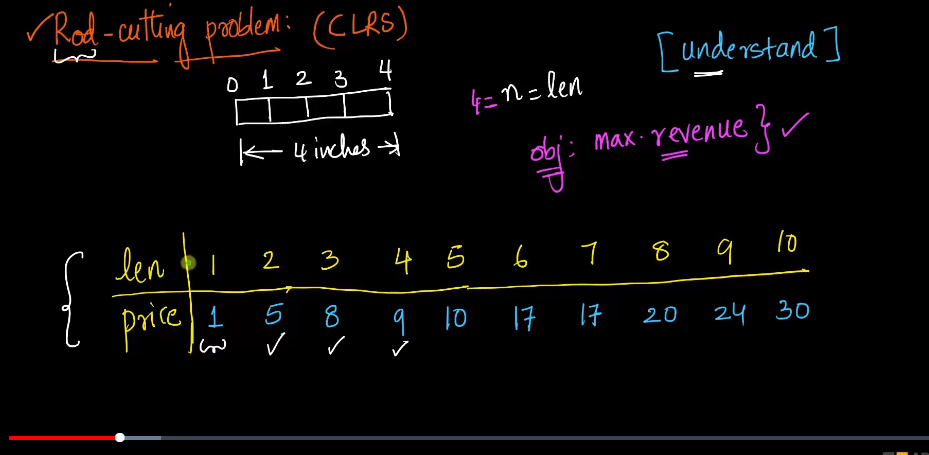
**Cutting a Rod | DP-13**

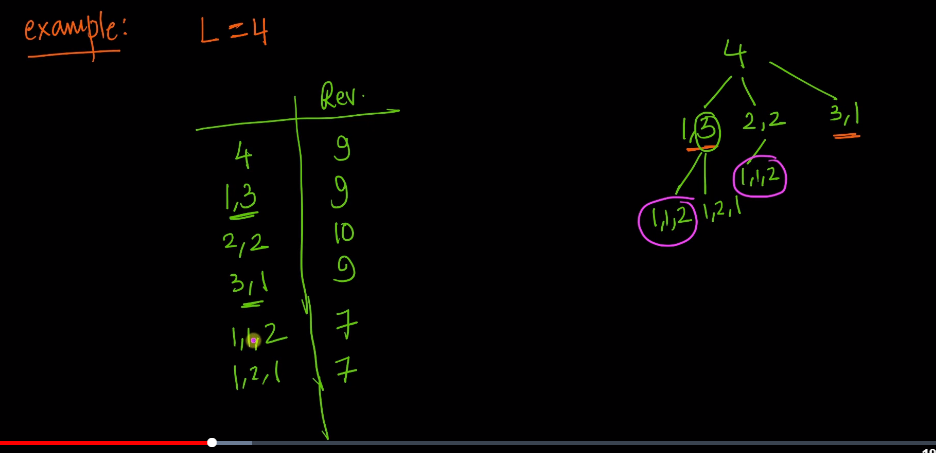
Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if length of the rod is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6) .

length | 1 2 3 4 5 6 7 8

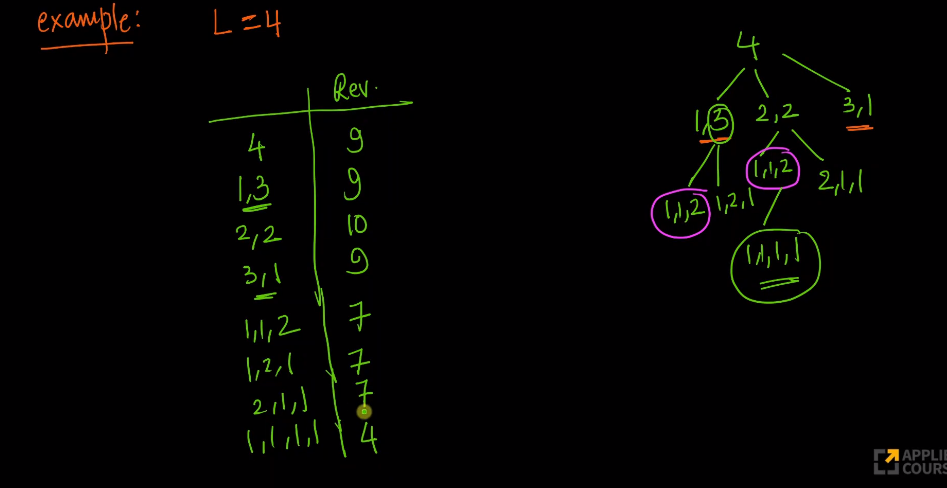
--------------------------------------------

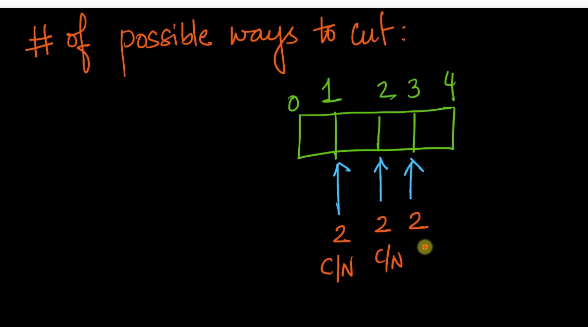
price | 1 5 8 9 10 17 17 20



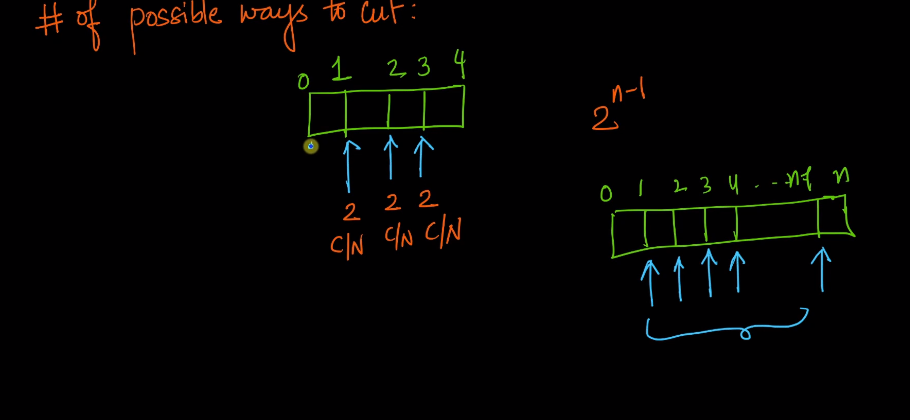


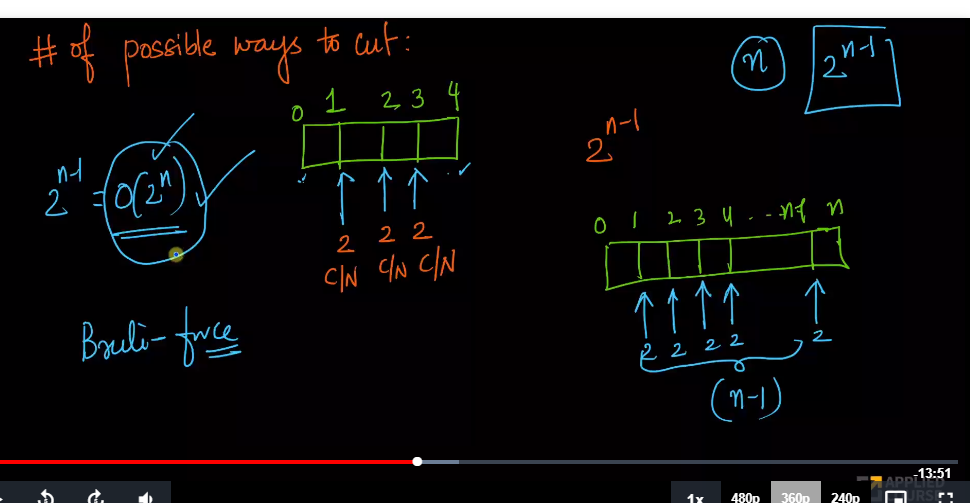
Here we have modified rod of the length 4 into differnt categories.



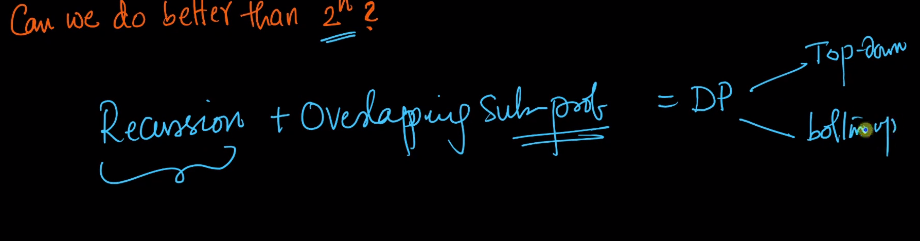


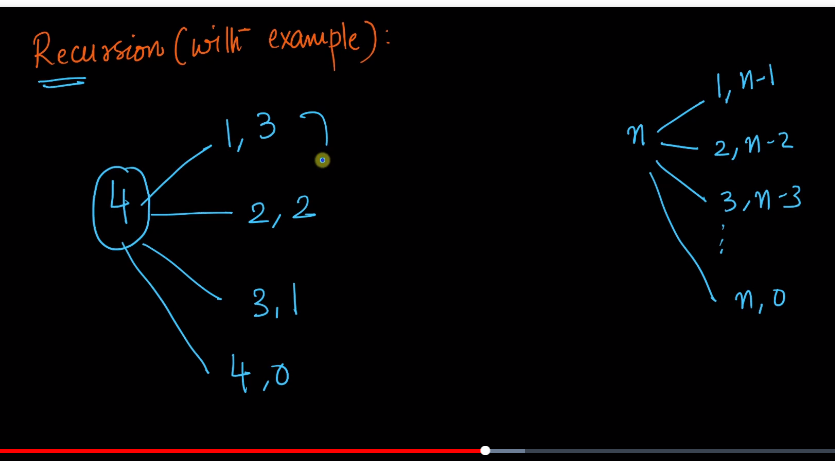
Here we can either cut the rod or not cut the rod in place 1,2,3. So total number will be 2(n-1).



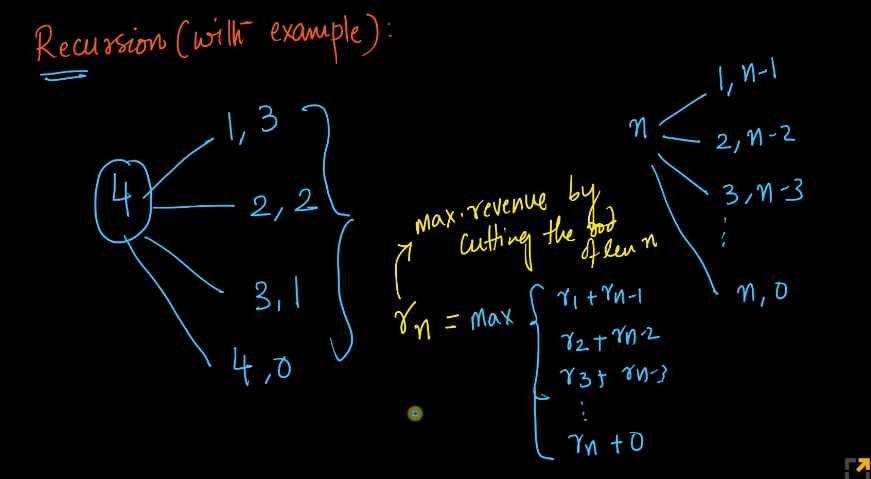


Time complexity will be 0(2n) just to cut the rod.



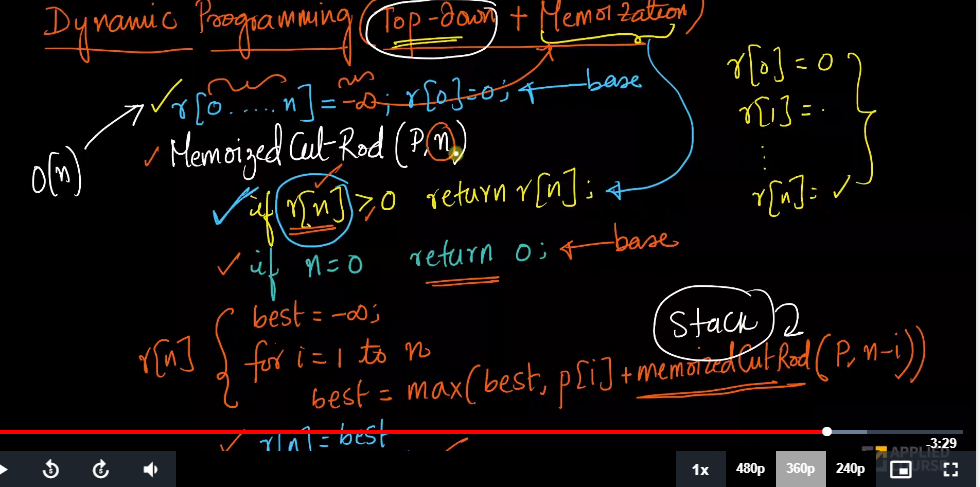


Here we need to compute all the computation shown here and take the maximum of it.

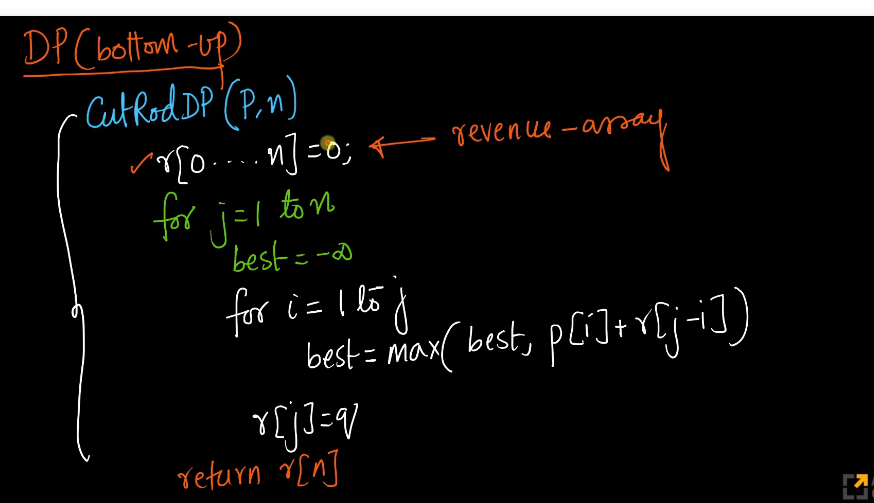




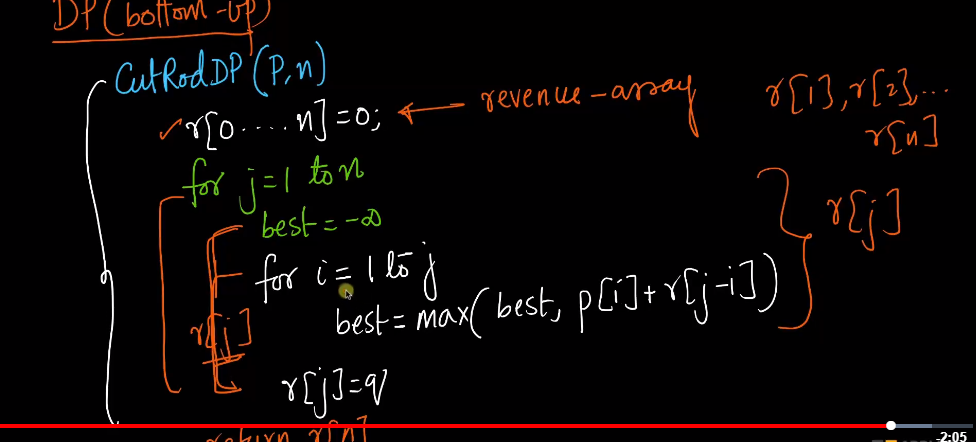




Here we have the recursive stack here.



Here we have a nested loop here.



Here in the inner loop, we are computing r[1],r[2] an all that.

#recursive approach

def rodCutting(arr,length):

if length<=0:

return 0

max\_rod=-1

for i in range(length):

max\_rod=max(max\_rod,arr[i]+rodCutting(arr,length-(i+1)))

return max\_rod

if \_\_name\_\_=='\_\_main\_\_':

arr=[2, 3, 7, 8, 9]

length=5

print("max rod",rodCutting(arr,length))

#dynamic Programming Approach

def rodCutting(arr,length):

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

dp[0]=0

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

max\_val=-1

for j in range(i):

max\_val=max(max\_val,arr[j]+dp[i-(j+1)])

dp[i]=max\_val

return dp[length]

if \_\_name\_\_=='\_\_main\_\_':

arr=[2, 3, 7, 8, 9]

length=5

print("max rod",rodCutting(arr,length))

**Count all possible paths in a Grid**

The problem is to count all the possible paths from top left to bottom right of a mXn matrix with the constraints that ***from each cell you can either move only to right or down***

**Examples :**

Input : m = 2, n = 2;

Output : 2

There are two paths

(0, 0) -> (0, 1) -> (1, 1)

(0, 0) -> (1, 0) -> (1, 1)

Input : m = 2, n = 3;

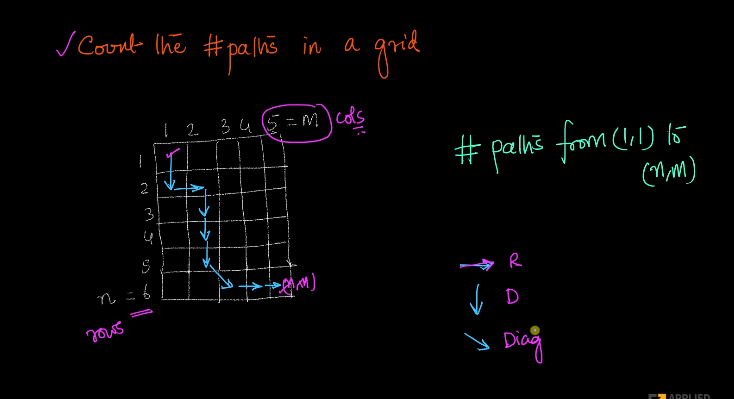
Output : 3

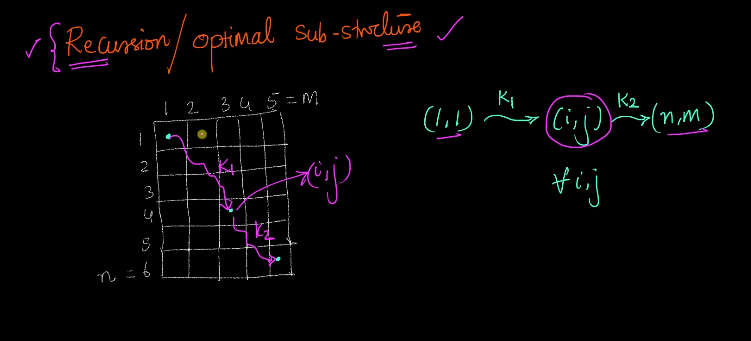
There are three paths

(0, 0) -> (0, 1) -> (0, 2) -> (1, 2)

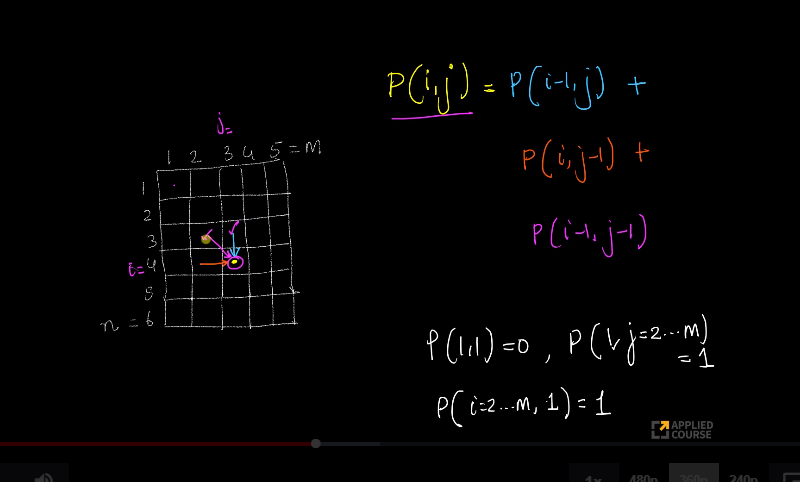
(0, 0) -> (0, 1) -> (1, 1) -> (1, 2)

(0, 0) -> (1, 0) -> (1, 1) -> (1, 2)



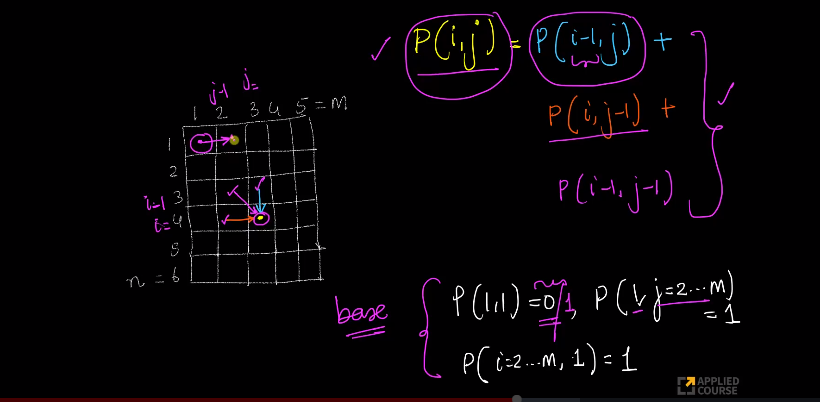


here we can post this problem as recursion where suppose k1 is total number of path to go from (1,1) to (i,j) and k2 is total number of path to go from (i,j) to (n,m).



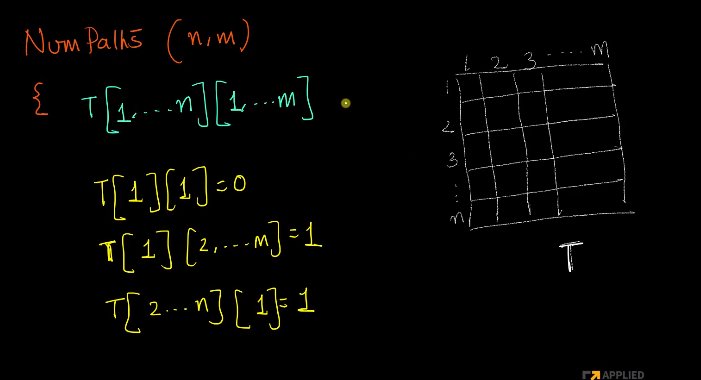
Here P(i-1,j) is the number of path to to rach p(i,j).

p(i,j) is total possible path to reach p(i,j).



When i=1, there is only one path to go from 1 to any number of j in a straight line.







Time complexity is 0(mn), space complexity is 0(mn).

# Returns count of possible paths to reach cell

# at row number m and column number n from the

# topmost leftmost cell (cell at 1, 1)

def numberOfPaths(m, n):

    # Create a 2D table to store

    # results of subproblems

    count = [[0 for x in range(m)] for y in range(n)]

    # Count of paths to reach any

    # cell in first column is 1

    for i in range(m):

        count[i][0] = 1;

    # Count of paths to reach any

    # cell in first column is 1

    for j in range(n):

        count[0][j] = 1;

    # Calculate count of paths for other

    # cells in bottom-up

    # manner using the recursive solution

    for i in range(1, m):

        for j in range(1, n):

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

    return count[m-1][n-1]

# Driver program to test above function

m = 3

n = 3

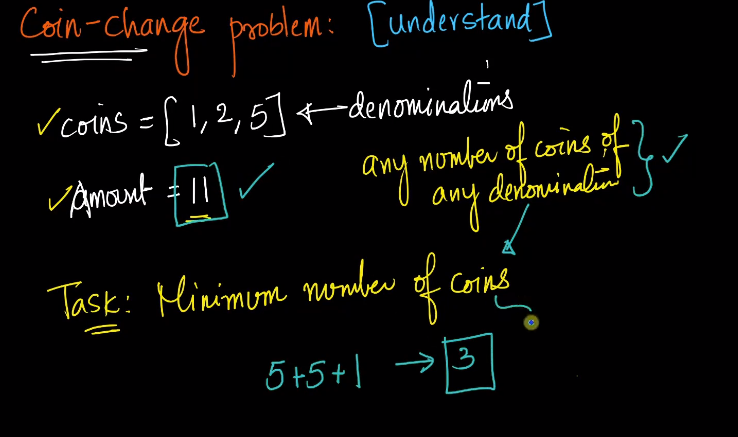
print( numberOfPaths(m, n))

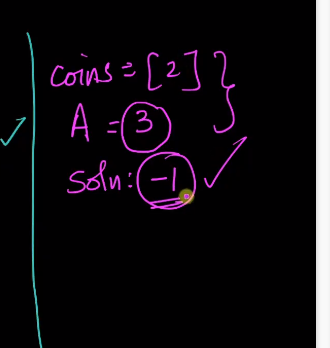
# This code is contributed by Aditi Sharma

**Coin Change**

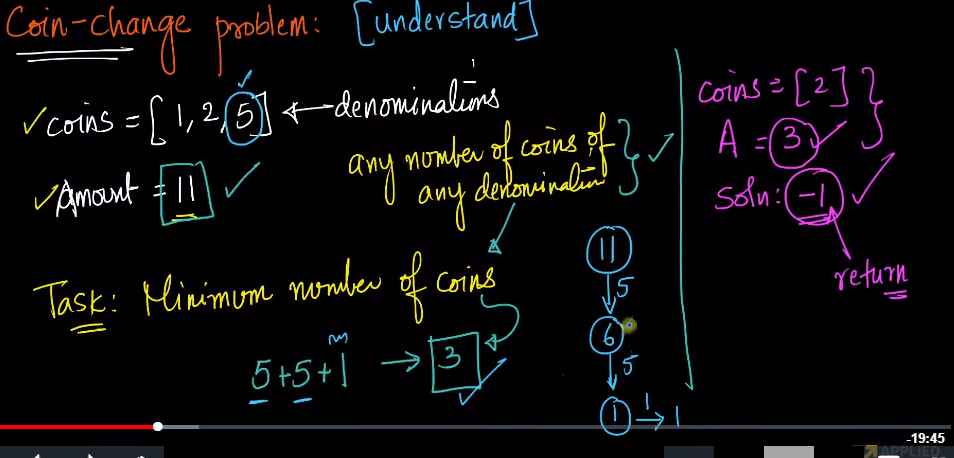
Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn’t matter.

For example, for N = 4 and S = {1,2,3}, there are four solutions: {1,1,1,1},{1,1,2},{2,2},{1,3}. So output should be 4. For N = 10 and S = {2, 5, 3, 6}, there are five solutions: {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}. So the output should be 5.

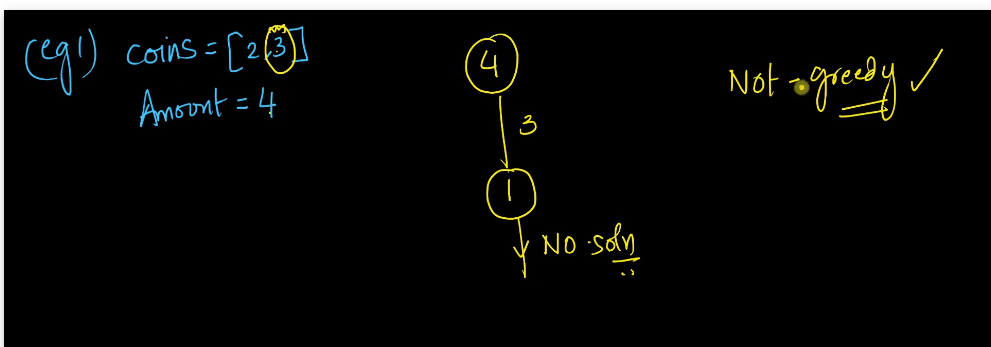




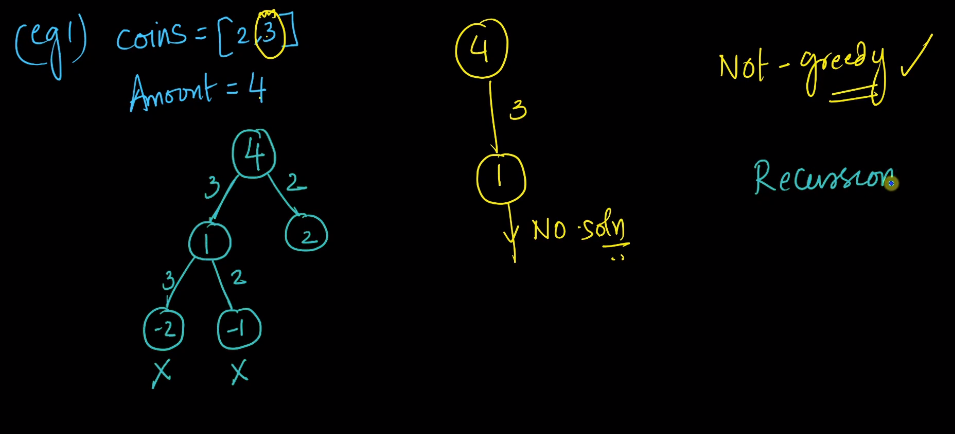
We need to return -1 when we can create the amount from denomination.



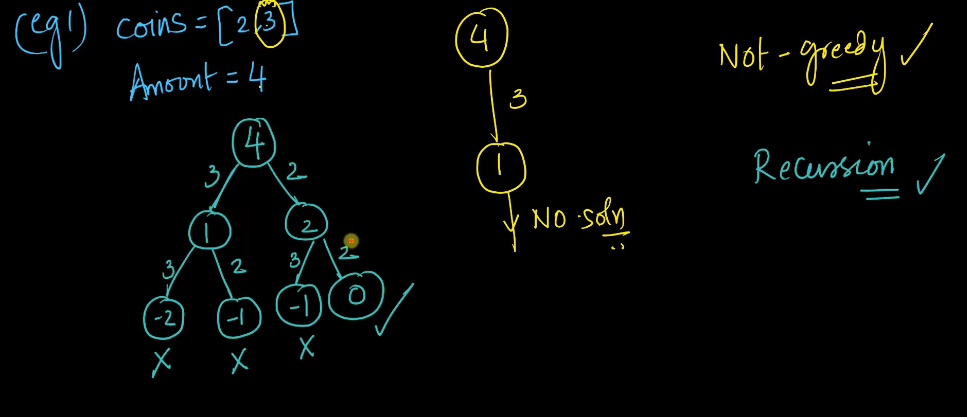
We can greedily pick 5 till the time we get less than 5 amount left in our example.

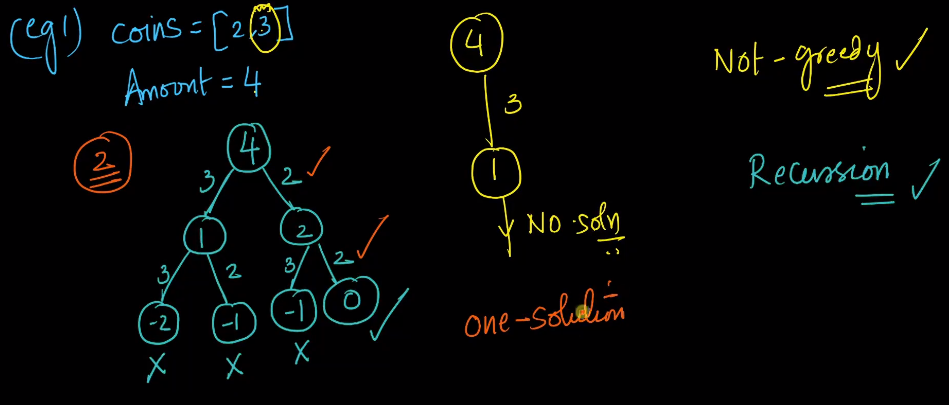


If we go for greedy approach, we cannot arrive the solution.

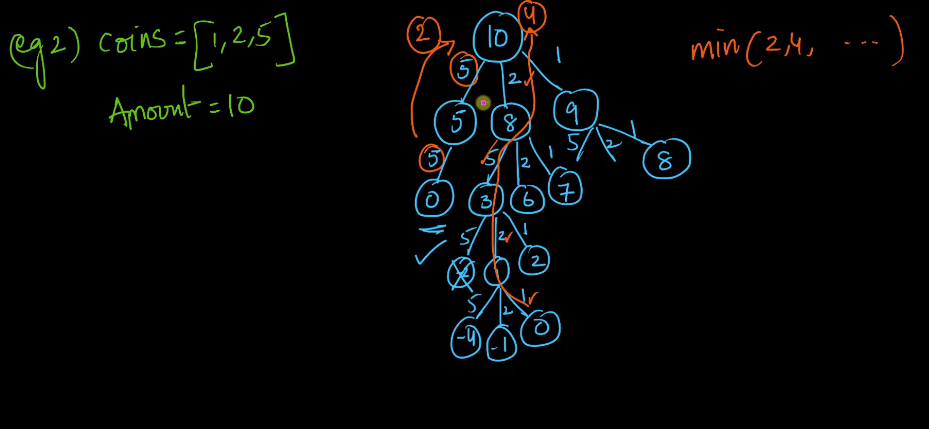


The moment we receive negtive numbers, we will stop .



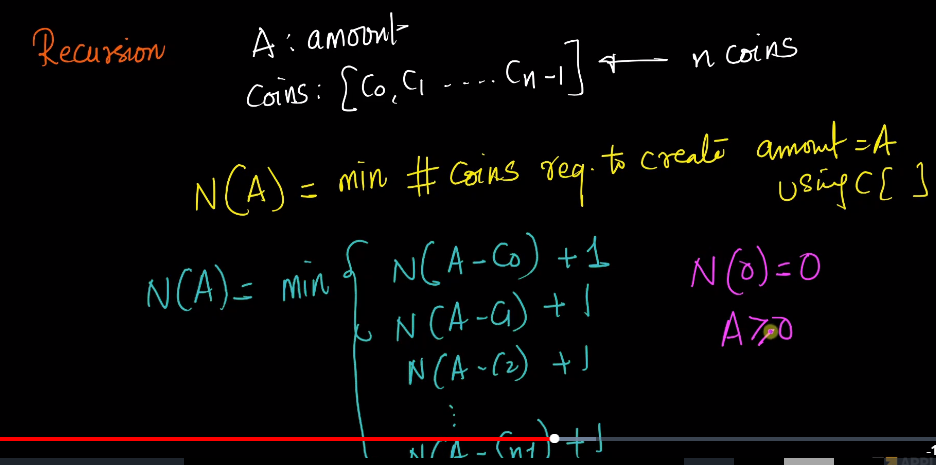
If we receive zero, we will track back.  


Here there is only one solution. We can more than one solution also.



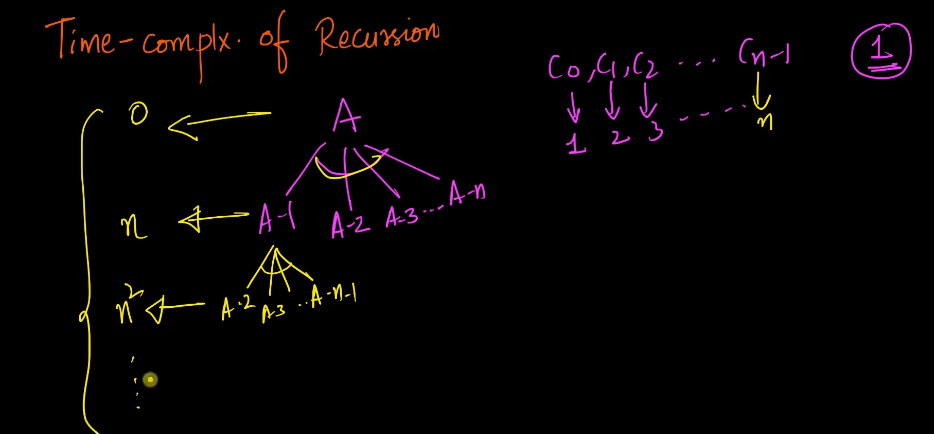
Here we can see overlapping subproblem.

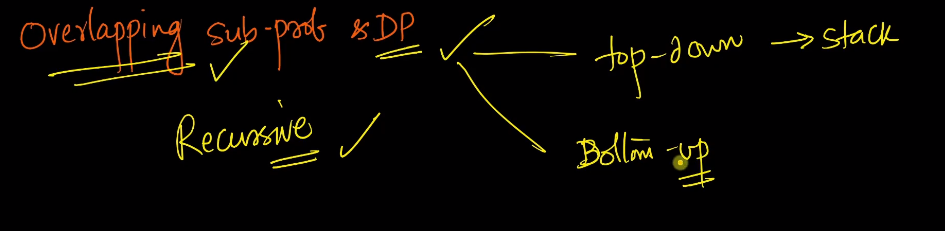
If we find the minimum solution in one of the subtree like 5,2,1, then we can add +1 to it to get the solution.

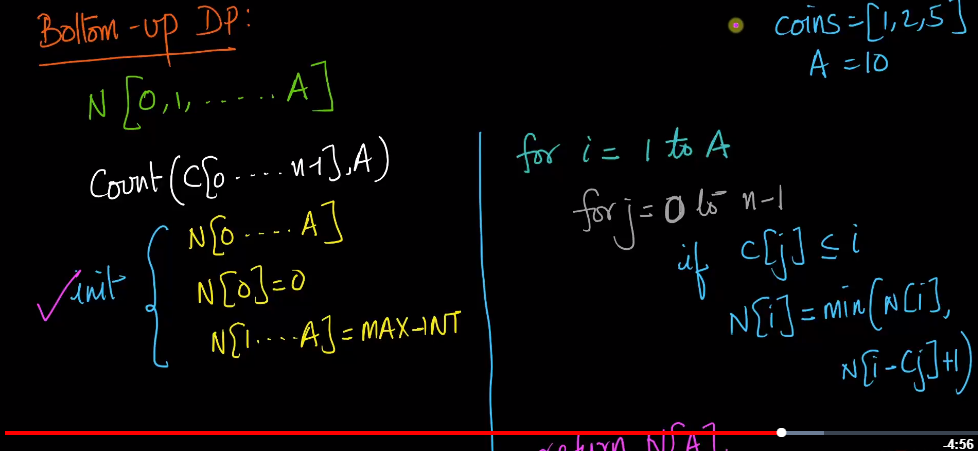




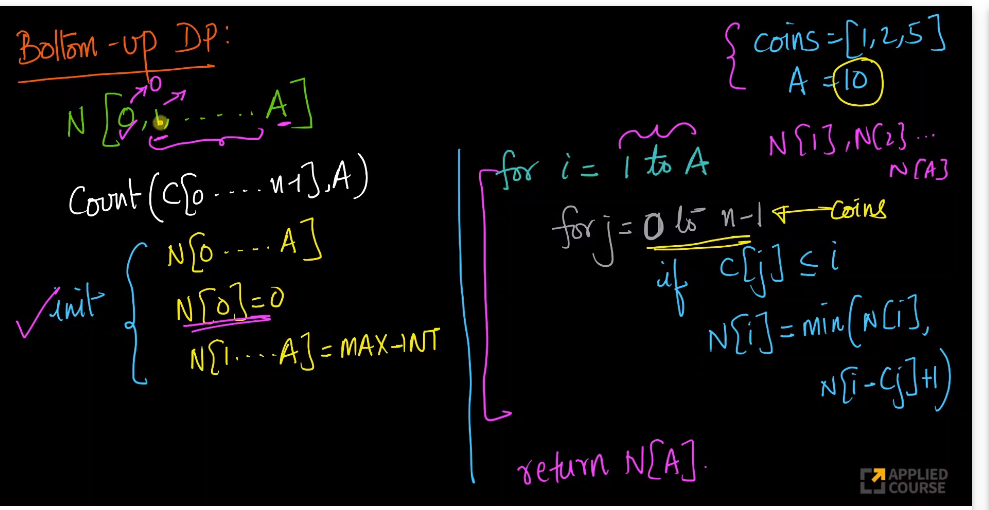
If we dont have any coins then we need to return -1.







N[0] means if the amount is zero cents, then how many we will use. If the amout is 1 cent, then how many we will use.



#recursion Solution

import sys

def minCoinChange(arr,amt,length):

if amt == 0:

return 0

res = sys.maxsize

for i in range(length):

if arr[i]<=amt:

temp=minCoinChange(arr,amt-arr[i],length)

if temp+1<res:

res=temp+1

return res

if \_\_name\_\_=='\_\_main\_\_':

arr = [1, 2, 3]

m = len(arr)

print(minCoinChange(arr, 15, m))

#bottom up approach

import sys

def minCoinChangeDP(arr,amt,m):

temp=0

dp=[0 for i in range(amt+1)]

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

dp[i]=sys.maxsize

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

for j in range(m):

if arr[j]<=i:

temp=dp[i-arr[j]]

if temp+1<dp[i]:

dp[i]=temp+1

return dp[amt]

if \_\_name\_\_=='\_\_main\_\_':

arr = [1, 2, 3]

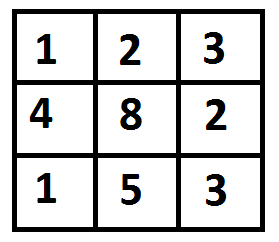
m = len(arr)

print(minCoinChangeDP(arr, 15, m))

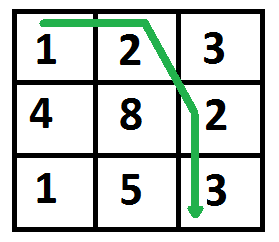
# Minimum Cost path Problem

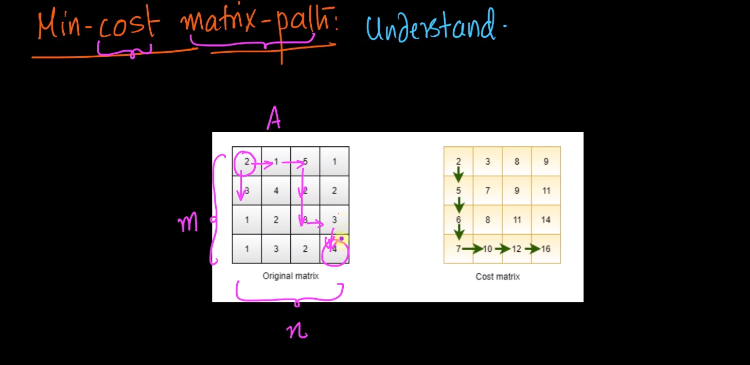
Given a cost matrix cost[][] and a position (m, n) in cost[][], write a function that returns cost of minimum cost path to reach (m, n) from (0, 0). Each cell of the matrix represents a cost to traverse through that cell. The total cost of a path to reach (m, n) is the sum of all the costs on that path (including both source and destination). You can only traverse down, right and diagonally lower cells from a given cell, i.e., from a given cell (i, j), cells (i+1, j), (i, j+1), and (i+1, j+1) can be traversed. You may assume that all costs are positive integers.

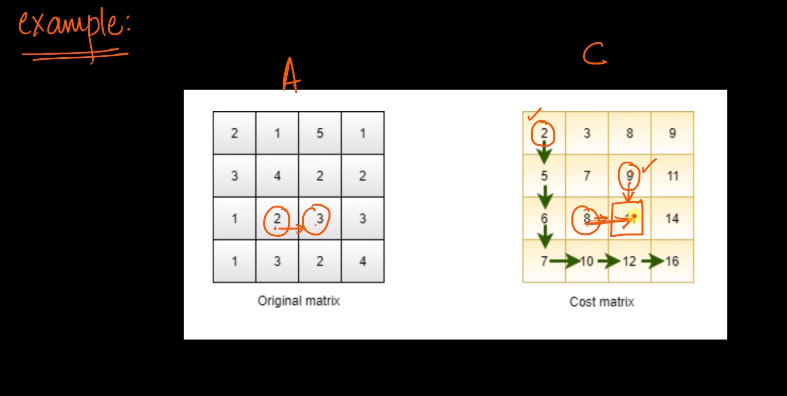
For example, in the following figure, what is the minimum cost path to (2, 2)?



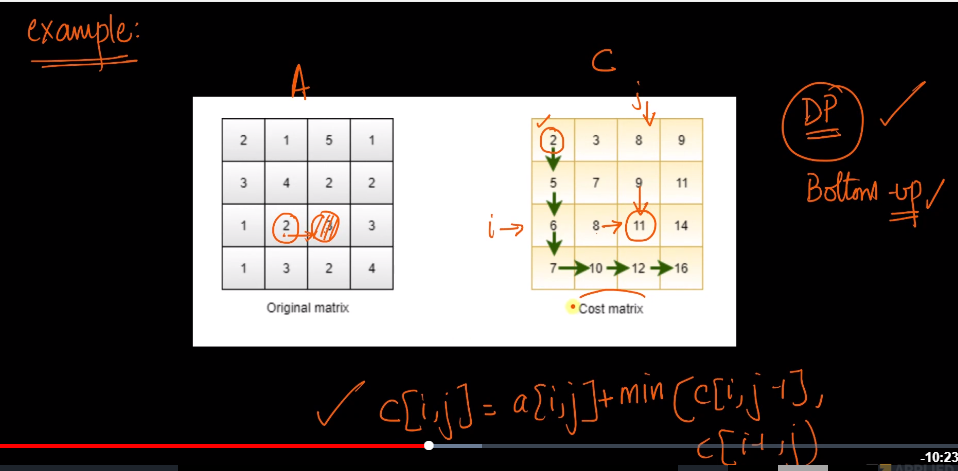
The path with minimum cost is highlighted in the following figure. The path is (0, 0) –> (0, 1) –> (1, 2) –> (2, 2). The cost of the path is 8 (1 + 2 + 2 + 3).

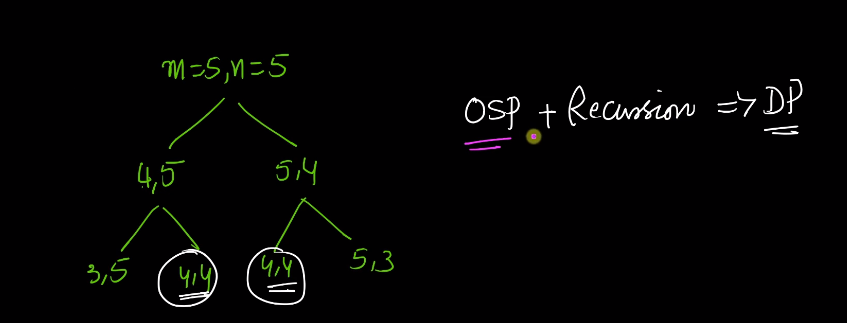


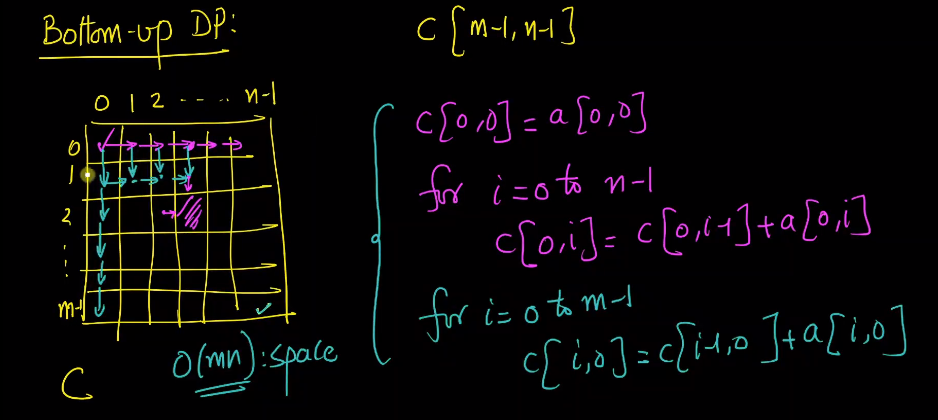


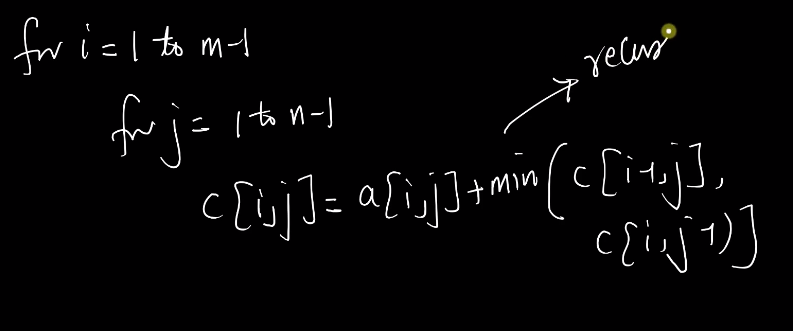


Here i can reach the cell in two ways, One from top and another from side.









#recursive approach

import sys

def minCost(arr,m,n):

if m==0 and n==0:

return arr[m][n]

elif m<0 or n<0:

return sys.maxsize

else:

return arr[m][n]+min(minCost(arr,m-1,n),

minCost(arr,m,n-1))

if \_\_name\_\_=='\_\_main\_\_':

cost= [ [1, 2, 3],

[4, 8, 2],

[1, 5, 3] ]

print(minCost(cost, 2, 2))

#dynamic Programming Approach

def minCost(arr,m,n):

dp=[[0 for x in range(m+1)] for x in range(n+1)]

dp[0][0]=arr[0][0]

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

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

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

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

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

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

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

return dp[m][n]

if \_\_name\_\_=='\_\_main\_\_':

cost= [ [1, 2, 3],

[4, 8, 2],

[1, 5, 3] ]

print(minCost(cost, 2, 2))

**Fill a N\*4 wall with 1\*4 bricks problem**

Given a number n, count number of ways to fill a n x 4 grid using 1 x 4 tiles.

Examples:

Input : **n = 1**

Output : 1

Input : **n = 2**

Output : 1

We can only place both tiles horizontally

Input : **n = 3**

Output : 1

We can only place all tiles horizontally.

Input : **n = 4**

Output : 2

The two ways are :

1) Place all tiles horizontally

2) Place all tiles vertically.

Input : **n = 5**

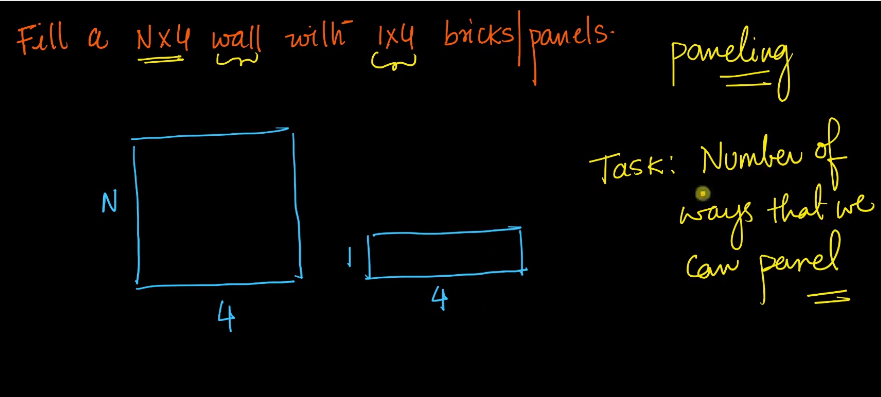
Output : 3

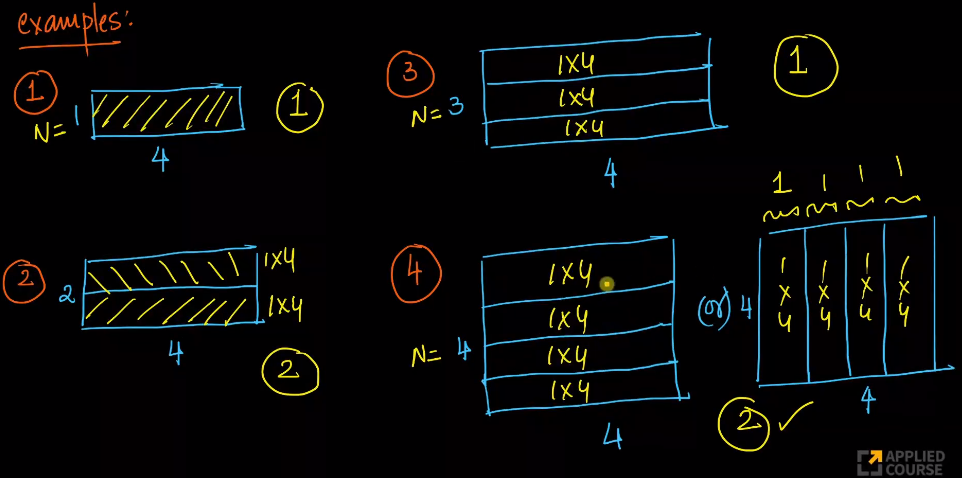
We can fill a 5 x 4 grid in following ways :

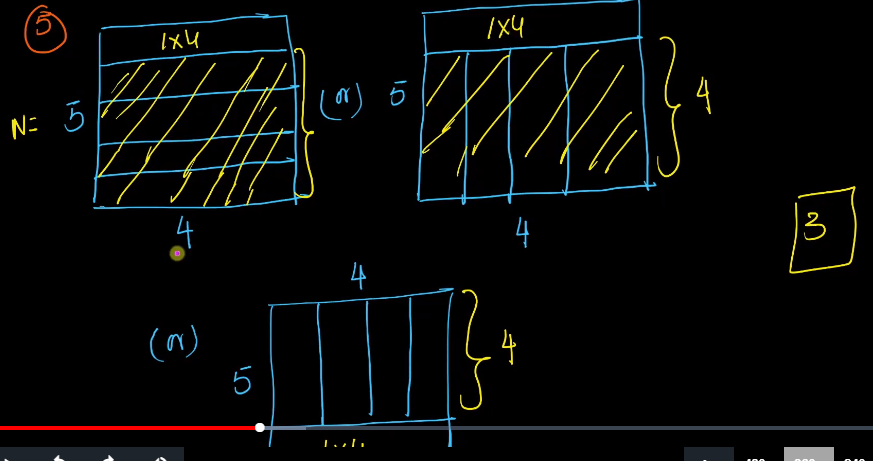
1) Place all 5 tiles horizontally

2) Place first 4 vertically and 1 horizontally.

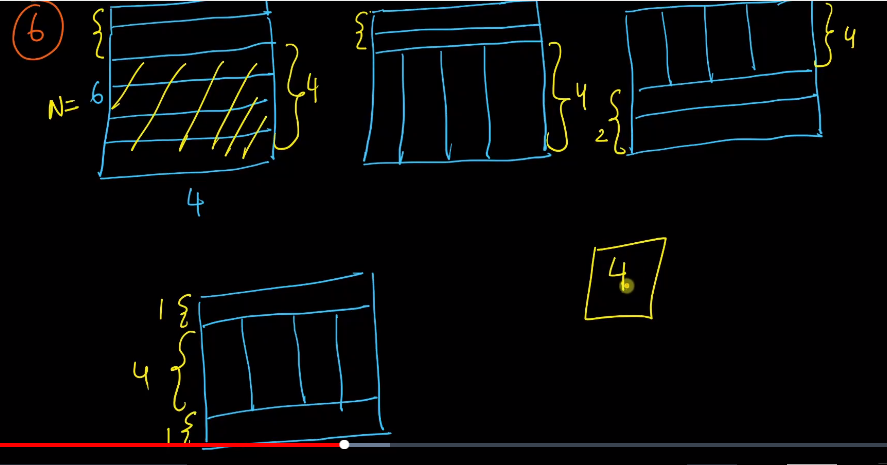
3) Place first 1 horizontally and 4 horizontally.

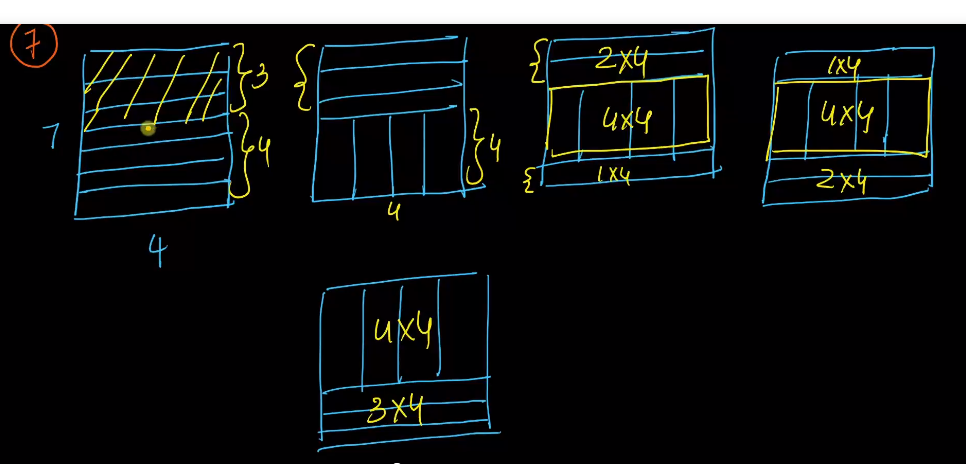


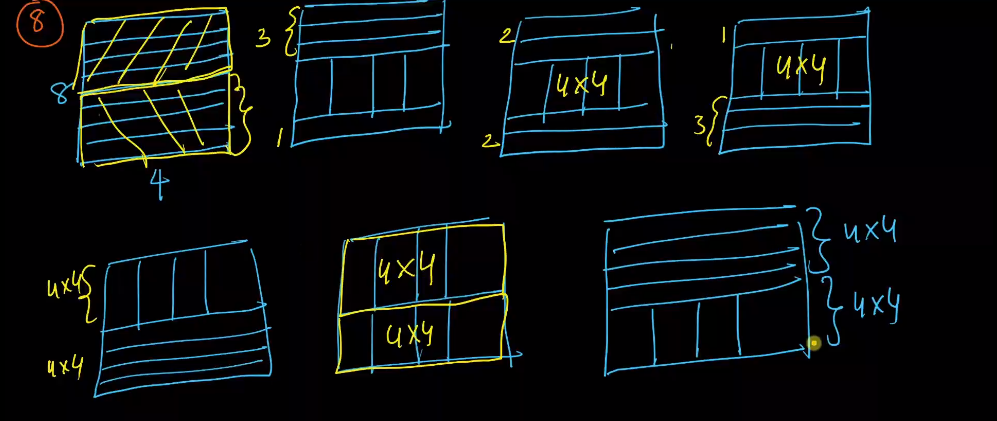




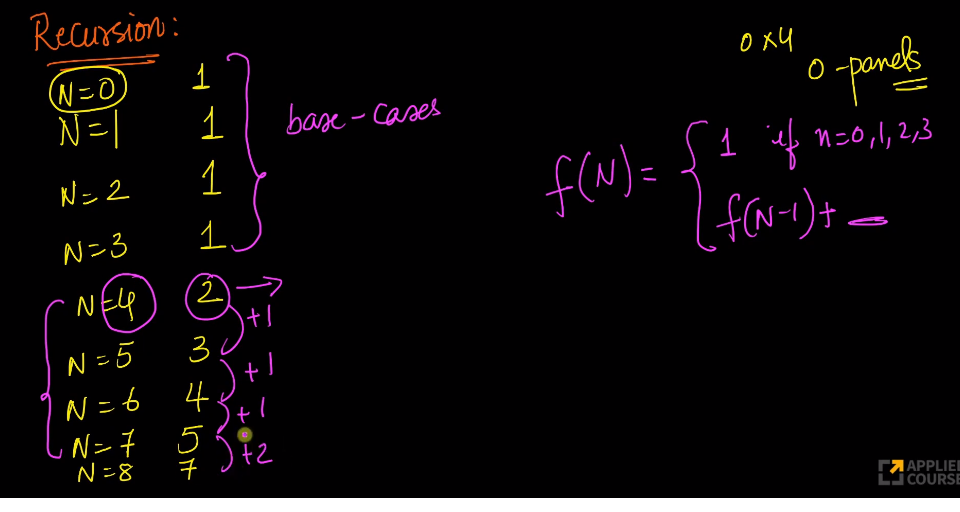
Here to solve for 5, we need to get the output from 4 and stack one the top of it.





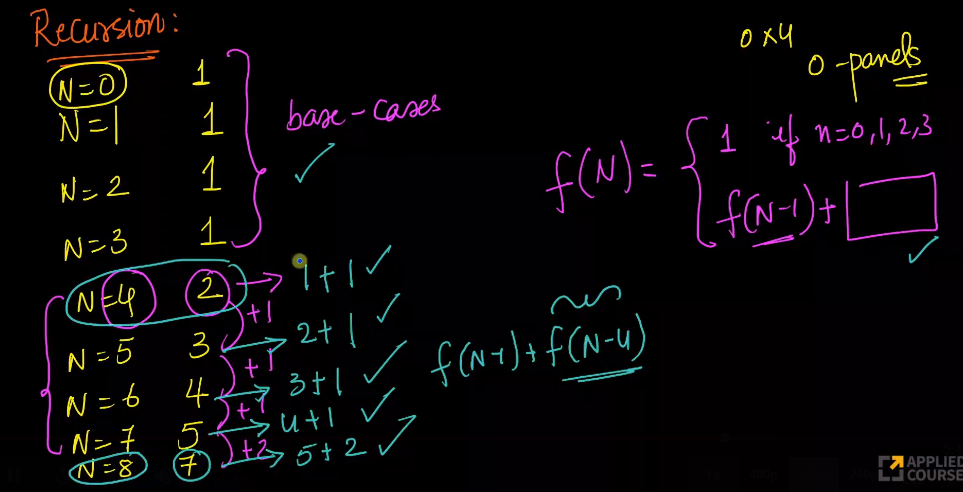


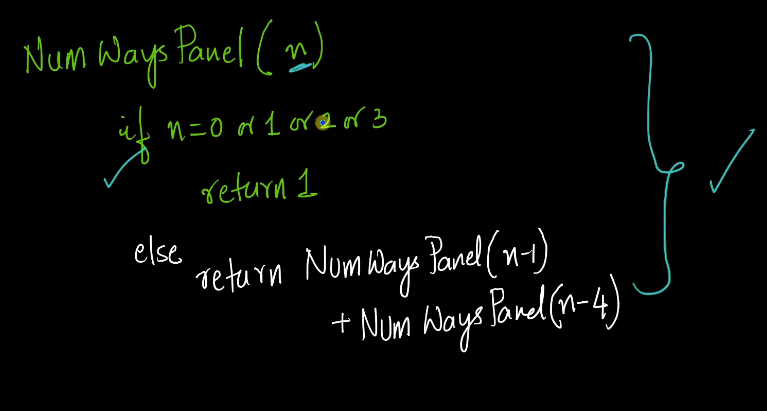
When N=0, we can panel this with 0 panel. It means we have one way to do this.

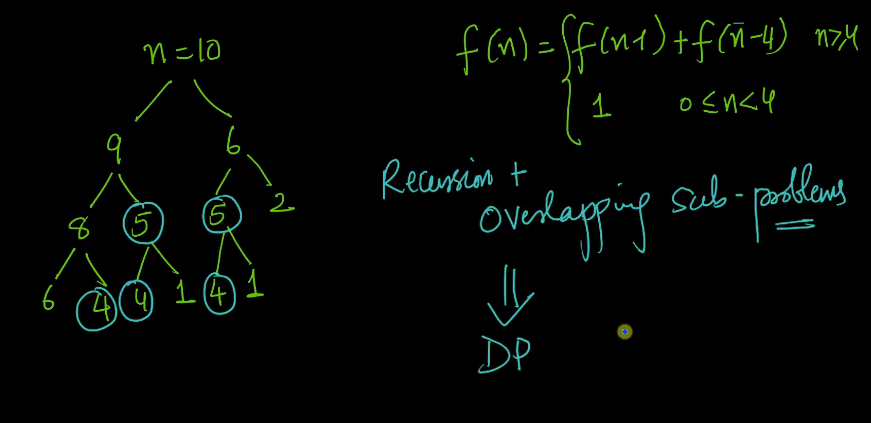


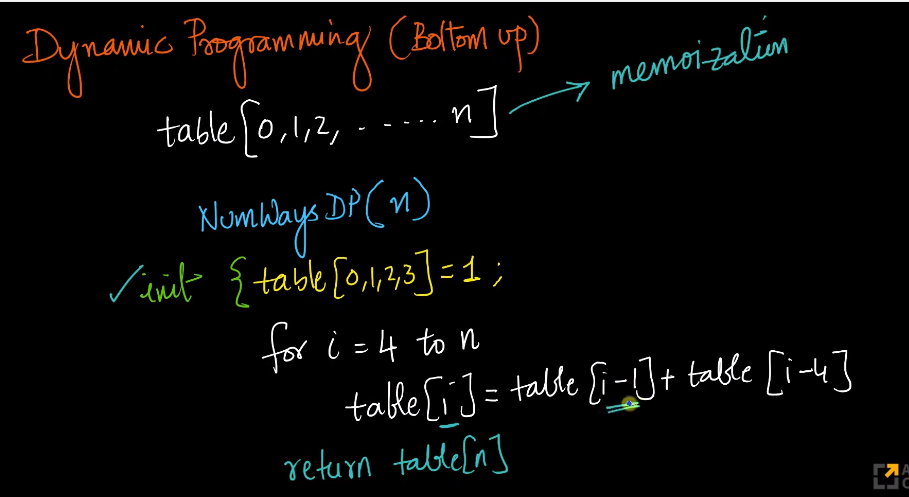
Here we can see that there is a sudden jump because at 8. We it is because 4\*4 is smart panel and we can stack it the way we want.











#recursive approach

def fillNumberWays(n):

if n==0 or n==1 or n==2 or n==3:

return 1

return fillNumberWays(n-1)+fillNumberWays(n-4)

if \_\_name\_\_=='\_\_main\_\_':

n=5

print(fillNumberWays(n))

#dynamic Programming Approach

def fillNumberWays\_dynamic(n):

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

dp[0],dp[1],dp[2],dp[3]=1,1,1,1

for i in range(4,n+1):

dp[i]=dp[i-1]+dp[i-4]

return dp[n]

if \_\_name\_\_=='\_\_main\_\_':

n=5

print(fillNumberWays\_dynamic(n))