Prajwalhonashetti143@gmail.com

Day 15 and 16:

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.

```
using System;
public class KnapsackProblem
{
    // Function to find the maximum value that can be put in a knapsack of capacity W
    public int Knapsack(int W, int[] weights, int[] values)
    {
        int n = weights.Length;
        int[,] dp = new int[n + 1, W + 1];

        for (int i = 1; i <= n; i++)
        {
            int weight = weights[i - 1];
            int value = values[i - 1];
            for (int w = 1; w <= W; w++)
            {
                if (weight <= w)
                {
                    dp[i, w] = Math.Max(dp[i - 1, w], dp[i - 1, w - weight] + value);
                }
                else
                {
```

```
            dp[i, w] = dp[i - 1, w];
        }
    }
}

int[] includedItems = new int[n];
int remainingWeight = W;
for (int i = n; i > 0; i--)
{
    if (dp[i, remainingWeight] != dp[i - 1, remainingWeight])
    {
        includedItems[i - 1] = 1;
        remainingWeight -= weights[i - 1];
    }
}

// Print the items included in the knapsack
Console.WriteLine("Items included in the knapsack:");
for (int i = 0; i < n; i++)
{
    if (includedItems[i] == 1)
    {
        Console.WriteLine($"Item {i + 1} (Weight: {weights[i]}, Value: {values[i]})");
    }
}

// Return the maximum value that can be achieved
return dp[n, W];
}
```

```
    public static void Main()
    {
        int[] weights = { 10, 20, 30 };
        int[] values = { 60, 100, 120 };
        int W = 50;

        KnapsackProblem knapsack = new KnapsackProblem();
        int maxValue = knapsack.Knapsack(W, weights, values);

        Console.WriteLine($"Maximum value in knapsack: {maxValue}");
    }
}
```

Explanation:

1 KnapsackProblem Class:

Knapsack Method:

- Takes in parameters W (capacity of the knapsack), weights (array of item weights), and values (array of item values).
- Uses a 2D array dp where dp[i, w] represents the maximum value that can be achieved with the first i items and a knapsack capacity of w.
- Iteratively fills up the dp table using dynamic programming to find the maximum value.
- Tracks which items are included in the knapsack by backtracking through the dp table.
- Prints out the items included in the knapsack.
- Returns the maximum value that can be achieved.

Main Method:

- Example usage of the Knapsack method with a sample set of items (weights, values) and knapsack capacity W.
- Prints out the maximum value that can be achieved with the given items and capacity.

For the input:

weights = {10, 20, 30}

values = {60, 100, 120}

W = 50 (knapsack capacity)

The output will be:

Items included in the knapsack:

Item 2 (Weight: 20, Value: 100)

Item 3 (Weight: 30, Value: 120)

Maximum value in knapsack: 220

Explanation of Output:

Items included in the knapsack: The items Item 2 (weight 20, value 100) and Item 3 (weight 30, value 120) are included in the knapsack because they give the maximum total value without exceeding the capacity W = 50.

Maximum value in knapsack: The maximum total value that can be achieved is 220.

Task 2: Longest Common Subsequence

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

To find the length of the Longest Common Subsequence (LCS) between two strings text1 and text2, we can use dynamic programming. Here's a Java implementation of the LCS function**:**

```
public class LongestCommonSubsequence {
    public int LCS(String text1, String text2) {
        int m = text1.length();
        int n = text2.length();

        // Create a 2D array to store the lengths of longest common subsequence
        int[][] dp = new int[m + 1][n + 1];
```

```
        // Build the dp array
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                if (i == 0 || j == 0) {
                    dp[i][j] = 0; // Base case: empty string has LCS of length 0
                } else if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j - 1] + 1; // Characters match
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]); // Characters do not match
                }
            }
        }

        return dp[m][n]; // Length of LCS
    }

    public static void main(String[] args) {
        String text1 = "abcde";
        String text2 = "ace";

        LongestCommonSubsequence solution = new LongestCommonSubsequence();
        int lcsLength = solution.LCS(text1, text2);

        System.out.println("Length of Longest Common Subsequence: " + lcsLength);
    }
}
```

Explanation:

1 LongestCommonSubsequence Class:

- **LCS Method**: Computes the length of the longest common subsequence between text1 and text2.
- **dp[i][j]** represents the length of LCS of **text1.substring(0, i)** and **text2.substring(0, j).**
- If characters **text1.charAt(i - 1)** and **text2.charAt(j - 1)** are equal, then **dp[i][j] = dp[i-1][j-1] + 1.**
- Otherwise, **dp[i][j] = max(dp[i-1][j], dp[i][j-1]).**
- **Base case: if i == 0 or j == 0, dp[i][j] = 0.**

Main Method: Example usage of the LCS method with two strings (text1 and text2).

For text1 = "abcde" and text2 = "ace", the LCS is "ace" with a length of 3.

Output:

The program will output: Length of Longest Common Subsequence: 3.

Output Explanation

For the given input strings:

text1 = "abcde"

text2 = "ace"

The program computes the length of the Longest Common Subsequence (LCS) between text1 and text2 using the LCS method. Here's the detailed output:

Text1: abcde

Text2: ace

Length of Longest Common Subsequence: 3