Dynamic Programming

General pointers :

We can apply Dynamic Programming when -

1. Repeating subproblems
2. Optimal substructure

COIN CHANGE PROBLEM

PROBLEM STATEMENT

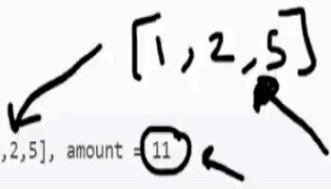
click to scroll output; double click to hide

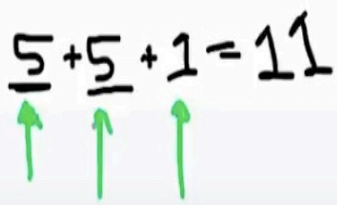
nge

Medium 5680 172 Add to List Share

You are given coins of different denominations and a total amount of money *amount*. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.





Example 1:

```
Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
```

Example 2:

```
Input: coins = [2], amount = 3
Output: -1
```

→ Why not GREEDY approach?

Ex- c = [1, 3, 5, 5] ; A = 7

start with 5:

① 5 + 5 = more than 7

② 5 + 4 = more

③ 5 + 3 = more

④⑤ 5 + 1 = 6 GOOD ✓

⑪ 5 + 1 + 1 = 7 PERFECT ✓

BUT,

if we start with 4 instead:

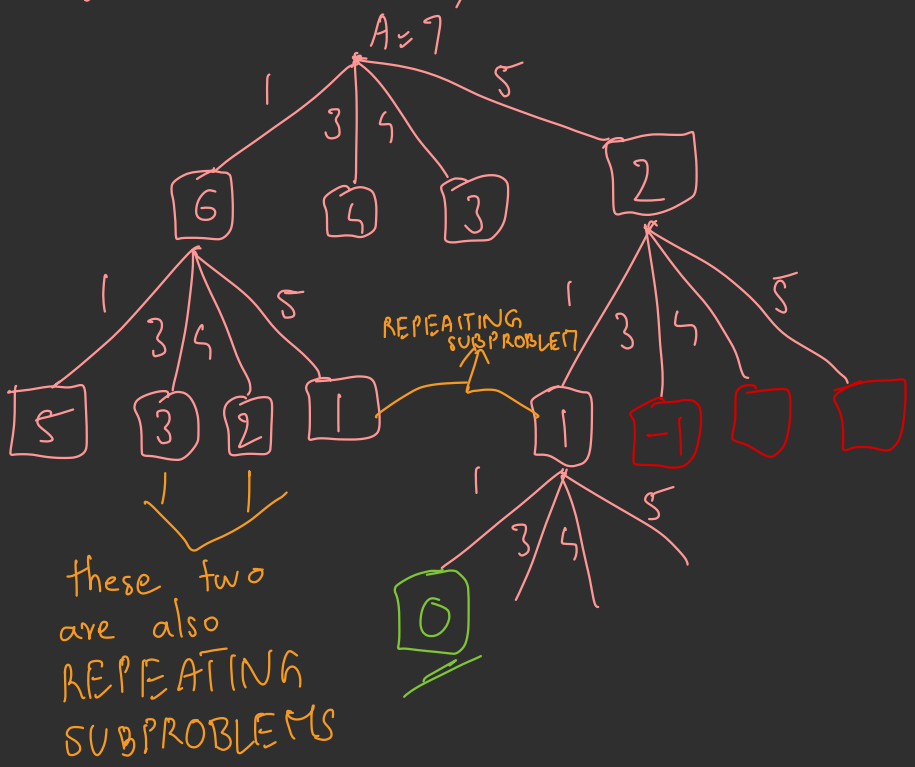
① 4 + 4 = more than 7

② 4 + 3 = 7 PERFECT ✓

∴ By starting with largest number (5), being GREEDY did not give us right result.

Repeating Subproblems-

C = [1, 3, 4, 5] ; A = 7



This is TOP-DOWN approach, which isn't optimal.

So, we use BOTTOM-UP D.P

So, we try DP - Bottom Up. A = 7 ; C = [1, 3, 4, 5]

Instead of starting from 7, we start from 0.

DP[0] → 0 (no. of coins to sum to 0)

DP[1] → 1

DP[2] → 1 coin to sum to 1, but we know that. So,

DP[2] → 1 + DP[1] → 2

↳ no. of coins of value 1

DP[1] ↳ cuz, after taking 1 coin of value 1, (2 - 1 = 1) amount of 1 is remaining to be calculated.

DP[3] → 1

DP[4] → 1

DP[5] → 1

DP[6] → 1 + DP[5] → 2

↳ no. of coins of value 1

DP[5] ↳ cuz, after taking 1 coin of value 1, (6 - 1 = 5) amount of 5 is remaining to be calculated.

(or)

DP[6] → 1 + DP[1] → 2

no. of coins of value 5

DP[1] ↳ cuz, after taking 1 coin of value 5, (6 - 5 = 1) amount of 1 is remaining to be calculated

(continued on next page)

To calculate DP[7]:

when we take coin with value 1 first:

DP[7] = 1 + DP[6]

↳ no. of coins of value 1

DP[6] cuz after you have selected 1 coin of value 1, (7-1=6) an amount of 6 is yet to be calc.

= 1 + 2

= [3] → potential answer

(we don't stop here, we want min., so, we check other coins as starting coin too)

when we take coin with value 3 first:

∴ DP[7] = 1 + DP[4]

no. of coins with value 3

DP[4] cuz after taking 1 coin of value 3, (7-3=4), an amount of 4 is yet to be calculated.

= 1 + 1

= [2] → potential answer

when we take coin with value 4 first:

∴ DP[7] = 1 + DP[3] = 1 + 1 = [2] → potential answer

when we take coin with value 5 first:

∴ DP[7] = 1 + DP[2] = 1 + 2 = [3] → potential answer

Least value of DP[7] is [2]. Therefore, [2] is ANS.

ALGORITHM -

coins = [1, 3, 4, 5]
amount = 7

```
def coinChangeDP(coins, amount):
    dP = [amount+1] * (amount+1)        //creates a list of 8 values like = [8,8,8,8,8,8,8,8]
    dP[0] = 0
    for i in range(1, amount+1):
        for c in coins:
            if (a-c) >= 0:
                dP[a] = min(dP[a], 1 + dP[a-c])

    if dP[amount] == amount+1:
        return -1
    else:
        return dP[amount]
```

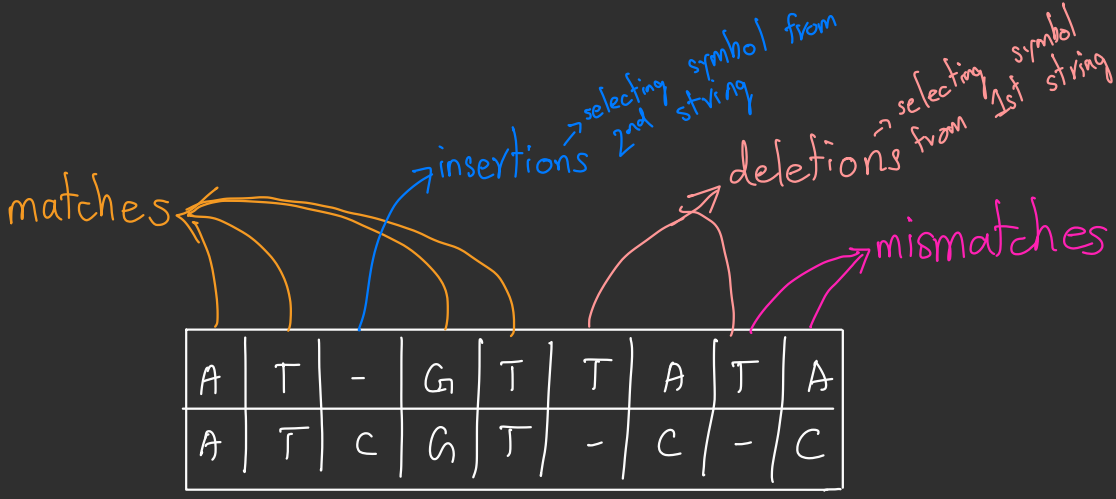
The alignment game

Longest Common Subsequence
Edit distance

① A T G T T A T A
A T C G T C C

② A T - G T T A T A
A T C G T C C (removing C)

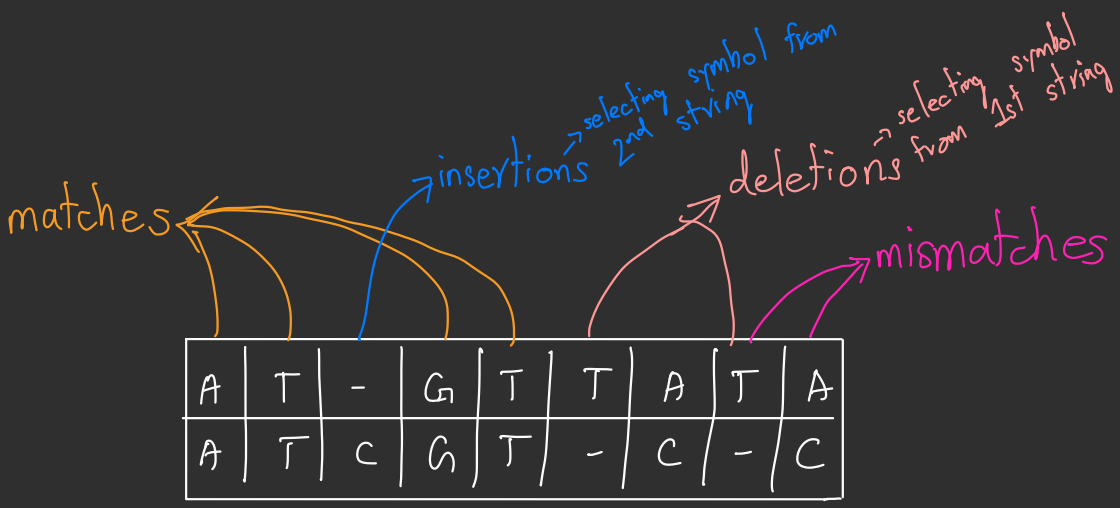
③ A T - G T T A T A
A T C G T - C - C



→ premium for every match

→ penalty for every mismatch & indel
(μ) (σ)

$$\#matches - (\mu) \times \#mismatches - (\sigma) \times \#indel$$



Common Subsequence-

Matches in an alignment of 2 strings forms their COMMON SUBSEQUENCE.

In this case, ATGT

Problem Statement

1143. Longest Common Subsequence

Medium 2553 31 Add to List Share

Given two strings `text1` and `text2`, return the length of their longest common subsequence.

A *subsequence* of a string is a new string generated from the original string with some characters(can be none) deleted without changing the relative order of the remaining characters. (eg, "ace" is a subsequence of "abcde" while "aec" is not). A *common subsequence* of two strings is a subsequence that is common to both strings.

If there is no common subsequence, return 0.

Example 1:

```
Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.
```

Example -

text_1 = abcde

text_2 = ace

HINT - a bcde

a ce

→ First letter is same, so keep it aside.

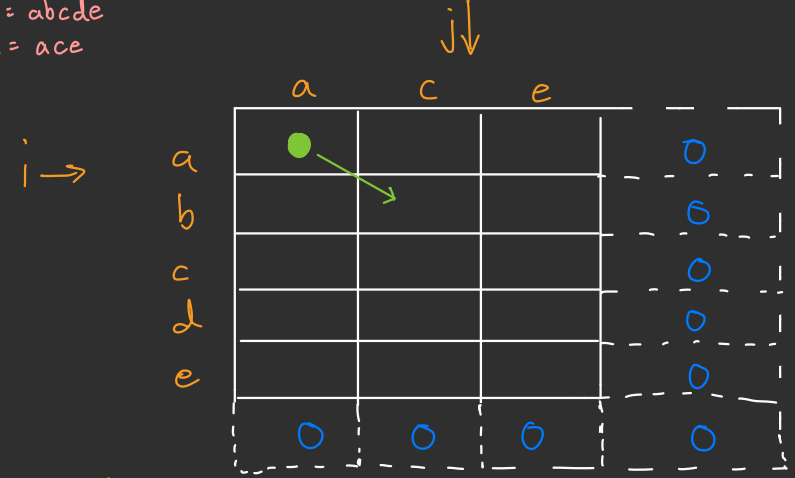
→ Focus on the rest of the string & add 1

→ This behaves like a subproblem

Problem will basically be solved using a 2-dimentional D.P approach.

2-Dimentional Dynamic Programming Approach:

text_1 = abcde
text_2 = ace

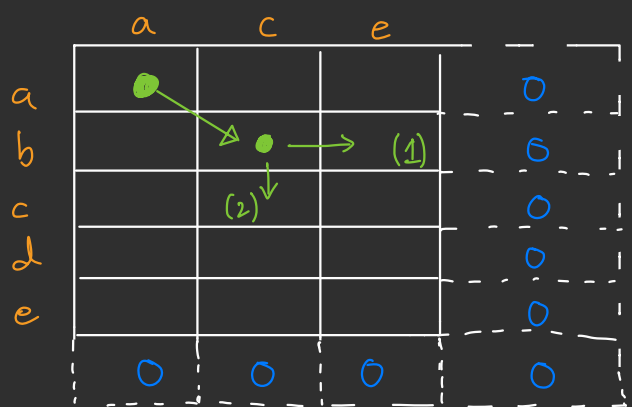


text_1 = i
text_2 = j

Equal: diagonal ↘
orelse: check ↓ & →

ace & abcde:

start with comparing a & a. These 2 are equal, so we move DIAGONALLY. Next, compare 'ce' with 'bcde'.



ce & bcde:

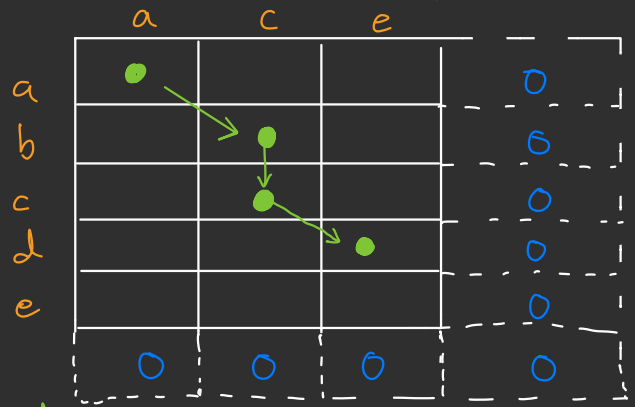
start with comparing c & b. These 2 are NOT equal, so we got 2 options.

GOING RIGHT → compare 'e' with 'bcde';

clearly, 1 is the longest subsequence.

GOING DOWN → compare 'ce' with 'cde'

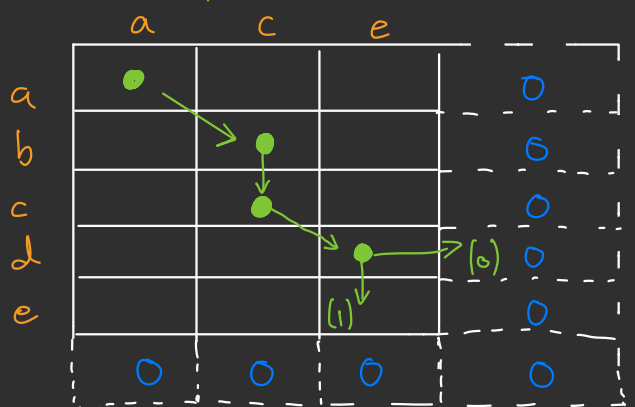
clearly, 2 is the longest subsq.



ce & cde:

start with comparing c & c. These 2 are equal, so move diagonally.

Next, compare 'e' & 'de'.



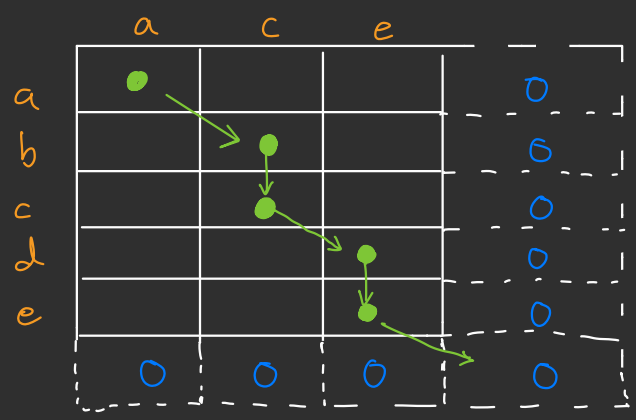
e & de:

'e' & 'd' not equal.

2 options - right or down. Whichever has max common subsequence, & put it in this current pos^n.

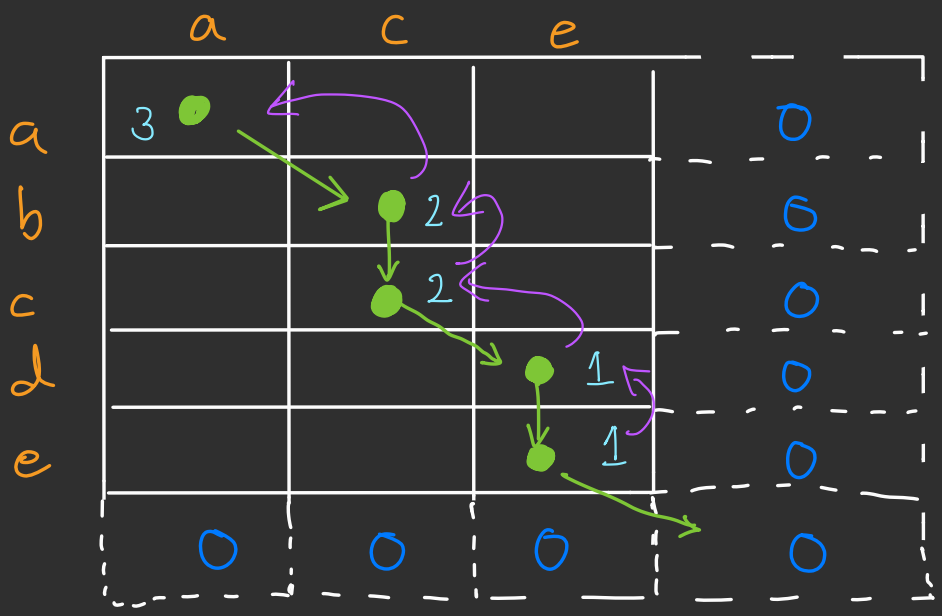
GOING RIGHT → compare <empty> with 'de'
so, 0 longest common subsq.

GOING DOWN → compare 'e' with 'e'.
so, 1 longest common subsq.



'e' with 'e':

compare 'c' & 'e'. These 2 are equal, so move DIAGONALLY. REACHED OUT OF BOUNDS so STOP.



Calculation :

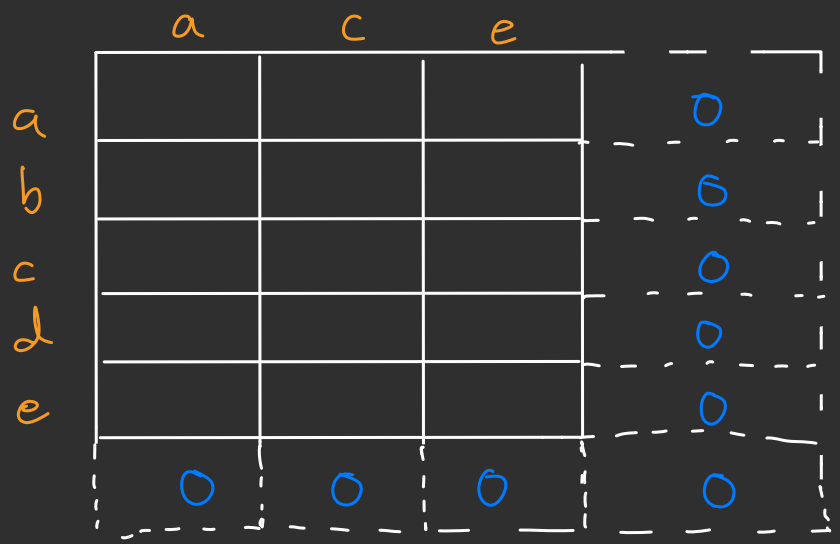
① When you find equal characters, add 1.

② When you find non-equal chars, place previous value in the box.

③ GO BACKWARDS → start from end.

BOTTOM-UP APPROACH.

Longest Common Subsequence will be 3 (first pos^n. in 2-D arr)



Approach for coding -

① Start from bottom

② Calculate value for all cells using the following idea -

(I) If characters match, then place

[1 + (element diagonally down right)]

(II) If characters don't match, then

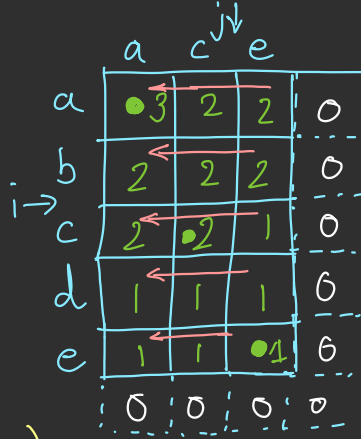
max(element down or element right)

③ Return FIRST element of 2-D array.

ALGORITHM:

```
def longestCommonSubsDP(text1, text2):
    dp = [[0 for j in range(0, len(text2)+1)]
          for i in range(0, len(text1+1)]
```

// initializing 2-D array with min(0) values




```
    for i in range(len(text1)-1, -1, -1):
        for j in range(len(text2)-1, -1, -1):
            if text1[i] == text2[j]:
                dp[i][j] = 1 + dp[i+1][j+1]
            else:
                dp[i][j] = max(dp[i+1][j], dp[i][j+1])
    return dp
```

// starting from end

// max cuz want largest value

Problem statement

72. Edit Distance

Hard 5539 66 Add to List Share

Given two strings `word1` and `word2`, return *the minimum number of operations required to convert* `word1` *to* `word2`.

You have the following three operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

Example 1:

```
Input: word1 = "horse", word2 = "ros"
Output: 3
Explanation:
horse -> rorse (replace 'h' with 'r')
rorse -> rose (remove 'r')
rose -> ros (remove 'e')
```

Basically,
Minimum no.of operations to convert string_1 into string_2.

BASE CASES:

① TEXT_1 : <empty>
TEXT_2 : <empty>

min. no.of operations to convert TEXT_1 to TEXT_2 = 0

② TEXT_1 : abc
TEXT_2 : <empty>

min. no.of operations to convert TEXT_1 to TEXT_2 = length of TEXT_1 (coz 3 DELETIONS)

③ TEXT_1 : <empty>
TEXT_2 : abc

min. no.of operations to convert TEXT_1 to TEXT_2 = length of TEXT_2 (coz 3 INSERTIONS)

(I) EQUAL:

WORD_1 = a b d

WORD_2 = a c d

if WORD_1[i] == WORD_2[j]:

(i+1, j+1)

else:

- insert : (i, j+1)
- delete : (i+1, j)
- replace : (i+1, j+1)

(II) NOT EQUAL:

(a) INSERT:

WORD_1 = a 'c' b d

WORD_2 = a c d

on inserting 'c', we know 2nd pos. is matching, so increment j.

(b) DELETE:

WORD_1 = a b d

WORD_2 = a c d

on deleting 'b', need to increment i.

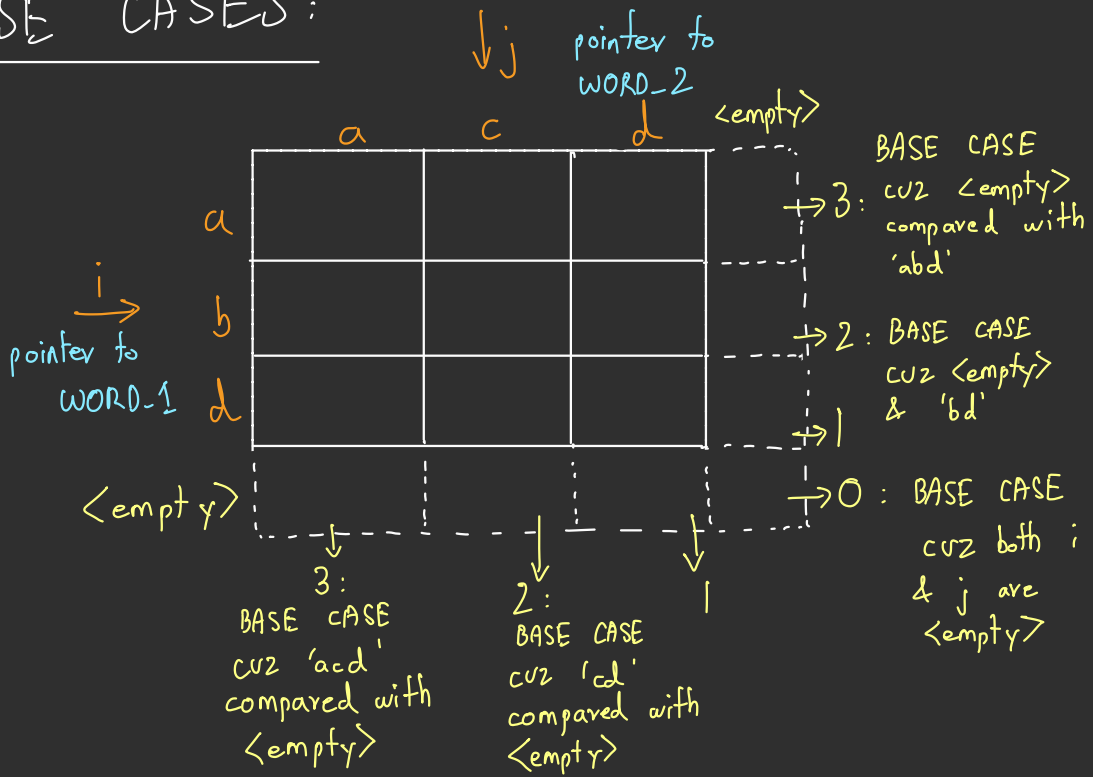
(c) REPLACE:

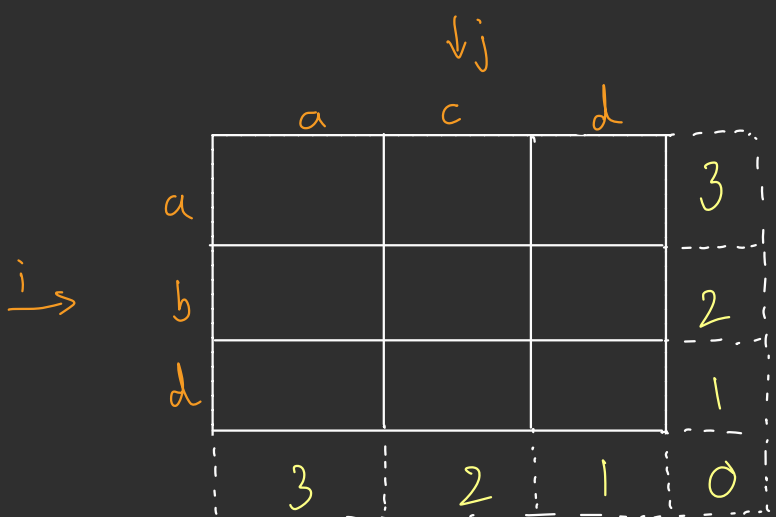
WORD_1 = a b c d

WORD_2 = a c d

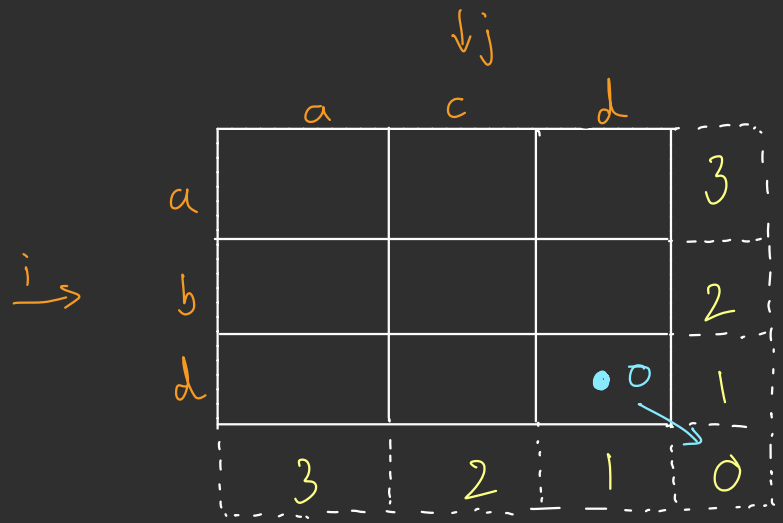
on replacing 'b' with 'c' need to increment both i & j.

BASE CASES:


j
pointer to WORD_2
a
c
d
<empty>
a
b
d
i
pointer to WORD_1
<empty>
3: BASE CASE cuz <empty> compared with 'abd'
2: BASE CASE cuz <empty> & 'bd'
1
0: BASE CASE cuz both i & j are <empty>
3: BASE CASE cuz 'acd' compared with <empty>
2: BASE CASE cuz 'cd' compared with <empty>
1

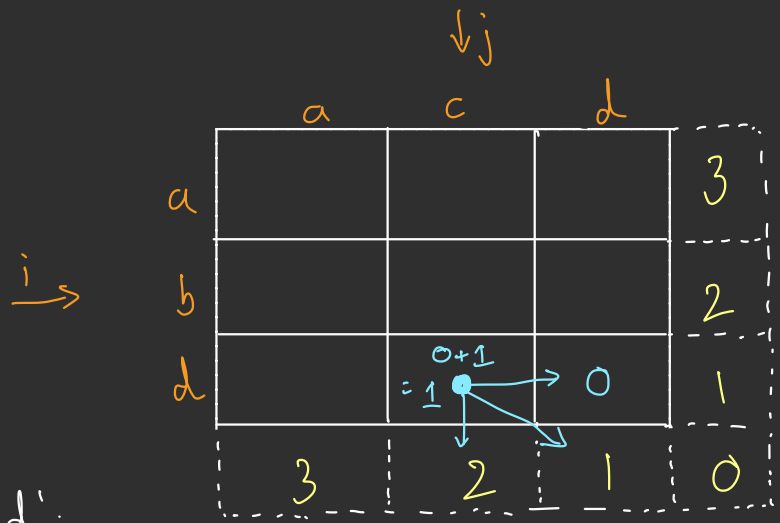

j
a
c
d
i
a
b
d
3
2
1
3
2
1
0

Bottom-Up Dynamic Programming:



'd' & 'd':

EQUAL → so, take diagonal right bottom



'c' & 'd':

NOT-EQUAL: so, minimum of insert, delete & replace.

(i, j+1) (i+1, j) (i+1, j+1)

AND add 1 (cuz 1 operation done)

DIFFERENCE B/W LONGEST COMMON SUBSEQUENCE:

word 1 = abcde

word 2 = ace

		a	c	e	
a		3	2	2	0
b		2	2	2	0
c		2	2	1	0
d		1	1	1	0
e		1	1	1	0
		0	0	0	0

EQUAL :

$1 + (i+1, j+1)$

not EQUAL:

$\max[(i, j+1), (i+1, j)]$

ANS → 3

AND EDIT DISTANCE :

word 1 : a b d

word 2 : a c d

		a	c	d	
a		1	2	2	3
b		2	1	1	2
d		2	1	0	1
		3	2	1	0

EQUAL:

$(i+1, j+1)$

NOT EQUAL :

$\min[(i, j+1), (i+1, j), (i+1, j+1)] + 1$

ANS → 1

When to add 1 in calculations:

In each case, you MAXIMIZE PROFIT when the elements being checked are EQUAL.

So, in LCS you MAXIMIZE PROFIT by adding 1 to [(i+1, j+1) element] if the elements are EQUAL. cuz you are trying to get big value so add 1.

In Edit Dist, you MAXIMIZE PROFIT by NOT adding 1 to [(i+1, j+1) element] if the eles. are EQUAL. cuz you are trying to get small value so DO NOT add 1.

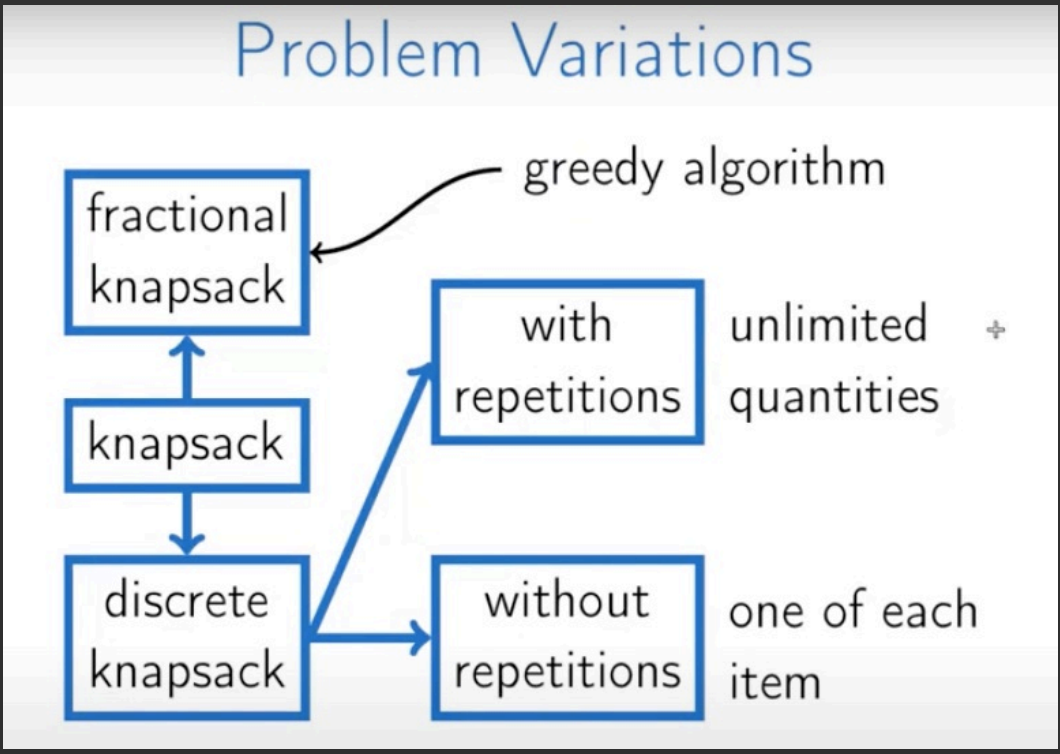
ALGORITHM -

```
def editDistanceDP(text1, text2):
    dp = [[max(len(text1), len(text2))+1 for j in range(len(text2))]
          for i in range(len(text1))]          // initializing 2-D array with MAX values

    for j in range(0, len(text2)+1):
        dp[len(text1)][j] = len(text2) - j
    for i in range(0, len(text1)+1):           // covering base cases
        dp[i][len(text2)] = len(text1) - i

    for i in range(len(text1)-1, -1, -1):      // starting from end
        for j in range(len(text2)-1, -1, -1):
            if text1[i] == text2[j]:
                dp[i][j] = dp[i+1][j+1]
            else:
                dp[i][j] = 1 + min(dp[i][j+1],
                                   dp[i+1][j], dp[i+1][j+1])
                                               // min cuz we want least
    return dp[0][0]
```

Knapsack Problem



Fraction Knapsack → GREEDY

Discrete Knapsack → DYNAMIC PROGRAMMING

FRACTIONAL KNAPSACK PROBLEM ALGORITHM (REVISION):

```
def knapsackGreed(M, weight, profit):
    d={}
    for i in range(0,len(weight)):          // creating dictionary of form
        t=[]                                    p/w : [w, p]
        t.append(weight[i])                 for each item
        t.append(profit[i])
        d[profit[i]/weight[i]] = t

    dk = list(d.keys())                     // sorting the dictionary
    dk.sort()
    sortedD = {i : d[i] for i in dk}
    res = 0
    for i in sortedD:                       // sortedD[i][0]=w
        if M>0 and sortedD[i][0]<M:             sortedD[i][1] = p
            M -= sortedD[i][0]
            res += sortedD[i][1]
        else:
            break
    if M>0:
        res += M*(sortedD[i][1]/sortedD[i][0])
                                            // res = res + M x (Pi)/(wi)
    return res
```

KNAPSACK PROBLEM: (DISCRETE) (LIMITED ITEMS)

(a.k.a 0/1 knapsack problem)

$p = [1, 2, 5, 6]$ capacity, $m = 8$

$w = [2, 3, 4, 5]$ no. of items, $n = 4$

(capacity) M, j↓

i→	p	w	0	1	2	3	4	5	6	7	8
			0	0	0	0	0	0	0	0	0
	1	2	1	0	1	1	1	1	1	1	1
	2	3	2	0							
	5	4	3	0							
	6	5	4	0							

When i=1: (ignore objects BELOW i)

- means only 1 item exists (of $w=2$, $p=1$)
- when $j=1$ (capacity is 1), item will NOT fit (as $w=2$)
- when $j=2$ (capacity is 2), item FITS & $p=1$
- for other j, capacity increases but considering only 1 item so 'p=1' for all other (j in i=1)

(capacity) M ↓j

i→			0	1	2	3	4	5	6	7	8
p	w	0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0								

when i=2: (include i=1 also)

- j=3 (capacity is 3), w_2 fits so p=2
  (cuz 3)
- for j=1 & j=2, val from previous row
- j=5, both w_1 & w_2 (w=5) can FIT & p=3
- SAME WAY FILL FOR i=3

(capacity) M j↓

i→

P	w	0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3
5	4	3	0	0	1	2	5	5	6	7
6	5	4	0	0	1	2	5	6	6	7

FORMULA :

$$V[i][j] = \max\left(V[i-1][j],\ V[i-1][j-w_i] + P_i \right)$$

i=4 : ① j=0 : $\max(V[3][0], V[3][-ve] + 6) \Rightarrow \max(0, \text{undefined})$

$\therefore$ i=4, j=0 → 0

② j=1 : $\max(V[3][1], V[3][-ve] + 6) \rightarrow 0$

③ j=2 : 1 ; ④ j=3 : 2 ; j=4 : 5

⑤ j=5 : $\max(V[3][5], V[3][0] + 6) \Rightarrow \max(5, 6)$

$\therefore$ 6

⑥ j=6 : 6 ; ⑦ j=7 : 7 ; ⑧ j=8 : 8

ALGORITHM: CODE-

① Initialize the 2-D Array, cover base cases.

rows → i → no. of items

columns → j → capacity (from 0 to CAPACITY(M))

② Go through each element in 2-D array & populate based on below conditions.

(i) if $j < w[i]$:

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

(ii) else :

$$dp[i][j] = \max(dp[i-1][j] \text{ or } p[i] + dp[i-1][j-w[i]])$$

③ Return $dp[len(w)][M]$

↓ no. of items ↳ max. capacity

ALGORITHM-

M → capacity ; w = [weights]
p = [profits]

```
def discreteKnapsackDP(M, w, p):
    dP = [[0 for j in range(0, M+1)] for i in range(0, len(w)+1)]
                                    //initializing 2-D array with 0s.
    wl = [0], pl = [0]
    for i in range(0, len(w)):
        wl.append(w[i]) & pl.append(p[i])   //to add '0' to 1st pos. in w & p

    for i in range(1, len(w)+1):
        for j in range(1, M+1):
            if j < w[i]:
                dP[i][j] = dP[i-1][j]
            else:
                dP[i][j] = max(dP[i-1][j], pl[i] + dP[i-1][j-wl[i]])
                                    //following conditions

    return dP[len(w)][M]
```

(write properly)

but,

When goal is to find large values (LongestCommonSubseq. & DiscreteKnapsack)

→ initialize array to 0.

→ use max in the programs

When goal is to find small values (CoinChange & EditDistance)

→ initialize array to max values

→ use min in the program