25/01

• DP on String → comparison → replacements

25k

* Longest Common Subsequences:

• what is subsequences?

abc → a, b, c, ab, bc, ac, abc, ""

maintain order and both are consecutive

'ac' are not consecutive but maintaining the order of 'abc'.

→ maintains the order in the string, it is called subsequence.

* → To print all the subsequences, using power set or recursion.

n [string length] → 2^n [no. of subsequences]

eg. S1 = abcd , S2 = baca

a	b → length = 1
b	a
c	c
d	ba
ab	bc
ac	ac → length = 2.
ad	aa
bc	ca
bd	
cd	
⋮	

→ ac is longest common subsequences.

Brute force: → exponential in nature.

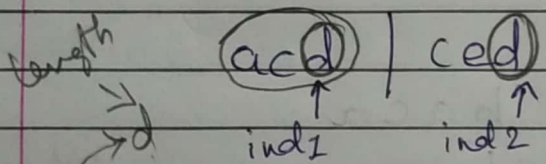
- Generate all subsequences & compare on way

eg. acd / ced

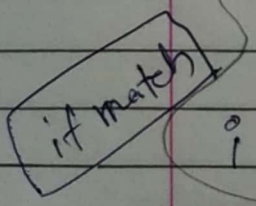
a	e
c	e
d	d
ac	ce
ad	cd
cd	ed
acd	ced

- Rules to write:

1. express in term of indexes ⇒ (ind1, ind2)
2. explore possibility of that index
3. take the best among them.

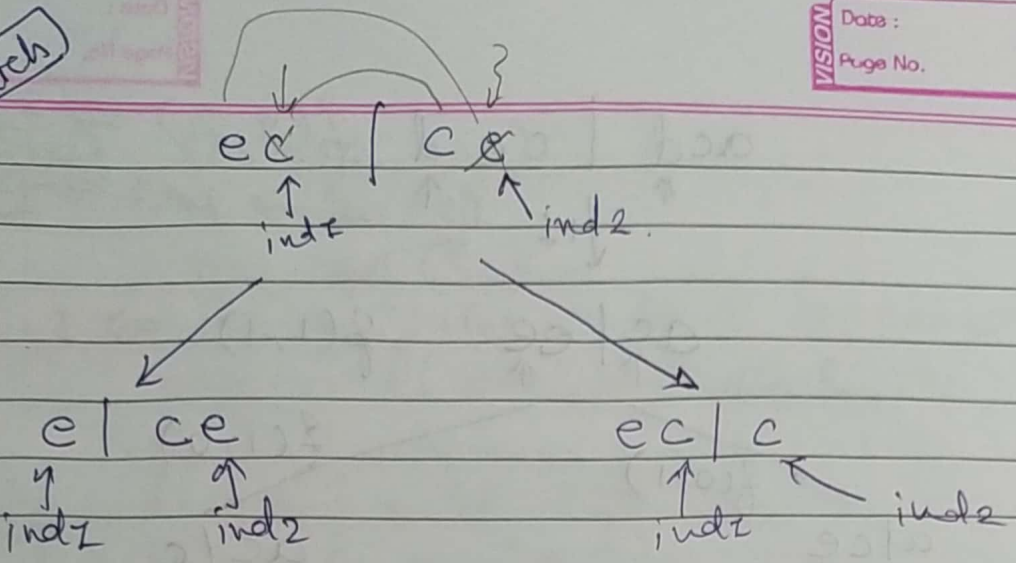


1 + ac / ce




```
if(S1[ind1] == S2[ind2])
    return 1 + f(ind1-1, ind2-1)
```

non-match



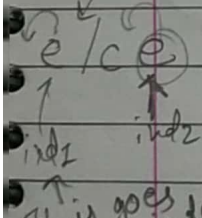
if not match

→ 0 + max(f(ind1 - 1, ind2), f(ind1, ind2 - 1))

the answer added will be zero for sure because there is no length that you are matching.

either solve this guy or either solve this guy whichever gives you the maximum answer is going to be the max answer.

f(ind1, ind2):

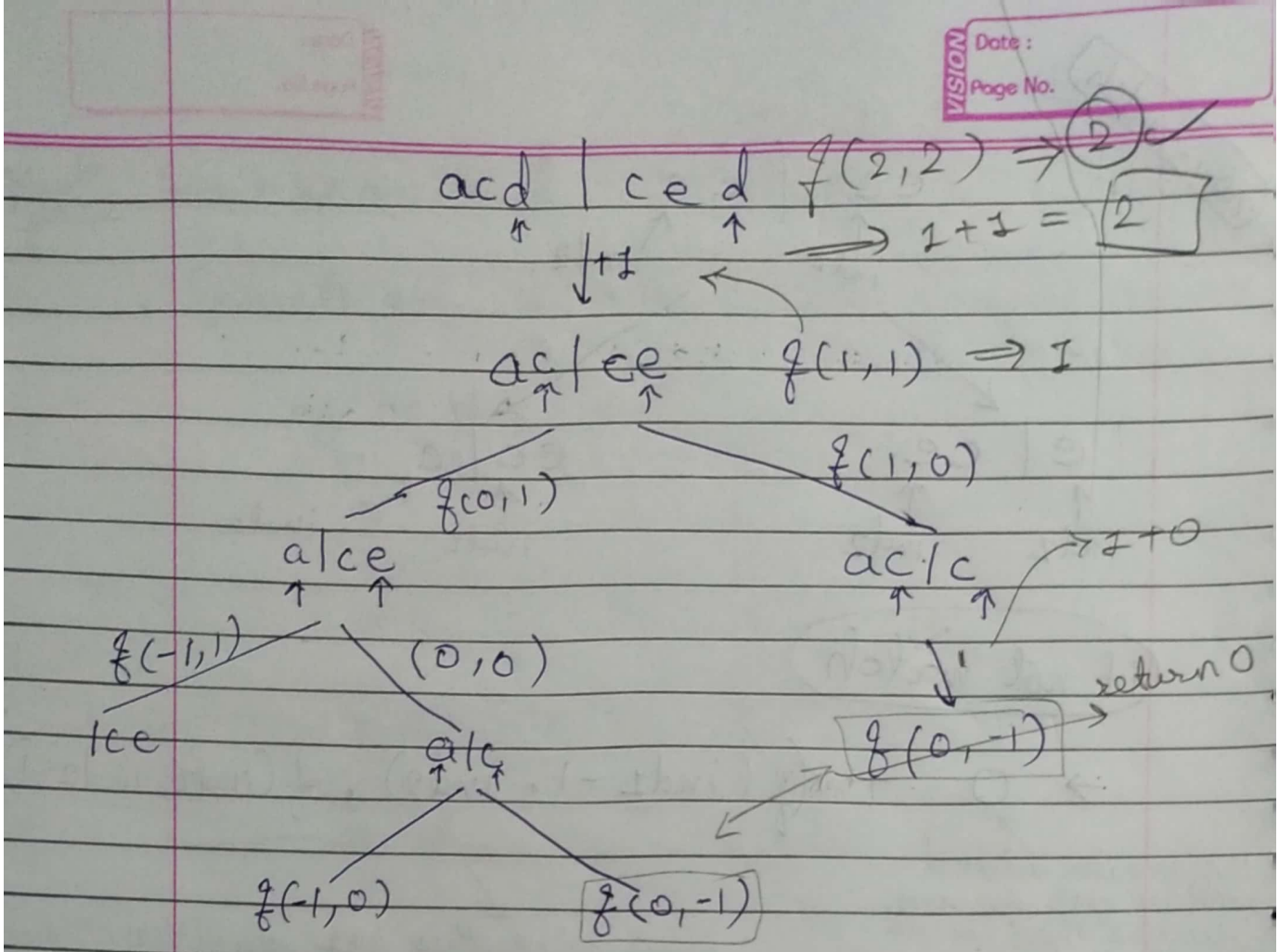


if it is goes negative index.

if (ind1 < 0 || ind2 < 0)
return 0

match { if (s1[ind1] == s2[ind2])
return 1 + f(ind1-1, ind2-1)

return 0 + max(f(ind1-1, ind2), f(ind1, ind2-1))



⇒ Recursion : $2^n \times 2^m$ ⇒ exponential.

optimised → memoization

If overlapping subproblems use memoization → ILU

⇒ f(i, j) → lcs of a
s1[0 ..... i] & s2[0 ..... j]

i → n
j → m

dp[n][m] → initialize with (-1).

```
def lcs(s1, s2):
    n = len(s1)
    m = len(s2)
    dp = [[-1]*m for i in range(n)]
    return f(n-1, m-1, s1, s2, dp)

def f(i, j, s1, s2, dp):
    if (i<0 or j<0):
        return 0
    if (dp[i][j] != -1) return dp[i][j]
    if (s1[i] == s2[j]):
        return 1 + f(i-1, j-1, s1, s2)
    else:
        return 0 + max(f(i-1, j, s1, s2),
                       f(i, j-1, s1, s2))
```

If:

→ `dp[i][j] = 1 + f(i-1, j-1, s1, s2, dp)`

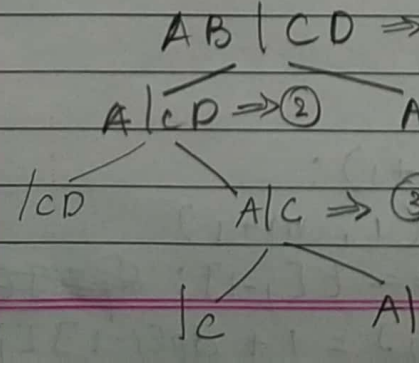
else:

→ `dp[i][j] = 0 + max(f(i-1, j, s1, s2, dp), f(i, j-1, s1, s2, dp))`

T.C ⇒ O(N*m)

S.C. ⇒ O(N*m) + O(N+m)

Auxiliary stack space
↳ match ⇒ 1+ ...
↳ not match ⇒ 2 directions

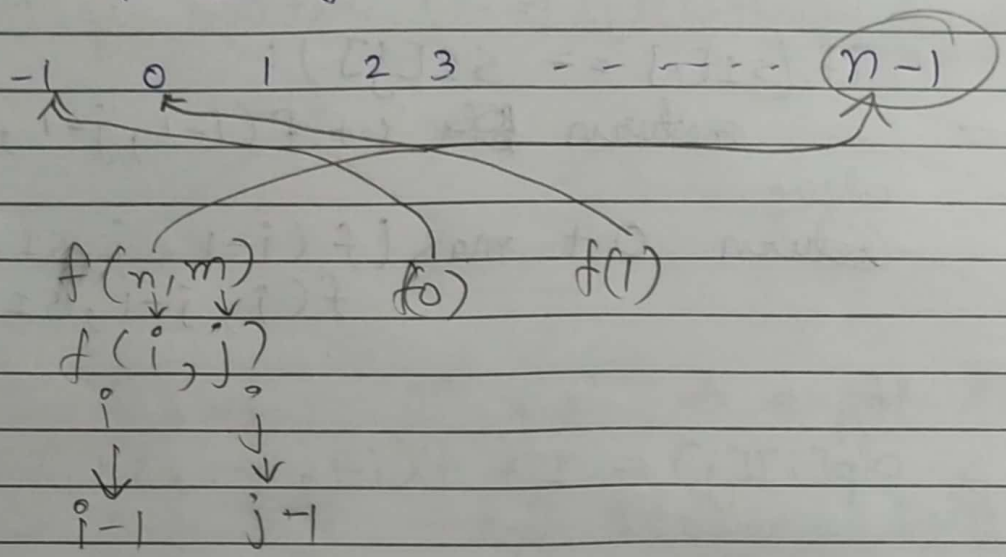


2+2
=4 steps.

• Tabulation / Bottom Up approach:

1. copy the base case.
2. write down the parameter in opposite fashion
3. copy the recurrence.

* • Shifting of index:



base case: if i == 0 or j == 0 :
return 0.

```
def lcs(s1, s2):
    n,m = len(s1), len(s2)
    dp[n+1][m+1] = [[-1]*m for i in range(n)]
    for i in range(n):
        dp[i][0] = 0
    for j in range(m):
        dp[0][j] = 0
    for i in range(1, n+1):
        for j in range(1, m+1)
            if s[i-1] == t[j-1]:
                dp[i][j] = 1 + dp[i-1][j-1]
```

dp[i] → will be 'cur'

for ——
  for ——
    if ——
    else:
      dp[i][j] = max (dp[i-1][j], dp[i][j-1])

return dp[n][m].

* Space optimization.

```
def lcs (s1, s2):
    n, m = len(s1), len(s2)

    prev = [0]*(m+1)
    cur = [0]*(m+1)

    for j in range(m+1):
        prev[j] = 0

    for i in range(1, n+1):
        for j in range(1, m+1):
            if s1[i-1] == s2[j-1]:
                cur[j] = 1 + prev[j-1]
            else:
                cur[j] = max(prev[j], cur[j-1])

        prev = cur

    return prev[m].
```