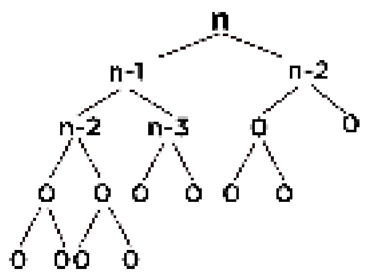
Dynamic Programming- 1

Introduction

Suppose we need to find the nth Fibonacci number using recursion that we have already found out in our previous sections. Let's directly look at its code:

```
def fibo(n):
      if(n <= 1):
            return n
      return fibo(n-1) + fibo(n-2)
```

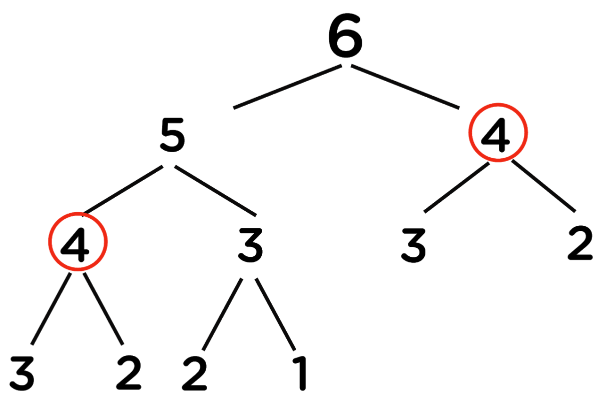
- Here, for every n, we need to make a recursive call to **f(n-1)** and **f(n-2)**.
- For **f(n-1)**, we will again make the recursive call to **f(n-2)** and **f(n-3).**
- Similarly, for **f(n-2)**, recursive calls are made on **f(n-3)** and **f(n-4)** until we reach the base case.
- The recursive call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

$$(2^0 + 2^1 + 2^2 + ... + 2^{n-1}) * k \approx 2^n k$$

- 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 6th Fibonacci number.



Important Observation:

- 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)).
- Both of these recursive calls are shown above in the outlining circle.
- Similarly, there are many other values for which we are repeating the recursive calls.
- 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. 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**.

- 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.

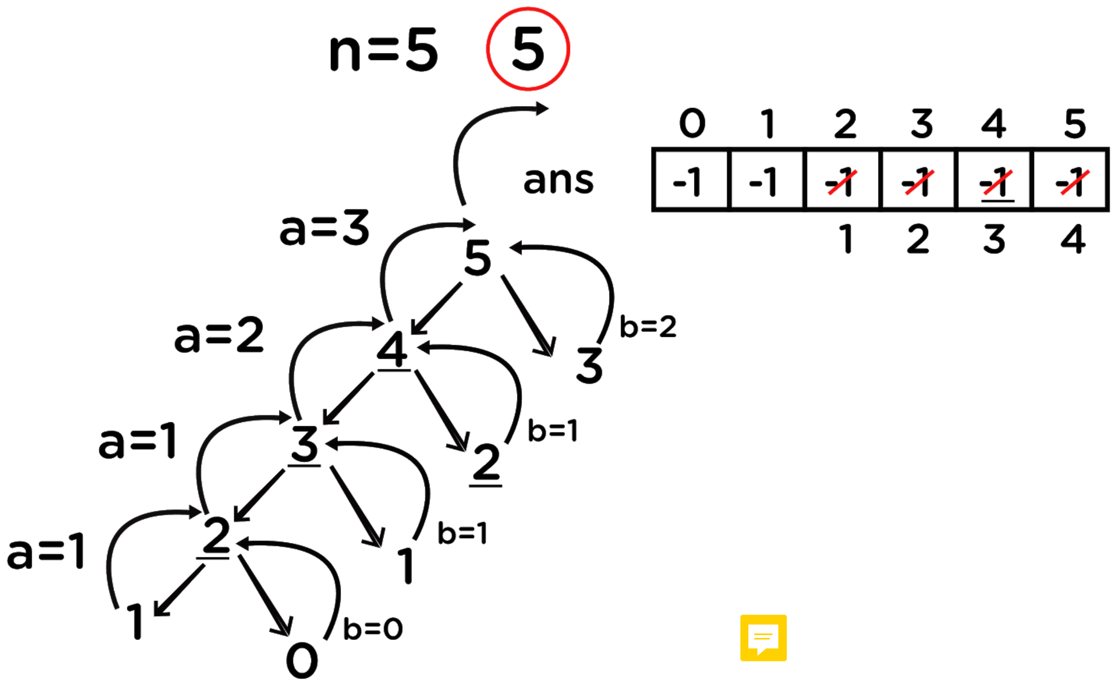
Now in this process of memoization, considering the above Fibonacci numbers example, it can be observed that the total number of unique calls will be at most **(n+1)** only.

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

```
def fibb(n):
      if n==0 or n==1:#Base
            return n

      if dp[n-1] == -1: #checking if has been already calculated
            ans1 = fibb(n-1,dp) # If being calculated first time
            dp[n-1] = ans1
      else:
            ans1 = dp[n-1]

      if dp[n-2] == -1: #checking if has been already calculated
            ans2 = fibb(n-2,dp)
            dp[n-2] = ans2
      else:
            ans2 = dp[n-2]
```

```
        myAns = ans1 + ans2 #final answer
        return myAns

n = int(input())
dp = [-1 for in range(n+1)]
ans = fibb(n,dp)
print(ans)
Le
```

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.

Summary

- Memoization is a **top-down approach,** where we save the previous answers so that they can be used to calculate future answers 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 5th index of the answer array as we already know that the i-th index contains the answer to the i-th value.

We are first trying to figure out the dependency of the current value on the previous values and then using them calculating 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. This approach of converting recursion into iteration is known as **Dynamic programming(DP).**

Let us now look at the DP code for calculating the nth Fibonacci number:

```python
def fibonacci(n):
    f = [0, 1]
    for i in range(2, n+1):
        f.append(f[i-1] + f[i-2]) #Bottom up approach
    return f[n]
```

Note: Generally, memoization is a recursive approach, and DP is an iterative approach.

For all further problems, we will do the following:

1. Figure out the most straightforward approach for solving a problem using recursion.
2. Now, try to optimize the recursive approach by storing the previous answers using memoization.
3. 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.)

Problem Statement: Min steps to 1

Given a positive integer n, find the minimum number of steps s, that takes n to 1. You can perform any one of the following three steps:

1. Subtract 1 from it. (n = n-1).
2. If its divisible by 2, divide by 2. (if n%2 == 0, then n= n/2).
3. If its divisible by 3, divide by 3. (if n%3 == 0, then n = n / 3).

Example 1: For n = 4:

STEP-1: n = 4/2 = 2

STEP-2: n = 2/2 = 1

Hence, the answer is **2.**

Example 2: For n = 7:

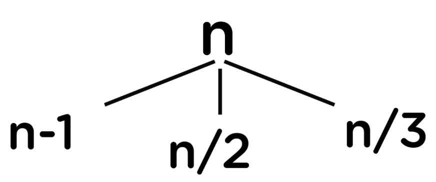
STEP-1: n = 7 - 1 = 6

STEP-2: n = 6/3 = 2

STEP-3: n = 2/2 = 1

Hence, the answer is **3**.

Approach: We are only allowed to perform the above three mentioned ways to reduce any number to 1.



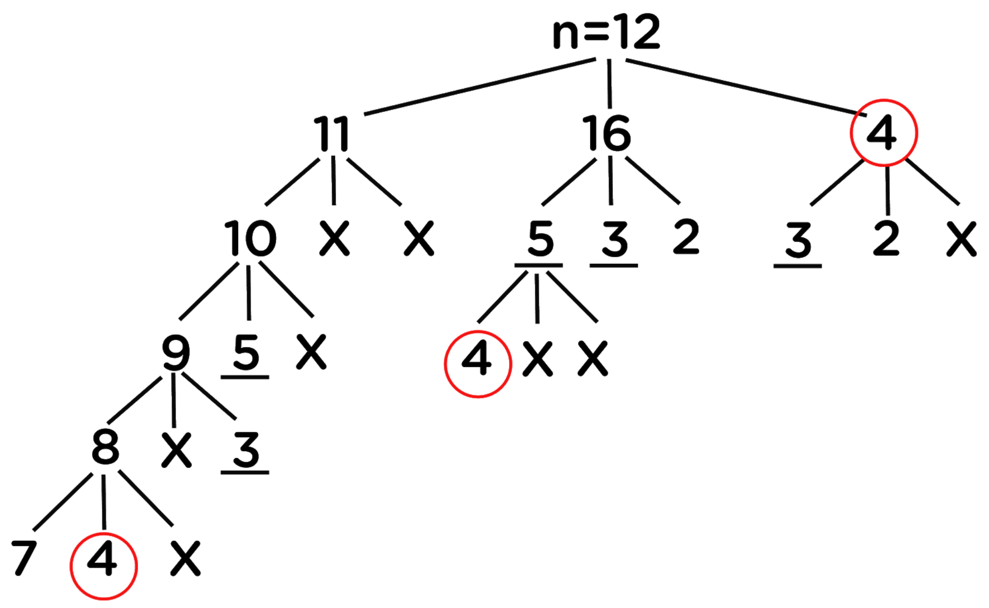
Let's start thinking about the brute-force approach first, i.e., recursion.

We will make a recursive call to each of the three steps keeping in mind that for dividing by 2, the number should be divisible by 2 and similarly for 3 as given in the question statement. After that take the minimum value out of the three obtained and simply add 1 to the answer for the current step itself. Thinking about the base case, we can see that on reaching 1, simply we have to return 0 as it is our destination value. Let's now look at the code:

```
def getMinSteps(n):
    # base case
    if (n <= 1):
        return 0

    x = getMinSteps(n-1) #Recursive call 1
    y = MAX_VALUE #Initialise to infinity to check divisibility
    Z = MAX_VALUE
    if (n%2 == 0):
        y = getMinSteps(n//2) #Recursive call 2
    if (n%3 == 0):
        z = getMinSteps(n//3) #Recursive call 3

    return min(x, min(y,z))+1
```

Now, we have to check if we can optimize the code that we have written. It can be done using memoization. But, for memoization to apply, we need to check if there are any overlapping sub-problems so that we can store the previous values to obtain the new ones. To check this let's dry run the problem for n = 12:

(Here X represents that the calls are not feasible as the number is not divisible by either of 2 or 3)



Here, if we blindly make three recursive calls at each step, then the time complexity will approximately be **O(3^n)**.

From the above, it is visible that there are repeating sub-problems. Hence, this problem can be optimized using memoization.

Now, we need to figure out the number of unique calls, i.e., how many answers we are required to save. It is clear that we need at most n+1 responses to be saved, starting from n = 0, and the final answer will be present at index n.

The code will be nearly the same as the recursive approach; just we will not be making recursive calls for already stored outputs. Follow the code and comments below:

```
def getMinStepsHelper(n, memo):
      # base case
    if (n == 1):
        return 0
    if (memo[n] != -1):
        return memo[n]

    res = getMinSteps(n-1, memo)

    if (n%2 == 0):
        res = min(res, getMinSteps(n//2, memo))
    if (n%3 == 0):
        res = min(res, getMinSteps(n//3, memo))

    # store memo[n] and return
    memo[n] = 1 + res
    return memo[n]

def getMinSteps(n):

    memo = [0 for i in range(n+1)]

    # initialize memoized array
    for i in range(n+1):
        memo[i] = -1

    return getMinStepsHelper(n, memo)
```

Time complexity has been reduced significantly to **O(n)** as there are only **(n+1)** unique iterations. Now, try to code the DP approach by yourself, and for the code, refer to the solution tab.

Problem Statement: Minimum Number of Squares

Given an integer N, find and return the count of minimum numbers, the sum of whose squares is equal to N.

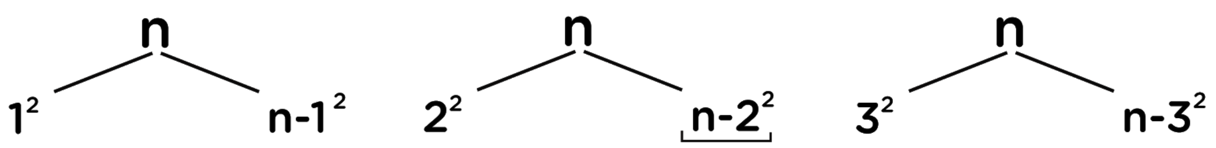
That is, if N is 4, then we can represent it as : {1^2 + 1^2 + 1^2 + 1^2} and {2^2}. The output will be 1, as 1 is the minimum count of numbers required. (x^y represents x raised to the power y.)

Example: For n = 12, we have the following ways:

- 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1
- 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 2^2
- 1^1 + 1^1 + 1^1 + 1^1 + 2^2 + 2^2
- 2^2 + 2^2 + 2^2

Hence, the minimum count is obtained from the 4-th option. Therefore, the answer is equal to 3.

Approach: First-of-all, we need to think about breaking the problems into two parts, one of which will be handled by recursion and the other one will be handled by us(smaller sub-problem). We can break the problem as follows:



And so on...

- In the above figure, it is clear that in the left subtree we are making ourselves try over a variety of values that can be included as a part of our solution.
- The right subtree's calculation will be done by recursion.

- Hence, we will just handle the **i^2** part, and **$(n-i^2)$** will be handled by recursion.
- By now, we have got ourselves an idea of solving this problem, the only thinking left is the loop's range on which we will be iterating, i.e., the values of **i** for which we will be deciding to consider while solving or not.
- As the maximum value up to which i can be pushed, to reach **n** is **$\sqrt{n}$** as **$(\sqrt{n}*\sqrt{n} = n)$.** Hence, we will be iterating over the range **(1 to $\sqrt{n}$)** and do consider each possible way by sending **$(n-i^2)$** over the recursion.
- This way we will get different subsequences and as per the question, we will simply return the minimum out of it.

This problem is left for you to try out using all the three approaches and for code, refer to the solution tab of the same.

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
```

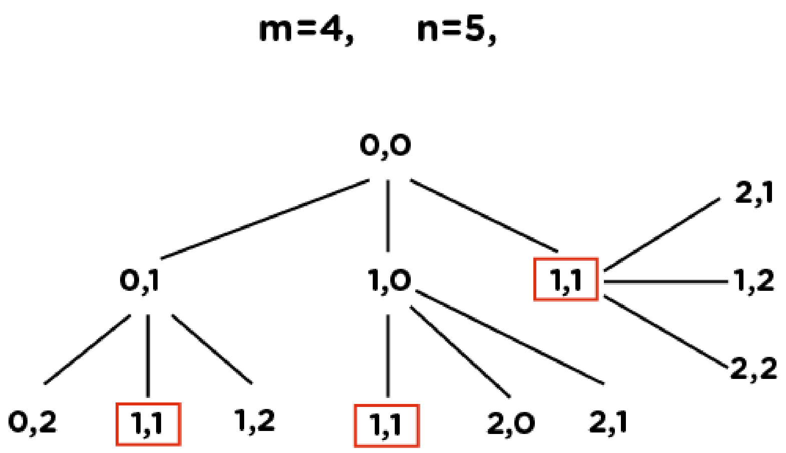
The path that should be followed is **3 -> 1 -> 8 -> 1**. Hence the output is **13**.

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:

```python
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:

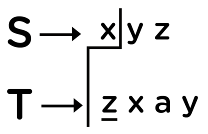
```
s1 = "x|yzar"
s2 = "x|qwea"
```

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.

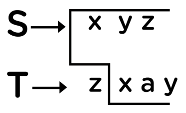
For example:

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

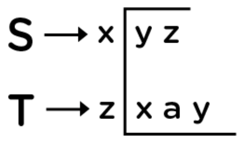
We can see that their first characters do not match so that we can call recursion over it in either of the following ways:



A =



B =



C=

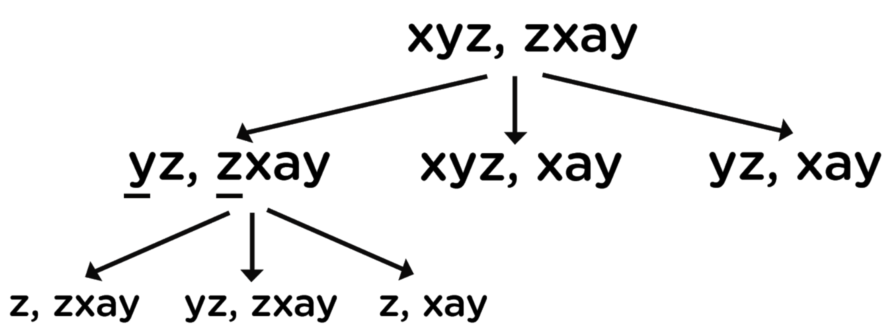
Finally, our answer will be:

```
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));
```

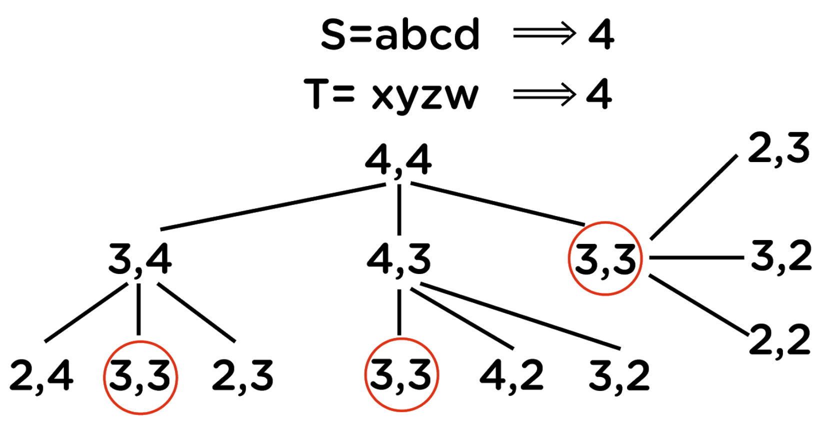
If we dry run this over the example: `s = "xyz"` and `t = "zxay"`, it will look something like below:



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:

```python
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:

```python
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