**Dynamic Programming**

Table of Contents

[**Theory** 2](#_Toc168908882)

[What is Dynamic Programming? 2](#_Toc168908883)

[**Sample Code** 2](#_Toc168908884)

[LEVEL 1: **EASY** 4](#_Toc168908885)

[1. Climbing Stairs 4](#_Toc168908886)

[2. House Robber 4](#_Toc168908887)

[3. Min cost climbing stairs 4](#_Toc168908888)

[4. Reach to one 4](#_Toc168908889)

[5. Coin change 4](#_Toc168908890)

[6. Longest Increasing Subsequence 4](#_Toc168908891)

[7. Stones 4](#_Toc168908892)

[8. Unique paths 4](#_Toc168908893)

[LEVEL 2: **Medium** 5](#_Toc168908894)

[1. Ninja training 5](#_Toc168908895)

[2. Longest common subsequence 5](#_Toc168908896)

[3. Longest repeating subsequence 5](#_Toc168908897)

[LEVEL 3: **Difficult** 6](#_Toc168908898)

[1. K-ordered longest common subsequence 6](#_Toc168908899)

[**SOLUTIONS:** 7](#_Toc168908900)

[**LEVEL 1:** 7](#_Toc168908901)

[**LEVEL 2:** 14](#_Toc168908902)

[**LEVEL 3:** 18](#_Toc168908903)

# **Theory**

## What is Dynamic Programming?

Dynamic programming (DP) is an optimization technique used to solve complex problems by breaking them down into simpler subproblems and storing the results of these subproblems to avoid redundant computations. It is particularly useful for problems with overlapping subproblems and optimal substructure properties.

**Key Concepts of Dynamic Programming:**

1. **Overlapping Subproblems**: Solving the same subproblems multiple times.
2. **Optimal Substructure**: The optimal solution to a problem can be constructed from optimal solutions of its subproblems.
3. **Memoization**: Storing results of subproblems to reuse them, thus saving computation time. Also called top-down approach.
4. **Tabulation**: Building a table in a bottom-up manner to store solutions of subproblems.

**Steps to approach Dynamic Programming problems:**

1. Find the recurrence relationship
2. Find the base case
3. Find way to store solutions of subproblems

# **Sample Code**

Fibonacci Series – Recursion:

def fib(n):

    if n<=1:

        return n

    else:

        return fib(n-1) + fib(n-2)

# fib : 0,1,1,2,3,5,8

print(fib(6)) #8

Here we see, our solution finding solution of same subproblems several time, which cause slowness. We can prevent this using DP, by saving solution of subproblems in an array.

Fibonacci Series –Top down:

Here we built solution from n to i, normally via recursion.

def fib\_top\_down(n, dp):

    if n<=1:

        return n

    if dp[n]!= -1:

        return dp[n]

    else:

        dp[n]= fib\_top\_down(n-1, dp) + fib\_top\_down(n-2, dp)

        return dp[n]

n=6

dp = [-1]\*(n+1)

print(fib\_top\_down(n, dp)) #8

Fibonacci Series – Bottom Up:

Here we built solution from i to n, normally via loop.

def fib\_bottom\_up(n):

    dp = [-1]\*(n+1)

    dp[0], dp[1] = 0, 1

    for i in range(2,n+1):

        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]

print(fib\_bottom\_up(6)) #8

Fibonacci Series – Bottom Up – Space Optimized:

In place of dp array we can use 2 variables for same purpose

def fib\_bottom\_up(n):

    prev = 1

    prev1 = 0

    curr = 0

    for i in range(2,n+1):

        curr = prev + prev1

        prev1 = prev

        prev = curr

    return prev

print(fib\_bottom\_up(6)) #8

# LEVEL 1: **EASY**

### Climbing Stairs

Link: <https://leetcode.com/problems/climbing-stairs/description/>

### House Robber

You are given money present in n adjacent houses, there is robber who wants to rob the houses. But he cannot rob from 2 adjacent houses. Find max loot of robber.

### Min cost climbing stairs

Link: <https://leetcode.com/problems/min-cost-climbing-stairs/>

### Reach to one

Given a number x, you can do 3 different operations on x: #1. Subtract 1 from it. #2 If it is divisible by 2, divide by 2. #3 If it is divisible by 3, divide by 3 .Find the minimum number of steps that it takes to get to 1 using only the above operations.

### Coin change

Link: <https://leetcode.com/problems/coin-change/description/>

### Longest Increasing Subsequence

Link: <https://leetcode.com/problems/longest-increasing-subsequence/description/>

### Stones

Link: <https://atcoder.jp/contests/dp/tasks/dp_k>

### Unique paths

Link: <https://leetcode.com/problems/unique-paths/description/>

# LEVEL 2: **Medium**

### Ninja training

Link: [https://leetcode.com/problems/climbing-stairs/description/](https://www.naukri.com/code360/problems/ninja-s-training_3621003)

### Longest common subsequence

Link: [https://leetcode.com/problems/climbing-stairs/description/](https://leetcode.com/problems/longest-common-subsequence/description/)

### Longest repeating subsequence

Link: [https://www.naukri.com/code360/problems/longest-repeating-subsequence](https://www.naukri.com/code360/problems/longest-repeating-subsequence_1118110?leftPanelTabValue=PROBLEM)

# LEVEL 3: **Difficult**

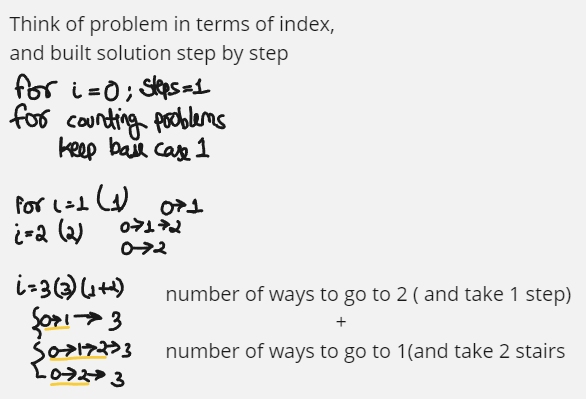
### K-ordered longest common subsequence

A k-ordered LCS is defined to be the LCS of two sequences if you are allowed to change at most k elements in the first sequence to any value you wish to. You are given 2 integer sequences and a number k. You can make max k changes in sequence 1 to get maximum LCS, find the max length of LCS.

# **SOLUTIONS:**

## **LEVEL 1:**

1. **Climbing Stairs**



# for n = number of ways to go to n-1 + number of ways to go to n-2

# 0->1 for making code easy

# 1->1

# 2 -> 2  (0->2 , 0->1->2)

class Solution:

    def climbStairs(self, n: int) -> int:

        dp = [1]\*(n+1)

        for i in range(2,n+1):

            dp[i] = dp[i-1] + dp[i-2]

        return dp[n]

class Solution:

    def climbStairs(self, n: int) -> int:

        def helper(n):

            if n==0: return 1

            if n==1: return 1

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

            dp[n] = helper(n-1) + helper(n-2)

            return dp[n]

        dp = [-1]\*(n+1)

        return helper(n)

1. **Loot house**

F[i] = max loot done till ith house, so F[i] = max ( arr[i] + F[i-2] , F[i-1] )

*#Loot HOUSE*

def lootBU(n,arr):

    dp=[0]\*(n)

    dp[0],dp[1] = arr[0],max(arr[0],arr[1])

    for i in range(2,n):

        dp[i] = max(arr[i]+dp[i-2] ,dp[i-1])

    print(dp)

    return dp[n-1]

arr = [6,2,3,9]

print(lootBU(len(arr),arr))

1. **Min cost climbing stairs**

Here dp[i] = cost of reaching at ith step.

#Bottom-up

class Solution:

    def minCostClimbingStairs(self, cost: List[int]) -> int:

        dp = [0]\*len(cost)

        dp[0] = cost[0]

        dp[1] = cost[1]

        for i in range(2,len(cost)):

            dp[i] = cost[i]+min(dp[i-1],dp[i-2])

        return min(dp[-1],dp[-2])

#Top-down

class Solution:

    def minCostClimbingStairs(self, cost: List[int]) -> int:

        n=len(cost)

        self.dp = [-1]\*n

        self.dp[0]=cost[0]

        self.dp[1]=cost[1]

        def helper(n):

            if self.dp[n]!=-1:

                return self.dp[n]

            else:

                self.dp[n] = cost[n] + min(helper(n-1),helper(n-2))

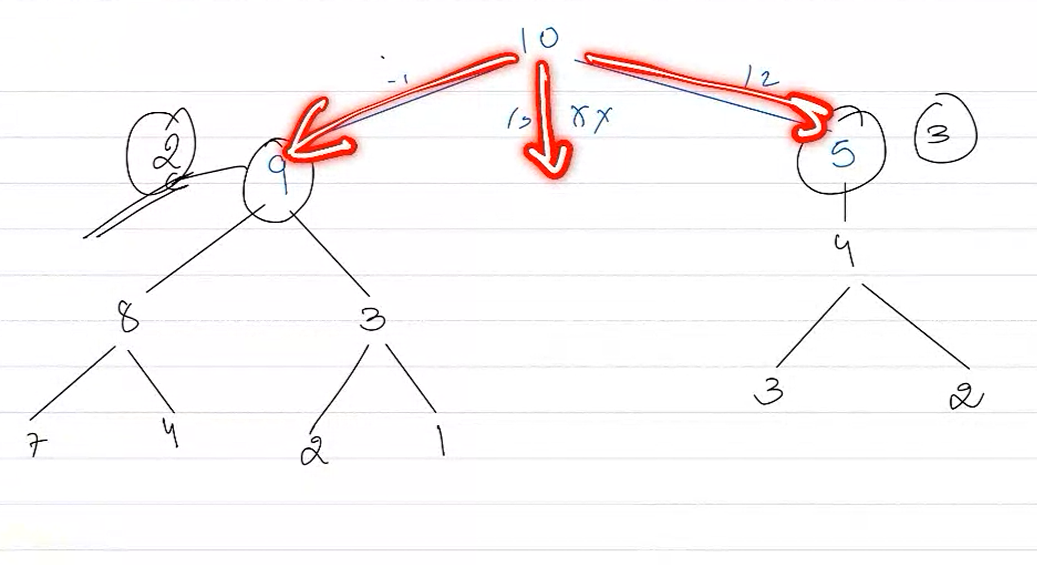
                return self.dp[n]

        helper(n-1)

        return min(self.dp[-1],self.dp[-2])

1. **Minimum steps to reach one**

If see local minimum, going from 10 to 5 is preferred than 10 to 9, however 5 takes more time to go to 1 than 9. Therefore in global way going to 9 is preferred.



*#Top down*

def minStepsToOneTD(n,dp):

    if n==1: return 0

    if n==2 or n==3: return 1

    if dp[n]!=0: return dp[n]

    div\_by\_3, div\_by\_2, less\_by\_1 = float('inf'),float('inf'),float('inf')

    if(n%3==0):

        div\_by\_3 = 1+minStepsToOneTD(n//3,dp)

    if(n%2==0):

        div\_by\_2 = 1+minStepsToOneTD(n//2,dp)

    less\_by\_1 = 1+minStepsToOneTD(n-1,dp)

    dp[n]=min(div\_by\_3, div\_by\_2, less\_by\_1)

    return dp[n]

n=7

dp=[0]\*(n+1)

print(minStepsToOneTD(n,dp))

*#Bottom Up*

def min\_steps\_to\_one(x):

    dp = [1]\*(x+1)

    dp[1]=0

    dp[2],dp[3]=1,1

    for i in range(4,x+1):

        dp[i]=1+min(dp[i-1], dp[i//2] if i%2==0 else x, dp[i//3] if i%3==0 else x)

    return dp[x]

print(min\_steps\_to\_one(10))

1. **Coin change**

#Top down

class Solution:

    def coinChange(self, coins: List[int], amount: int) -> int:

        def helper\_td(n , coins , dp):

            if n==0: return 0

            if dp[n]!=-1:return dp[n]

            temp = float('inf')

            for i in coins:

                if i <= n:

                    temp = min(temp,1+helper\_td(n-i,coins,dp))

            dp[n] = temp

            return dp[n]

        dp = [-1]\*(amount+1)

        ans=helper\_td(amount , coins, dp)

#Bottom up  (performs better both space and time wise)

class Solution:

    def coinChange(self, coins: List[int], amount: int) -> int:

        dp = [float('inf')]\*(amount+1)

        dp[0]=0

        for i in range(1,amount+1):

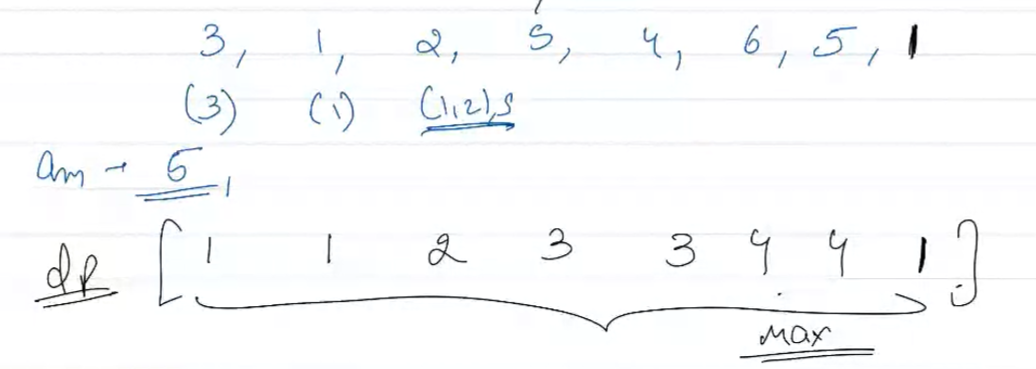
            for j in coins:

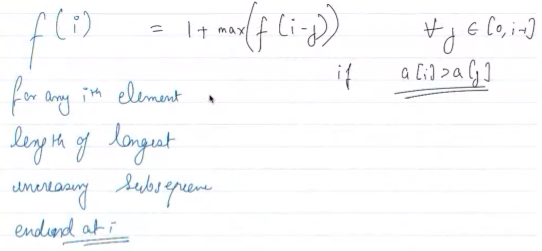
                if j<=i:

                    dp[i] = min(dp[i], 1+dp[i-j])

        return dp[amount] if dp[amount]!=float('inf') else -1

1. **Longest Increasing subsequence**





class Solution:

    def lengthOfLIS(self, nums: List[int]) -> int:

        n = len(nums)

        dp =[1]\*n

        for i in range(n):

            for j in range(i):

                if nums[i]>nums[j]:

                    dp[i] = max(dp[i] , 1+dp[j])

        return max(dp)

1. **Stone**

If k==0 : any player who reaches this state will lose

If k<min(a): here also if any player reaches this state will lose

So winning or losing depends on state and independent of who plays. So if state=k is winning state, player one wins else player 2.

State K is winning state if any state ( k-a[i] ) is losing state for all a[i] in array a. Meaning if first player can push second player to any losing state then first player can win. But if all ( k-a[i] ) states are winning states, then kth state is losing state.

n , k = map(int,input().split())

a = list(map(int,input().split()))

dp = [-1]\*(k+1)

#at state 0 all will lose

dp[0]=0

#if a=[2,3] so at 2 and 3, player takes all stone and next player won't have any stone to pick.

#so that will be winning state.

for i in range(1,k+1):

    flag=0

    for j in a:

        if j<=i and dp[i-j]==0: #can send next player to any losing state

            flag=1

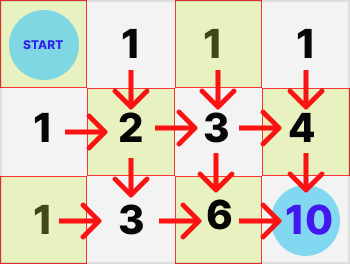
    dp[i]=flag

print("First" if dp[k]==1 else "Second")

1. **Unique Paths**

To go to m,n. we have 2 options. (1) go down from n-,m (2)go right from n,m-1.

By this logic. For all m=0 or n=0 there is only one way. So value=1.



class Solution:

    def uniquePaths(self, m: int, n: int) -> int:

        dp = [[1] \* n for i in range(m)]

        for i in range(1, m):

            for j in range(1, n):

                dp[i][j] = dp[i - 1][j] + dp[i][j - 1]

        return dp[m - 1][n - 1]

**Using combination**

For any given M x N grid, each unique path (no matter which one it is) requires you to move right from the starting point N - 1 times and move down from the starting point M - 1 times. Hence, regardless of the order you choose to move right or down, you need to make a total of (M - 1) + (N - 1) = M + N - 2 moves.

Then, out of the M + N - 2 moves, we need to select M - 1 moves to move right and the remaining N - 1 moves to move down. This essentially is why this problem boils down to combinatorics, because we need to calculate how many different ways we can select M - 1 moves from M + N - 2 moves (or equivalently, N - 1 moves from M + N - 2 moves).

class Solution:

    def uniquePaths(self, m: int, n: int) -> int:

        return math.comb(m+n-2, m-1)  # or math.comb(m+n-2, n-1)

## **LEVEL 2:**

1. **Ninja Training**

def ninjaTraining(n: int, points: List[List[int]]) -> int:

    # Write your code here.

    dp = [[0]\*3 for i in range(n)]

    dp[0][0] = points[0][0]

    dp[0][1] = points[0][1]

    dp[0][2] = points[0][2]

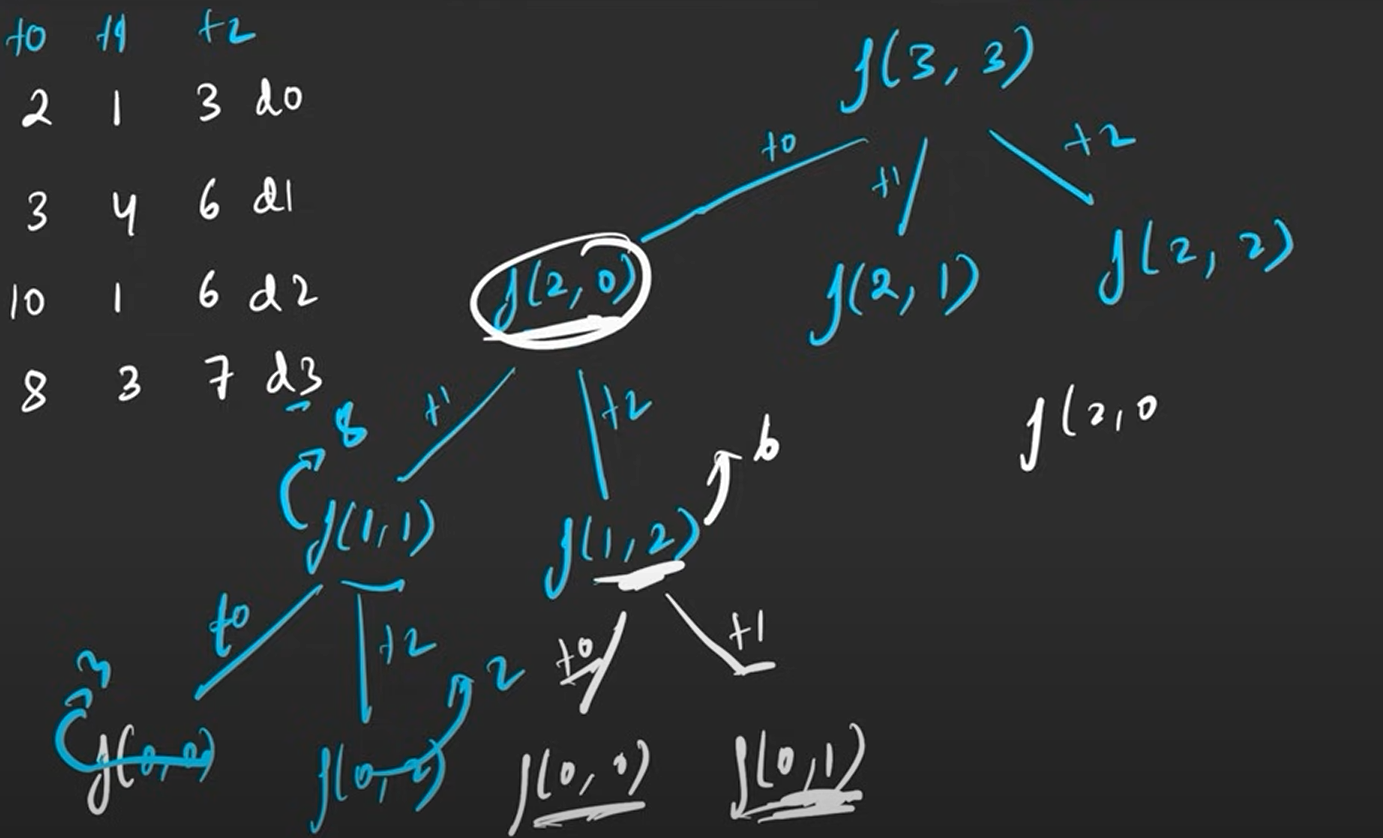
    for i in range(1,n):

        dp[i][0] = points[i][0] + max(dp[i-1][1] , dp[i-1][2])

        dp[i][1] = points[i][1] + max(dp[i-1][0] , dp[i-1][2])

        dp[i][2] = points[i][2] + max(dp[i-1][0] , dp[i-1][1])

    return max(dp[n-1])



Can further do space optimization as in each step we just need to get the previous state. So in place of saving whole dp, just save the previous state.

1. **Longest common subsequence**

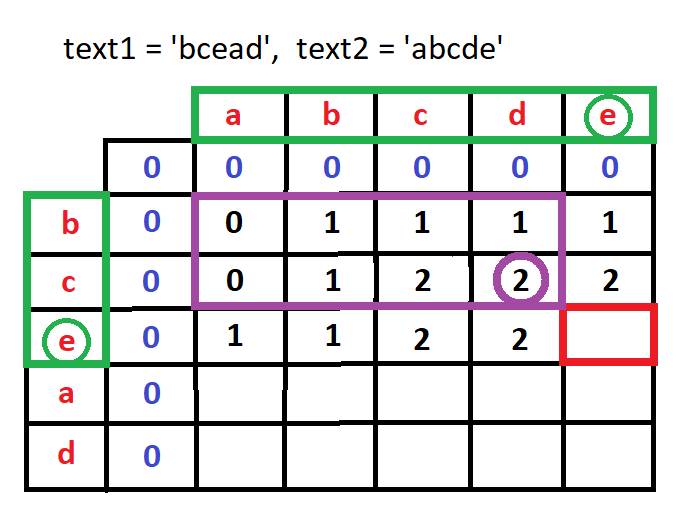
Here recursive relation is:

Eg:- st1= “abc” , st2=“adb” , I know st1[0] == st2[0],

So to get lcs, do **lcs(“abc”, “adb”) = 1 + lcs(“bc”,”db”)**

Eg2:- st1= “pqrs” , st2= “xqor” , I know st1[0] != st2[0]

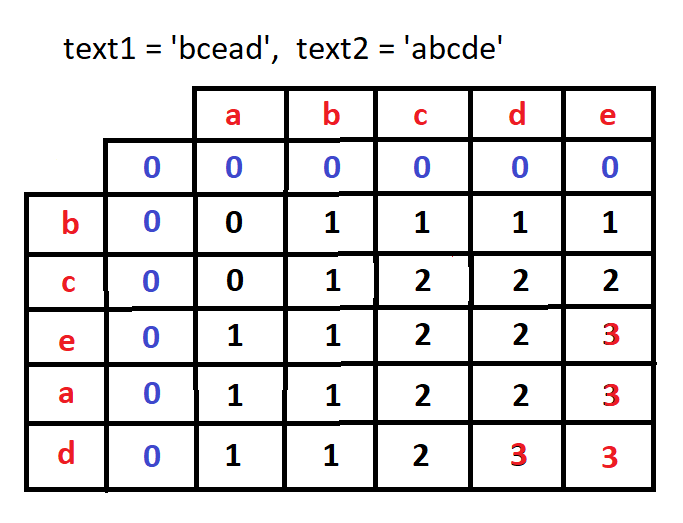
So to get lcs, do **lcs(“pqrs”, “xqor”) = max( lcs(“qrs” , “xqor”) , lcs(“pqrs”, “qor”) )**



For this red box:

Till now, text1=”bce” and text2=”abcde”

Now since last values are equal, it is equal to 1+lcs(“bc”, “abcd”) =3



class Solution:

    def longestCommonSubsequence(self, text1: str, text2: str) -> int:

        n,m=len(text1),len(text2)

        dp = [[0]\*(m+1) for i in range(n+1)] #m+1 rows and n+1 columns

        for i in range(1,n+1):

            for j in range(1,m+1):

                if text1[i-1]==text2[j-1]:

                    dp[i][j] = 1+dp[i-1][j-1]

                else:

                    dp[i][j] = max(dp[i-1][j] ,dp[i][j-1])

        return dp[n][m]

**## Demo run**

       ''  a  c  e

 ''    [0, 0, 0, 0]

  a    [0, 1, 1, 1]

  b    [0, 1, 1, 1]

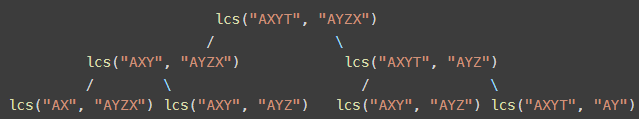
  c    [0, 1, 2, 2]

  d    [0, 1, 2, 2]

  e    [0, 1, 2, 3]

**For top down**

Will start looking from last values str[-1] and str[-2]



#Normal Recursion

class Solution:

    def longestCommonSubsequence(self, text1: str, text2: str) -> int:

        n = len(text1)

        m = len(text2)

        def helper(i, j):

            if i==0 or j==0: return 0

            if text1[i-1] == text2[j-1]:

                return helper(i-1,j-1)+1

            else:

                return  max(helper(i-1, j), helper(i, j-1))

        return helper(n,m)

Top down approach

class Solution:

    def longestCommonSubsequence(self, text1: str, text2: str) -> int:

        n = len(text1)

        m = len(text2)

        def helper(i, j, dp):

            if i==0 or j==0: return 0

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

            if text1[i-1] == text2[j-1]:

                dp[i][j] = helper(i-1,j-1, dp)+1

            else:

                dp[i][j] =  max(helper(i-1, j, dp), helper(i, j-1, dp))

            return dp[i][j]

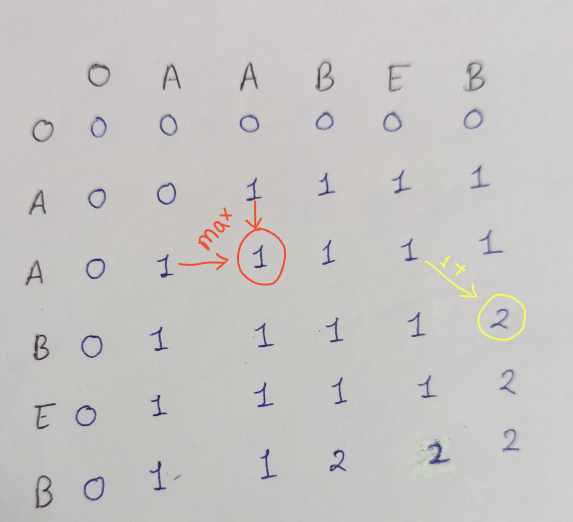
        dp = [[-1]\*(m+1) for i in range(n+1)]

        return helper(n,m,dp)

1. **Longest repeating subsequence**

Treat it as same as LCS on same string

But equal condition will have extra check that i != j,



def longestRepeatingSubsequence(st, n):

    dp = [[0]\*(n+1) for \_ in range(n+1)]

    for i in range(1,n+1):

        for j in range(1,n+1):

            if i!=j and st[i-1]==st[j-1]:

                dp[i][j] = 1 + dp[i-1][j-1]

            else:

                dp[i][j] = max(dp[i][j-1], dp[i-1][j])

    return dp[n][n]

## **LEVEL 3:**

1. **K-ordered LCS**

Same as like LCS, but one new case.

New case. Now we can change k values in seq1 to make it’s ith value equal to jth value of seq2.

So when seq1[i] != seq2[j], then 2 cases, (1) we can use k and make both values equal, (2) don’t use k and proceed as normal. Final ans is maximum of both.

def KOrderedLCS(seq1, seq2, k):

    def helper(i, j, k, dp):

        if i==0 or j==0: return 0

        if dp[i][j][k]!=-1:

            return dp[i][j][k]

        if seq1[i-1] == seq2[j-1]:

            dp[i][j][k] = 1+ helper(i-1,j-1,k,dp)

        else:

            if k>0:

                #we replacing a value in seq1,so will act as seq1[i]==seq2[j]

                temp = 1+ helper(i-1 ,j-1 , k-1, dp)

                #Case we not replacing any value

                temp2 =  max(helper(i-1,j ,k ,dp), helper(i, j-1, k, dp))

                dp[i][j][k] = max(temp, temp2)

            else:

                dp[i][j][k] = max(helper(i-1, j, k, dp), helper(i, j-1, k, dp))

        return dp[i][j][k]

    n,m=len(seq1),len(seq2)

    # dp of size n\*m\*k

    dp = [[[-1 for \_ in range(k+1)] for \_ in range(m+1)] for \_ in range(n+1)]

    return helper(n,m,k,dp)

seq1 = [1, 2, 3, 4, 5]

seq2 = [5, 3, 1, 4, 2]

k=1

print(KOrderedLCS(seq1, seq2, k))

**## Demo run**

eg:-

seq1 = [1, 2, 3, 4, 5]

seq2 = [5, 3, 1, 4, 2]

k = 1

Ans = 3

You can change the first element of the first sequence to 5 to get the LCS comprising of the sequence (5, 3, 4)

Bottom Up

def KOrderedLCS(seq1, seq2, k):

    n, m = len(seq1), len(seq2)

*# Initialize the dp table with zero values*

    dp = [[[0 for \_ in range(k+1)] for \_ in range(m+1)] for \_ in range(n+1)]

*# Iterate through all positions of seq1 and seq2*

    for i in range(1, n+1):

        for j in range(1, m+1):

            for z in range(k+1):

                if seq1[i-1] == seq2[j-1]:

                    dp[i][j][z] = dp[i-1][j-1][z] + 1

                else:

                    if z > 0:

*# Option 1: Replace seq1[i-1] with seq2[j-1]*

                        replace = dp[i-1][j-1][z-1] + 1

*# Option 2: Do not replace, just move in one of the sequences*

                        dont\_replace = max(dp[i-1][j][z], dp[i][j-1][z])

                        dp[i][j][z] = max(replace, dont\_replace)

                    else:

*# When no replacements are allowed*

                        dp[i][j][z] = max(dp[i-1][j][z], dp[i][j-1][z])

    return dp[n][m][k]

*# Read input*

seq1 = [1, 2, 3, 4, 5]

seq2 = [5, 3, 1, 4, 2]

k=1

*# Print the result*

print(KOrderedLCS(seq1, seq2, k))