BACKSTRACKING

* It is a recursion based concept.
* Backtracking is problem solving technique.

⇒ "It involves finding a solution incrementally by trying different options & undoing them if they lead to a dead end."

* It is commonly used in situations where we need to explore multiple posibilities to solve a problem.

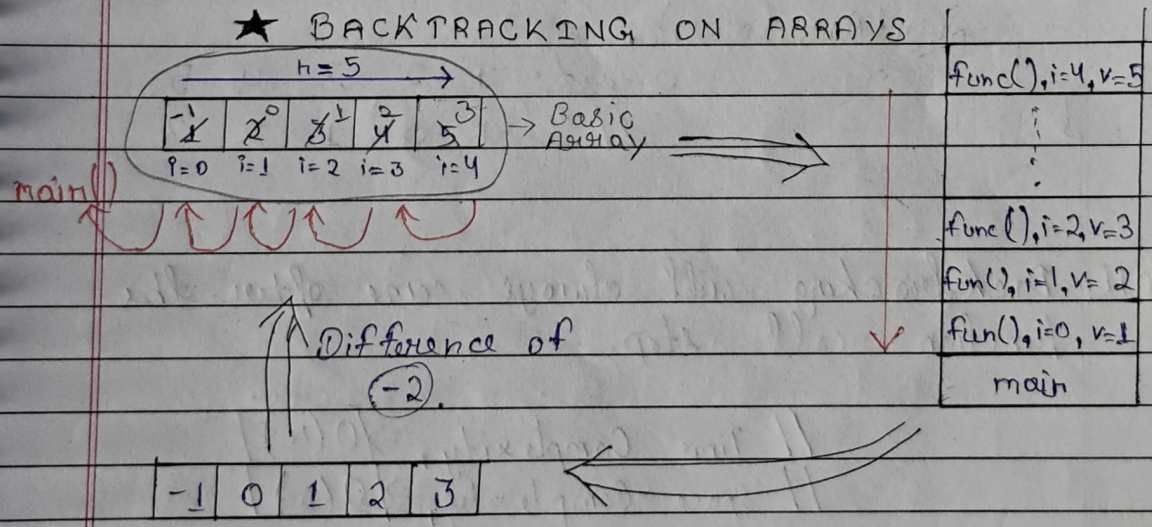
Ex→ Seaxching path in a Maze
or Sudoku.

★ TYPES OF BACKTRACKING

* Problems associated with backtracking can be categorized into 3 categories :-

1. Decision Problems : we search for a feasible solution.
2. Optimization Problems : we search for the best solution.
3. Enumeration Problems : we find set of all possible feasible solutions to the problems of this type.

★ BACKTRACKING ON ARRAYS



⇒
```
public class Backtracking {
    public static void changeArr (int arr[], int i, int val) {
        // base case
        if (i == arr.length) {
            printArr (arr);
            return;
        }
        // recursion
        arr [i] = val;
        changeArr (arr, i+1, val+1); // function call step
        arr [i] = arr [i] - 2; // backtracking step
    }
    public static void printArr (int arr[]) {
        for (int i=0; i<arr.length; i++) {
            System.out.print (arr[i] + " ");
        }
        System.out.println ();
    }
    public static void main (String args[]) {
        int arr[] = new int [5];
        changeArr (arr, 0, 1);
        printArr (arr);
    }
}
```

Output: 1 2 3 4 5
-1 0 1 2 3

#NOTE: Backtracking will always come after the function call step.

// Time Complexity, ⇒ O(n)
// Space Complexity, ⇒ O(n)

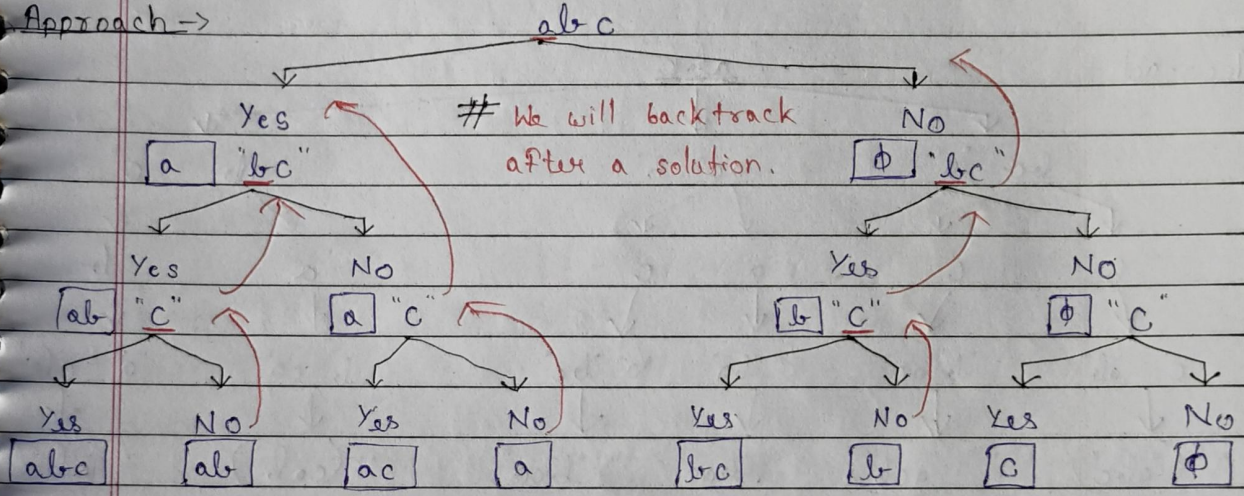
DATE

200

★ Find Subsets

* "abc" is a set then Subsets → 'a', 'b', 'c', 'ab', 'bc', 'ac', 'abc', null(ϕ)
* If there are 'n' number of elements in a set then the subsets will be 2^n.

Approach →



∴ Subsets ⇒ abc, ab, ac, a, bc, b, c, ϕ (null)

⇒

```
public class Backtracking {
    public static void findSubsets (String str, int i, String ans) {
        if (i == str.length()) {
            if (ans.length() == 0) {                    // TC = O(n * 2^n)
                System.out.println("null");             // SC = O(n)
            } else {
                System.out.println(ans);
            }
            return;
        }
        findSubsets(str, i+1, ans);                     // No
        findSubsets(str, i+1, ans + str.charAt(i));     // Yes
    }
}
```

Input ⇒ findSubsets(str, "", 0);

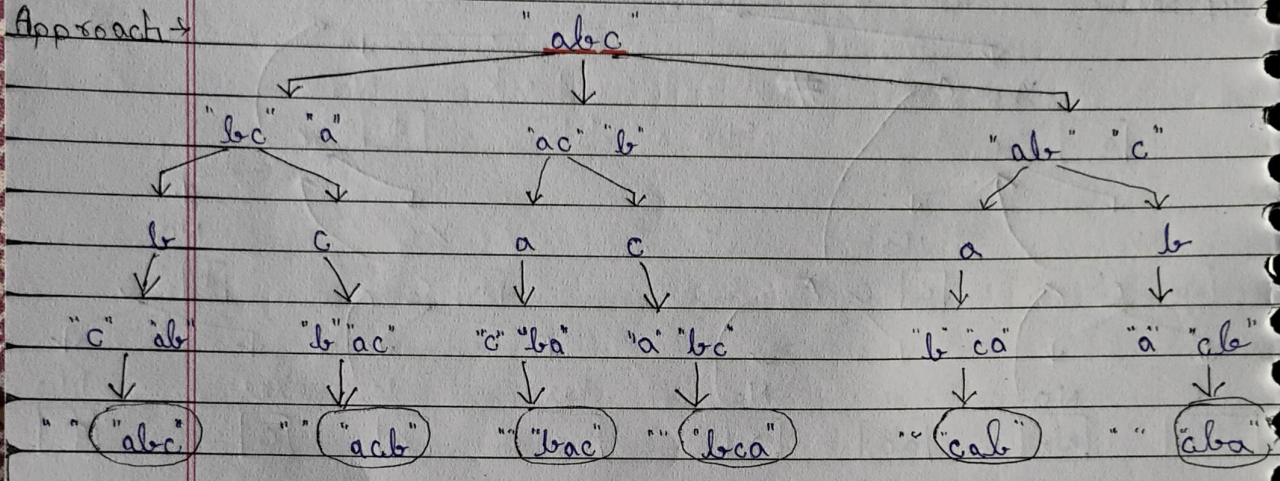
★ FIND PERMUTATIONS

* Find & print all permutations of a String.
(means arrangement.)

i.e., "abc" → abc, acb, bac, bca, cab, cba

* n elements → then → n! permutations

Approach →



⇒
```
public static void findPermutation (String str, String ans){
    //base case
    if (str.length() == 0) {
        System.out.println (ans);
        return;
    }                                   // O(n * n!)
    //recursion
    for (int i=0; i<str.length(); i++) {
        char curr = str.charAt(i);
        String NewStr = str.substring(0,i) + str.substring(i+1);
        findPermutation (NewStr, ans + curr);
    }
}
```

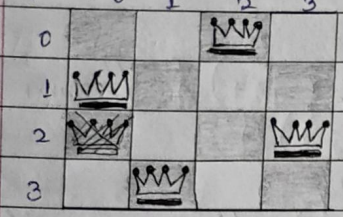
* Input → String str = "abc";
findPermutation (str, "");

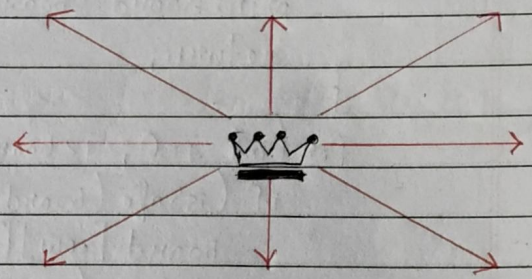
DATE

202

★ Place N queens on an N×N chessboard such that no 2 queens can attack each other.

N = 4





Approach →

Q	Q	X	X
X	X	X	Q
X	X	X	X

→ अगर किसी Row में Queen नहीं आई तो Backtracking use होगा और फिर Queen Rearrange होगा।

//N QUEENS ALL POSSIBLE WAYS

⇒
```
public class Backtracking {
    public static boolean isSafe(char board[][], int row, int col) {
        for (int i=row-1; i>=0; i--) {
            if (board[i][col] == 'Q') {
                return false;
            }
        }
        for (int i=row-1, j=col-1; i>=0 && j>=0; i--, j--) {
            if (board[i][j] == 'Q') {
                return false;
            }
        }
        for (int i=row-1, j=col+1; i>=0 && j<board.length; i--, j++) {
            if (board[i][j] == 'Q') {
                return false;
            }
        }
        return true;
    }
```

→ //vertical up

→ //diagonal Left up

→ //diagonal right up

```
public static void nQueens (char board[][], int row) {
    if (row == board.length) {
        printBoard (board);                    // base case
        return;
    }
    for (int j=0; j<board.length; j++) {  //column Loop
        if (isSafe (board, row, j)) {
            board [row][j] = 'Q';
            nQueens (board, row+1); //function call
            board [row][j] = 'x'; //backtracking step
        }
    }
}

public static void printBoard (char board[][]) {
    System.out.println ("------chess board -------");
    for (int i=0; i<board.length; i++) {
        for (int j=0; j<board.length; j++) {
            System.out.print (board [i][j] + " ");
        }
        System.out.println ();
    }
}

public static void main (String args []) {
    int n = 4;
    char board[][] = new char [n][n];
    //initialise
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            board [i][j] = 'x';
        }
    }                                  // Time Complexity, O(n!)
    nQueens (board, 0);
}
```

★ N-QUEENS - count ways

* Count total no. of ways in which we can solve N queens problem.

⇒
```
public class Backtracking {
    public static boolean isSafe(char board[][], int row, int col){
        same code

    }
    public static void nQueens(char board[][], int row){
        if(row == board.length){
            count++;
            return;
        }
        same code

    }
    public static void printBoard(char board[][]){
        same code

    }
    static int count = 0;
    public static void main(String args[]){
        int n = 4;
        same code

        nQueens(board, 0);
        System.out.println("Total ways to solve n Queens: "+count);
    }
}
```

★ N-QUEENS : Print 1 solutions

* Check if problem can be solved & print only 1 solution to N Queens problem.

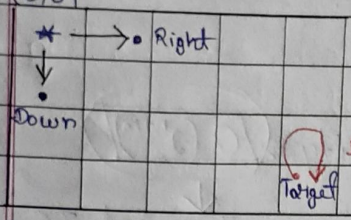
⇒

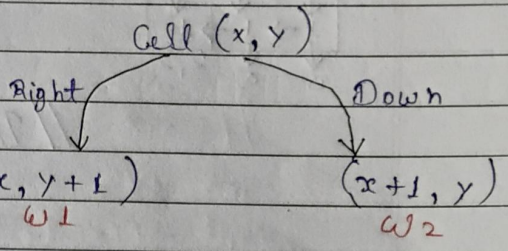
```
public static boolean isSafe (char board[][], int row, int col) {
                    Same code
}
public static boolean nQueens (char board[][] int row) {
    if (row == board.length) {
        count ++;
        return true;
    }
    for (int j=0; j<board.length; j++) {
        if (isSafe (board, row, j)) {
            board [row][j] = 'Q';
            if (nQueens (board, row+1)) {
                return true;
            }
            board [row][j] = 'x';
        }
    }
    return false;
}
public static void printBoard (char board[][]) {
                    Same code
} static int count = 0;
public static void main (String args[]) {
                    same code
    if (nQueens (board, 0)) {
        System.out.println ("Solution is Possible");
        printBoard (board);
    } else {
        System.out.println ("Solution is not Possible");
    }
}
```

★ GRID WAYS – Find no. of ways to reach from (0,0) to (N-1, M-1) in a N x M Grid.

* Allowed moves – right or down.

(0,0)





∴ Recursive or Recurrence equation will be,

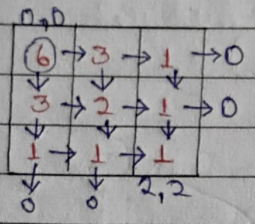
$$\Rightarrow f(x,y) = f(x+1,y) + f(x,y+1)$$

Down w_1 Right w_2.

* Target से Target पे जाने के लिए अगर कोई पूछे कितने Ways तो ⇒ **Total 1 way.**

⇒
```
public class Backstracking {
    public static int gridWays (int i, int j, int n, int m) {
        //base case
        if (i==n-1 && j==m-1) { //condition for last cell
            return 1;
        } else if (i==n || j==n) { //boundary cross condition
            return 0;
        }
        int w1 = gridWays (i+1, j, n, m);
        int w2 = gridWays (i, j+1, n, m);
        return w1 + w2;
    }
    public static void main (String args[]) {
        int n=3, m=3;
        System.out.println (gridWays (0, 0, n, m));
    }
}
```

* Dry Run for 3 x 3



∴ 6 ways.

// Time Complexity, ⇒ $O(2^{n+m})$

Very bad TC.

* Math Trick to solve GridWays.

Total Ways ⇒ $\frac{(n-1+m-1)!}{(n-1)!\,(m-1)!}$

Ex ⇒ $n=3$, $m=3$ ⟶ $\frac{(3-1+3-1)!}{(3-1)!\,(3-1)!}$

$\frac{4!}{2!\,2!} = \frac{4\times3\times2\times1}{2\times2}$

⇒ 6 ways

// Time Complexity, $O(n+m)$

★ SUDOKU - Write a function to complete a Sudoku.

		8						
4	9		1	5	7			2
		3			4	1	9	
1	8	5		6			2	
				2			6	
9	6		4		5	3		
	3			7	2			4
	4	9		3			5	7
8	2	7			9		1	3

// If you have written any number in a cell so remember that NUMBER SHOULDN'T REPEATED IN THE

→ SAME ROW
→ SAME COLUMN
→ SAME GRID
⇒ NUMBER SHOULD BE 1-9.

⇒
```
public class Sudoku {
    public static boolean SudokuSolver (int sudoku, int row, int col){
        //base case
        if (row == 9) {
            return true;
        }
        //recursion
        int nextRow = row, nextCol = col + 1;
        if (col + 1 == 9) {
            nextRow = row + 1;
            nextCol = 0;
        }
        if (sudoku [row][col] != 0) {
            return SudokuSolver (sudoku, nextRow, nextCol);
        }
        for (int digit = 1; digit <= 9; digit++) {
            if (isSafe (sudoku, row, col, digit)) {
                sudoku [row][col] = digit;
                if (SudokuSolver (sudoku, nextRow, nextCol)){
                    return true;
                }
                sudoku [row][col] = 0;
            }
        }
        return false;
    }
public static boolean isSafe (int sudoku[][], int row, int col, int digit){
    //column
    for (int i=0; i<=8; i++) {
        if (sudoku [i][col] == digit) {
            return false;
        }
    }
```

```
//row
for (int j=0 ; j<=8 ; j++) {
    if (sudoku [row] [j] == digit ) {
        return false;
    }
}
// Grid
int sr = (row/3) * 3;
int sc = (col/3) * 3;
//3×3 grid
for (int i=sr ; i<sr+3 ; i++ ) {
    for (int j=sc ; j<sc+3 ; j++) {
        if (sudoku [i] [j] == digit) {
            return false;
        }
    }
}
return true;
}
public static void printSudoku (int Sudoku[][]) {
    for (int i=0; i<9 ; i++) {
        for (int j=0 ; j<9 ; j++) {
            System.out.println (sudoku [i] [j] + " ");
        }
        System.out.println ();
    }
}
```

// The vacant spaces are denoted as 0 inside the code.

```
public static void main (String args[]) {
    int sudoku[][] = { {0, 0, 8, 0, 0, 0, 0, 0, 0},
    {4, 9, 0, 1, 5, 7, 0, 0, 2},
    {0, 0, 3, 0, 0, 4, 1, 9, 0},
    {1, 8, 5, 0, 6, 0, 0, 2, 0},
    {0, 0, 0, 0, 2, 0, 0, 6, 0},
    {9, 6, 0, 4, 0, 5, 3, 0, 0},
    {0, 3, 0, 0, 7, 2, 0, 0, 4},
    {0, 4, 9, 0, 3, 0, 0, 5, 7},
    {8, 2, 7, 0, 0, 9, 0, 1, 3} };
    if (SudokuSolver (sudoku, 0, 0)) {
        System.out.println ("------ Solution Exists ------");
        printSudoku (sudoku);
    } else {
        System.out.println ("----Solution not Exists------");
    }
}
}
```

Output →

```
------Solution Exists ---------
2 1 8 3 9 6 7 4 5
4 9 6 1 5 7 8 3 2
7 5 3 2 8 4 1 9 6
1 8 5 7 6 3 4 2 9
3 7 4 9 2 8 5 6 1
9 6 2 4 1 5 3 7 8
5 3 1 6 7 2 9 8 4
6 4 9 8 3 1 2 5 7
8 2 7 5 4 9 6 1 3
```

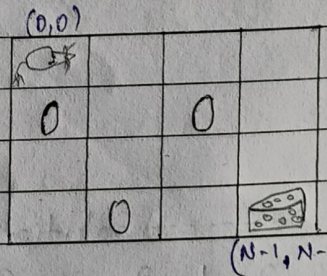
DATE

211

Ques 1. Rat in a Maze - You are given a starting position for a rat which is stuck in a maze at an initial point (0, 0). The maze would be given in the form of a square matrix of order N * N where the cells with value 0 represent the maze's blocked locations while value 1 is the open/available path that rat can take to reach its destination. The rat's destination is of (N-1, N-1).

Your task is to find all the possible paths that the rat can take to reach its destination from source.

The possible directions that it can take to move in the maze are 'U' (up) i.e. (x, y-1), 'D' (down) i.e., (x, y+1), 'L' (left) i.e. (x-1, y), 'R' (right) i.e., (x+1, y).



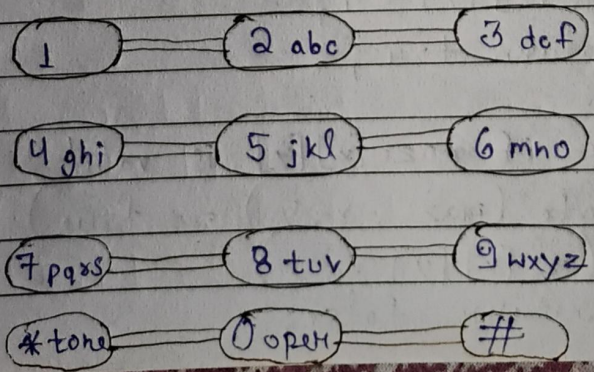
* Sample Input : int maze[][] = {{1, 0, 0, 0},
{1, 1, 0, 1},
{0, 1, 0, 0},
{1, 1, 1, 1}}

* Sample Output : 1 0 0 0
1 1 0 0
0 1 0 0
0 1 1 1

⇒
```
public class Backstracking {
    public static void printSolution (int sol [][]) {
        for (int i=0; i<sol.length; i++) {
            for (int j=0; j<sol.length; j++) {
                System.out.print (" "+sol [i][j] + " ");
            }
            System.out.println ();
        }
    }
    public static boolean isSafe (int maze [][], int x, int y) {
        //if (x,y) outside maze then return false.
        return (x>=0 && x<maze.length && y>=0 &&
                y<maze.length && maze [x][y] == 1);
    }
    public static boolean solveMaze (int maze [][]) {
        int N = maze.length;
        int sol [][] = new int [N][N];
        if (solveMazeUtil (maze, 0, 0, sol) == false) {
            System.out.print ("Solution doesn't Exist");
            return false;
        }
        printSolution (sol);
        return true;
    }
    public static boolean solveMazeUtil (int maze[][], int int int x, y, sol[][]
        if (x==maze.length-1 && y==maze.length-1 && maze[x][y]==1
            sol [x][y] = 1;
            return true;
        }
        //check if maze [x][y] is valid
        if (isSafe (maze, x, y) == true) {
            if (sol [x][y] == 1)
                return false;
        }
```

```
        sol[x][y] = 1;
        if (solveMazeUtil(maze, x+1, y, sol))
            return true;
        if (solveMazeUtil(maze, x, y+1, sol))
            return true;
        sol[x][y] = 0;
        return false;
    }
    return false;
}
public static void main(String args[]) {
    int maze[][] = { {1, 0, 0, 0},
                     {1, 1, 0, 1},
                     {0, 1, 0, 0},
                     {1, 1, 1, 1} };
    solveMaze(maze);
}
}
```

Ques 2. Keypad Combinations – Given a string containing digits from 2-9 inclusive, print all possible letter combinations that the number could represent. You can print the answer in any order.
A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.



1. Sample Input : digits = "23"
Sample Outputs "ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf".

2. Sample Input : digits = "2"
Sample Output: "a", "b", "c"

3. Sample Input : digits = ""
Sample Output : "".

⇒
```
public class Solution {
    final static char[][] L = {{}, {}, {'a', 'b', 'c'},
                               {'d', 'e', 'f'}, {'g', 'h', 'i'}, {'j', 'k', 'l'},
                               {'m', 'n', 'o'}, {'p', 'q', 'r', 's'}, {'t', 'u', 'v'},
                               {'w', 'x', 'y', 'z'}};
    public static void letterCombinations (String D) {
        int len = D.length();
        if (len == 0) {
            System.out.println ("");
            return;
        }
        bfs (0, len, new StringBuilder(), D);
    }
    public static void bfs (int pos, int len, StringBuilder sb, String D) {
        if (pos == len) {
            System.out.println (sb.toString());
        }
        else {
            char[] letters = L[Character.getNumericValue
                               (D.charAt (pos))];
            for (int i = 0; i < letters.length; i++) {
                bfs (pos+1, len, new StringBuilder (sb).
                     append (letters[i]), D);
            }
        }
    }
    public static void main (String args[]) {
        letterCombinations ("2");
    }
}
```

Ques 3. Knight's Tour - Given a N*N board with the Knight placed on the first block of an empty board. Moving according to the rules of chess, Knights must visit each square exactly once. Print the order of each cell in which they are visited.

⇒
```
public class Solution {
    static int N = 8;
    public static boolean isSafe(int x, int y, int sol[][]) {
        return (x >= 0 && x < N && y >= 0 && y < N
                && sol[x][y] == -1);
    }
    public static void printSolution(int sol[][]) {
        for (int x = 0; x < N; x++) {
            for (int y = 0; y < N; y++) {
                System.out.print(sol[x][y] + " ");
            }
            System.out.println();
        }
    }
    public static boolean solveKT() {
        int sol[][] = new int[8][8];
        for (int x = 0; x < N; x++) {
            for (int y = 0; y < N; y++) {
                sol[x][y] = -1;
            }
        }
        int xMove[] = {2, 1, -1, -2, -2, -1, 1, 2};
        int yMove[] = {1, 2, 2, 1, -1, -2, -2, -1};
        //As the Knight starts from cell (0,0)
        sol[0][0] = 0;
```

```
if (!solveKTUtil (0, 0, 1, sol, xMove, yMove)) {
    System.out.println ("Solution does not exist");
    return false;
}
else {
    printSolution (sol);
}
return true;
}
public static boolean solveKTUtil (int x, int y, int movei,
                                   int sol[][], int xMove[],
                                   int yMove[]) {
int k, next_x, next_y;
if (movei == N * N) {
    return true;
}
for (k=0; k<8; k++) {
    next_x = x + xMove[k];
    next_y = y + yMove[k];
    if (isSafe (next_x, next_y, sol)) {
        sol[next_x][next_y] = movei;
        if (solveKTUtil (next_x, next_y, movei+1,
                         sol, xMove, yMove)) {
            return true;
        }
        else {
            sol[next_x][next_y] = -1;
        }
    }
}
return false;
}
```

```
    public static void main (String args[]) {
        solveKT();
    }
}
```