DAY 21:

ASSIGNMENT 1:

Task 1: The Knight's Tour Problem

Create a function bool SolveKnightsTour(int[,] board, int moveX, int moveY, int moveCount, int[] xMove, int[] yMove) that attempts to solve the Knight's Tour problem using backtracking. The function should return true if a solution exists and false otherwise. The board represents the chessboard, moveX and moveY are the current coordinates of the knight, moveCount is the current move count, and xMove[], yMove[] are the possible next moves for the knight. Fill the chessboard such that the knight visits every square exactly once. Keep the chessboard size to 8x8.

ANSWER:

```
public class KnightsTour {

    private static final int N = 8;

    // Check if (x, y) is a valid move
    private static boolean isValidMove(int x, int y, int[][] board) {
        return (x >= 0 && x < N && y >= 0 && y < N && board[x][y] == -1);
    }

    // Utility function to print the solution board
    private static void printSolution(int[][] board) {
        for (int x = 0; x < N; x++) {
            for (int y = 0; y < N; y++) {
                System.out.printf("%2d ", board[x][y]);
            }
            System.out.println();
        }
    }
```

```
// The main function to solve the Knight's Tour problem
public static boolean solveKnightsTour() {
    int[][] board = new int[N][N];

    // Initialize the board with -1 indicating unvisited squares
    for (int x = 0; x < N; x++) {
        for (int y = 0; y < N; y++) {
            board[x][y] = -1;
        }
    }

    // Possible moves for the knight
    int[] xMove = {2, 1, -1, -2, -2, -1, 1, 2};
    int[] yMove = {1, 2, 2, 1, -1, -2, -2, -1};

    // Starting position of the knight
    board[0][0] = 0;

    // Start from 0,0 and explore all tours using solveKTUtil()
    if (!solveKTUtil(0, 0, 1, board, xMove, yMove)) {
        System.out.println("Solution does not exist");
        return false;
    } else {
        printSolution(board);
    }

    return true;
}

// A recursive utility function to solve Knight's Tour problem
```

```
    private static boolean solveKTUtil(int x, int y, int moveCount, int[][] board, int[] xMove, int[]
yMove) {
        int nextX, nextY;
        if (moveCount == N * N) {
            return true;
        }

        // Try all next moves from the current coordinate x, y
        for (int k = 0; k < 8; k++) {
            nextX = x + xMove[k];
            nextY = y + yMove[k];
            if (isValidMove(nextX, nextY, board)) {
                board[nextX][nextY] = moveCount;
                if (solveKTUtil(nextX, nextY, moveCount + 1, board, xMove, yMove)) {
                    return true;
                } else {
                    // Backtracking
                    board[nextX][nextY] = -1;
                }
            }
        }

        return false;
    }

    // Main function to execute the solution
    public static void main(String[] args) {
        solveKnightsTour();
    }
}
```

Explanation:

1. Initialization:

- The chessboard (board) is initialized to -1, marking all squares as unvisited.

- The knight starts at the first block (0, 0), so board[0][0] is set to 0.

2. Movement Arrays:

- xMove and yMove arrays contain the possible moves a knight can make.

3. Recursive Backtracking:

- The solveKTUtil function is used to recursively attempt to solve the problem.

- If moveCount equals N * N, it means the knight has visited all squares, and the function returns true.

- The function tries all possible moves from the current position. If a move is valid, it updates the board and calls itself recursively with the new position and incremented move count.

- If a move doesn't lead to a solution, it backtracks by resetting the square to -1.

4. Solution Output:

- If a solution is found, printSolution is called to print the board. If no solution exists, a message is printed.

Running the main method will attempt to solve the Knight's Tour problem and print the solution if one exists.