→ storage.

→ check if ans is pre-calculated

→ store your ans in storage before returning.

Q) Given 2 strings, find the length of longest common subsequence in 2 strings. [L.C.S]

① s1 : a b b c d g f

s2 : b a c d e g f

ans = 5

② s1 : k l a g r i p

s2 : l g i g k m

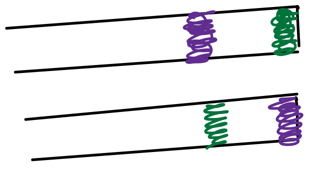
ans = 3

idea. → Consider all subsequence of s1 & s2 and compare to get longest common subsequence.

LCS(s1(0-N-1) , s2(0-m-1))

s1[n-1] == s2[m-1]

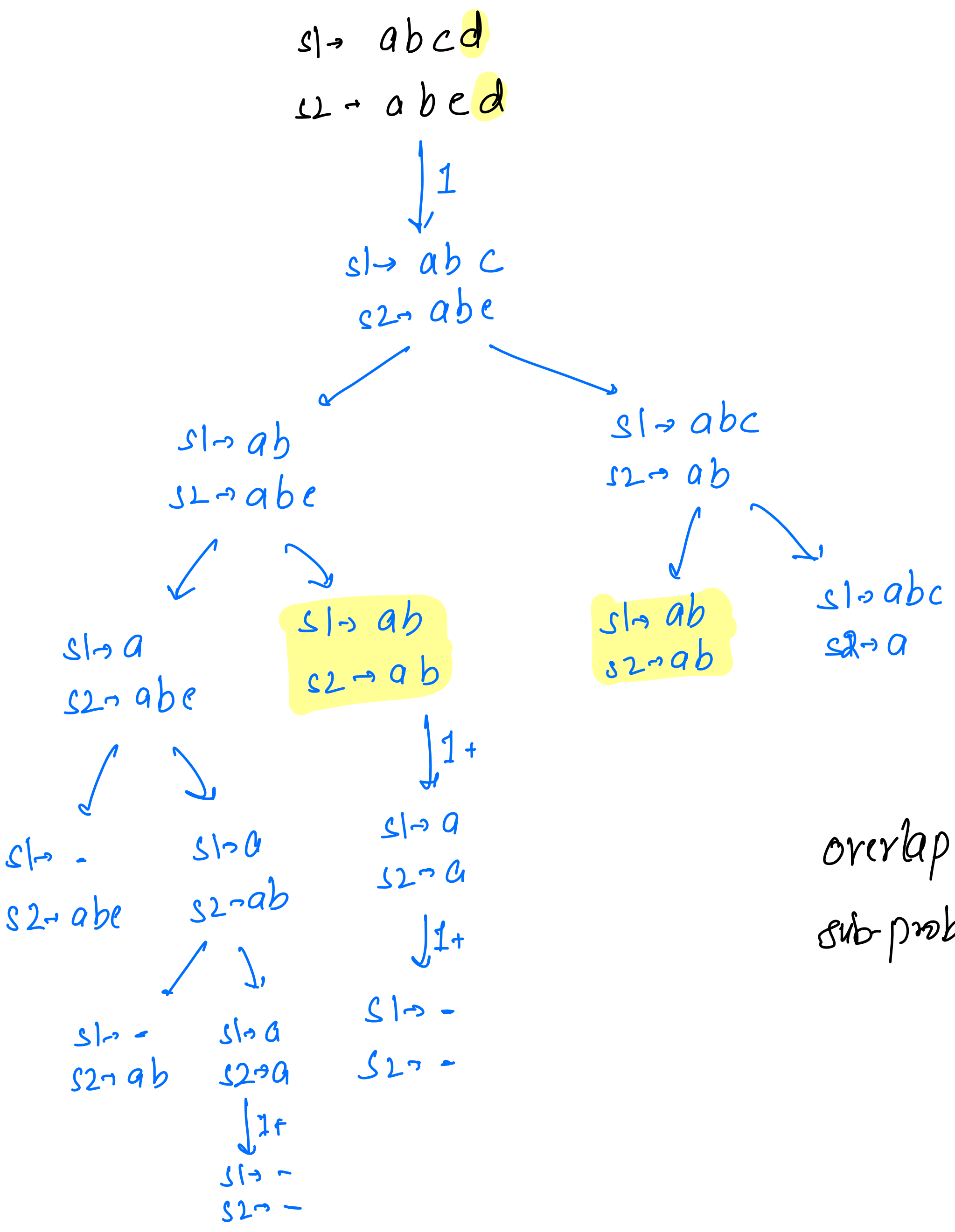
not equal.



1 + LCS(s1(0,N-2), s2(0,m-2))

max [LCS(s1(0,N-2), s2(0,m-1)), LCS(s1(0,N-1), s2(0,m-2))]

optimal substructure



ans = 3.

overlapping sub-problems ✓

$$\text{LCS}(s1, s2, \underset{N-1}{\underset{\downarrow}{i}}, \underset{m-1}{\underset{\downarrow}{j}}) = \begin{cases} s1[i] == s2[j] & 1 + \text{LCS}(s1, s2, i-1, j-1) \\ s1[i] \neq s2[j] & \max\left[\begin{array}{l} \text{LCS}(s1, s2, i-1, j), \\ \text{LCS}(s1, s2, i, j-1) \end{array}\right] \end{cases}$$

int dp[N][m]

pseudo-code.

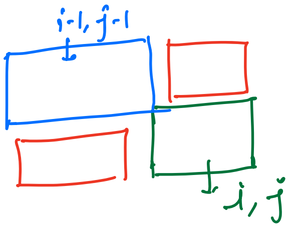
```
int dp[N][m] // initialise -1
                     N-1  m-1
int lcs( s1, s2, i, j){
    if(i<0 || j<0){return 0} // empty string
    if( dp[i][j] != -1) { return dp[i][j] }
    if ( s1[i] == s2[j] ){
        dp[i][j] = 1 + lcs( s1, s2, i-1, j-1) ;
    }
    else {
        dp[i][j] = Max( lcs(s1,s2, i-1,j) ,lcs(s1,s2,i,j-1))
    }
    return dp[i][j] ;
}
```

T.C → O(N*m)
S.C → O(N*m)

Bottom-up.

equal . $dp[i][j] = 1 + dp[i-1][j-1]$

unequal → $dp[i][j] = Max(dp[i-1][j], dp[i][j-1])$



j

		-	M	A	I	C	A
		0	1	2	3	4	5
- i	0	0	0	0	0	0	0
K	1	0	0	0	0	0	0
A	2	0	0	1	1	1	1
I	3	0	0	1	2	2	2
Y	4	0	0	1	2	2	2
A	5	0	0	1	2	2	3

S1 → KAIYA
S2 → MAICA

dp[N+1][M+1]

pseudo-code.

```
int dp[N+1][m+1]
// initialise 0th row & 0th col with 0
for( i=1 ; i ≤ N ; i++){
    for( j=1 ; j ≤ m ; j++){
        if ( s[i-1] == s[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];
```

T.C → O(N*m)
S.C → O(N*m)

Edit Distance

Given two string s1 & s2. Convert s1 to s2 by performing some operations.

[find minimum cost to convert s1 to s2.]

insert → $cost_i$
delete → $cost_d$
replace → $cost_r$

$cost_i \rightarrow 2$, $cost_d \rightarrow 2$, $cost_r \rightarrow 3$

① $s1 = a\overset{b}{\downarrow}c$
$s2 = a\ b\ c$

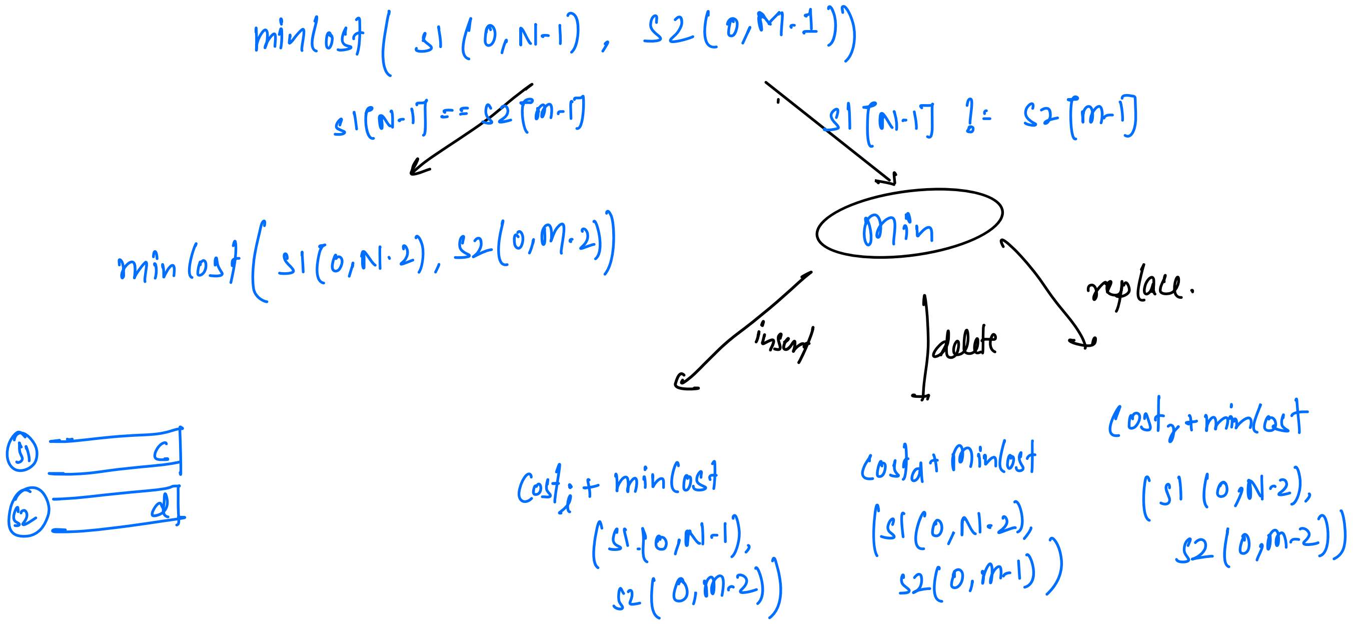
ans = 2

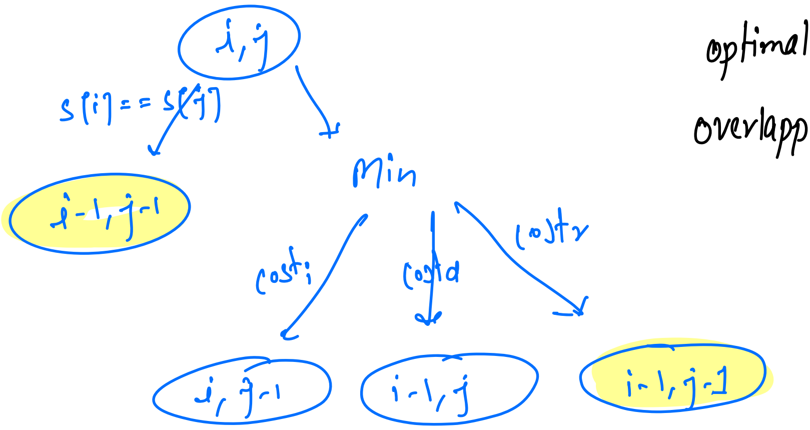
② $s1 = a\ b\ \overset{e}{\overset{\uparrow}{c}}\ \cancel{d}$
$s2 = a\ b\ e$

ans = 5

③ $s1 = a\overset{b}{\downarrow}c\ \overset{g}{\cancel{d}}\ x\ \cancel{y}$
$s2 = a\ b\ c\ g\ x$
I R D

ans = 7.


minCost (s1 (0, N-1) , S2 (0, M-1))
s1[N-1] == s2[m-1]
s1[N-1] != s2[m-1]
min Cost (s1(0,N-2), s2(0,m-2))
Min
insert
delete
replace.
Cost_i + minCost (s1(0,N-1), s2 (0,m-2))
Cost_d + MinCost (s1(0,N-2), s2(0,m-1))
Cost_r + minCost (s1 (0,N-2), s2 (0,m-2))
s1
c
s2
d



optimal substructure

overlapping sub-problems.

#pseudo-code

```
    int dp[N][M] // initialise -1
int minCost( string s1, string s2, int i, int j){
        if( i < 0  && j < 0) { return 0}
        else if ( i < 0){
            //only option is to insert
            return cost_i * (j+1);
        }
        else if ( j < 0){
            return cost_d * (i+1);
        }

        if (dp[i][j] != -1) { return dp[i][j] }

        if ( s1[i] == s2[j] ){
            dp[i][j] = minCost( s1, s2, i-1, j-1);
        }
        else {
            dp[i][j] = Min [ cost_i + minCost(s1, s2, i, j-1)
                             cost_d + minCost( s1, s2, i-1, j)
                             cost_r + minCost(s1, s2, i-1, j-1) ]
        }
        return dp[i][j]
}
```

[T.C → O(N*m)
S.C → O(N*m)]

[#bottom-up #todo]

Wildcard Pattern Matching

check if s1 & s2 are matching ?

s2 can contain ? and * .

? → match with any single character

* → match with 0 or any no. of characters.

① s : a b a cd ⇒ true.
p : a b a c d

② s : a b a c d ⇒ true.
p : a ? a ? d

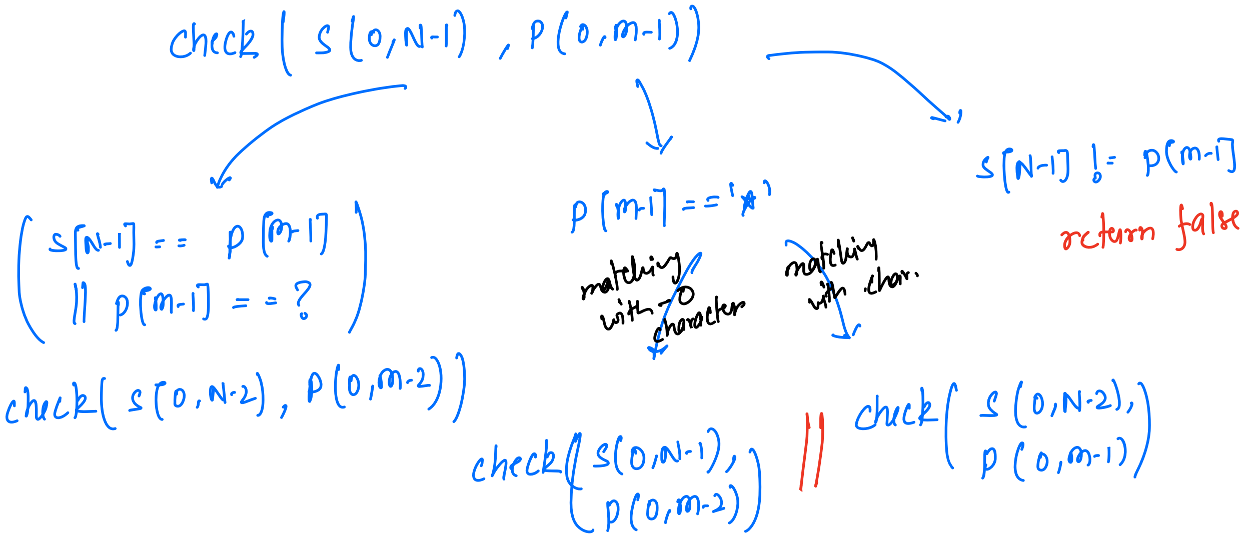
③ s : a b b a c ⇒ true.
p : a * c

④ s : x b b z z c ⇒ false.
p : x * z * x

⑤ s : x b b z z c ⇒ true.
p : x * z * *

⑥ s : x b b z z ⇒ false.
p : x * z * * * ? z

s → ab
p → ab*


check (s (0,N-1) , p (0,m-1))

(s[N-1] == p[m-1] || p[m-1] == ?)

check(s[0,N-2), p(0,m-2))

p[m-1] == '*'

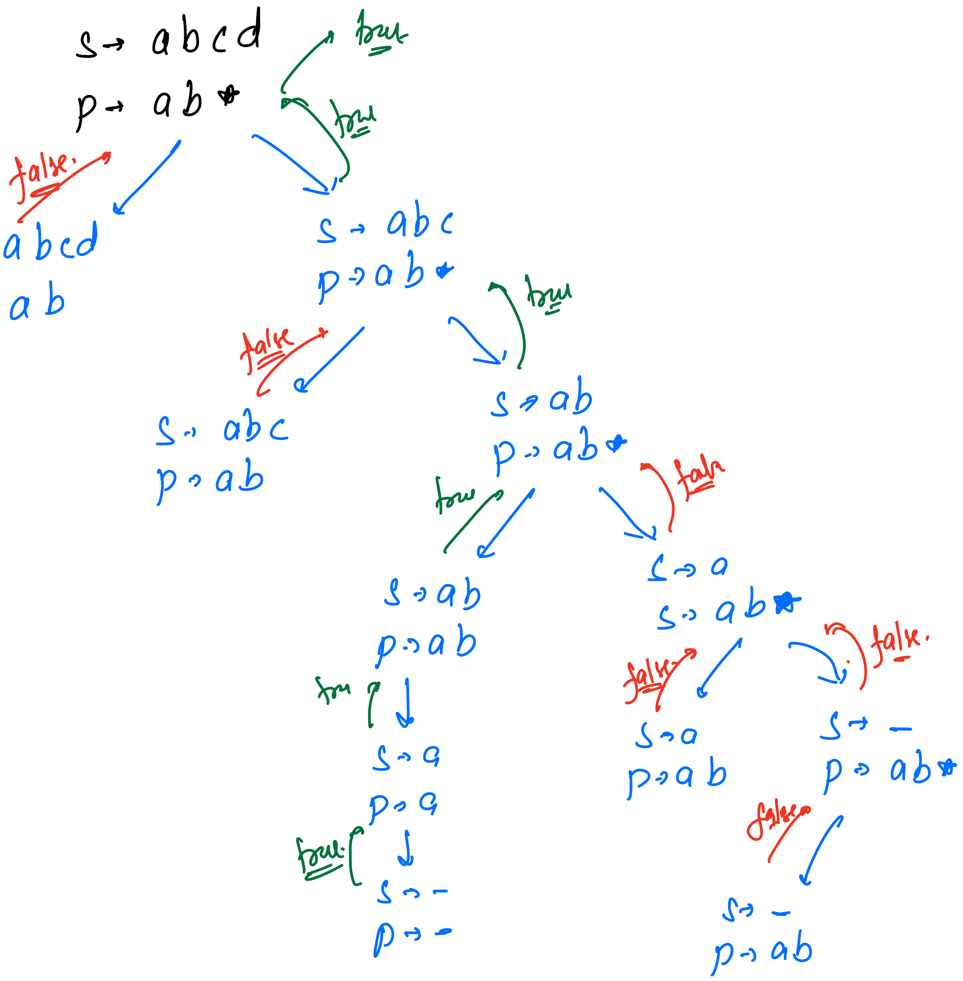
matching with 0 character

matching with char.

check(s(0,N-1), p(0,m-2)) || check(s(0,N-2), p(0,m-1))

s[N-1] != p[m-1]

return false



```
dp[i][j] = [ s[i] == P[j] || p[j] == '?'   →   dp[i-1][j-1]
           [ p[j] == '*'                   →   dp[i-1][j] || dp[i][j-1]
           [ s[i] != p[j]                  →   return false.
```

```
if ( i < 0 && j < 0 ) { return true }
else if ( j < 0 ) return false ;
else if ( i < 0 ) {
{
        if only '*' are remaining in pattern → true
        otherwise → return false
}
```

{ # code #todo. }