Dynamic Programming - II

Let us now move to some advanced-level DP questions

Problem Statement: Knapsack

Given the weights and values of 'N' items, we are asked to put these items in a knapsack, which has a capacity 'C'. The goal is to get the maximum value from the items in the knapsack. Each item can only be selected once, as we don't have multiple quantities of any item.

For Example :

Items: {Apple, Orange, Banana, Melon}
Weights: {2, 3, 1, 4}
Values: {4, 5, 3, 7}
Knapsack capacity: 5

Possible combinations that satisfy the given conditions are:

Apple + Orange (total weight 5) => 9 value
Apple + Banana (total weight 3) => 7 value
Orange + Banana (total weight 4) => 8 value
Banana + Melon (total weight 5) => 10 value

This shows that **Banana + Melon** is the best combination, as it gives us the maximum value, and the total weight does not exceed the capacity.

Approach: First-of-all, let's discuss the brute-force-approach, i.e., the recursive

approach. There are two possible cases for every item, either to put that item into the knapsack or not. If we consider that item, then its value will be contributed towards the total value, otherwise not. To figure out the maximum value obtained by maintaining the capacity of the knapsack, we will call recursion over these two cases simultaneously, and then will consider the maximum value obtained out of the two.

If we consider a particular weight 'w' from the array of weights with value 'v' and the total capacity was 'C' with initial value 'Val', then the remaining capacity of the knapsack becomes 'C-w', and the value becomes 'Val + v'.
Let's look at the recursive code for the same:

```
public int knapsack(int[] weight, int[] values,int i, int n, int maxWeight)
{
      // Base case : if the size of array is 0 or we are not able to add
      // any more weight to the knapsack
      if(n == i || maxWeight == 0) {
            return 0;
      }

      // If the particular weight’s value extends the limit of knapsack’s
      // remaining capacity, then we have to simply skip it
      if(weight[i] > maxWeight) {
            return knapsack(weight, values, i+1, n, maxWeight);
      }

      // Recursive calls
      //1. Considering the weight
      int x = knapsack(weight, values, i+1, n, maxWeight - weight[i]) +
                                                              values[i];
      // 2. Skipping the weight and moving forward
      int y = knapsack(weight, values, i+1, n, maxWeight) + values[i];

      // finally returning the maximum answer among the two
      return Math.max(x, y);
}
```

Now, the memoization and DP approach is left for you to solve. For the code, refer to the solution tab of the same. Also, figure out the time complexity for the same by running the code over some examples and by dry running it.

Problem Statement: LCS (Longest Common Subsequence)

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

Note: Subsequence is a part of the string which can be made by omitting none or some of the characters from that string while maintaining the order of the characters. If s1 and s2 are two given strings then z is the common subsequence of s1 and s2, if z is a subsequence of both of them.

Example 1:

```
s1 = "abcdef"
s2 = "xyczef"
```

Here, the longest common subsequence is "cef"; hence the answer is 3 (the length of LCS).

Example 2:

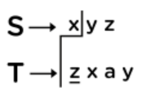
```
s1 = "ahkolp"
s2 = "ehyozp"
```

Here, the longest common subsequence is "hop"; hence the answer is 3.
Approach: Let's first think of a brute-force approach using recursion. For LCS, we have to match the starting characters of both strings. If they match, then simply we can break the problem as shown below:

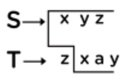
```
s1 = "x|yzar"
s2 = "x|qwea"
```

The rest of the LCS will be handled by recursion. But, if the first characters do not match, then we have to figure out that by traversing which of the following strings, we will get our answer. This can't be directly predicted by just looking at them, so we will be traversing over both of them one-by-one and check for the maximum value of LCS obtained among them to be considered for our answer.

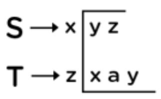
For example: Suppose, string s = "xyz" and string t = "zxay". We can see that their first characters do not match so that we can call recursion over it in either of the following ways:



A =



B =



C=

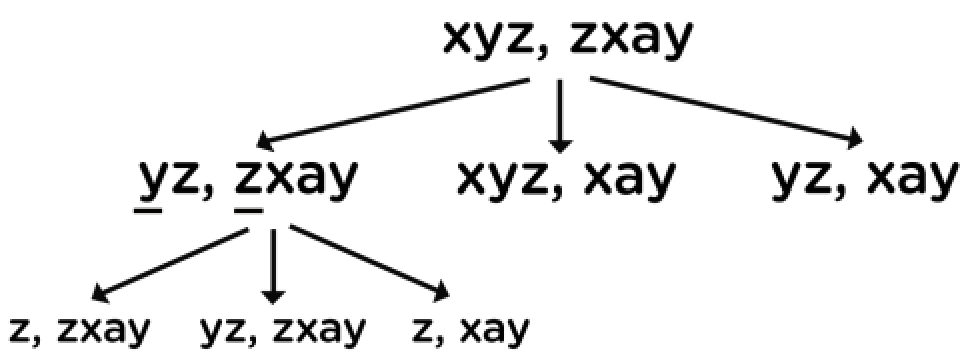
Finally, our answer will be:

```
LCS = max(A, B, C)
```

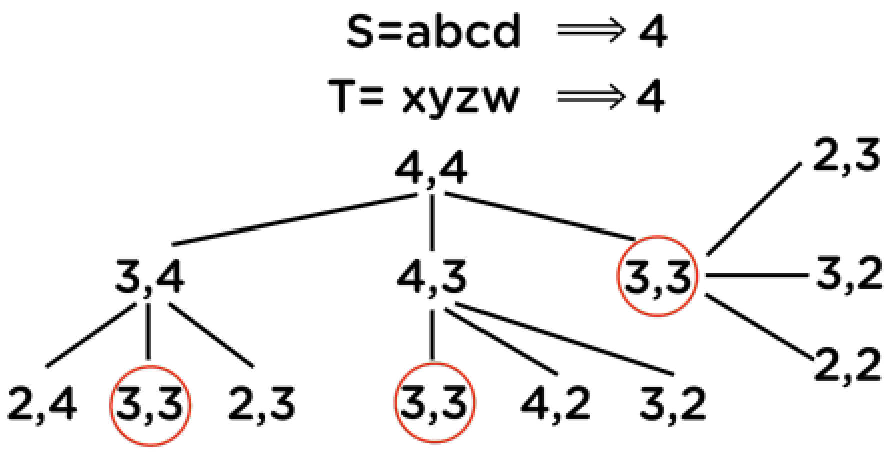
Check the code below and follow the comments for a better understanding.

```
public int lcs(string s, string t) {
      // Base case
      if(s.size() == 0 || t.size() == 0) {
              return 0;
      }

      // Recursive calls
      if(s[0] == t[0]) {
              return 1 + lcs(s.substr(1), t.substr(1));
      }
      else {
              int a = lcs(s.substring(1), t); // discarding the first
                                             // character of string s
              int b = lcs(s, t.substring(1));// discarding the first
                                             // character of string t
              int c = lcs(s.substring(1), t.substring(1));// discarding the
                                                // first character of both
              return Math.max(a, Math.max(b, c)); // Small calculation
      }
}
```

If we dry run this over the example: s = "xyz" and t = "zxay", it will look something like below:



Here, as for each node, we will be making three recursive calls, so the time complexity will be exponential and is represented as O(2^(m+n)), where m and n are the lengths of both strings. This is because, if we carefully observe the above code, then we can skip the third recursive call as it will be covered by the two others. Now, thinking about improving this time complexity... Consider the diagram below, where we are representing the dry run in terms of its length taken at each recursive call.



As we can see there are multiple overlapping recursive calls, the solution can be optimised using memoization followed by DP. So, beginning with the memoization approach, as we want to match all the subsequences of the given two strings, we have to figure out the number of unique recursive calls.

For string s, we can make at most length(s) recursive calls, and similarly, for string t, we can make at most length(t) recursive calls, which are also dependent on each other's solution. Hence, our result can be directly stored in the form of a 2-dimensional array of size (length(s)+1) * (length(t) + 1) as for string s, we have 0 to length(s) possible combinations, and the same goes for string t.So for every index 'i' in string s and 'j' in string t, we will choose one of the following two options:

1. If the character s[i] matches t[j], the length of the common subsequence would be one plus the length of the common subsequence till the i-1 and j-1 indexes in the two respective strings. 2. If the character s[i] does not match t[j], we will take the longest subsequence by either skipping i-th or j-th character from the respective strings .

Hence, the answer stored in the matrix will be the LCS of both strings when the length of string s will be 'i' and the length of string t will be 'j'. Hence, we will get the final answer at the position matrix[length(s)][length(t)]. Moving to the code

```
public int lcs_mem(String s, String t, int[][] output) {
	int m = s.length();
	int n = t.length();

	// Base case
	if(m == 0 || n == 0) {
		return 0;
	}

	// Check if ans already exists
	if(output[m][n] != -1) {
		return output[m][n];
	}

	int ans;
	// Recursive calls
	if(s[0] == t[0]) {
		ans = 1 + lcs_mem(s.substring(1), t.substring(1), output);
	}
	else {
		int a = lcs_mem(s.substring(1), t, output);
		int b = lcs_mem(s, t.substring(1), output);
		int c = lcs_mem(s.substring(1), t.substring(1), output);
		ans = Math.max(a, Math.max(b, c));
	}

	// Save your calculation
	output[m][n] = ans;

	// Return ans
```

```
	return ans;
}

public int lcs_mem(String s, String t) {
	int m = s.length();
	int n = t.length();
	int[][] output = new int[m+1][n+1];
	for(int i = 0; i <= m; i++) {
		for(int j = 0; j <= n; j++) {
			output[i][j] = -1;  // Initializing the 2D array with -1
		}
	}
	return lcs_mem(s, t, output);
}
```

Now, converting this approach into the DP code:

```
public int lcs_DP(String s, String t) {
      int m = s.length();
      int n = t.length();
      // declaring a 2D array of size m*n
      int[][] output = new int[m+1][n+1];

      // Fill 1st row
      for(int j = 0; j <= n; j++) { // as if string t is empty, then the
            output[0][j] = 0;       //lcs(s, t) = 0
      }

      // Fill 1st col
      for(int i = 1; i <= m; i++) { // as if string s is empty, then the
            output[i][0] = 0;       // lcs(s, t) = 0
      }

      for(int i = 1; i <= m; i++) {
            for(int j = 1; j <= n; j++) {
                  // Check if 1st char matches
                  if(s[m-i] == t[n-j]) {
                        output[i][j] = 1 + output[i-1][j-1];
                  }
                  else {
                        int a = output[i-1][j];
                        int b = output[i][j-1];
                        int c = output[i-1][j-1];
                        output[i][j] = Math.max(a, Math.max(b, c));
                  }
            }
      }
      return output[m][n];      // final answer
}
```

Time Complexity: We can see that the time complexity of the DP and memoization approach is reduced to O(m*n) where m and n are the lengths of the given strings.

Problem Statement: Min Cost Path

Given an integer matrix of size **m*n**, you need to find out the value of minimum cost to reach from the cell **(0, 0) to (m-1, n-1)**. From a cell **(i, j)**, you can move in three directions : **(i+1, j), (i, j+1) and (i+1, j+1).** The cost of a path is defined as the sum of values of each cell through which the path passes.

For example, The given input is as follows:

```
3 4
3 4 1 2
2 1 8 9
4 7 8 1
```

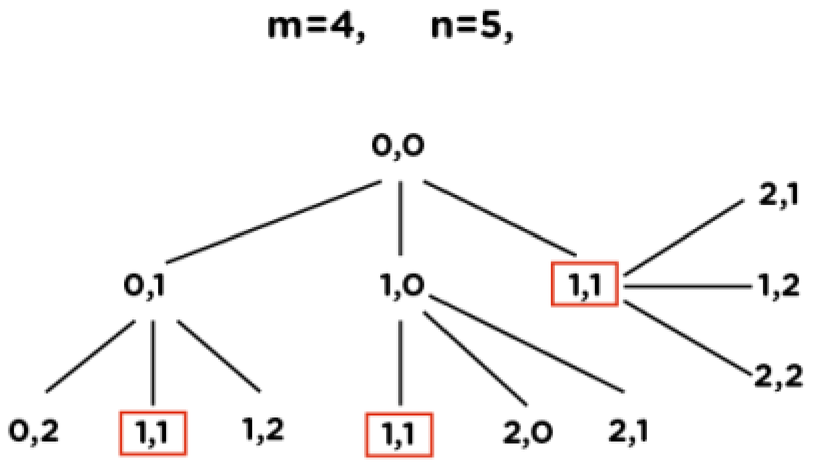
The path that should be followed is 3 -> 1 -> 8 -> 1. Hence the output is 13.

Approach:

- Thinking about the recursive approach to reach from the cell (0, 0) to (m-1, n-1), we need to decide for every cell about the direction to proceed out of three.
- We will simply call recursion over all the three choices available to us, and finally, we will be considering the one with minimum cost and add the current cell's value to it.
- Let's now look at the recursive code for this problem:

```java
public int minCostPath(int[][] input, int m, int n, int i, int j) {
      // Base case: reaching out to the destination cell
      if(i == m- 1 && j == n- 1) {
            return input[i][j];
      }

      if(i >= m || j >= n) {  // checking for within the constraints or not
            return Integer.MAX_VALUE; //if not, returning +infinity so that
      }                              // it will not be considered as the answer


      // Recursive calls
      int x = minCostPath(input, m, n, i, j+1); // Towards right direction
      int y = minCostPath(input, m, n, i+1, j+1);// Towards diagonal
      int z = minCostPath(input, m, n, i+1, j);  // Towards down direction

      // Small Calculation: figuring out the minimum value and then adding
     // current cells value to it
      int ans = Math.min(x, Math.min(y, z)) + input[i][j];
      return ans;
}

public int minCostPath(int[][] input, int m, int n) {
      // we will be using a helper function
      return minCostPath(input, m, n, 0, 0);  // as wee need to keep the
}                                         //track of current row and column
```

Let's dry run the approach to see the code flow. Suppose, m = 4 and n = 5; then the recursive call flow looks something like below:



Here, we can see that there are many repeated/overlapping recursive calls(for example: (1,1) is one of them), leading to exponential time complexity, i.e., O(3^n). If we store the output for each recursive call after their first occurrence, we can easily avoid the repetition. It means that we can improve this using memoization.

Now, let's move on to the Memoization approach. In memoization, we avoid repeated overlapping calls by storing the output of each recursive call in an array. In this case, we will be using a 2D array instead of 1D, as we already discussed in our previous lectures that the storage used for the memoization is generally the same as the one that recursive calls use to their maximum. Refer to the memoization code (along with the comments) below for better understanding

```
public int helper(int[][]input, int m, int n, int i, int j, int[][] output)
{
    if(i == m- 1 && j == n- 1) {     // Base case
        return input[i][j];
    }

    if(i >= m || j >= n) {
        return Integer.MAX_VALUE;
    }

    // Check if ans already exists
    if(output[i][j] != -1) {
        return output[i][j];     // as each cell stores its own ans
    }

    // Recursive calls
    int x = helper(input, m, n, i, j+1, output);
    int y = helper(input, m, n, i+1, j+1, output);
    int z = helper(input, m, n, i+1, j, output);

    // Small Calculation
    int a = Math.min(x, Math.min(y, z)) + input[i][j];

    // Save the answer for future use
    output[i][j] = a;
```

```
    return a;
}

public int minCostPath_Mem(int[][] input, int m, int n, int i, int j) {
    int[][] output = new int[m][];

    for(int i = 0; i < m; i++) {
        output[i] = new int[n];
        for(int j = 0; j < n; j++) {
            output[i][j] = -1;  // Initialising the output array by -1.
                               // Here, -1 denotes that the value of the
                               //current cell is unknown and could be
                               //replaced only after we  find the same
        }
    }
    return helper(input, m, n, i, j, output);
}
```

Here, we can observe that as we move from the cell (0,0) to (m-1, n-1), in general, the i-th row varies from 0 to m-1, and the j-th column runs from 0 to

n-1. Hence, the unique recursive calls will be a maximum of (m-1) * (n-1), which leads to the time complexity of O(m*n).
To get rid of the recursion, we will now proceed towards the DP approach. The DP approach is simple. We just need to create a solution array (lets name that as ans), where:

```
ans[i][j] = minimum cost to reach from (i, j) to (m-1, n-1)
```

Now, initialise the last row and last column of the matrix with the sum of their values and the value, just after it. This is because, in the last row or column, we can reach there from their forward cell only (You can manually check it), except the cell (m-1, n-1), which is the value itself.

```
ans[m-1][n-1] = cost[m-1][n-1]
ans[m-1][j] = ans[m-1][j+1] + cost[m-1][j]  (for 0 < j < n)
ans[i][n-1] = ans[i+1][n-1] + cost[i][m-1] (for 0 < i < m)
```

Next, we will simply fill the rest of our answer matrix by checking out the minimum among values from where we could reach them. For this, we will use the same formula as used in the recursive approach:

```
ans[i][j] = Minimum(ans[i+1][j], ans[i+1][j+1], ans[i][j+1]) + cost[i][j]
```

Finally, we will get our answer at the cell (0, 0), which we will return. The code looks as follows:

```
public int minCost_DP(int[][] input, int m, int n) {
      int[][] ans = new int[m][n];

      ans[m-1][n-1] = input[m-1][n-1];

      // Last row
      for(int j = n - 2; j >= 0; j--) {
            ans[m-1][j] = input[m-1][j] + ans[m-1][j+1];
      }

      // Last col
      for(int i = m-2; i >= 0; i--) {
            ans[i][n-1] = input[i][n-1] + ans[i+1][n-1];
      }

      // Calculation using formula
      for(int i = m-2; i >= 0; i--) {
            for(int j = n-2; j >= 0; j--) {
                ans[i][j] = input[i][j] + Math.min(ans[i][j+1],
                                      Math.min(ans[i+1][j+1], ans[i+1][j]));
            }
      }
      return ans[0][0];     // Our Final answer as discussed above
}
```

Note: This is the bottom-up approach to solve the question using DP.