Dynamic Programming- 2

Let us now move to some advanced-level DP questions, which deal with 2D arrays.

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:

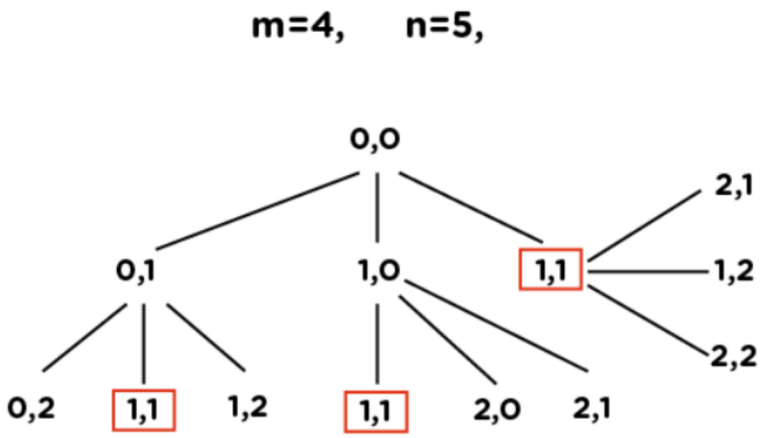
- Thinking about the **recursive approach** to reach from the cell **(0, 0)** to **(m-1, n-1)**, we need to decide for every cell about the direction to proceed out of three.
- We will simply call recursion over all the three choices available to us, and finally, we will be considering the one with minimum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

```
import sys

# Returns cost of minimum cost path from (0,0) to (m, n) in mat[R][C]
def minCost(cost, m, n):
    if (n < 0 or m < 0):
        return sys.maxsize
    elif (m == 0 and n == 0):
        return cost[m][n]
    else:
        return cost[m][n] + min( minCost(cost, m-1, n-1),
                                 minCost(cost, m-1, n),
                                 minCost(cost, m, n-1) )

#A utility function that returns minimum of 3 integers */
def min(x, y, z):
    if (x < y):
        return x if (x < z) else z
    else:
        return y if (y < z) else z
```

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:

```
import sys
def minCost(cost,i,j,n,m,dp):

      # Special Case
      if i == n-1 and j==m-1:
            return cost[i][j]

      # Base Case
      if i>=n or j>=m:
            return sys.maxsize

      if dp[i][j+1] == sys.maxsize:
            ans1 = minCost(cost,i,j+1,n,m,dp)
            dp[i][j+1] = ans1
      else:
            ans1 = dp[i][j+1]

      if dp[i+1][j] == sys.maxsize:
            ans2 = minCost(cost,i+1,j,n,m,dp)
            dp[i+1][j] = ans2
```

```
        else:
                ans2 = dp[i+1][j]

        if dp[i+1][j+1] == sys.maxsize:
                ans3 = minCost(cost,i+1,j+1,n,m,dp)
                dp[i+1][j+1] = ans3
        else:
                ans3 = dp[i+1][j+1]

        ans= cost[i][j]+min(ans1, ans2, ans3)
        return ans

cost = [[1,5,11],[8,13,12],[2,3,7],[15,16,18]]
n=4
m=3
dp= [[sys.maxsize for j in range(m+1) for i in range(n+1)]
ans = minCost(cost, 0,0,4,3,dp)
print(ans)
```

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, initialize 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] = min(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:

```
R = 3
C = 3

def minCost(cost, m, n):

    ans = [[0 for x in range(C)] for x in range(R)]

    ans[0][0] = cost[0][0]

    # Initialize first column of total cost(tc) array
    for i in range(1, m+1):
        ans[i][0] = ans[i-1][0] + cost[i][0]

    # Initialize first row of tc array
    for j in range(1, n+1):
        ans[0][j] = ans[0][j-1] + cost[0][j]

    # Construct rest of the tc array
    for i in range(1, m+1):
        for j in range(1, n+1):
            min_temp = min(ans[i-1][j-1],ans[i-1][j],ans[i][j-1])
            ans[i][j] = min_temp + cost[i][j]

    return ans[m][n]
```

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

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:

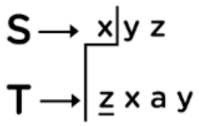
```
s1 = "x|yzar"
s2 = "x|qwea"
```

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:

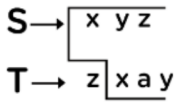
Suppose, string `s = "xyz"` and string `t = "zxay"`.

We can see that their first characters do not match so that we can call recursion over it in either of the following ways:

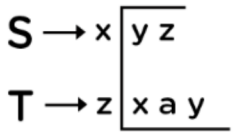


A =

B =



C=



Finally, our answer will be:

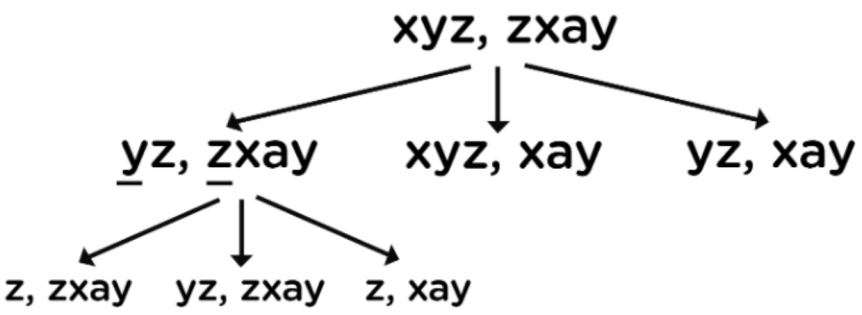
```
LCS = max(A, B, C)
```

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

```
def lcs(s, t, m, n):

    if m == 0 or n == 0: #Base Case
       return 0;
    elif s[m-1] == t[n-1]:
       return 1 + lcs(s, t, m-1, n-1);
    else:
       return max(lcs(s, t, m, n-1), lcs(s, t, m-1, n));
```

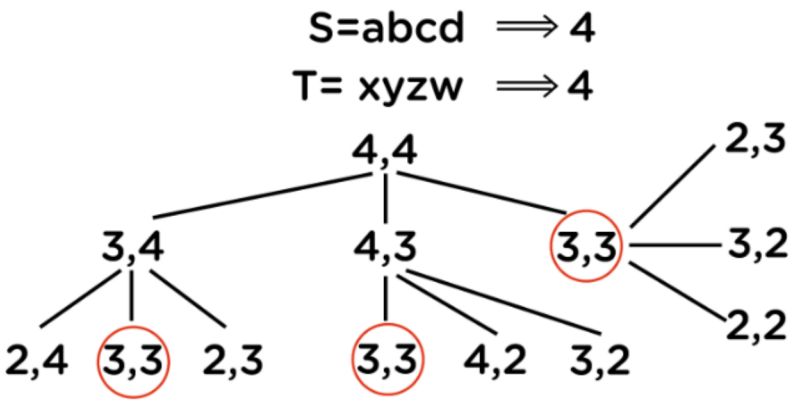
If we dry run this over the example: `s = "xyz"` and `t = "zxay"`, it will look something like below:



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 over 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 optimized 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:

```
N = 0
M = 0

def lcs(s, t, i, j, memo):

    # one or both of the strings are fully traversed

    if i == N or j == M:
        return 0

    # if result for the current pair is already present in
    # the table

    if memo[i][j] != -1:
        return memo[i][j]

    # check if the current characters in both the strings are equal

    if s[i] == t[j]:

        # check for the next characters in both the strings

        memo[i][j] = lcs(s, t, i + 1, j + 1, memo) + 1
    else:

        memo[i][j] = max(lcs(s,t,i,j+1,memo), lcs(s,t,i+1,j,memo))

    return memo[i][j]
```

Now, converting this approach into the **DP** code:

```
def lcs(s , t):
    # find the length of the strings
    m = len(s)
    n = len(t)

    # declaring the array for storing the dp values
    L = [[None]*(n+1) for i in xrange(m+1)]

    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                L[i][j] = 0
            elif s[i-1] == t[j-1]:
                L[i][j] = L[i-1][j-1]+1
            else:
                L[i][j] = max(L[i-1][j] , L[i][j-1])

    # L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]
    return L[m][n]
```

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: 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:

```
def knapSack(W, wt, val, n):

    # Base Case: if the size of array is 0 or we are not able to add
    # any more weight to the knapsack
    if n == 0 or W == 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 (wt[n-1] > W):
        return knapSack(W, wt, val, n-1)
    else:#Recursive Calls
        return max(val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
            knapSack(W, wt, val, n-1))
```

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.

Practice problems:

The link provided below contains 26 problems based on Dynamic programming and numbered as A to Z, A being the easiest, and Z being the toughest.

https://atcoder.jp/contests/dp/tasks