwise direction), i.e. in the order ***(r-2, c+1), (r-1, c+2), (r+1, c+2), (r+2, c+1), (r+1, c-1), (r+1, c-2), (r-1, c-2) and (r-2, c-1)***.

After making a move to one of the 8 possible, we will check if the current cell is valid or not, i.e. whether the value of the row or column has gone out of bound or not (< 0 or >= size of grid).

Also, the current move can be invalid if it had been already visited in any of the previous moves. This is because we are not allowed to visit any cell twice, i.e. we must visit every cell exactly once.

Hence, we will also return (invalid configuration), if the current cell is already filled with value > 0 (a move number) or not.

Thus these are the checks to find whether the current cell is an invalid move or not:

1. **row < 0 or row >= n**
2. **col < 0 or col >= n**
3. **chess[row][col] > 0 (already filled)**

Now, the only condition left is the ***base case***?

We will print the current configuration when we have already visited all the cells exactly once, i.e. the current move is equal to n2 where n = size of grid.

But before printing, we should fill the current cell with the last move (= n^2). And after printing, we will make the current cell back to 0 (empty), as we are backtracking (to print all possible configurations), and then return.

***Pseudo Code/ Algorithm***

1. If ***invalid move*** (row < 0, row >= n, col < 0, col >= n or chess[row][col] > 0), then return without printing.
2. If the ***base case*** is hit (got till the last move = n2), then fill the cell with move no, print the configuration, mark the cell empty back again (=0), and backtrack/return.
3. Before Recurring, mark the cell with the current move number.
4. Recur for all possible 8 moves with modified values of row and col, and a unit increment of the move number.
5. After returning from all 8 calls, mark the cell as empty again (=0) and return.

***Implementation***

*Note*: Before reading the Code, we recommend that you must try to come up with the solution on your own. Now, hoping that you have tried by yourself, here is the Java code.

import java.io.\*;

import java.util.\*;

public class Main {

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

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

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

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

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

int[][] chess = new int[n][n];

printKnightsTour(chess, r, c, 1);

}

public static void printKnightsTour(int[][] chess,

int r, int c, int upcomingMove){

if(r < 0 || c < 0 || r >= chess.length

|| c >= chess.length || chess[r][c] != 0){

return;

} else if(upcomingMove == chess.length \* chess.length){

chess[r][c] = upcomingMove;

displayBoard(chess);

chess[r][c] = 0;

return;

}

chess[r][c] = upcomingMove;

printKnightsTour(chess, r - 2, c + 1, upcomingMove + 1);

printKnightsTour(chess, r - 1, c + 2 , upcomingMove + 1);

printKnightsTour(chess, r + 1, c + 2 , upcomingMove + 1);

printKnightsTour(chess, r + 2, c + 1 , upcomingMove + 1);

printKnightsTour(chess, r + 2, c - 1 , upcomingMove + 1);

printKnightsTour(chess, r + 1, c - 2 , upcomingMove + 1);

printKnightsTour(chess, r - 1, c - 2 , upcomingMove + 1);

printKnightsTour(chess, r - 2, c - 1 , upcomingMove + 1);

chess[r][c] = 0;

}

public static void displayBoard(int[][] chess){

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

for(int j = 0; j < chess[0].length; j++){

System.out.print(chess[i][j] + " ");

}

System.out.println();

}

System.out.println();

}

}

Java Code is written and explained by our team in the [solution video](https://www.youtube.com/watch?v=SP880DBRJ_8) from timestamp *[3:20 , 12:00]*.

I suggest you draw a recursion tree running your code line by line and see how backtracking works. It is explained using an example in the same video from *[12:00, 13:45]*.

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

For each cell, we have 8 decisions to choose from, and since there are n2 cells, the time complexity will be ***O(8(n^2)).***

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

The maximum number of recursive calls/depth of the recursion tree can be equal to the number of cells. Hence the recursion stack call will take atmax O(n2) space.

However, we are not using any extra data structure, hence the solution is said to have O(1) auxiliary space.

***Asked in Companies***: *Amazon*

Hope that you liked the article on *Knight’s Tour*.

Subscribe to Pepcoding’s youtube channel for more such amazing content on Data Structures & Algorithms and follow the resources available for all students in the resources section of Pepcoding’s website.

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