NAME :- Jangili Ravali

EMAIL :- jangiliravali9@gmail.com

Day 15 , 16 and 17 :-

Task 1: Knapsack Problem

Write a function int Knapsack(int W, int[] weights, int[] values) in C# that determines the maximum value of items that can fit into a knapsack with a capacity W. The function should handle up to 100 items. Find the optimal way to fill the knapsack with the given items to achieve the maximum total value. You must consider that you cannot break items, but have to include them whole.

a possible implementation of the Knapsack problem in C#:

```
using System;

class KnapsackProblem
{
   public static int Knapsack(int W, int[] weights, int[] values)
   {
      int n = weights.Length;
int[,] K = new int[n + 1, W + 1];

      for (int i = 0; i <= n; i++)
      {
         for (int w = 0; w <= W; w++)
         {
            if (i == 0 || w == 0)
K[i, w] = 0;
            else if (weights[i - 1] <= w)
```

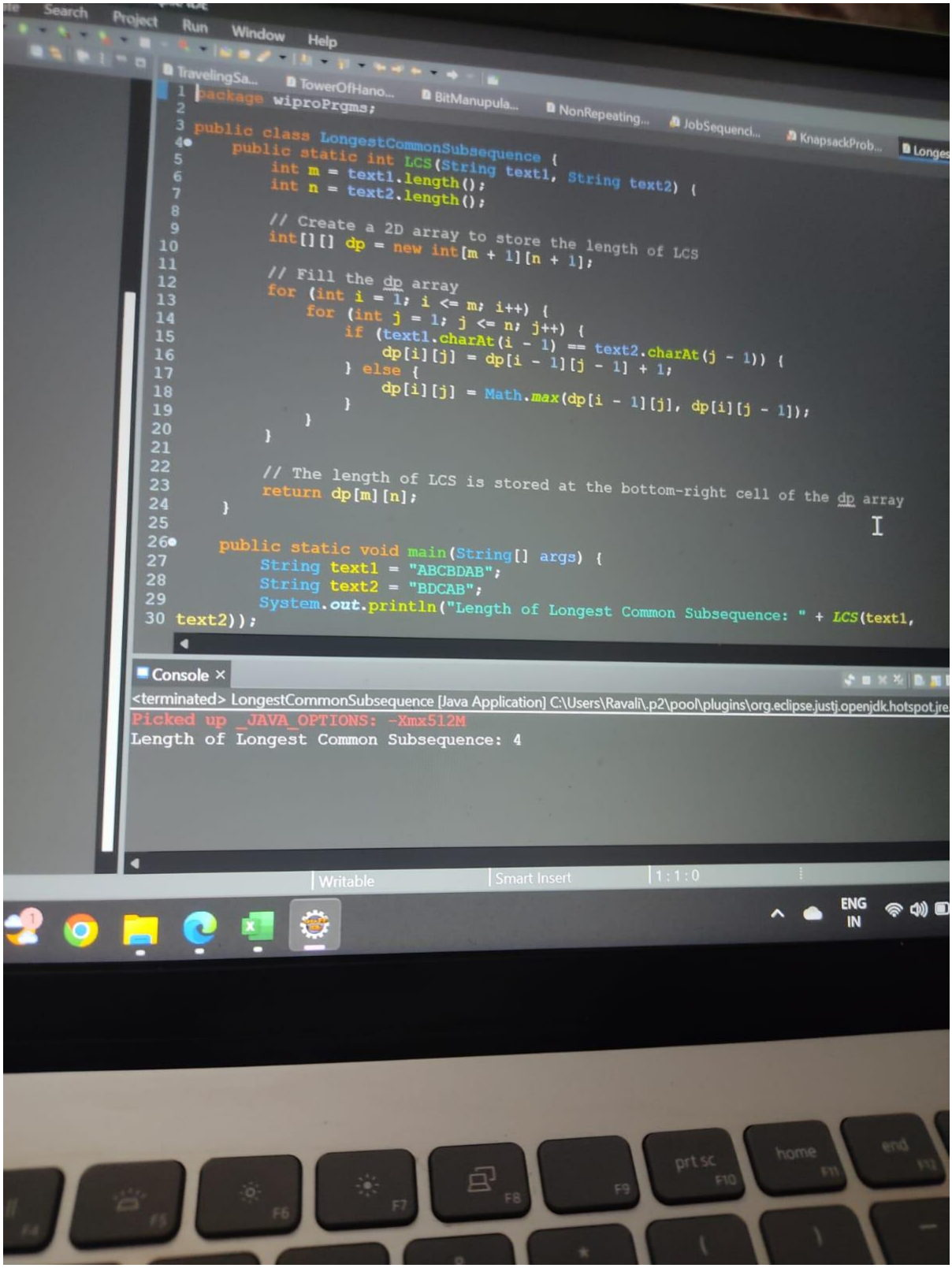
```
                K[i, w] = Math.Max(values[i - 1] + K[i - 1, w - weights[i - 1]], K[i - 1, w]);
else
                K[i, w] = K[i - 1, w];
          }
       }

       int result = K[n, W];
return result;
   }

   static void Main(string[] args)
   {
       int[] values = { 60, 100, 120 };
int[] weights = { 10, 20, 30 };        int
W = 50;
       Console.WriteLine("Maximum value that can be obtained is " + Knapsack(W, weights, values));
   }
}
```

This implementation finds the maximum value that can fit into a knapsack with a capacity W given the weights and values of items. It uses dynamic programming to solve the problem efficiently. You can adjust the values, weights, and W variables to test different scenarios.

Task 2: Longest Common Subsequence

Implement int LCS(string text1, string text2) to find the length of the longest common subsequence between two strings.

LongestCommonSubsequence



You can run this code and it will output the length of the longest common subsequence between the strings text1 and text2.

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.

a C# implementation of the SolveKnightsTour function:

```
using System;

class KnightsTour
{
    static int N = 8;

    static bool IsSafe(int x, int y, int[,] board)
    {
        return (x >= 0 && x < N && y >= 0 && y < N && board[x, y] == -1);
    }

    static void PrintSolution(int[,] board)
    {
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                Console.Write(board[i, j] + " ");
            }
```

```
            Console.WriteLine();
        }
    }

    static bool SolveKnightsTour(int[,] board, int x, int y, int moveCount, int[] xMove, int[] yMove)
    {
        if (moveCount == N * N)
        {
            PrintSolution(board);
return true;
        }

        for (int k = 0; k < 8; k++)
        {
            int nextX = x + xMove[k];
            int nextY = y + yMove[k];

            if (IsSafe(nextX, nextY, board))
            {
                board[nextX, nextY] = moveCount;                if (SolveKnightsTour(board, nextX,
nextY, moveCount + 1, xMove, yMove))
                {
                    return true;
                }
else
                {
                    board[nextX, nextY] = -1;
                }
            }
        }
```

```
        return false;
    }

    static void Main()
    {
        int[,] board = new int[N, N];
for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                board[i, j] = -1;
            }
        }

        int[] xMove = { 2, 1, -1, -2, -2, -1, 1, 2 };
int[] yMove = { 1, 2, 2, 1, -1, -2, -2, -1 };

        board[0, 0] = 0;

        if (!SolveKnightsTour(board, 0, 0, 1, xMove, yMove))
        {
            Console.WriteLine("Solution does not exist");
        }
    }
}
```

This code uses backtracking to solve the Knight's Tour problem on an 8x8 chessboard. It checks for all possible moves and recursively explores them until a solution is found or all possibilities are exhausted.

Task 2: Rat in a Maze

implement a function bool SolveMaze(int[,] maze) that uses backtracking to find a path from the top left corner to the bottom right corner of a maze. The maze is represented by a 2D array where 1s are paths and 0s are walls. Find a rat's path through the maze. The maze size is 6x6.

```
using System;


class RatInMaze {
private const int N = 6;


   // A utility function to print solution matrix
static void PrintSolution(int[,] sol) {        for (int i
= 0; i < N; i++) {            for (int j = 0; j < N; j++)
              Console.Write(" " + sol[i, j] + " ");
          Console.WriteLine();
       }
   }


   // A utility function to check if x, y is valid index     static bool
IsSafe(int[,] maze, int x, int y) {        // if (x, y outside maze) return false
return (x >= 0 && x < N && y >= 0 && y < N && maze[x, y] == 1);


}
   /* This function solves the Maze problem using Backtracking.
   It mainly uses solveMazeUtil() to solve the problem.     It
returns false if no path is possible, otherwise return     true and
prints the path in the form of 1s. Please note     that there may
be more than one solutions, this function     prints one of the
feasible solutions.*/
```

```
   static bool SolveMazeUtil(int[,] maze, int x, int y, int[,] sol) {
      // if (x, y is goal) return true
if (x == N - 1 && y == N - 1) {
sol[x, y] = 1;           return true;
      }

      // Check if maze[x][y] is valid       if
(IsSafe(maze, x, y) == true) {           // mark
x, y as part of solution path           sol[x, y] =
1;

         /* Move forward in x direction */          if
(SolveMazeUtil(maze, x + 1, y, sol) == true)
return true;

         /* If moving in x direction doesn't give
solution then Move down in y direction */          if
(SolveMazeUtil(maze, x, y + 1, sol) == true)
return true;

         /* If none of the above movements work then
BACKTRACK: unmark x, y as part of solution           path
*/          sol[x, y] = 0;          return false;
      }

      return false;

}
   /* This function solves the Maze problem using Backtracking.
```

```
   It mainly uses solveMazeUtil() to solve the problem.     It
returns false if no path is possible, otherwise return     true and
prints the path in the form of 1s. Please note     that there may
be more than one solutions, this function     prints one of the
feasible solutions.*/     static bool SolveMaze(int[,] maze) {
      int[,] sol = {
         {0, 0, 0, 0, 0, 0},
         {0, 0, 0, 0, 0, 0},
         {0, 0, 0, 0, 0, 0},
         {0, 0, 0, 0, 0, 0},
         {0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0}
      };

      if (SolveMazeUtil(maze, 0, 0, sol) == false) {
Console.WriteLine("Solution doesn't exist");           return
false;
      }

      PrintSolution(sol);
return true;
   }

   // Main Method        public static void
Main(string[] args) {       int[,] maze = {
         {1, 0, 0, 0, 0, 0},
         {1, 1, 1, 1, 1, 0},
         {0, 0, 0, 0, 1, 0},
         {0, 1, 1, 1, 1, 1},
         {0, 1, 0, 0, 0, 1},
{0, 1, 1, 1, 1, 1}
```

```
        };

        SolveMaze(maze);
    }
}
```

This program will print a 6x6 matrix where the path taken by the rat is represented by 1s, and walls are represented by 0s.

Task 3: N Queen Problem

Write a function bool SolveNQueen(int[,] board, int col) in C# that places N queens on an N x N chessboard so that no two queens attack each other using backtracking. Place N queens on the board such that no two queens can attack each other. Use a standard 8x8 chessboard.

```
using System;

class NQueens {
static int N = 8;

    static void PrintSolution(int[,] board) {
for (int i = 0; i < N; i++) {           for (int j = 0;
j < N; j++) {
            Console.Write(board[i, j] + " ");
        }
        Console.WriteLine();
```

```
        }
    }

    static bool IsSafe(int[,] board, int row, int col) {        int
i, j;

        for (i = 0; i < col; i++)
if   (board[row,   i]   ==   1)
return false;

        for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
if (board[i, j] == 1)               return false;

        for (i = row, j = col; j >= 0 && i < N; i++, j--)
if (board[i, j] == 1)               return false;

        return true;
    }

    static bool SolveNQueenUtil(int[,] board, int col) {
        if (col >= N)
return true;

        for (int i = 0; i < N; i++) {
if (IsSafe(board, i, col)) {
board[i, col] = 1;

            if (SolveNQueenUtil(board, col + 1))
                return true;
```

```
            board[i, col] = 0; // backtrack
          }
        }

        return false;
    }

    static bool SolveNQueen(int[,] board, int col) {        if
(!SolveNQueenUtil(board, col)) {
Console.WriteLine("Solution does not exist");
            return false;
        }

        PrintSolution(board);
return true;
    }

    static void Main(string[] args) {
int[,] board = {
            {0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0}
        };

        SolveNQueen(board, 0);
```

```
    }
}
```

This code defines a function SolveNQueen that takes an empty 8x8 chessboard (board) and the column number (col) as input. It solves the N Queen problem using backtracking and prints the solution if it exists.
