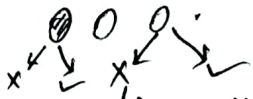
SAMIR PAUL

Dynamic Programming



DP → Enhanced Recursion
Parent of DP is Recursion

Identify DP → ① Choice Option → Using Recursion
↓
② asking optimal (minimum/maximum)

again calling 2 function
⇓
DP can be use

code DP first write Recursive soution

DP = Recursion + Storage
= Memoization

Tabular / Matrix
Bottom-Up DP

Variation of DP Questions:

1) 0-1 Knapsack (6 questions) → Subset Sum
→ Equal sum Partition
→ Count of Subset
2) Unbounded knap sack (5)
3) Fibonoci (7)
4) LCS (15)
5) LIS (10)
6) Kadane's Algorithm (6)
7) DP on Trees (4)
8) DP on Grid (14)
9) Matrix chain Multiplication (7)
10) Others (5)

0-1 Knapsack Problem

⇓ Minor changes

1) Subset sum
2) Equal Sum Partition
3) Count of Subset Sum ~~Diff~~
4) Minimum Subset Sum Diff
5) Target Sum
6) Number of Subset Sum with given Diff.

Knap Sack Problem

- Fractional knapsack → greedy approach
- 0/1 knapsack → DP
- Unbounded knapsack → DP

Input:

	I_1	I_2	I_3	I_4	
weight [] :	1	3	4	5	→ kg
Value [] :	1	4	5	7	→ ₹

W = 10 kg ↓ max capacity

⇓ max Profit ? bag

Identify → choice → include or ~~not~~ not include
Optimal → max Profit ₹

Memoization ⇒ Top-Down DP
Matrix DP ⇒ Bottom Up DP ⇒ Tabular

0/1 knapsack Recursion

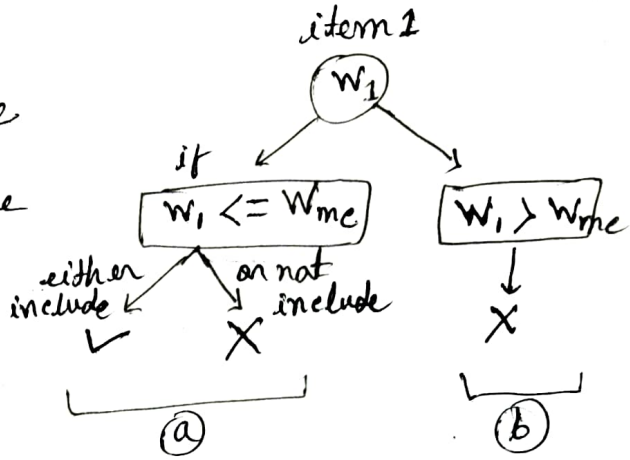
Input: Weight : [1, 3, 4, 5] kg
Values: [1, 4, 5, 7] ₹
$W_{max\ capacity}$: 7 kg → Wmc

output ⇒ maxprofit

Steps) Choice Diagram → Recursive code

def knapsack{
  Base condition
  Choice Diagram
}

Base case → think of the smallest valid input



Recursive Code

```
n = len(weight)
def knapsack(weight, values, Wmc, n):
    if (n==0 or Wmc=0) : return 0
a   if Weight[n-1] <= Wmc :
        return max(values[n-1] + knapsack(weight, values, Wmc-Weight[n-1],n-1),
                   knapsack(weight, value, Wmc, n-1))
                   value if include        value if not include
b   elif weight[n-1] > Wmc :
        return knapsack(weight, values, Wmc, n-1)
```

⇓ Memorisation

constraints n, W
$n \leq 100$
$W_{mc} \leq 1000$

dp[102][1002] ⇒ make matrix (each element -1)

only add these to the recursive code and memoraze.

```
def knapsack(wt, val, Wmc, n):
    if n==0 or Wmc==0 : return 0
✓   if dp[n][Wmc] != -1 : return dp[n][Wmc]
    if wt[n-1] <= Wmc :
        return dp[n][Wmc] = max(val[n-1] + knapsack(wt, val, Wmc-wt[n-1], n-1),
                                knapsack(wt, val, Wmc, n-1))
    if wt[n-1] > Wmc :
        return dp[n][Wmc] = knapsack(wt, val, Wmc, n-1)
```

Base condition of Recursive Solution ⇒ Initialisation of Top-Down approach (1st row, 1st column)

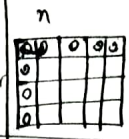
Recursive Code

```
if n==0 or W==0 : return 0
```

```
if wt[n-1] <= W :
    return max(val[n-1] + knapsack(wt, val, W-wt[n-1], n-1), knapsack(wt, val, W, n-1))
elif wt[n-1] > W :
    return knapsack(wt, val, W, n-1)
```

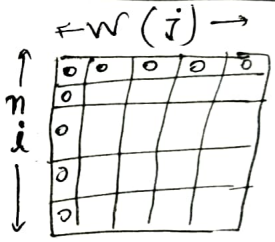
Top-down Code

```
for i in range(n+1):
    for j in range(W+1):
        if i==0 or j==0:
            dp[i][j] = 0
```



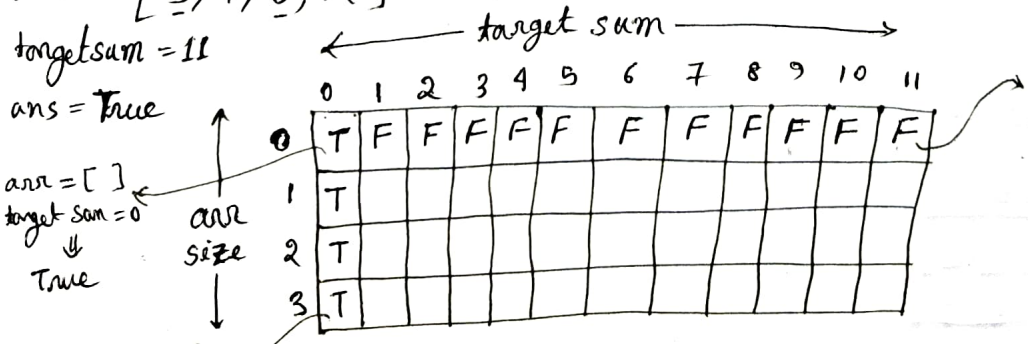

```
if wt[n-1] <= W :
    dp[n][W] = max(val[n-1] + dp[n-1][W-wt[n-1]], dp[n-1][W])
elif wt[n-1] > W :
    dp[n][W] = dp[n-1][W]
```

```
dp = [[0]*(W+1) for i in range(n+1)]
for i in range(n+1):
    for j in range(W+1):
        if i==0 or j==0 : dp[i][j] = 0

for i in range(1, n+1):
    for j in range(1, W+1):
        if wt[j-1] <= W :
            dp[i][j] = max(val[i-1] + dp[i-1][j-wt[j-1]], dp[i-1][j])
        else:
            dp[i][j] = dp[i-1][j]

return dp[n][W]
```



Subset Sum Problem: Check if we can find the target sum from the subset of the array.

arr = [3, 7, 8, 12]
targetsum = 11
ans = True



arr = [ ]
target sum = 0
⇓
True

arr = [3, 7, 8, 12]
targetsum = 0
⇓
True (Empty Subarray)

arr = [ ]
targetsum = 11
⇒ Not Possible
False

Symmetricity of Subset sum with 0/1 Knapsack

0/1 knapsack

wt [ ] W

val [ ]

if wt[i-1] <= j :

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

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

Subset Sum

wt[] ⇒ arr[] W ⇒ Target Sum

val[] ⇒ Ignore

if arr[i-1] <= j :

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

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

Equal Sum Partition

arr = [1, 3, 2, 2, 5, 7] ⇒ if sum(arr) % 2 != 0 return False

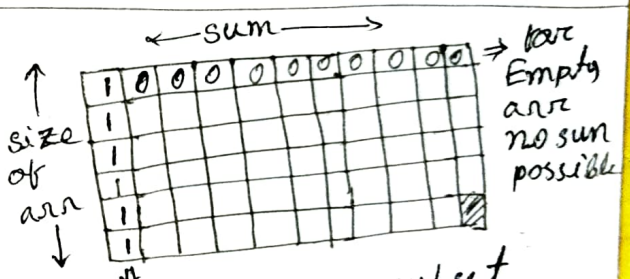
else: target = sum(arr) // 2

apply subset sum on target.

Count of Subset sum with a given Sum :

arr = [2, 3, 5, 6, 8, 10] Sum = 10

ans = 3 ⇒ {2, 3, 5}, {2, 8}, {10}



Just change the "or" of subset sum ⇒ '+'

if arr[i-1] <= j :

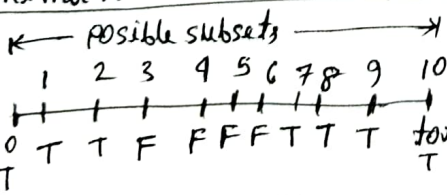
dp[i][j] = dp[i-1][j-arr[i-1]] + dp[i-1][j]

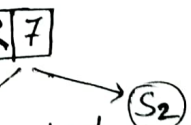
include do not include

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

Minimum Subset Sum Difference

arr = [1 | 2 | 7]





Find min(S2 - S1) (S2 smaller)

S1 + S2 = totalsum

⇒ S2 = totalsum - S1

∴ min(totalsum - 2·S1)

in subsetsum problem

size of array ↓	possible sums → 0	1	2	3	4	5	6	7	8	9	10
0	T	F	F	F	F	F	F	F	F	F	F
1	T	T	F	F	F	F	F	F	F	F	F
2	T										
3	T	T	T	F	F	F	F	T	T	T	T

2 can be a subsetsum for size 3 arr

7 can be a subset sum

⇒ This row denotes ⇒ if size of arr is 3 then what are the posible subset sums are possible

∴ From 3rd row we get,
possible subset sum for arr = [1,2,7] is
subset sums = [0, 1, 2, 7, 8, 9, 10] ⇒ S1
n = len(subset sums)
mn = minsubsetsumdiff = float("inf")

```
for S1 in subset sums:
    mn = min(mn, totalsum - 2*S1)
return mn
```

Count the number of subsets with a given difference:

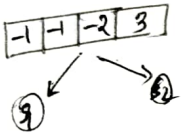
arr = [1,1,2,3] diff = 1

$S_1 - S_2 = diff$ — ①
$S_1 + S_2 = totalsum$ — ②
① + ② ⇒ $S_1 = \frac{totalsum + diff}{2}$

So, we have to find now count of subsets whose sum is $\frac{totalsum+diff}{2}$

∴ count of subset sum

Target Sum: arr = [1,1,2,3] target = 1 assign '+' and '−' before elements of arr such that sum(arr) == target

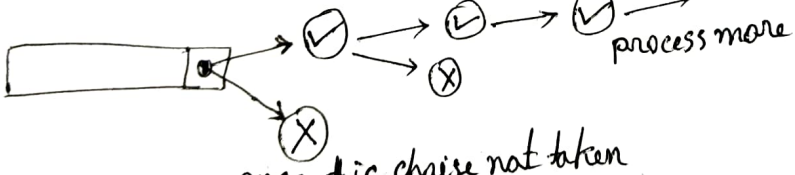


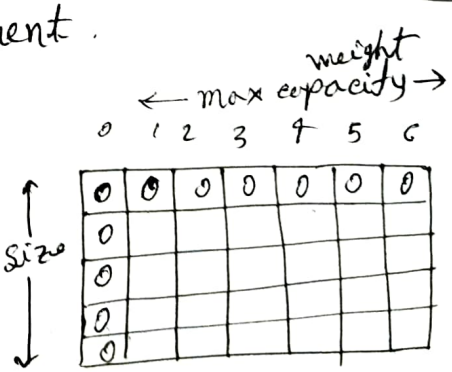
basically we are again deviding arr into two subset such that difference b/w subset sums equals target.

∴ $S_1 - S_2 == target$
⇒ $2S_1 = target + sum(arr)$
⇒ $S_1 = \frac{target + sum(arr)}{2}$ ⇒ Find number of possible S_1 ⇒ answer

---

Unbounded knapsack: multiple occurance of an element.





0-1 knapsack

```
if wt[i-1] <= j:
    dp[i][j] = max(val[i-1] + dp[i-1][j - wt[i-1]], dp[i-1][j])
else:
    dp[i][j] = dp[i-1][j]
```

Unbounded knapsack

```
if wt[i-1] <= j:
    dp[i][j] = max(val[i-1] + dp[i][j - wt[i-1]], dp[i-1][j])
else:
    dp[i][j] = dp[i-1][j]
```

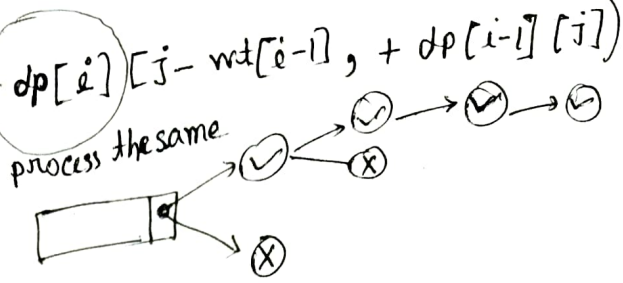
Taken call the same for further processing

⊗ not taken so call for remaining

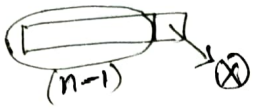
Unbounded knapsack ⇒ ① Rod Cutting , ② Coin Change I, ③ coin change II , ④ Maximum ribon cut

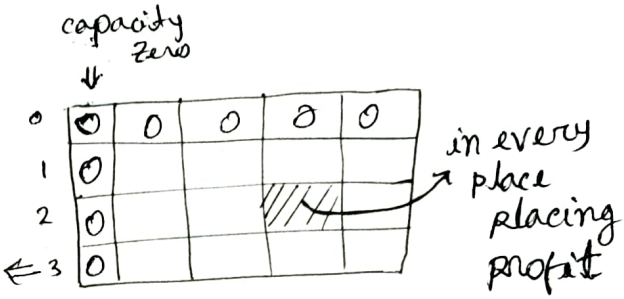
if wt[i-1] <= j :

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



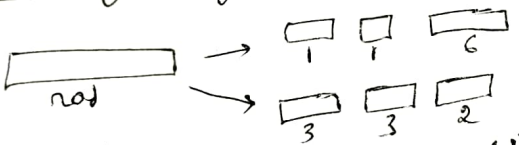
else: dp[i][j] = dp[i-1][j]





size of arr is 3 but for empty array or not taking any element profit = 0

Rod Cutting length = 8 prices = [1, 5, 8, 9, 10, 17, 17, 20]



val ⇒ prices

Weight ⇒ length = [(i+1) for i in range(n)]

if length[i-1] <= j :

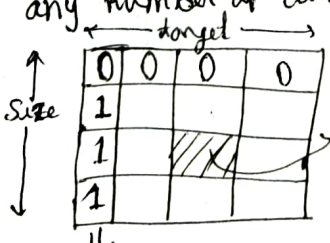
dp[i][j] = max(prices[i-1] + dp[i][j - length[i-1]] , dp[i-1][j])

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

↳ as weight of Unbounded knapsack is length → possible length in which we can cut the rod. array here.

Coinchange: any number of coins can be used ⇒ Unbounded knapsack

⇓ Maximum Combination of coins to make target.



⇒ as size = 0 ; so no coin can be taken ⇒ combination = 0

From size 1 empty arr no coins target 0 is possible.

if coin[i-1] <= j : dp[i][j] = dp[i][j - coin[i-1]] + dp[i-1][j]

add as total combination is needed

as multiple time one coin can be taken

process remaing coins except i

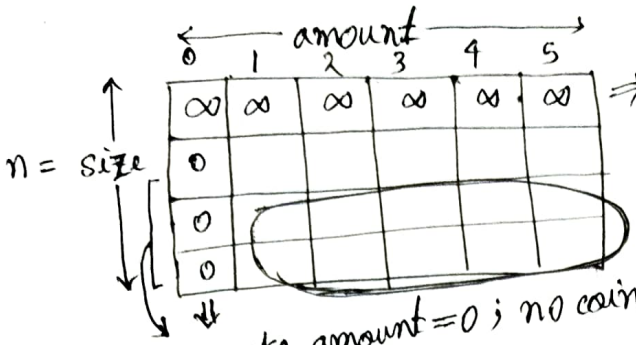
else : dp[i][j] = dp[i-1][j]

as No val array is present in this problem so not taking val

Coin Change II → Minimum number of coins required to make the amount

coin = [1, 2, 5), amount = 11

Unbounded Knapsack
as any no. of coins can be taken



⇒ for array of size = 0 to make amount ∞ numbers of coins required.

so dp[i][0] = 0 ; 0 < i ≤ n

** In this question we have to initialise 1st raw also **

↳ for j in range (1, amount+1):
  if j % coin[0] == 0: ⇒ the amount (j) is multiple of 1st element of coin[0]
    dp[1][j] = j // coin[0] ⇒ no. of coins needed to make j

for i in range (2, n):
  for j in range (1, amount+1):
    if coins[i-1] <= j:
      dp[i][j] = min((1) + dp[i][j - coins[i-1]] , dp[i-1][j])
    else: dp[i][j] = dp[i-1][j]

(1) → Instead of val[i-1] ∞ = 1 for taking current coin.

Longest Common Subsequence

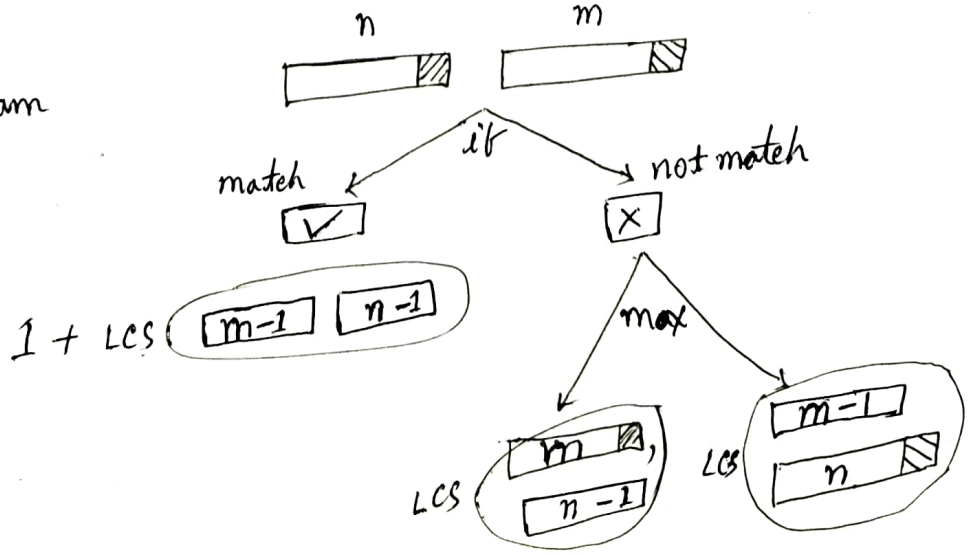
X: (a) (b) c (d) g (h)
Y: (a) (b) e (d) f (h) r
⇒ a b d h

(4) ~

m = len(X)
n = len(Y)

Recursive :

choise diagram

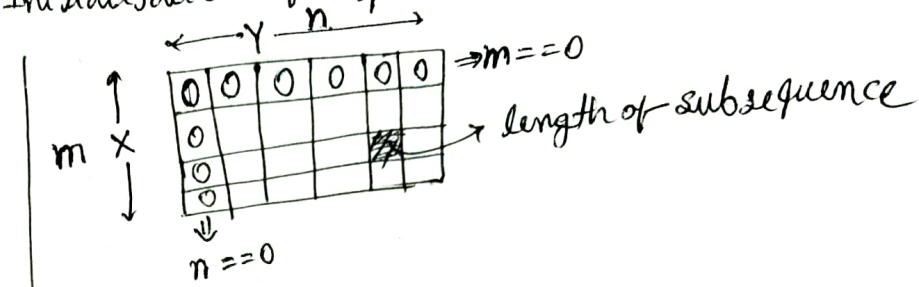


```
def LCS(X, Y, m, n):
    if m==0 or n==0: return 0          ⇒ Base condition
    if X[m-1] == Y[n-1]: return 1 + LCS(X, Y, m-1, n-1)
    else: return max(LCS(X, Y, m, n-1), LCS(X, Y, m-1, n))
```

from choise diagram

⇓ Top down dP.

Base case of recursion ⇒ Initialisation of Top-down DP




```
if m==0 or n==0:
    return 0
```

```
if X[m-1]==Y[n-1]:
    return 1 + LCS(X,Y,m-1,n-1)
```

```
if X[m-1] == Y[n-1]:
    dp[m][n] = 1 + dp[m-1][n-1]
```

```
else:
    return max(LCS(X,Y,m,n-1), LCS(X,Y,m-1,n))
```

```
else:
    dp[m][n] = max(dp[m][n-1], dp[m-1][n])
```

change $m \to i$; $n \to j$

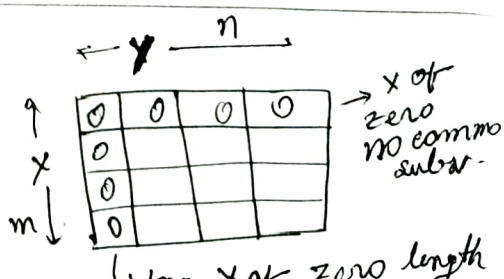
```
dp = [[0]*(n+1) for i in range(m+1)]
for i in range(1, m+1):
    for j in range(1, n+1):
        if X[i-1] == Y[j-1]: dp[i][j] = 1 + dp[i-1][j-1]
        else: dp[i][j] = max(dp[i][j-1], dp[i-1][j])
return dp[m][n]
```

Longest Common Substring:

X = a b c d e ⇒ Continuous common substring.
Y = a b f c e ⇒ length = 2

1 1+1 0 1 1

2 ↳ Discontinuous part 0




```
res = 0
if X[i-1] = Y[j-1]:
    dp[i][j] = 1 + dp[i-1][j-1] ; res = max(res, dp[i][j])
else: dp[i][j] = 0
```

as after discontinuity again assigning dp[i][j] = 0 so storing max length in res.

Length of Shortest Common Super Sequence :

X : AGGTAB (m)
Y : GXTXAYB (n)
} eliminate Common subsequence (X+Y − LCS)

LCS = GTAB ⇒ length(LCS) = 4

AGGTAB + GXTXAYB − GTAB

⇒ m + n − len(LCS)

⇒ 6 + 7 − 4 = 9

Minimum number of Insertion and deletion to convert string X to string Y :

X : heap (m)
Y : pea (n)

p ✗ e a ✗ ⟶ pea
LCS = ea

add : 1
delete : 2

In X, LCS will remain intact and remaining characters will be removed.
as LCS is common subsiquence so optimal sol^n can be found using LCS DP

X ⟶ Y
delete ↘ LCS ↗ add
(via)

LCS : heap / pea > ea

heap ⟶ pea
2 Deletion ↘ ea ↗ 1 add

∴ number of insertion = len(Y) − len(LCS)
∴ number of Deletion = len(X) − len(LCS)

Longest Palindromic Subsequence : input X = "a g b c b a" ⇒ only one string.

How to Find Similarities between LPS and LCS ?

Longest Palindromic Subsequence (LPS)	Longest common subsequence (LCS)
Input : 1 string problem : Longest (optimal) output : length (int)	Input : 2 string problem : Longest (optimal) output : length (int)

the other string may be hidden !

X = "agbcba" , ∴ Y = reverse(X) = "abcbga"

X = a g b c b a
Y = a b c b g a
> LCS = "abcba"
longest palindrom

∴ LPS(X) ≡ LCS(X, reverse(X))

Minimum no. of Deletion/insertion in a string to make it a palindrome:

input = "a g b c b a"

Minimum no. of deletion from input ⇒ maximum length of palindrome.

∴ max len palindrom = "a b c b a" ⇒ no. of deletion = len(input) − len(LPS)

∴ length of LPS $\propto \frac{1}{\text{no. of deletion}}$ = len("a g b c b a") − len("a b c b a")

∴ Minimum no. of deletion = len(input) − LPS = 6 − 5 = 1

Print Shortest Common Supersequence:

input: X = "a c b c f" (m)
Y = "a b c d a f" (n)

Supersequence is a string in which both X and Y are present eleminating multiple values that occurs at same time.

X = a c b c f
Y = a b c d a f > Supersequence = a c b c d a f

In X+Y, LCS exist 2 times, so eleminating 1 LCS we get SCS

Use print LCS tecnique.

dp=

	∅	a	b	c	d	a	f
∅	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1
c	0	1	1	2	2	2	2
b	0	1	2	2	2	2	2
c	0	1	2	3	3	3	3
f	0	1	2	3	3	3	4

LCS = a b c f
LCS = 4

This path of finding LCS is the supersequence which contains both X and Y.

```
i = m; j = n
while i > 0 and j > 0:
    if X[i-1] = Y[j-1]:
        res += X[i-1] # or Y[j-1]
        i -= 1; j -= 1
    else:
        if dp[i][j-1] > dp[i-1][j]:
            res += Y[j-1]
            j -= 1
        else:
            res += X[i-1]
            i -= 1
```

If any of i == 0 or j == 0 then we have to add remaining also as we have to add all X and Y elements

```
while i > 0:
    res += X[i-1]
    i -= 1
while j > 0:
    res += Y[j-1]
    j -= 1
return res[::-1]
```

Longest Repeating Subsequence : input S = "AABEBCDD"

S = AABEBCDD ⇒ ABD

X = AABEBCDD
Y = AABEBCDD
> use LCS with 1 extra condition i != j

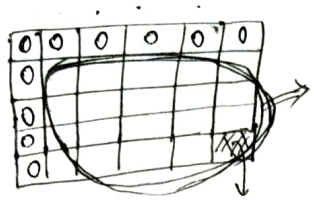
$X_A \to 0, 1 \to (0,1)$
$Y_A \to 0, 1$ ✓

$X_B \to 2, 4$
$Y_B \to 2, 4$ ✓

$X_E \to 3$
$Y_E \to 3$
✗

$X_D \to 6, 7$
$Y_D \to 6, 7$ ✓
$i = 6$
$j = 7$

dp =



return dp[-1][-1]

if X[i-1] == Y[j-1] and i != j :
dp[i][j] = 1 + dp[i-1][j-1]
else:
dp[i][j] = max(dp[i][j-1], dp[i-1][j])

check in the given 2 string one is subsequence of other or not.

if one string is the subsequence of others then the LCS of these 2 string equal to min(len(A), len(B)).

Alternative if len(A) < len(B) : ⇒ A and B are list of given input strings

```
i = 0
while i < len(A):
    if i == len(B): return False
    if A[i] == B[i] : i += 1
    else B.pop(i)
return A[:] == B[:len(A)]
```

Is subsequence problem

Matrix Chain Multiplication : arr[] = [40, 20, 30, 10, 30]

For array of length n, there will be n-1 matrix.

$[\ ]_{2\times3} \times [\ ]_{3\times6} = [\ ]_{2\times6}$ ⇒ Cost or no. of operation = 2×3×6

$[a \times b] \times [b \times c] = [a \times c]$

this dimentions must be same for matrix multiplication.

From are 4 matrices of dimentions 10×20, 20×30, 30×10, 10×30
A B C D

Most cost efficient way is (A*(B*C))*D

cost or no. of operations

$(A*(B*C))*D$

$(40\times20*(20\times30*30\times10))*10\times30$

$(40\times20*20\times10)*10\times30$

$(40\times10)*10\times30$

40×30

$20*30*10 + 40*20*10 + 40*10*30$

$= 26000$

arr = | A | B | C | D |

i K j

solve(arr, i, K)

solve(arr, K+1, j)

arr[i-1] * arr[K] * arr[j]

Recursive: Find i, j

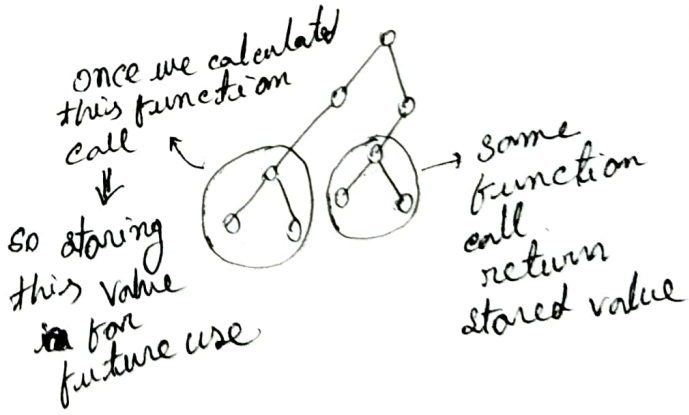
```
def solve(arr, i, j):
    if i >= j: return 0
    for K in range(i, j):
        temp = solve(arr, i, K) +
               solve(arr, K+1, j) +
               arr[i-1] * arr[K] * arr[j]
        ans = min(ans, temp)
    return ans
```

Memoization

dp[1001][1001]

-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

```
ans = IntMax
def solve(arr, i, j):
    if i >= j: return 0
    if dp[i][j] != -1:
        return dp[i][j]
    for K in range(i, j):
        temp = solve(arr, i, K) + solve(arr, K+1, j) +
               arr[i-1] * arr[K] * arr[j]
        ans = min(temp, ans)
    dp[i][j] = ans
    return dp[i][j]
```



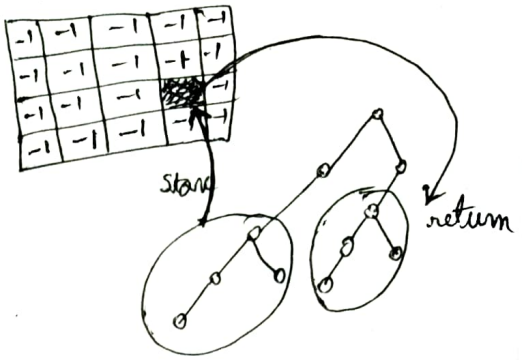
Palindrome Partitioning :

Minimum no. of partitions to make every partition palindrome.

s = "nitik" → No. of partitions

n | iti | k → 2 partitions

Memoization



global dp [constrain of len(s) +1] [constrain of len(s) +1] → every cell -1

```
def solve(s, i, j):
    if i >= j: return 0
    if ispalindrome(s, i, j): return 0
    if dp[i][j] != -1: return dp[i][j]
    ans = sys.maxsize
    for k in range(i, j):
        temp = 1 + solve(s, i, k) + solve(s, k+1, j)
        ans = min(ans, temp)
    dp[i][j] = ans
    return dp[i][j]
```

this line can be further optimized by checking whether solve(s,i,j) calculated before or not

```
def ispalindrome(s, i, j): return s[i:j+1] == s[i:j+1][::-1]
```

optimisation :

```
if dp[i][k] != -1:
    left = dp[i][k]
else:
    left = solve(s, i, k)
    dp[i][k] = left
if dp[k+1][j] != -1:
    right = dp[k+1][j]
else: right = solve(s, k+1, j)
    dp[k+1][j] = right
temp = 1 + left + right
```

Boolean Parenthesization

total no. of ways to make S = True

"|" → or, & → and, ^ → XOR

input ⇒ S = T^F|F

output = 2 ((T^F)|F), (T^(F|F))

S = T | F & T ^ F

i K K K j

if (T|F&)T(^F) ✗

✗ K

i = 0; j = len(s)-1

∴ K = K+2 ✓

So K can come in the index of operators only and i, j at indeces of T and F

(T or F and T) xor (F)

i K j

(i to K-1) (K+1 to j)

XOR

T ^ T → F

F ^ F → F

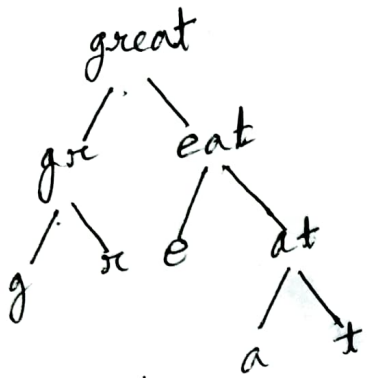
T ^ F → T

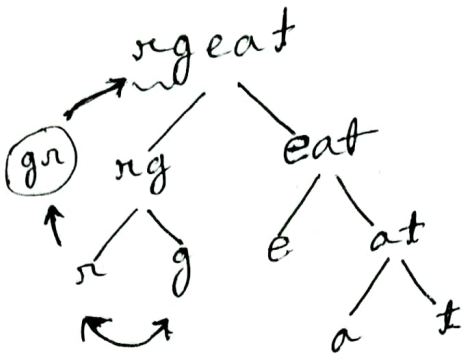
F ^ T → T

Scrambled String: s_1 = "great" , s_2 = "rgeat"

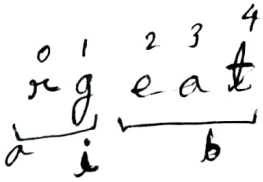
return True or False if s_1 can be scrambled as s_2 or not.

represent s_1 as binary tree





n = 5



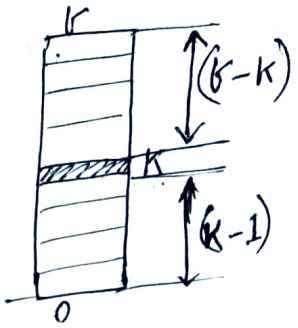
solve(a, b)

Memoization:

```
mp = { }   # hashmap to store calculated values or can also take dp matrix to store
def solve(a,b):
    if len(a) != len(b): return False   # If not equal substring then can not be scrambled.
    if a == b: return True
    if len(a) <= 1: return False   # len == 1 single character and not equal or empty str.
    key = a + "-" + b   # for storing value in map.
    if key in mp: return mp[key]   # previously calculated for so (a,b) that is stored in mp so don't need to calculate again.
    flag = False
    n = len(a)
    for i in range(1, n):
        noswap = solve(a[:i], b[:i]) and solve(a[i:], b[i:])
        swap = solve(a[:i], b[n-i:]) and solve(a[i:], b[:n-i])

        if noswap or swap: # any one is True
            flag = True
            break
    mp[key] = flag
    return mp[key]
return solve(s1, s2)
```

Egg Dropping Problem:

e → no of egg, f → no. of floor.



Solve(e, f)

if egg Break → Solve(e-1, k-1)
if break at k then may break for lowers floors. so check for k-1

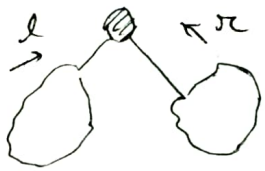
not Break → Solve(e, f-k)
if not break at k then will not break for lower floors. so attempt for upper floors of k

Base Case: (i) If no. of floor is 0 then need 0 attempts.
(ii) if 1 floor need 1 attempt, (iii) If 1 egg, in worst case need to check for all f floors so f attempts.

```
dp = [[-1] * (1001) for i in range(1001)]
def Solve(e, f):
    if f==0 or f==1 : return f
    if e ==1 : return f
    if dp[e][f] != 1 : return dp[e][f]      # previously calculated for solve(e, f) at dp[e,f)
    ans = sys.maxsize
    for k in range(1, f+1):      # try from 1 to f floor, drop every floor and find worst case (max) for that floor; Find min of all 1 to f floors.
        temp = 1+max(
                Solve(e-1, k-1),      → egg break, now we have e-1 remaining eggs and k-1 floors, because above k floor all eggs will break.
                Solve(e, f-k)         → egg not break, now we remain e eggs and f-k floor because before k (included) all eggs will remain
                )
        ans = min(ans, temp)
    dp[e][f] = ans  # storing ans for future use.
return Solve(e,f)
```

Time Complexity = $O(n^2 \times f)$
Space Complexity = $O(k \times n)$

DP on Tree: General Structure of code




```
def solve(node, res):
```

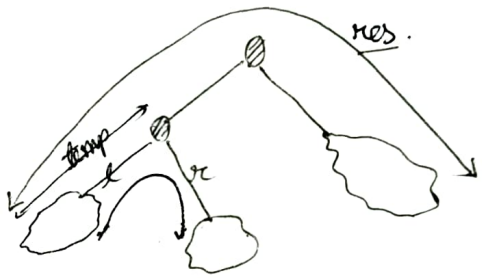
Base Case →
```
if node == None: return 0
```

Hypothesis →
```
l = solve(node.left, res)
r = solve(node.right, res)
```

Induction →
```
temp = calculate temp ans.        → 1 + max(l, r)
ans = max/min(temp, relation)     → 1 + l + r
res = max/min(res, ans)
return temp
```

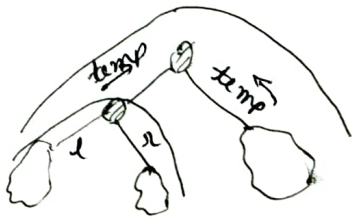
if have to return something else res then return temp

Diameter of Binary Tree:




```
self.res = 0
def solve(root):
B [ if not root: return 0
H [ l = solve(root.left)
    r = solve(root.right)
I [ temp = 1 + max(l, r)
    ans = max(temp, 1 + l + r)      → bar including the current root and left, right as longest path.
    self.ans = max(self.ans, ans)
    return temp                     → if longest path extends further
solve(root)
return self.res.
```

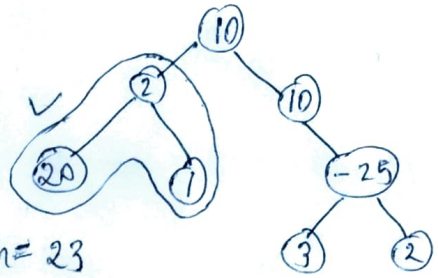
Binary Tree max PathSum:




```
self.res = root.val
def Solve(root):
B [ if not root: return 0
H [ l = solve(root.left)
    r = solve(root.right
I [ temp = max(root.val + max(l, r), root.val)      → max single straight path containing current root and any of max sum subtree or only root if l and r are -ve.
    ans = max(temp, root.val + l + r)               → either temp or total subtree
    self.res = max(self.res, ans)
    return temp
Solve(root)
return self.res
```

Maximum Path Sum of Binary tree from leaf node to leaf node:

```
self.res = root.val
def solve(root):
    if not root: return 0
    l = solve(root.left)
    r = solve(root.right
    temp = max(l,r) + root.val
    ans = max(temp, l+r+root.val)
    self.res = max(ans, self.res)
    return temp
solve(root)
return self.res
```



sum = 23

SAMIR PAUL