**Dynamic Programming**

Table of Contents

[**Theory** 3](#_Toc207350270)

[What is Dynamic Programming? 3](#_Toc207350271)

[**Sample Code** 3](#_Toc207350272)

[LEVEL 1: **Linear DP** 5](#_Toc207350273)

[1. Climbing Stairs 5](#_Toc207350274)

[2. House Robber 5](#_Toc207350275)

[3. House Robber 2 5](#_Toc207350276)

[4. Jump Game 5](#_Toc207350277)

[5. Stones 5](#_Toc207350278)

[LEVEL 2: **Multi-Dimensional DP** 6](#_Toc207350279)

[1. Vacation 6](#_Toc207350280)

[2. Find the Maximum Length of Valid Subsequence 2 6](#_Toc207350281)

[LEVEL 3: **String DP** 7](#_Toc207350282)

[1. Longest Increasing Subsequence(LIS) 7](#_Toc207350283)

[2. Longest common subsequence 7](#_Toc207350284)

[3. Print Longest common subsequence 7](#_Toc207350285)

[4. Longest common substring 7](#_Toc207350286)

[5. Minimum number of deletions and insertions 7](#_Toc207350287)

[6. Longest Palindromic Subsequence 7](#_Toc207350288)

[7. Minimum Insertion Steps to Make a String Palindrome 7](#_Toc207350289)

[8. Shortest Common Supersequence 7](#_Toc207350290)

[LEVEL 4: **DP on Grid** 8](#_Toc207350291)

[1. Unique Paths 8](#_Toc207350292)

[2. Minimum path sum 8](#_Toc207350293)

[3. Grid 1 8](#_Toc207350294)

[Type 5: **Knapsack** 9](#_Toc207350295)

[1. Knapsack-1 9](#_Toc207350296)

[2. Coin Change 9](#_Toc207350297)

[3. Subset Sum Problem 9](#_Toc207350298)

[4. Partition Equal Subset Sum 9](#_Toc207350299)

[5. Partition with Given Difference 9](#_Toc207350300)

[Type 6: **Kadane’s Algorithm** 10](#_Toc207350301)

[1. Maximum Subarray 10](#_Toc207350302)

[2. Best time to Buy or Sell Stock 10](#_Toc207350303)

[3. Best time to Buy or Sell Stock II 10](#_Toc207350304)

[Type 7: **DP on Graph** 11](#_Toc207350305)

[1. Longest Path 11](#_Toc207350306)

[LEVEL 8: **Miscellaneous** 12](#_Toc207350307)

[1. K-ordered longest common subsequence 12](#_Toc207350308)

[**SOLUTIONS:** 13](#_Toc207350309)

[**Type 1: Linear DP** 13](#_Toc207350310)

[**TYPE 2: Multi-Dimensional DP** 16](#_Toc207350311)

[**TYPE 3: String DP** 19](#_Toc207350312)

[**TYPE 4: DP on Grids** 27](#_Toc207350313)

[**TYPE 5: Knapsack** 30](#_Toc207350314)

[**TYPE 6: Kadane’s Algorithm** 39](#_Toc207350315)

[**Type 7: DP on graph** 41](#_Toc207350316)

[**Type 8: Miscellaneous** 42](#_Toc207350317)

# **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 code is 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: **Linear DP**

For such questions you need to find the repetitive part of solution and improve it by saving result in array or somewhere, classic example is Fibonacci series. Also use when there is multiple way to do something and we need optimal way, min-max scenarios.

### Climbing Stairs

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

### House Robber

Link: <https://leetcode.com/problems/house-robber/description/>

### House Robber 2

Link: <https://leetcode.com/problems/house-robber-ii/>

### Jump Game

Link: <https://leetcode.com/problems/jump-game/>

### Stones

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

# LEVEL 2: **Multi-Dimensional DP**

### Vacation

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

### Find the Maximum Length of Valid Subsequence 2

Link: <https://leetcode.com/problems/find-the-maximum-length-of-valid-subsequence-ii/>

# LEVEL 3: **String DP**

These problems focus on operations over strings, such as finding subsequence or transformations.

### Longest common subsequence

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

### Print Longest common subsequence

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

### Longest common substring

Link: <https://www.geeksforgeeks.org/problems/longest-common-substring1452/1>

### Minimum number of deletions and insertions

Link: <https://www.geeksforgeeks.org/problems/minimum-number-of-deletions-and-insertions0209/1>

### Longest Palindromic Subsequence

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

### Minimum Insertion Steps to Make a String Palindrome

Link: [https://leetcode.com/problems/minimum-insertion-steps-to-make-a-string-palindrome/](https://leetcode.com/problems/minimum-insertion-steps-to-make-a-string-palindrome/description/)

### Shortest Common Supersequence

Link: [https://leetcode.com/problems/minimum-insertion-steps-to-make-a-string-palindrome/](https://leetcode.com/problems/shortest-common-supersequence/)

# LEVEL 4: **DP on Grid**

These problems involve navigating a 2D grid, making optimal decisions at each cell.

### Unique Paths

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

### Minimum path sum

Link: <https://leetcode.com/problems/minimum-path-sum/>

### Grid 1

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

# Type 5: **Knapsack**

These problems revolve around selecting items with given weights and values to maximize or minimize a certain criterion.

### Knapsack-1

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

### Coin Change

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

### Subset Sum Problem

Link: <https://www.geeksforgeeks.org/problems/subset-sum-problem-1611555638/1>

### Partition Equal Subset Sum

Link: <https://leetcode.com/problems/partition-equal-subset-sum/description/>

### Partition with Given Difference

Link: <https://www.geeksforgeeks.org/problems/partitions-with-given-difference/1>

# Type 6: **Kadane’s Algorithm**

Kadane’s Algorithm is a dynamic programming approach used to solve the Maximum Subarray Sum Problem — i.e., to find the contiguous subarray within a 1D array of numbers that has the largest sum.

### Maximum Subarray

Link: <https://leetcode.com/problems/maximum-subarray/description/>

### Best time to Buy or Sell Stock

Link: <https://leetcode.com/problems/best-time-to-buy-and-sell-stock/description/>

### Best time to Buy or Sell Stock II

Link: <https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/>

# Type 7: **DP on Subsequence**

* We are not restricted to *contiguous* elements (unlike Kadane or sliding window).
* We choose a subsequence by skipping or including elements while keeping order.
* State = "best solution ending at index i considering elements before it."

### Longest Increasing Subsequence(LIS)

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

# LEVEL 9: **Miscellaneous**

### 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.

1. Palindrome Partitioning 2

Link: <https://leetcode.com/problems/palindrome-partitioning-ii/description/>

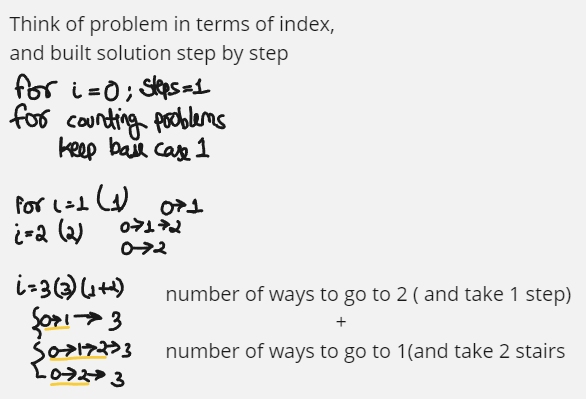
1. Longest Path

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

# **SOLUTIONS:**

## **Type 1: Linear DP**

1. **Climbing Stairs**



# for n = number of ways to go to n-1 + number of ways to go to n-2 (Fibonacci series)

# 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. **House Robber**

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. **House Robber 2**

Just like previous question,

Only difference is first and last house can not be looted together. So we will run the code to both subarrays, one without first house and one without last house and will return maximum from wither case.

def helper(*house*):

    def maxLoot(*n*,*dp*):

*if* *n*<0: *return* 0

*if* *n*==0: *return* *house*[0]

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

*dp*[*n*] = max(maxLoot(*n*-1,*dp*) , maxLoot(*n*-2,*dp*)+*house*[*n*])

*return* *dp*[*n*]

    n = len(*house*)

    dp=[-1]\*n

*return* maxLoot(n-1,dp)

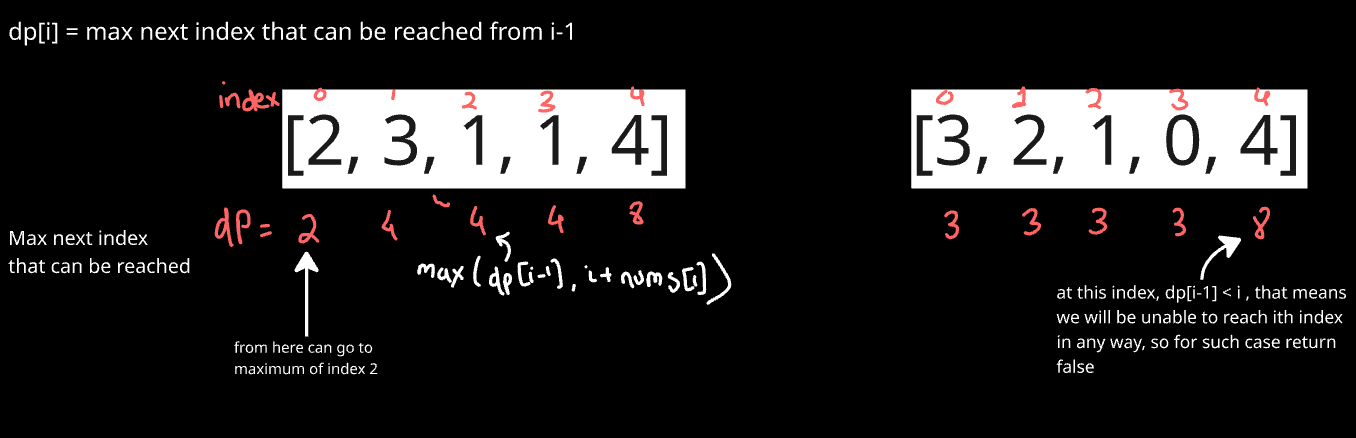
def houseRobber(*valueInHouse*):

*if* len(*valueInHouse*)==1:

*return* *valueInHouse*[0]

*return* max(helper(*valueInHouse*[:-1]) , helper(*valueInHouse*[1:]))

1. **Jump Game**



class Solution:

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

        n = len(nums)

        dp=[0]\*(n)

        dp[0] = nums[0]

        for i in range(1,n):

            if dp[i-1]<i:

                return False

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

        return True

**5. 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)

dp[0]=0 #at state 0 all will lose

#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

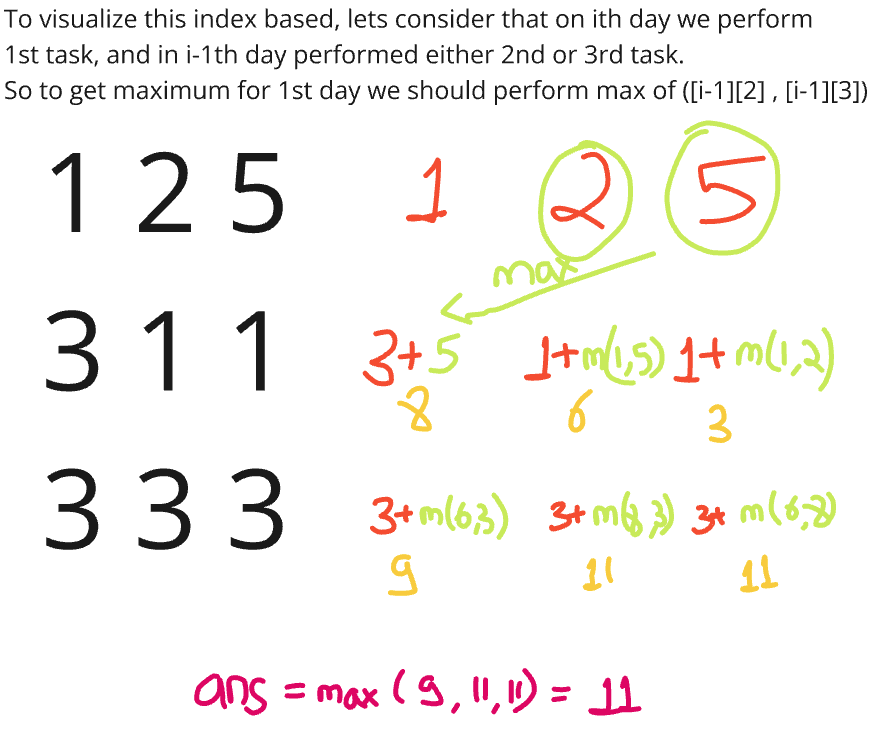
            flag=1

    dp[i]=flag

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

## **TYPE 2: Multi-Dimensional DP**

1. **Vacation**



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.

def vacation(N, points):

    dp = [[0]\*3 for \_ 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])

N = int(input())

points=[]

for \_ in range(N):

    points.append(list(map(int,input().split())))

print(vacation(N,points))

**2. Find the Maximum Length of Valid Subsequence 2**

**Concept:** we want

(a+b)%k = (b+c)%k …. == a%k = c%k (means alternating remainders should be same)

We have 2 valid patterns:

* All elements have same remainder : r,r,r,r,r….
* Two remainders that alternate : r1, r2, r1, r2, r1, r2, r1 ……

Thus can use DP : **dp[i][j] =** The length of longest valid subsequence where second-to-last element had remainder I and last element had remainer j.

As we iterate through nums, we’ll update this table.

class Solution:

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

        dp = [[0] \* k for \_ in range(k)]

        res = 0

        for num in nums:

            num %= k

            for prev in range(k):

                dp[prev][num] = dp[num][prev] + 1

                res = max(res, dp[prev][num])

        return res

**Explanation:**

dp[prev][num] = dp[num][prev] + 1

**num:** remainder of current number we’re adding

**prev:** a possible remainder for element before the current number

The new length of a subsequence ending in (prev,num) is 1+ the old length of a sequence that was ending in (num,prev).

Since alternating pattern, so for remainder (r1,r2) previous should be (r2,r1)

Time: O(n\*k)

Space: O(k\*\*2)

## **TYPE 3: String DP**

**1. Longest common subsequence**

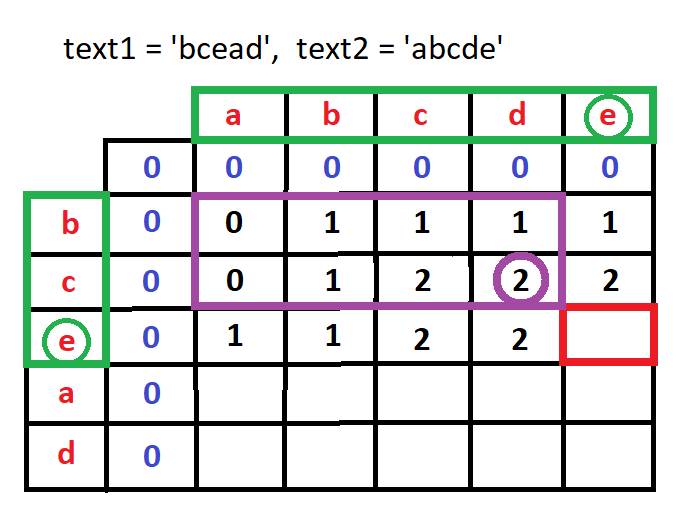
Here recursive relation is:

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]

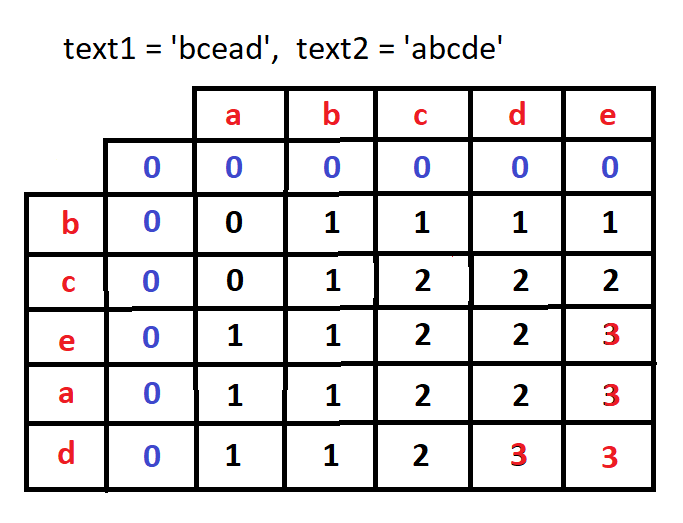
So to get lcs, do **lcs(“pqrs”, “xqor”) = max( lcs(“qrs” , “xqor”) , lcs(“pqrs”, “qor”) )**



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]

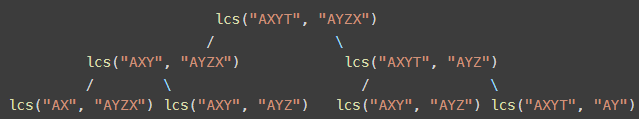
  b    [0, 1, 1, 1]

  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)

#Recursion + memorization = DP

class Solution:

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

        n, m = len(text1), len(text2)

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

        def helper(i,j):

            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] =  1+helper(i-1,j-1)

            else:

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

            return dp[i][j]

        return helper(n,m)

**3. Print Longest Common Subsequence**

Here will keep code same as LCS, now length of LCS will be at dp[n][m], from that place we will iterate back to top checking from where the value really got increased. If s1[i-1] = s2[j-1] , then it will be part of ans, else check up and left, from where value got increased and iterate till top.

def lcs(s1, s2):

    n , m = len(s1) , len(s2)

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

    for i in range(1, n+1):

        for j in range(1, m+1):

            if s1[i-1] == s2[j-1]:

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

            else:

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

    #now maximum value is present at dp[n][m]

    #will iterate form n,m to 0,0 and check at what places value of dp[i][j] increased

    i, j = n, m

    ans=''

    while(i>0 and j>0):

        if s1[i-1] == s2[j-1]:

            ans += s1[i-1]

            i-=1

            j-=1

        elif dp[i-1][j]>dp[i][j-1]:

            i-=1

        else:

            j-=1

    return ans[::-1] # since iterated from bottom to top, got ans in reversed order

s1 = input()

s2 = input()

print(lcs(s1, s2))

**4. Longest Common Substring**

A **subsequence** is a sequence derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

While a **substring** must be continues.

For substring, we just need few modifications in LCS:

1. If s1[i] != s2[j], at this point substring chain is broken, so make dp[i][j] as 0
2. Largest substring can be present at any point, even in mid of 2 substring. So ans is not just present on dp[n][m]. But it is max value present in whole dp. So keep track of that.

class Solution:

    def longestCommonSubstr(self, s1, s2):

        # code here

        n,m = len(s1), len(s2)

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

        mx=0

        for i in range(1,n+1):

            for j in range(1,m+1):

                if s1[i-1]==s2[j-1]:

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

                    mx = max(mx,dp[i][j])

                else:

                    dp[i][j]=0

        return mx

**5. Minimum number of deletions and insertions**

For heap to become pea

Heap 🡪 ea 🡪 pea

Basically s1 ----deleletion-----> LCS ------insertion---->s2

No of deletion: len(s1)-len(lcs)

No of insertion: len(s2)-len(lcs)

class Solution:

    def minOperations(self, s1, s2):

        # code here

        n,m = len(s1),len(s2)

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

        def rec(n,m,dp):

            if n==0 or m==0:

                return 0

            if dp[n][m]!=-1:

                return dp[n][m]

            if s1[n-1]==s2[m-1]:

                dp[n][m] = 1+ rec(n-1,m-1,dp)

            else:

                dp[n][m] = max(rec(n-1,m,dp), rec(n,m-1,dp))

            return dp[n][m]

        lcs = rec(n,m,dp)

        ans = (n-lcs) + (m-lcs)

        return ans

**6. Longest Palindromic Subsequence**

#can be considered equivalent to finding LCS of string with it's reverse

class Solution:

    def longestPalindromeSubseq(self, s: str) -> int:

        def helper(i,j):

            if i==0 or j==0:

                return 0

            if dp[i][j]!=-1:

                return dp[i][j]

            if s1[i-1]==s2[j-1]:

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

            else:

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

            return dp[i][j]

        s1 = s

        s2 = s[::-1]  #reverse of original string

        n = len(s)

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

        helper(n,n)

        return dp[n][n]

**7. Minimum Insertion Steps to Make a String Palindrome**

First we will get the length of longest palindrome possible from given string. This can be done by finding LCS between string and it’s reverse.

**Eg**: for aebchba: longest palindrom is abcba or abhba

We are left with e,h or e,c, so if asked how many deletion or insertion(in this question) required to make string palindrome, it will be 2.

After 2 insertion it becomes: aebchcbea or aebhchbea

class Solution:

    def minInsertions(self, s: str) -> int:

        s1 = s

        s2 = s[::-1]

        n = len(s)

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

        for i in range(1,n+1):

            for j in range(1,n+1):

                if s1[i-1]==s2[j-1]:

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

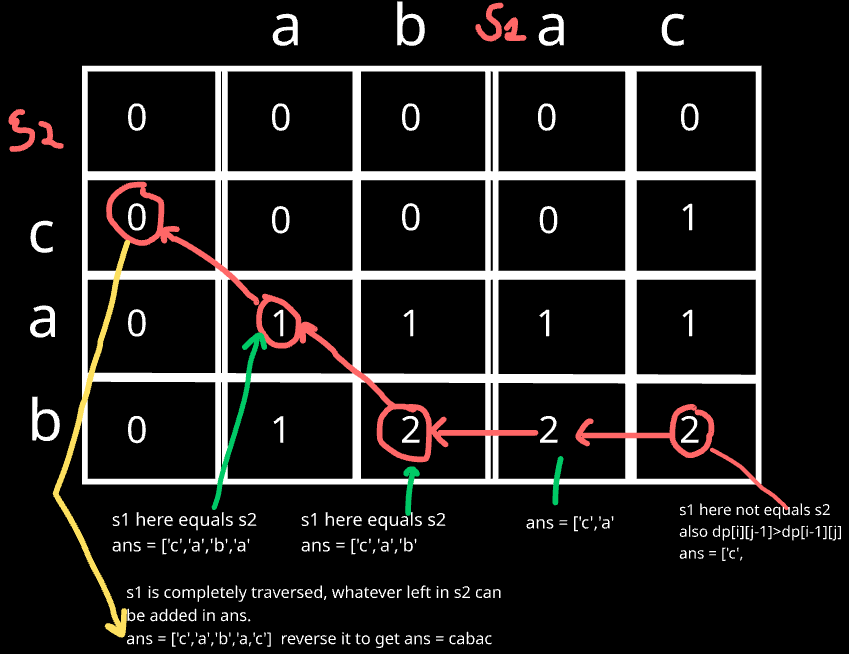
                else:

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

        lcs\_length = dp[n][n]

        return n-lcs\_length

**8. Shortest Common Supersequence**



Just similar to printing the LCS.

Just change logic when values are not equal, take the value from element whose value is more.

class Solution:

    def shortestCommonSupersequence(self, str1: str, str2: str) -> str:

        # Step 1: Find the longest common subsequence using dynamic programming

        m, n = len(str1), len(str2)

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

        # Fill the dp table

        for i in range(1, m + 1):

            for j in range(1, n + 1):

                if str1[i-1] == str2[j-1]:

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

                else:

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

        # Step 2: Construct the shortest common supersequence

        # Start from the bottom right of the dp table

        i, j = m, n

        result = []

        while i > 0 and j > 0:

            if str1[i-1] == str2[j-1]:

                # If the characters are the same, add it once

                result.append(str1[i-1])

                i -= 1

                j -= 1

            elif dp[i-1][j] > dp[i][j-1]:

                # If coming from top has higher value, take character from str1

                result.append(str1[i-1])

                i -= 1

            else:

                # Otherwise, take character from str2

                result.append(str2[j-1])

                j -= 1

        # Add remaining characters from str1 (if any)

        while i > 0:

            result.append(str1[i-1])

            i -= 1

        # Add remaining characters from str2 (if any)

        while j > 0:

            result.append(str2[j-1])

            j -= 1

        # Reverse the result to get the final supersequence

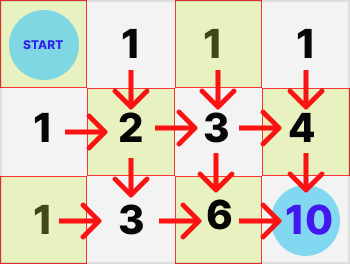
        return ''.join(result[::-1])

## **TYPE 4: DP on Grids**

1. **Unique Paths**

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

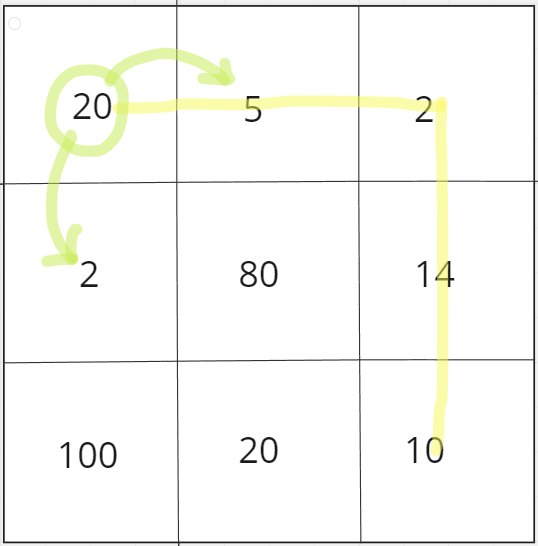
Then, out of the M + N - 2 moves, we need to select M - 1 moves to move right and the remaining N - 1 moves to move down. This essentially is why this problem boils down to combinatorics, because we need to calculate how many different ways we can select M - 1 moves from M + N - 2 moves (or equivalently,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

**2. Minimum Path Sum**



If we use greedy strategy here. Then we must be going to 2, instead of 5.

But 5 gives us best results globally. So for this case as we have global consideration to get best results, we use dp to check all possible paths and get best results.

class Solution:

    def minPathSum(self, grid: List[List[int]]) -> int:

        n = len(grid)

        m = len(grid[0])

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

        for i in range(n):

            for j in range(m):

                if i==0 and j==0:

                    dp[i][j] = grid[i][j]

                else:

                    up,left = float('inf'),float('inf')

                    if i>0:

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

                    if j>0:

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

                    dp[i][j] = min(up,left)

        return dp[n-1][m-1]

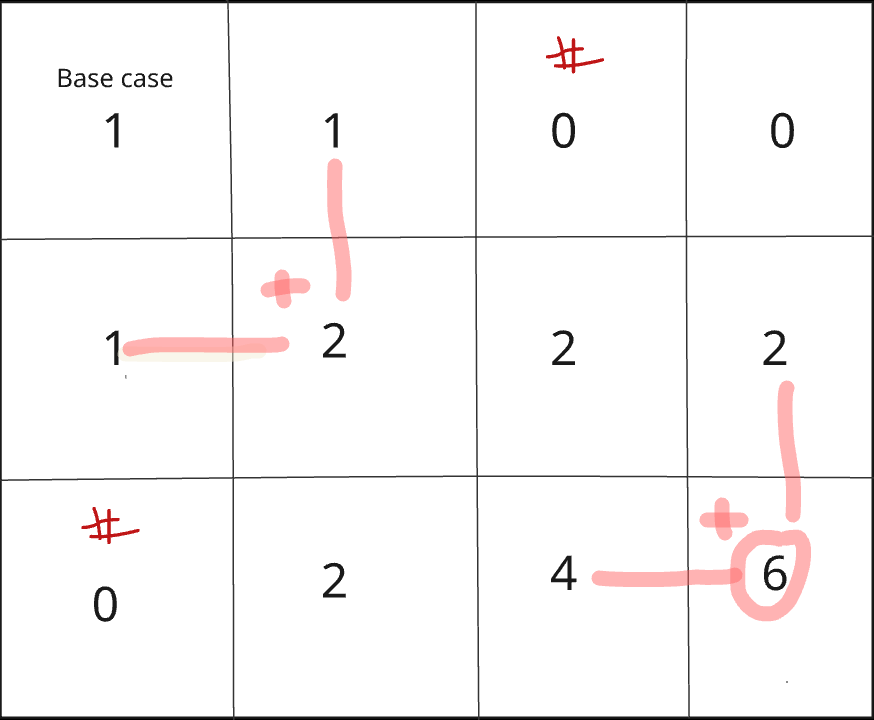
**3. Grid 1**

You can **only** come into a cell (i, j) from either:

* The **cell above**: (i-1, j)
* The **cell to the left**: (i, j-1)

If a cell is **blocked (#)**, you cannot reach it — so paths to it = 0.

**Base Case:** If you're at the starting cell (0,0) and it's not blocked, you have **exactly 1 path** (just staying there).



mod = 10\*\*9 + 7

def grid\_paths(h, w, dp, grid):

    for i in range(h):

        for j in range(w):

            if grid[i][j] == '#':

                dp[i][j] = 0

            else:

                if i > 0:

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

                if j > 0:

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

                dp[i][j] %= mod

    return dp[h-1][w-1]

h, w = map(int, input().split())

grid = [input() for \_ in range(h)]

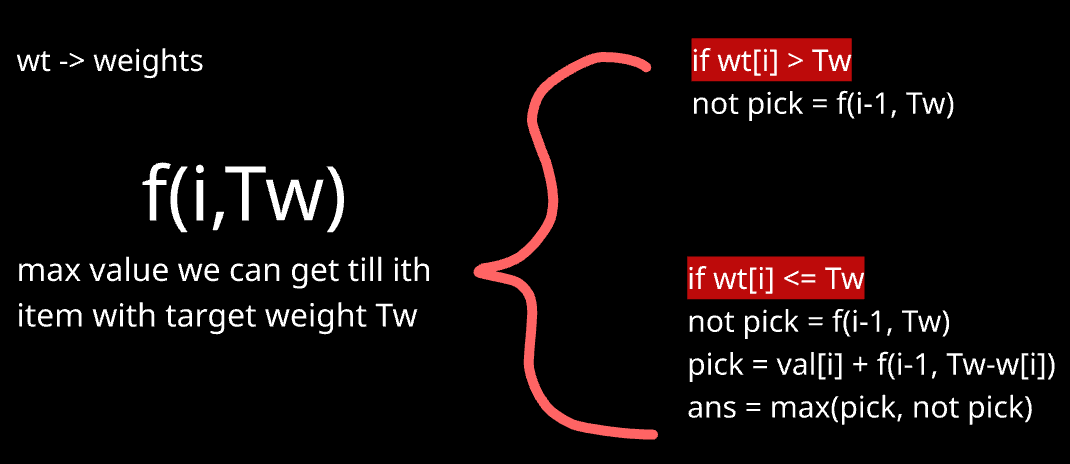
dp = [[0] \* w for \_ in range(h)]

dp[0][0] = 1  # Start cell (Base case)

print(grid\_paths(h, w, dp, grid))

## **TYPE 5: Knapsack**

1. **Knapsack-1**



iin=lambda:map(int,input().split())

def knapsack(i, Tw, dp, wt, val):

    if i == 0: # for base case: if i=0 and you can pick a object, so must pick

        if wt[0] <= Tw:

            return val[0]

        return 0

    if dp[i][Tw] != -1:

        return dp[i][Tw]

    not\_pick = knapsack(i-1, Tw, dp, wt, val)

    pick = 0

    if wt[i] <= Tw:

        pick = val[i] + knapsack(i-1, Tw-wt[i], dp, wt, val)

    dp[i][Tw] = max(pick, not\_pick)

    return dp[i][Tw]

#main method

N,Tw=iin()

wt,val = [],[]

for \_ in range(N):

    w,v = iin()

    wt.append(w)

    val.append(v)

# Initialize dp with -1

dp = [[-1] \* (Tw + 1) for \_ in range(N)]

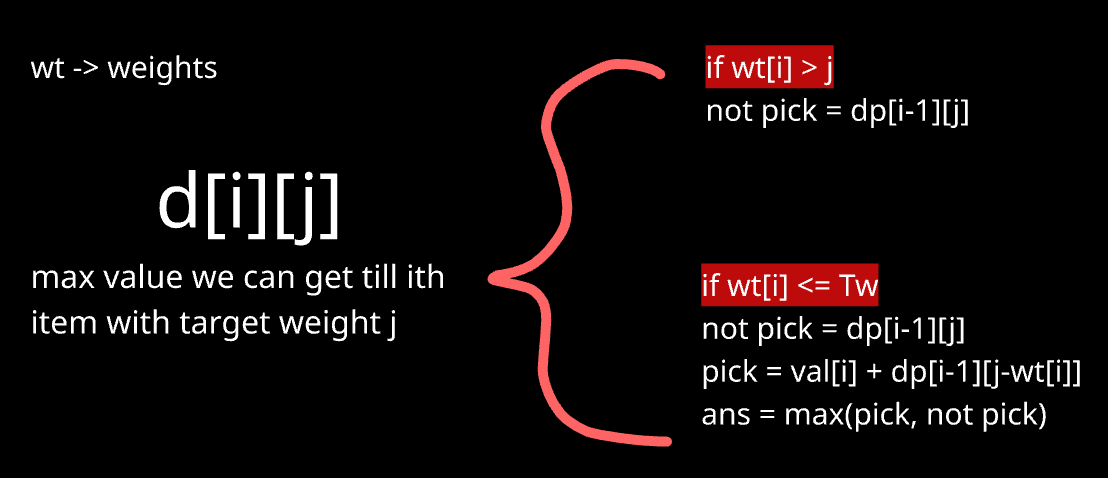
ans= knapsack(N-1,Tw,dp,wt,val)

print(ans)

Bottom up

Can’t do in 1D DP, as eg lets say object of weight 3 is present, Now for dp[6], dp[6]= take object of weight 3 + dp[6-3] . So in this case 3 is considered twice. So here use 2D dp, which make sure every object is considered only once. (This are called **0/1 Knapsack**)

In coin change multiple pick is allowed, so can use 1D DP.(Unbounded Knapsack)



iin=lambda:map(int,input().split())

#main method

N,Tw=iin()

wt,val = [],[]

for \_ in range(N):

    w,v = iin()

    wt.append(w)

    val.append(v)

# Initialize dp with -1

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

#fill row wise (0->tw+1)

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

    for j in range(1,Tw+1):

        if(wt[i-1]<=j):

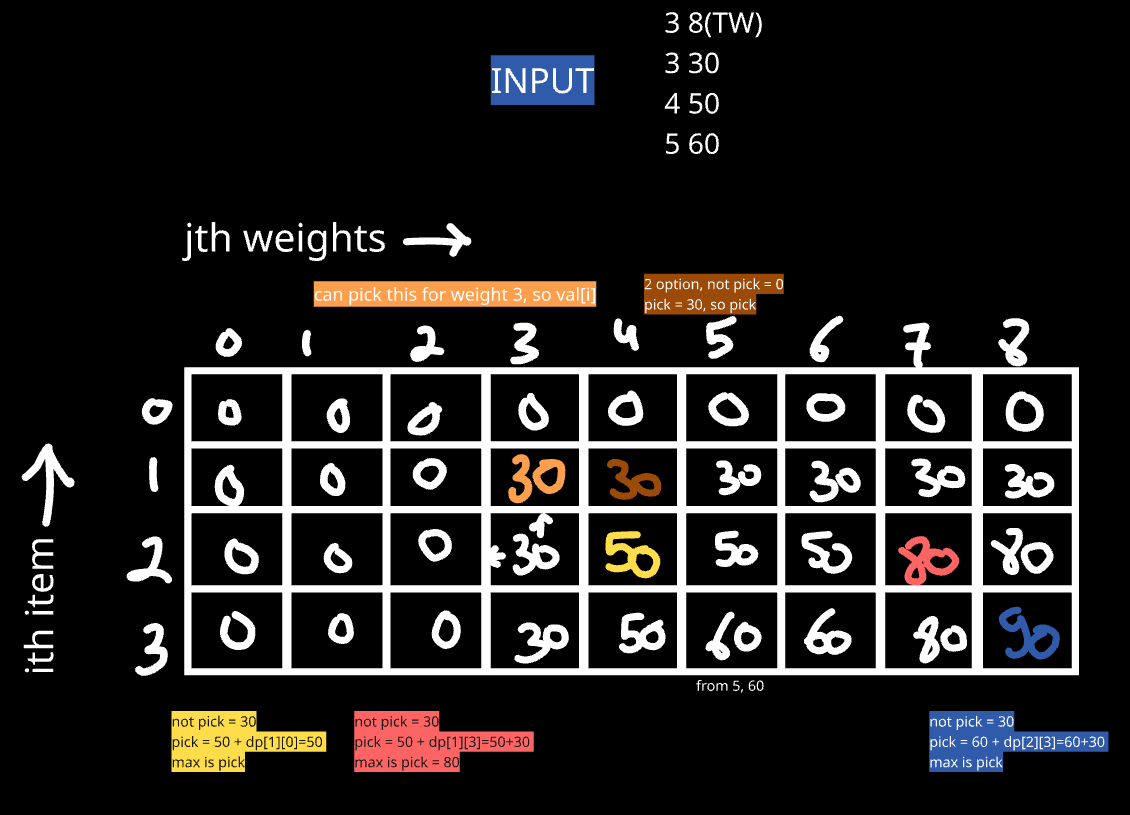
            dp[i][j]=max(dp[i-1][j], val[i-1] + dp[i-1][j-wt[i-1]])

        else:

            dp[i][j]=dp[i-1][j]

print(dp)

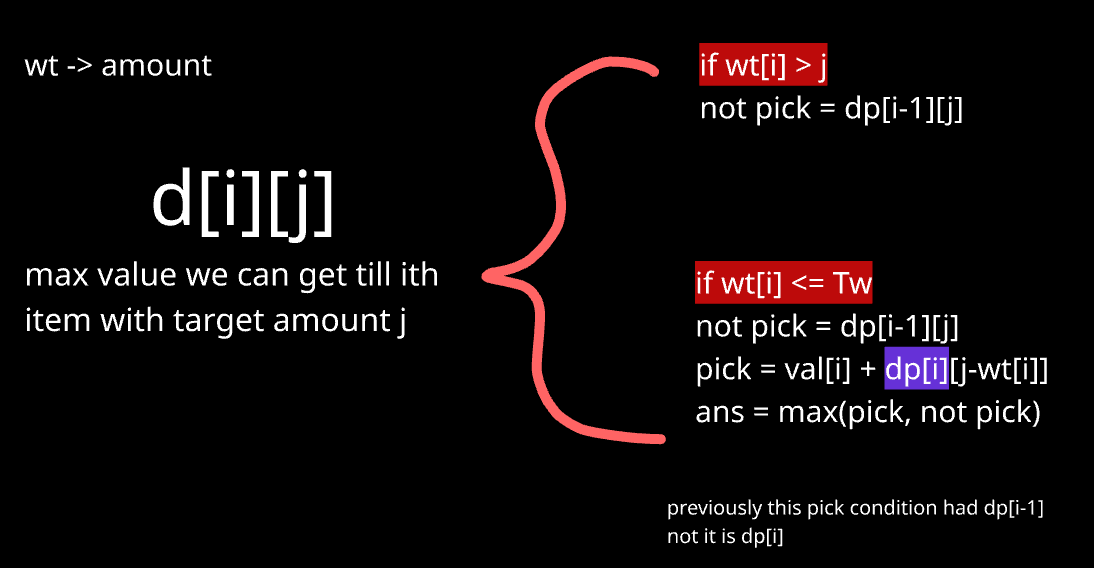
print(dp[N][Tw])



Here using this approach, as one weight can be picked only once.

1. **Coin Change**

**Approach 1 :** Just like previous one, but there since same value can’t be pick twice, so in pick case we were doing dp[i-1], but here same I can be picked again, so in pick condition don’t change to i-1 and check again for ith (This are called **unbounded Knapsack**)



For base condition

Since we iterate row wise, we have to fill 0th row for base condition. now if we have target and only coins[0] as coin, so maximum coins we can put is target//coins[0]. So will do that.

class Solution:

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

        n = len(coins)

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

        for t in range(amount+1):

            if t%coins[0]==0 :

                dp[0][t] = t//coins[0]

            else:

                dp[0][t] = float('inf')

        #moving row wise

        for i in range(1,n):

            for j in range(amount+1):

                not\_pick = dp[i-1][j]

                pick = float('inf')

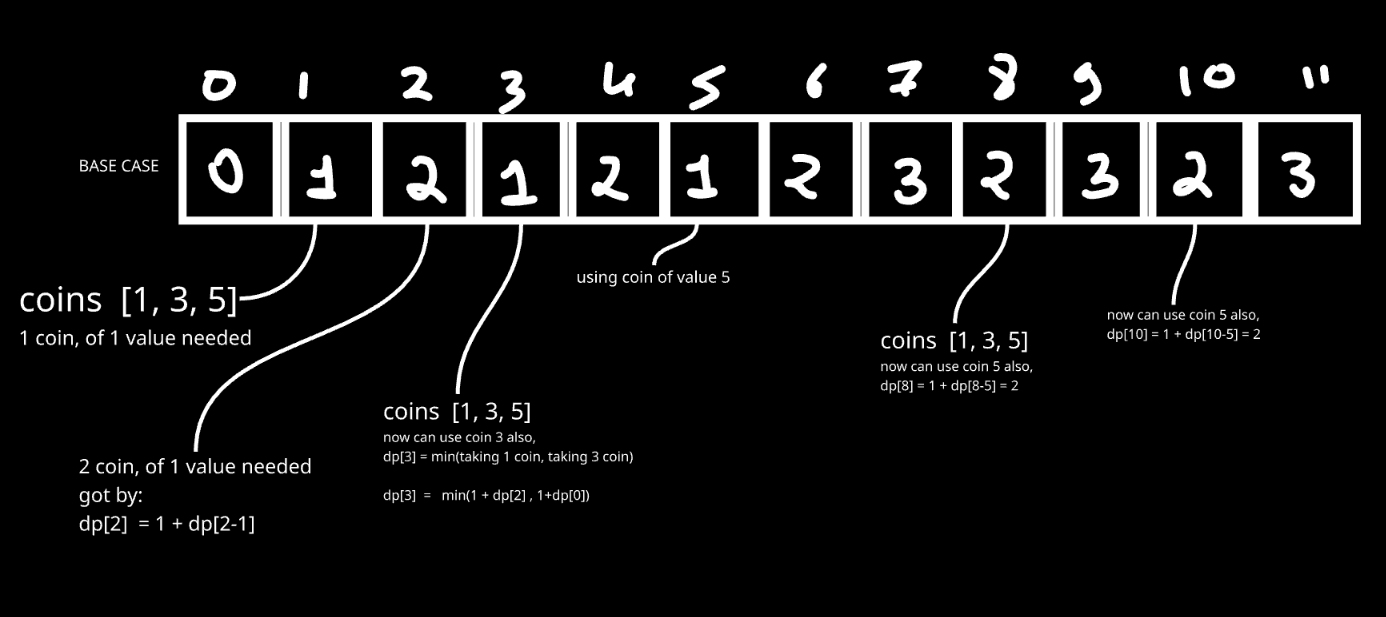
                if coins[i] <= j:

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

                dp[i][j] = min(pick,not\_pick)

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

**Approach 2 :** Just loop from 0 to amount, and for each value, check which coin from coins give minimum ans and put that in 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

#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)

1. **Subset Sum**

**Approach 1 :** Using plain recursion and doing pick and non-pick . But this will give **TLE**

class Solution:

    def isSubsetSum (self, arr, sum):

        # code here

        def checkSubs(sub\_sum,i):

            if i==len(arr):

                if sub\_sum==sum:

                    return True

                return False

            else:

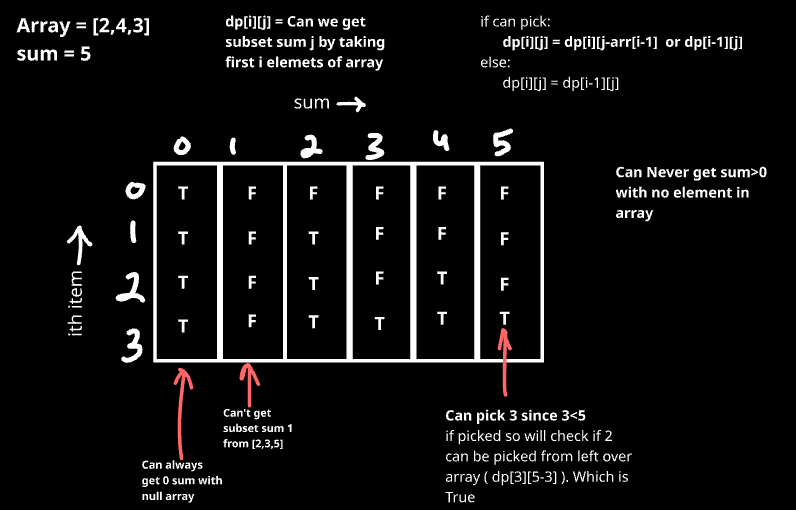
                pick = checkSubs(sub\_sum+arr[i],i+1)

                not\_pick = checkSubs(sub\_sum,i+1)

                return pick or not\_pick

        return checkSubs(0,0)

**Approach 2:** Using Knapsack pattern. This is equivalent to 1st knapsack problem. Our sack can handle weight (sum) and we are given list of weights. Here value is not given as we just need to say true or false.



class Solution:

    def isSubsetSum (self, arr, sum):

        # code here

        n = len(arr)

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

        #for 0 sum all is True

        for i in range(n+1):

            dp[i][0]=True

        for i in range(1,n+1):

            for j in range(1, sum+1):

                if arr[i-1] <= j:  #so can pick

                    dp[i][j] = dp[i-1][j-arr[i-1]] or dp[i-1][j]

                else:      #if can't pick. ans = whatever we got till i-1 array

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

        return dp[n][sum]

1. **Partition Equal Subset Sum**

**Approach:** It is equivalent to previous problem. Here if sum of array is s, we need to find subarray with sum s/2. If this is possible, by default other subarray will also have sum=s/2

class Solution:

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

        s = sum(nums)

        if s%2!=0:  #if sum is even (then only it can split into 2)

            return False

        #if we get one subarray of sum s/2, by default other one will have sum s/2

        #same as finding subarray with given sum

        rs = s//2

        n = len(nums)

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

        #for 0 sum all is True

        for i in range(n+1):

            dp[i][0]=True

        for i in range(1,n+1):

            for j in range(1, rs+1):

                if nums[i-1] <= j:  #so can pick

                    dp[i][j] = dp[i-1][j-nums[i-1]] or dp[i-1][j]

                else:      #if can't pick. ans = whatever we got till i-1 array

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

        return dp[n][rs]

1. **Partitions with Given difference**

This problem can be asked in 3 variants:

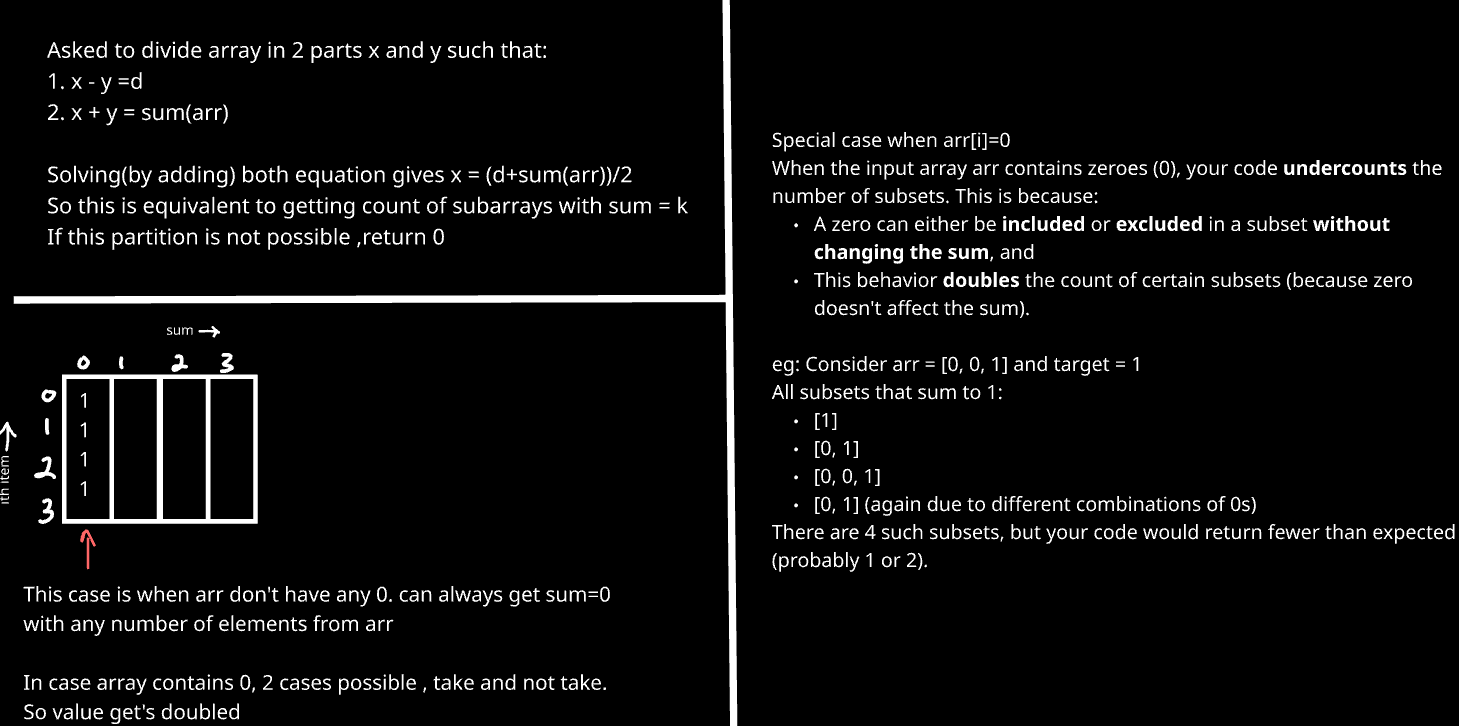
1. Number of subsets with given sum: in that we will just need to count subset sums=k. So it will include 2nd half of solution we added below

Link: <https://www.geeksforgeeks.org/problems/perfect-sum-problem5633/1>

2. Second is this one for which we added our code below

3. Target sum: Here we have to add +/- to have given target. After adding + for few and - for other we will have sum(a,d,e) – sum(b,c)= target. So this is exactly equal to the code added below.

Link: <https://leetcode.com/problems/target-sum/description/>



class Solution:

    def countPartitions(self, arr, d):

        n = len(arr)

        MOD = 10\*\*9 + 7

        total = sum(arr)

        # If partition not possible

        if (total + d) % 2 != 0 or d > total:

            return 0

        s = (total + d) // 2

        # DP: count subsets with sum = s

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

        dp[0][0] = 1

        for i in range(1, n + 1):

            if arr[i - 1] == 0:

                dp[i][0] = dp[i - 1][0] \* 2

            else:

                dp[i][0] = dp[i - 1][0]

        for i in range(1, n + 1):

            for j in range(s + 1):

                if arr[i - 1] <= j:

                    dp[i][j] = (dp[i - 1][j - arr[i - 1]] + dp[i - 1][j]) % MOD

                else:

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

        return dp[n][s]

## **TYPE 6: Kadane’s Algorithm**

**1. Maximum Subarray**

class Solution:

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

        curr=0

        maxSum=nums[0]

        for num in nums:

            if curr<0:  #when sum becomes negative, start from 0

                curr = 0

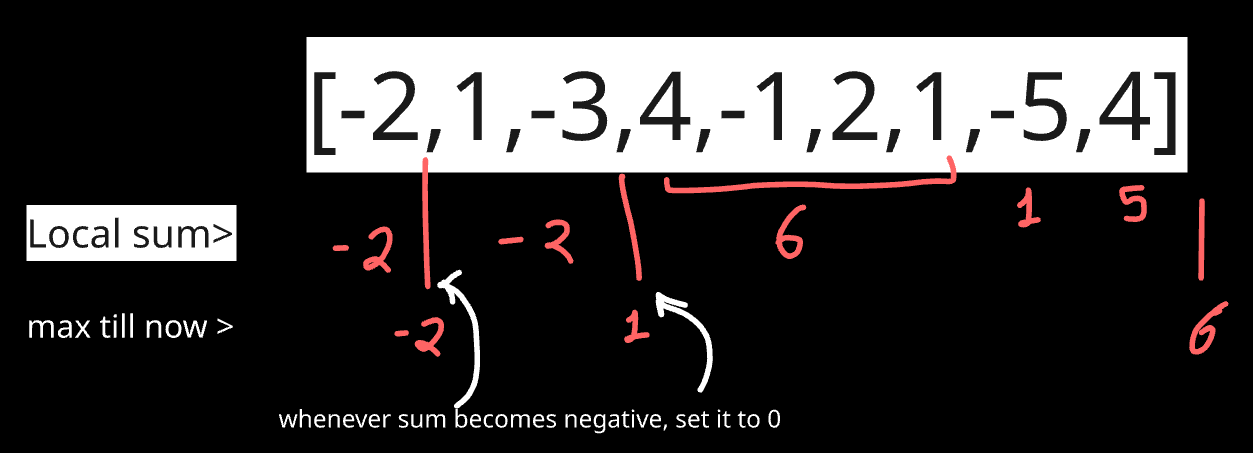
            curr+=num

            maxSum = max(maxSum,curr)

        return maxSum

At each position i, we have two choices:

* Extend the previous subarray: max\_ending\_here + arr[i]
* Start a new subarray from current element: arr[i]



**Why is Kadane’s Algorithm Dynamic Programming?**

Because it builds the solution using:

* **Optimal substructure**: Solution at index i depends on solution at i-1.
* **Overlapping sub problems**: Repeated computation of subarray sums.
* But it uses **space optimization** (constant space), which is why it may not seem like "typical" DP.

1. **Best time to buy and sell stock**

To get maximum profit on ith day, you have to buy that at minimum price till 1->i-1th day. So we will keep track of minimum value.

class Solution:

    def maxProfit(self, prices: List[int]) -> int:

        min\_price = prices[0]

        profit=0

        for price in prices[1:]:

            profit = max(profit, price-min\_price)

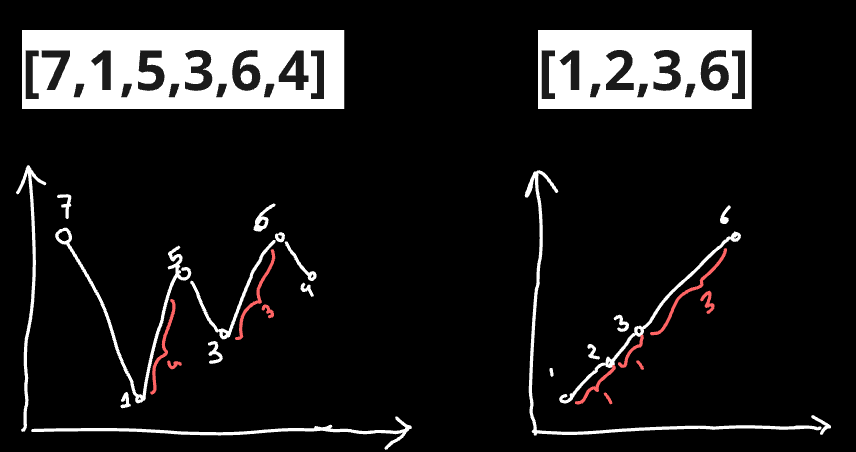
            min\_price = min(min\_price, price)

        return profit

1. **Best time to buy and sell stock II**

Here we can buy and sell multiple times, so better is that buy every time possible if next one is having more price.

Check from graph, whenever there is rise.



class Solution:

    def maxProfit(self, prices: List[int]) -> int:

        profit=0

        for i in range(1,len(prices)):

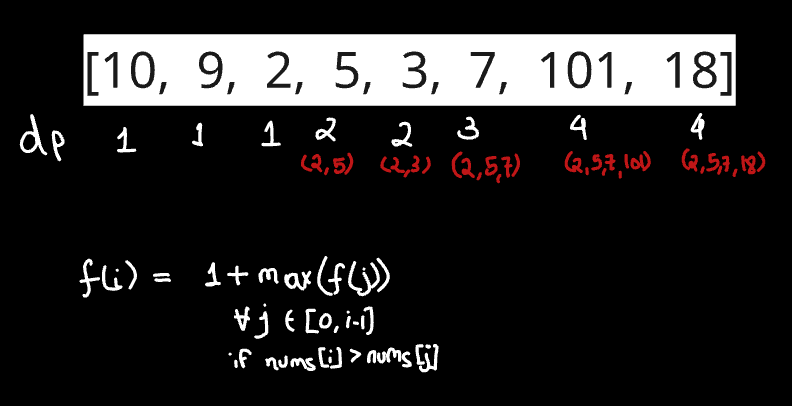
            if prices[i]-prices[i-1]>0:

                profit += prices[i]-prices[i-1]

        return profit

## **Type 7: DP on Subsequence**

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)

## **Type 8: Miscellaneous**

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))

1. **Palindrome Partitioning 2**

Move from 0 to n-1, and make all possible cuts. After each cut, we get 2 halves, apply same logic on given halves and hence it becomes recursion.

K moves from 0->n-1, base condition is when i>j (not possible string, so return 0, or i==j, single character, so already a palindrome (so 0 partition required)

class Solution:

    def minCut(self, s: str) -> int:

        n = len(s)

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

        # Step 1: Precompute palindrome table

        is\_pal = [[False] \* n for \_ in range(n)]

        for length in range(1, n + 1):

            for i in range(n - length + 1):

                j = i + length - 1

                if s[i] == s[j]:

                    if length <= 2:

                        is\_pal[i][j] = True

                    else:

                        is\_pal[i][j] = is\_pal[i + 1][j - 1]

# Step 2: Apply Recursion

        def rec(i, j):

            if i >= j:

                return 0

            if is\_pal[i][j]:

                return 0

            if dp[i][j] != -1:

                return dp[i][j]

            min\_cuts = float("inf")

            for k in range(i, j):

                if is\_pal[i][k]:  # small optimization

                    if dp[k+1][j]!=-1:  #if rec(k+1,j) is already computed, use that value

                        temp = 1+dp[k+1][j]

                    else:

                        temp = 1 + rec(k + 1, j)

                    min\_cuts = min(min\_cuts, temp)

            dp[i][j] = min\_cuts

            return dp[i][j]

        return rec(0, n - 1)

**Optimization explanation:**

Your original recursion was:

for k in range(i, j):

    temp = 1 + rec(i, k) + rec(k + 1, j)

    min\_cuts = min(min\_cuts, temp)

This means:

1. You cut the string s[i..j] into two parts: s[i..k] and s[k+1..j].
2. You compute **minimum cuts for both parts**.
3. You add 1 cut for splitting between them.

But notice: here we are **solving both halves**, even if the left half s[i..k] is **not a palindrome**.  
That’s wasteful because the only time we care about a cut at position k is when the **first part (s[i..k]) is a palindrome**. Otherwise, that partition is invalid for a palindrome partitioning.

**The Optimization Idea**

* If s[i..k] is **already a palindrome**, then we don’t need to compute its minimum cuts — it contributes **0 cuts**.
* So instead of: **1 + rec(i, k) + rec(k+1, j)** we reduce it to: **1 + rec(k+1, j)** (because rec(i, k) = 0 when s[i..k] is palindrome). This avoids unnecessary recursion calls on the left half.
* Even on **1 + rec(k+1, j),**  in place of directly calling the function, we check if it is already calculated. So that we can use already calculated answer present in dp.

**3. Longest Path**

**Intitution:**

Each node asks:

* “What’s the longest path if I start from each of my neighbors?”

The length of the path from me is:

* 1 + max(longest path from all my neighbors)

If I have no neighbors (i.e., terminal node), my longest path is 0.

**Key points:**

* Only works for **DAGs** (Directed Acyclic Graphs).
* You are **not counting nodes**, but **number of edges** in the longest path.
* For each node, recursively calculate the longest path starting from it.
* Store results in dp[] to avoid recomputation (memoization).

import sys

sys.setrecursionlimit(10 \*\* 7) # else was getting runtime exceed

def longest\_path(n, adj, dp):

    if dp[n] != -1:

        return dp[n]

    max\_len = 0

    for neighbour in adj[n]:

        max\_len = max(max\_len, 1 + longest\_path(neighbour, adj, dp))

    dp[n] = max\_len

    return dp[n]

# Input

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

adj = [[] for \_ in range(n + 1)]

for \_ in range(m):

    x, y = map(int, input().split())

    adj[x].append(y)

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

# Find longest path from every node

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

    longest\_path(i, adj, dp)

print(max(dp))