Todays Content:

a) Longest common subsequence

b) Edit distance

c) Regex Matching

Dp ?

a. Recursion ✓

b. SubProblems Overlapping:

Dp says : Solve a single subProblem only once, store it & reuse it.

Steps:

1. dp table = invalid initialization
2. SubProblem 1st time:

   Solve it, store it, return it

   SubProblem >= 2nd time,

   re-use it.
3. TC: O(# No: of sub Problems) * (Time taken for each)

10. Given 2 strings $S_1 : N$ & $S_2 : M$

Find length of longest common subsequence → {order of indices}

→ {Pick any indices we want}

Eg1:

	0	1	2	3	4	5	6
S_1 :	f	g	d	c	b	e	a
S_2 :	f	g	e	h	c	a	

fgeca → S_1* missing order

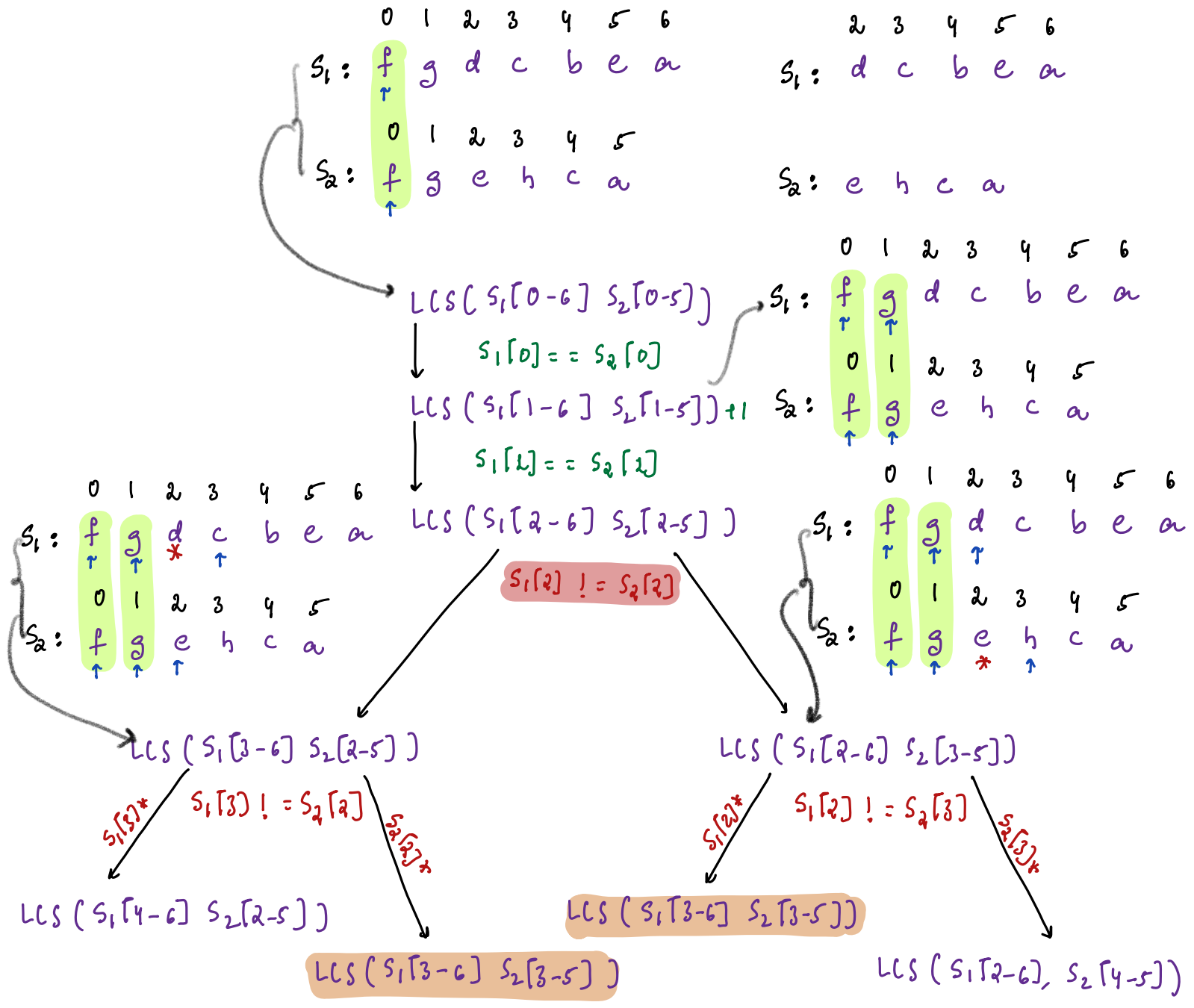
ans = fgca / fgea / : 4

Eg2:

	0	1	2	3	4	5	6
S_1 :	p	i	r	g	a	l	k
S_2 :	m	k	g	i	g	l	

ans = igl / 3.

Idea: # Get LCS between $S_1[0-6]$ && $S_2[0-5]$



Recursion: Ass: Calulate LCS between $S_1[i, N-1]$ & $S_2[j, M-1]$

```
int LCS(char s1[], char s2[], int i, int j){
    if( i==N || j==M) { return 0} // i==N: S1 is empty,

    if( s1[i] == s2[j]){
        return LCS(s1, s2, i+1, j+1)+1
    }
    else{
        return max{LCS(s1, s2, i+1, j) , LCS(s1, s2, i, j+1)}
    }
}
```

S_1 : i i+1 .. N-1
S_2 : j j+1 .. M-1
→ Consider them in ans sequence.

S_1 : i i+1 .. N-1
S_2 : j j+1 .. M-1

S_1 : i i+1 .. N-1
S_2 : j j+1 .. M-1

Using Dp:

// invalid: ↶
`int dp[N][M] = -1;`

dpstate: dp[i][j] = LCS between $S_1[i..N-1]$ && $S_2[j...M-1]$)

Is dp[N-1][M-1] is valid : LCS $S_1[N-1, N-1]$ && $S_2[M-1, M-1]$

Is dp[N][M] is valid : LCS $S_1[N, N-1]$ x

```
int LCS(char s1[], char s2[], int i, int j){ TC: O(N*M) SC: O(N*M)
    if( i==N || j==M) { return 0}

    if(dp[i][j] == -1){
        if( s1[i] == s2[j]){
            dp[i][j] = LCS(s1, s2, i+1, j+1)+1
        }
        else{
            dp[i][j] = max{LCS(s1, s2, i+1, j) , LCS(s1, s2, i, j+1)}
        }
    }
    return dp[i][j];
}
int solve(char s1[], char s2[]){ TC: O(  ) SC: O(  )
    int N = s1.length, M = s2.length;
    return LCS(s1, s2, 0, 0);
}
```

Edit Distance:

Given 2 Strings S_1 & S_2 min operations to perform on S_1 so that $S_1 \to S_2$

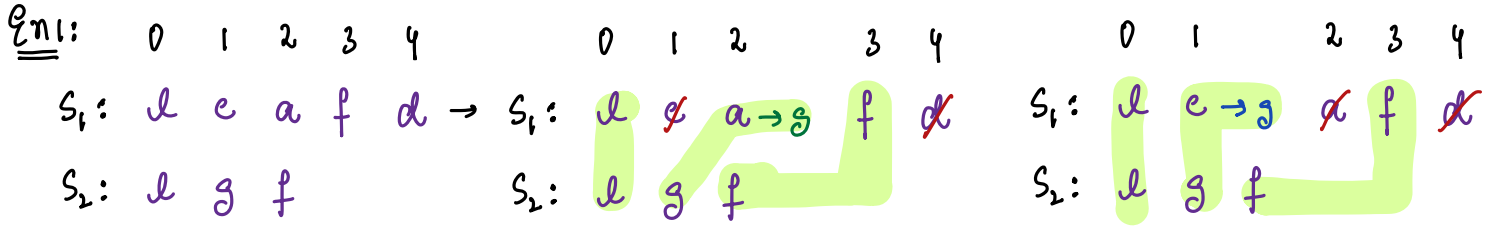
In 1 operation of S_1.

i → We can insert any character in S_1 at any pos

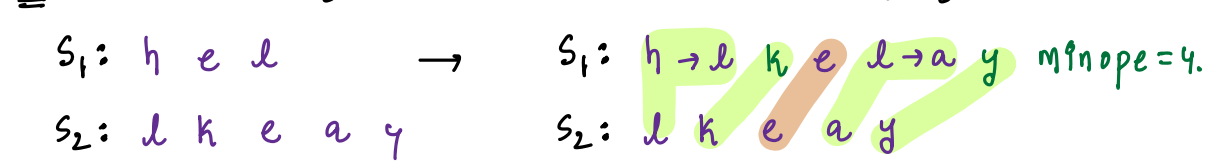
d → We can delete any character in S_1 at any pos

r → We can replace any character in S_1 at any pos, with any char

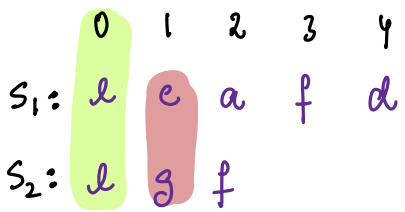
Ex1:



Ex2:



Idea: # Get Edit distance between $S_1[0\ 4]$ && $S_2[0\ 2]$



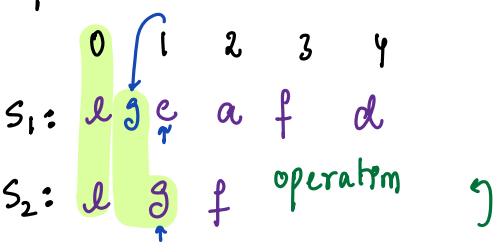
Edit($S_1[0-4]$ $S_2[0\ 2]$)

$S_1[0] == S_2[0]$

Edit($S_1[1-4]$ $S_2[1\ 2]$)

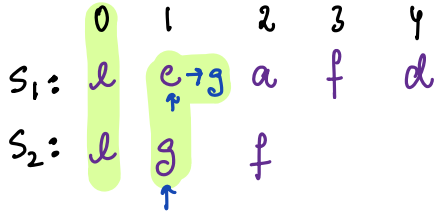
$S_1[1] \neq S_2[1]$

Operation: Insert



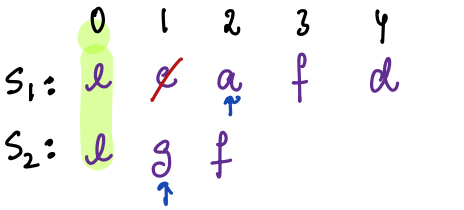
Edit($S_1[1-4]$ $S_2[2-2]$) +1

replace



Edit($S_1[2-4]$ $S_2[2-2]$) +1

delete



Edit($S_1[2-4]$ $S_2[1-2]$) +1

```
int   Edit( char s1[], char s2[], int i, int j){
    if ( i == N && j == M){   // Both s1 & s2 are exhausted.
        return 0;   // s1 = ""  s2 = ""  min op req to convert s1 -> s2:
    }
                    // s1 is exhausted  -> add all characters in s1
    if( i == N){    // s1 = "",  s2 = j j+1 .. M-1   How many: [j .. M-1] ?
        return M-j     min op req to convert s1 -> s2:  M-1-j+1 = M-j
    }
                    // s2 is exhausted.  -> delete all characters in s1
    if( j == M){    // s1 = i i+1 .. N-1   s2 = "";  :  How many : [i .. N-1]
        return N-i     min op req to convert s1 -> s2: N-1-i+1 = N-i
    }

    if( s1[i] == s2[j]){          s1 : i | i+1 .. N-1 |
        return Edit( s1, s2, i+1, j+1)    s2 : j | j+1 .. M-1 |
    }
    else{
            insert:                      replace:                      delete
            s1 : s2[j] i  i+1 .. N-1      s1 : i->s2[j] i+1 .. N-1      s1 : i  i+1 .. N-1
            s2 :   j     j+1 .. M-1       s2 : j       j+1 .. M-1       s2 : j  j+1 .. M-1
    } return min{ Edit(s1, s2, i, j+1) +1   Edit(s1, s2, i+1, j+1) +1   Edit(s1, s2, i+1, j) +1 }
}
```

Using Dp:

dpState:

int dp[ ][ ]; dp(i,j) = Min Edit distance S_1:[i..N-1] && S_2:[j..M-1]

int Edit(char s_1[], char s_2[] int i, int j){ → TODO

}

10:55pm break

Regex Matching?

Given Text(T) & Pattern(P) check if both are same or not

T → In Text it contain alphabets

P → With alphabets, it contains ?, *

? → It can match any 1 character

* → It can match any number of continous characters, {0, 1, 2..}

Ex1:

	0	1	2	3	4
T:	a	p	p	l	e
P:	a	?	*		e

matching

Ex2:

	0	1	2	3	4	5
T:	a	p	p	l	a	e
P:	a	*	a		?	e

not matching.

Ex3:

	0	1	2
T:	a	n	t
P:	a	?	*	*	t

matching.

Ex4:

T = ""

P = * * * *

matching

Ex5:

T = ""

P = ""

matching

Ex6:

T = c a b

P = a *

not matching.

→ Check if T[0-4] is matching with P[0-3]

	0	1	2	3	4
T:	e	l	p	p	a
P:	e	?	*		a

Reg(T[0-4] P[0 3])

↓ T[0] == P[0]

Reg(T[1-4] P[1-3]) → T: e l p p a, P: e ? * a

↓ P[1] == ?,

Reg(T[2-4] P[2-3]) : * : {0 1 2 3 ..}

P[ ] == *

	0	1	2	3
T:	a	b	c	
P:	?	b	c	

	0	1	2	3	4
T:	e	l	p	p	a
P:	e	?	*		a

Reg(T[2-4] P[3-3])

* not matching
* leave it

T = l
& match it &
stay there

	0	1	2	3	4
T:	e	l	p	p	a
P:	e	?	*		a

Reg(T[3-4] P[2-3])

TODO: Overlapping Sub Prob

We still stay at star to make choice

Ass: Calulate if Text[i..N-1] is matching Pattern[j..M-1]

```
boolean Regex(char T[], char P[], int i, int j){
    if ( i==N && j==M){ // Both T & P are exhausted → empty:
        return true
    }
    if( i == N){ // Text exhausted:            j j+1 .. M-1
        int c = 0;                  // T = "",  P = * * * *  }
        for(l=j; l<M; l++){
            if(P[l]=='*'){
                c=c+1
            }
        }
        return c == M-j
    }

    if( j==M){ // Pattern is exhausted
        return false;              // T = i i+1 .. N-1  P = "";
    }

    if( T[i]==P[j] || P[j]=='?' ){
        return { Regex(T, P, i+1, j+1) }
    }
    else if( P[j]=='*'){
        return { Regex(T, P, i, j+1)  or  Regex(T, P, i+1, j) }
    }
    else{
        return False;
    }
}
```

It can match, if all characters in P from [j..M-1] are *

How many stars = M-1-j+1 = M-j stars

Note: In this condition we are calulate no: of *'s in range [j..M-1] ≈ range query:

Optimize Range Query: Prefix Sum ~ Suffix Sum

Prefix Sum → Sum from 0; Suffix Sum → Sum from last index

Case T[i]==P[j] || P[j]=='?':

T : i | i+1 . . N-1
P : j | j+1 . . M-1

T : i | i+1 . . N-1
P : j=? | j+1 . . M-1

Case P[j]=='*':

T : i | i+1 . . N-1
P : j=* | j+1 . . M-1

T : i | i+1 . . N-1
P : j=* | j+1 . . M-1

Else case:

T : i | i+1 . . N-1
P : j ≠ ? * | j+1 . . M-1

Dp Code:

dp[i][j] = Match Text[i..N-1] with Pattern[j..M-1]

int dp[N][M]
- = -1 : not solved
- = 0 : its false
- = 1 : its true:

Note: if we keep dp table as boolean we cannot initialize with invalid value.

↳ create it in main & pass.

```
int Regex(char T[], char P[], int i, int j, int suf[]){
    if( i==N && j==M){
        return true
    }
    if( i==N){
        int c=0;
        // Need no: of stars from [j..M-1]
        return suf[j] == M-j
    }
    for(k=j;k<M;k++){
        if(P[k]=='*'){
            c=c+1
        }
    }
    return c==M-j

    if( j==M){// Pattern is exhausted
        return false;        // T = i i+1..N-1  P = "";
    }

    if( dp[i][j]==-1){
        if(T[i]==P[j] || P[j]=='?'){
            dp[i][j] = Regex(T,P,i+1,j+1,suf)
        }
        else if(P[j]=='*'){
                    T:  i   i+1..N-1
                    P:  j=* j+1..M-1
            dp[i][j] = { Regex(T,P,i,j+1,suf) | Regex(T,P,i+1,j,suf) }
        }
        else{
            return false;
        }
    }
    return dp[i][j]
}
```

↳ Bitwise OR

En: SuffInSum:

	0	1	2	3	4	5	6	7	
P:	*	*	a	b	?	*	*	*	
suf[8]:	5	4	3	3	3	3	2	1	0

suf[i]: No: of stars from i... last index:

```
int solve(char T[], char P[]){
    int M = P.length;
    int suf[M] = 0;
    int c = 0;
    // Create suf[] for stars.
    for(int i = M-1; i >= 0; i--){
        if(P[i] == '*'){ c = c+1; suf[i] = c;}
        else{ suf[i] = c;}          // for optimisation.
    }
    return regen(T, P, 0, 0, suf)
}
```