# Dynamic Programming and Memoization

## What is Dynamic Programming?

Dynamic Programming is a programming paradigm, where the idea is to memoize the already computed values instead of calculating them again and again in the recursive calls. This optimization can reduce our running time significantly and sometimes reduce it from exponential-time complexities to polynomial time.

## Introduction

We know that Fibonacci numbers are defined as follows

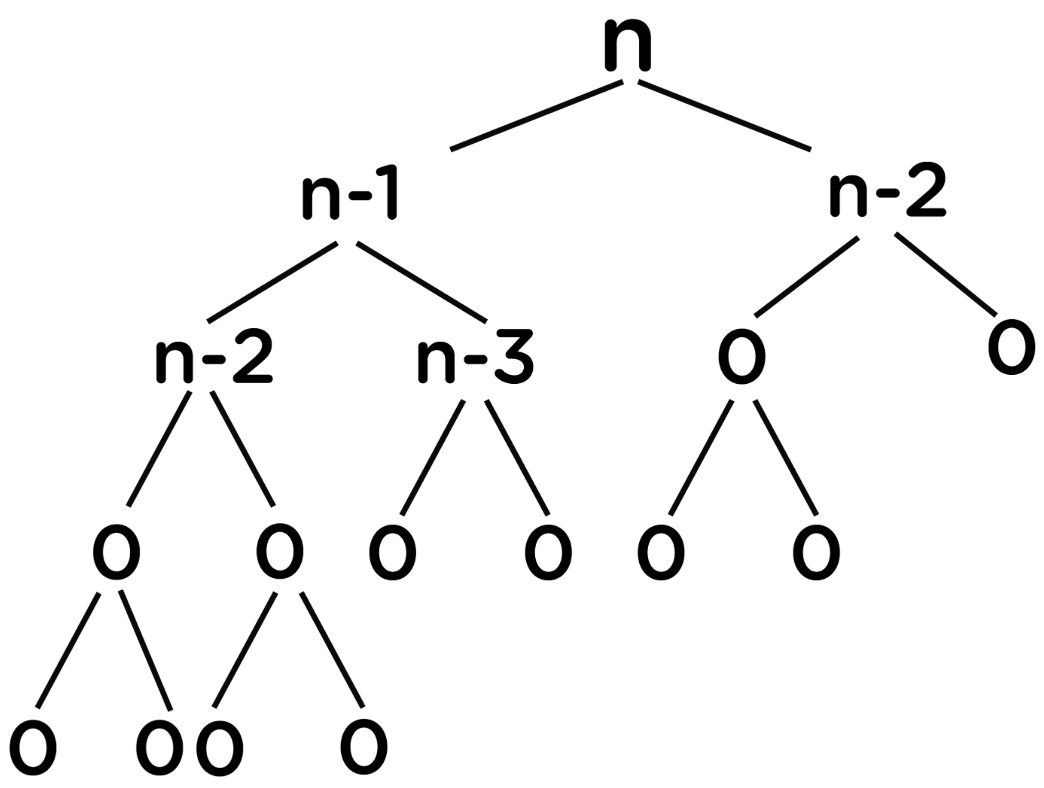
| | |
|---|---|
| **fibo(n)** = n | for n <= 1 |
| **fibo(n)** = fibo (n - 1) + fibo (n - 2) | otherwise |

Suppose we need to find the nth Fibonacci number using recursion that we have already found out in our previous sections.

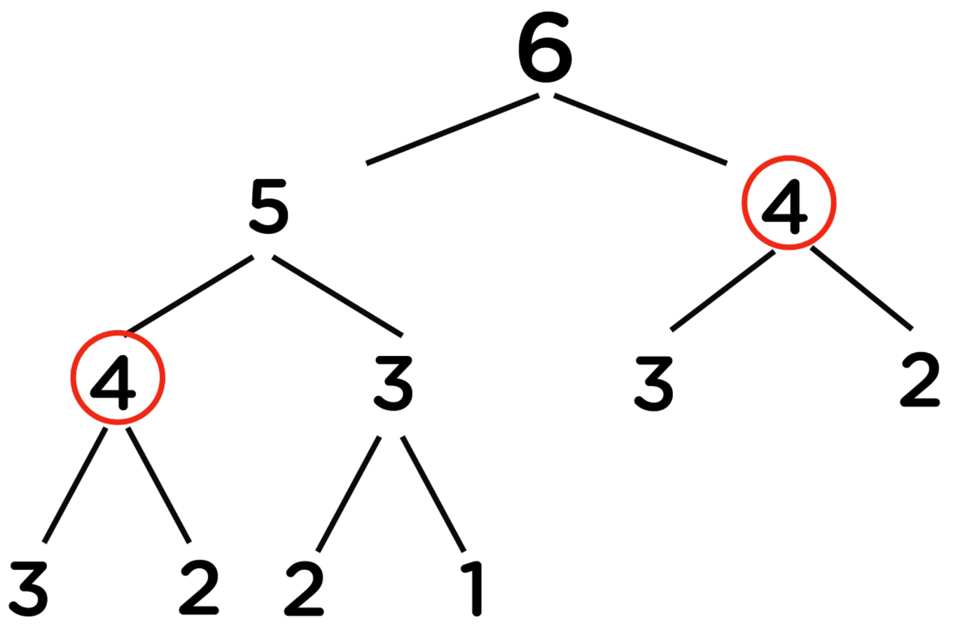
```
function fibo(n):
        if(n <= 1)
                return n

        return fibo(n-1) + fibo(n-2)
```

- Here, for every n, we need to make a recursive call to **f(n-1)** and **f(n-2)**.
- For **f(n-1)**, we will again make the recursive call to **f(n-2)** and **f(n-3).**
- Similarly, for **f(n-2)**, recursive calls are made on **f(n-3)** and **f(n-4)** until we reach the base case.
- The recursive call diagram will look something like shown below:



- At every recursive call, we are doing constant work(k) (addition of previous outputs to obtain the current one).
- At every level, we are doing (**2^n) * K** work (where n = 0, 1, 2, …).
- Since reaching 1 from n will take n calls, therefore, at the last level, we are doing (2^(**n-1)) *** k work.
- Total work can be calculated as:(2^0 + 2^1 + 2^2 + ... + 2^(n-1)) * k  ≃  (2^n) * k
- Hence, it means time complexity will be **O(2^n)**.
- We need to improve this complexity. Let's look at the example below for finding the 6**th** Fibonacci number.



**Important Observations**

- We can observe that there are repeating recursive calls made over the entire program.
- As in the above figure, for calculating f(5), we need the value of f(4) (first recursive call over f(4)), and for calculating f(6), we again need the value of f(4) (second similar recursive call over f(4)).
- Generally, while recursing, there are repeated recursion calls, which increases the time complexity of the program.

To overcome this problem, we will store the output of previously encountered values (preferably in arrays as these are most efficient to traverse and extract data). Next time whenever we will be making the recursive calls over these values, we will directly consider their already stored outputs and then use these in our calculations instead of calculating them over again.

This way, we can improve the running time of our code.

## Memoization

This process of storing each recursive call's output and then using them for further calculations preventing the code from calculating these again is called **Memoization**.

- Notice that or answer cannot be -1. Hence if we encounter any value equal to it, we can know that this value is yet to be completed. We could have used any other value as well that cannot be a possible answer. Let us take -1 as of now.
- To achieve this in our example we will simply take an answer array, initialized to -1.
- Now while making a recursive call, we will first check if the value stored in this answer array corresponding to that position is -1 or not.
- If it is -1, it means we haven't calculated the value yet and need to proceed further by making recursive calls for the respective value.
- After obtaining the output, we need to store this in the answer array so that next time, if the same value is encountered, it can be directly used from this answer array.

Now in this process of memoization, considering the above Fibonacci numbers example, it can be observed that the total number of unique calls will be at most **(n+1)** only.

Let's look at the memoization code for Fibonacci numbers below:

**Pseudocode:**

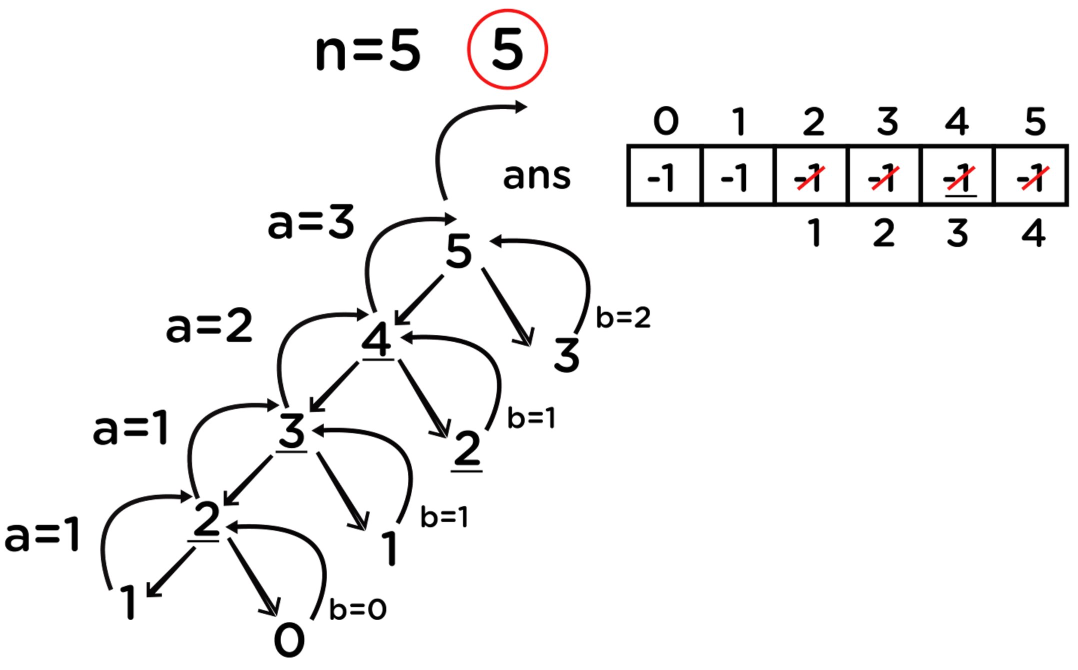
```
function fibo(n, dp)
    //   base case
    if n equals 0 or n equals 1
        return n

    //   checking if has been already calculated
    if dp[n] not  equals -1
        return dp[n]

    //   final ans
    myAns = fibo(n - 1) + fibo(n - 2)
    dp[n] = myAns

    return myAns
```

Let's dry run for n = 5, to get a better understanding:



Again, if we observe carefully, we can see that for any number, we are not able to make a recursive call on the right side of it. This means that we can make at most 5+1 = 6 (n+1) unique recursive calls which reduce the time complexity to O(n) which is highly optimized as compared to simple recursion.

## Top-down approach

- Memoization is a **top-down approach,** where we save the answers to recursive calls so that they can be used to calculate future answers when the same recursive calls are to be evaluated and improve the time complexity to a greater extent.
- Finally, what we are doing is making a recursive call to every index of the answer array and calculating the value for it using previous outputs stored.
- Recursive calls terminate over the base case, which means we are already aware of the answers that should be stored in the base case's indexes.
- In cases of Fibonacci numbers, these indexes are 0 and 1 as f(0) = 0 and f(1) = 1. So we can directly allot these two values to our answer array and then use these to calculate f(2), which is f(1) + f(0), and so on for every other index.
- This can be simply done iteratively by running a loop from i = (2 to n).
- Finally, we will get our answer at the 5**th** index of the answer array as we already know that the i-th index contains the answer to the i-th value.

## Bottom-up approach

We are first trying to figure out the dependency of the current value on the previous values and then using them to calculate our new value. Now, we are looking for those values which do not depend on other values, which means they are independent (the base case's values as these are the smallest problems about which we are already aware of). Finally, we will follow a **bottom-up approach** to reach the desired index.

Let us now look at the DP code for calculating the n**th** Fibonacci number:

**Pseudocode:**

```
function fibonacci(n):

    f = array[n+1]
    //  base case
    f[0] = 0
    f[1] = 1

    for i from 2 to n:
        //  calculating the f[i] based on the last two values
        f[i] = f[i-1] + f[i-2]

    return f[n]
```

**General Steps**

- Figure out the most straightforward approach for solving a problem using recursion.
- Now, try to optimize the recursive approach by storing the previous answers using memoization.
- Finally, replace recursion by iteration using dynamic programming. (It is preferred to be done in this manner because recursion generally has an increased space complexity as compared to iteration methods.)

## Min Cost Path

**Problem Statement:** 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:

**Pseudocode:**

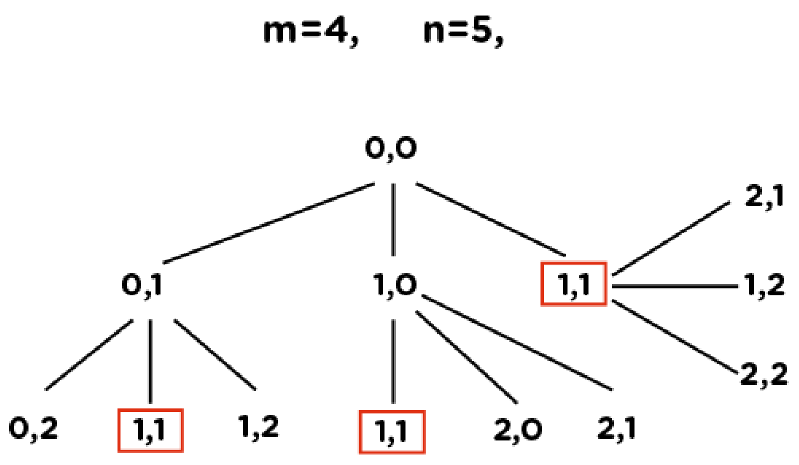
```
function minCost(cost,m , n, dp)
        //  base case
        If m == 0 AND n == 0
                return cost[m][n]

        //  outside the grid
        if m < 0 or n < 0
                return infinity

        recursionResult1 = minCost(cost, m-1, n , dp)
        recursionResult2 = minCost(cost, m, n-1 , dp)
        recursionResult3 = minCost(cost, m-1, n-1 , dp)
        myResult = min(recursionResult1, recursionResult2, recursionResult3) + cost[m][n]

        return myResult
```

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 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 the storage used for the memoization depends on the **states**, which are basically the necessary variables whose value at a particular instant is required to calculate the optimal result.

Refer to the memoization code (along with the comments) below for better understanding:

**Pseudocode:**

```
function minCost(cost,m , n, dp)
        //   base case
        If m == 0 AND n == 0
                return cost[m][n]

        //   outside the grid
```

```
if m < 0 or n < 0
        return infinity

if dp[m][n] != -1
        return dp[m][n]

recursionResult1 = minCost(cost, m-1, n , dp)
recursionResult2 = minCost(cost, m, n-1 , dp)
recursionResult3 = minCost(cost, m-1, n-1 , dp)
myResult = min(recursionResult1, recursionResult2, recursionResult3) + cost[m][n]

//   store in dp
dp[m][n] = myresult

return myResult
```

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:

**Pseudocode:**

```
function minCost(cost, m, n)
    ans = array[m+1][n+1]
    ans[0][0] = cost[0][0]

    //  Initialize first column of ans array
    for i from 1 to m
        ans[i][0] = ans[i-1][0] + cost[i][0]

    //  Initialize first row of ans array
    for j from 1 to n
        ans[0][j] = ans[0][j-1] + cost[0][j]

    //  Construct rest of the ans array
    for i from 1 to m
        for j from 1 to
            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.

**Time Complexity:** 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)**.

**Space Complexity:** Since we are using an array of size (m*n) the space complexity turns out to be **O(m*n)**.

## LCS (Longest Common Subsequence)

**Problem statement**: 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).

**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**"

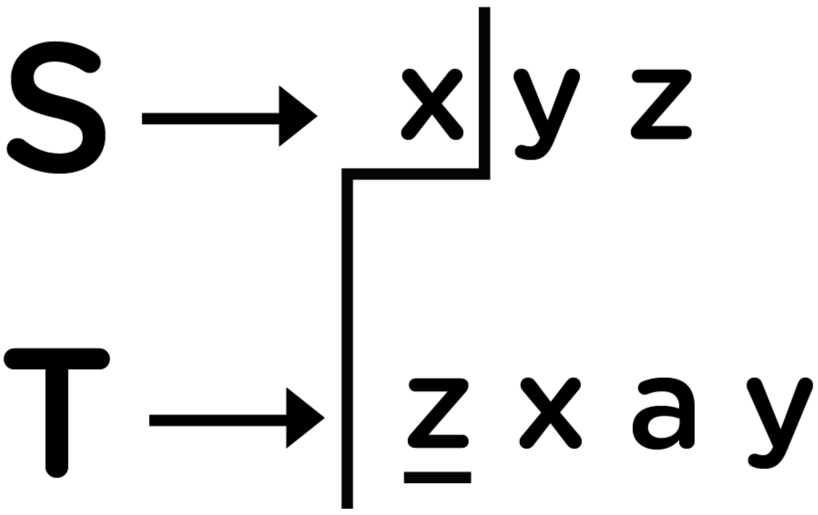
The rest of the LCS will be handled by recursion. But, if the first characters do not match, then we have to figure out that by traversing which of the following strings, we will get our answer. This can't be directly predicted by just looking at them, so we will be traversing over both of them one-by-one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

**For example:**

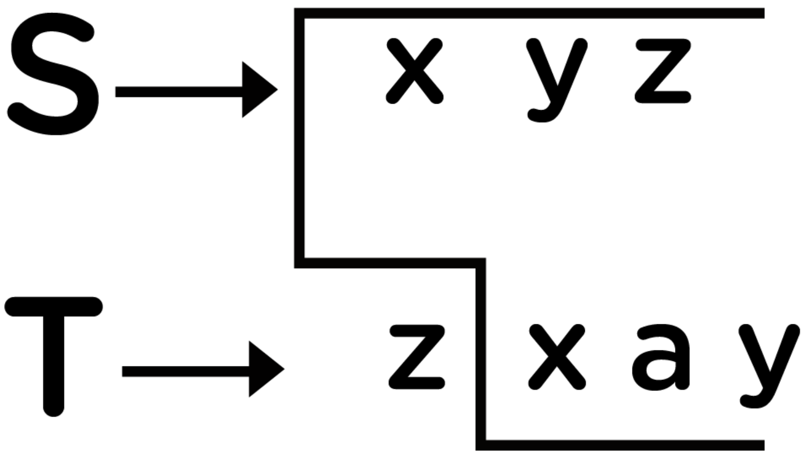
Suppose, string s = "**xyz**" and string t = "**zxay**" .

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

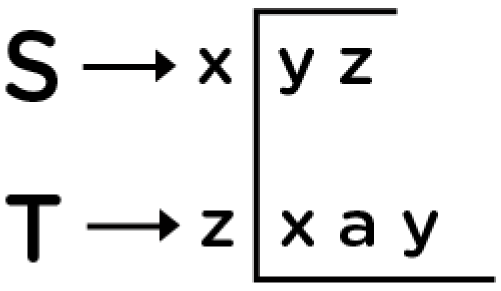
A =



B =



C =



Finally, our answer will be: **LCS = max(A, B, C)**

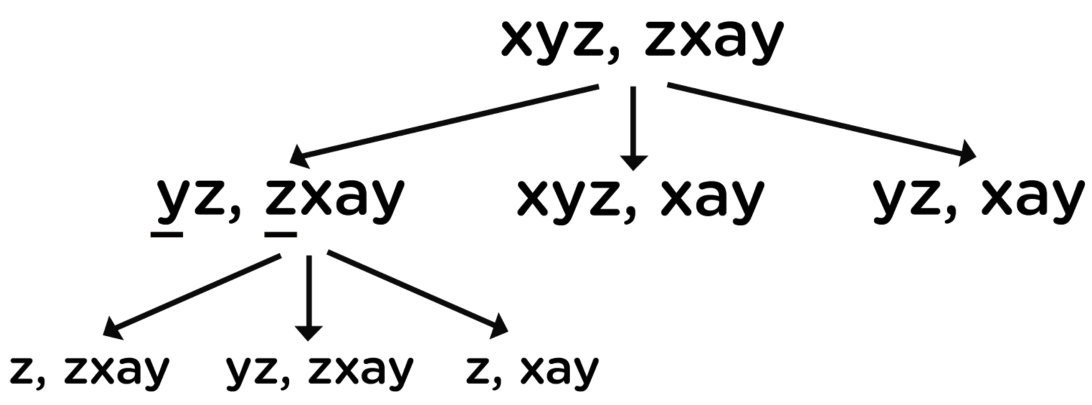
Check the code below and follow the comments for a better understanding.

**Pseudocode:**

```
function lcs(s, t, m, n):
        //   Base Case
        if m equals 0 or n equals 0
        return 0

    //   match m -1th character of s with n -1 th character of t
    else if s[m-1] equals 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))
```

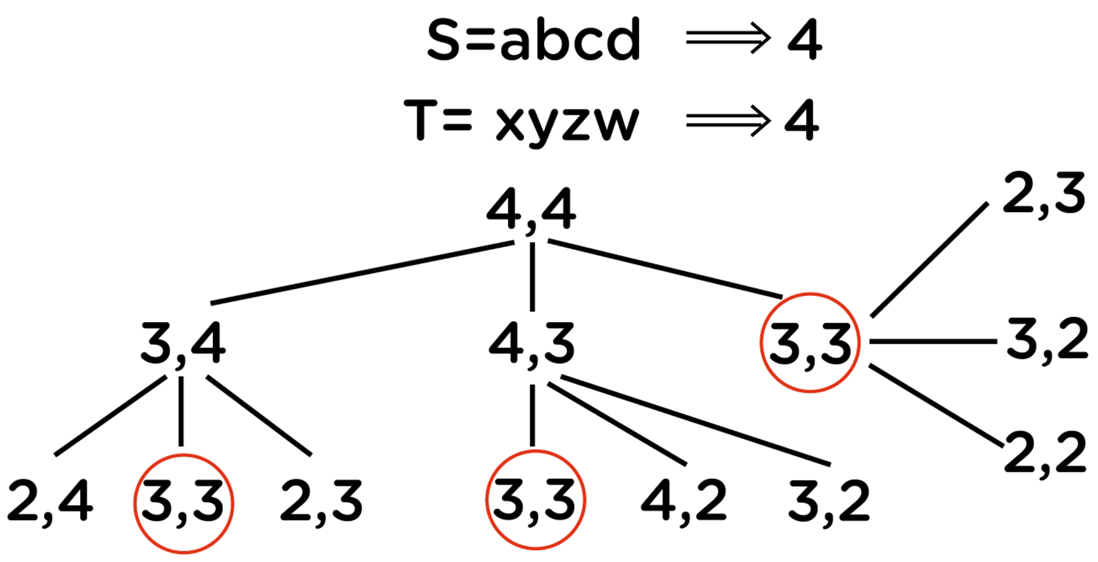
If we dry run this over the example: s = "xyz" and t = "zxay", it will look something like below:



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:

**Pseudocode:**

```
function LCS(s, t, i, j, memo)
        //   one or both of the strings are fully traversed
        if i equals len(s) or j equals len(t)
                return 0

        //   if result for the current pair is already present in the table
        if memo[i][j] not equals -1
                return memo[i][j]

        if s[i] equals t[j]
                //   check for the next characters in both the strings
                memo[i][j] = lcs(s, t, i + 1, j + 1, memo) + 1
        else
                //   check if the current characters in both the strings are equal
                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:

**Pseudocode:**

```
function LCS(s , t)

        //   find the length of the strings
        m = len(s)
        n = len(t)

        //   declaring the array for storing the dp values
        L = array[m + 1][n + 1]

        for i from 0 to m
                for j from 0 to n
                if i equals 0 or j equals 0
                                L[i][j] = 0
                else if s[i-1] equals 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.

**Space Complexity:** Since we are using an array of size (m*n) the space complexity turns out to be **O(m*n)**.

## Applications of Dynamic programming

- They are often used in machine learning algorithms, for eg Markov decision process in reinforcement learning.
- They are used for applications in interval scheduling.
- They are also used in various algorithmic problems and graph algorithms like Floyd warshall's algorithms for the shortest path, the sum of nodes in subtree, etc.