Q3

A)

we define a function called m: length of the first string

n: length of the second string.

we consider words A and B as arrays of characters. $A = a_0 a_1 \ldots a_m$, $B = b_0 b_1 \ldots b_n$

We first initailize two-dimensional array $DP_{(m+1)*(n+1)}$ as 0 to store all the distance between substring of the words. DP_{ij} denotes the edit distance of $a_0 a_1 \ldots a_i$ and $b_0 \ldots b_j$.

we use a bottom up manner.

$DP_{ij} =$

1. If i=0: it means the first string is empty, only option is to insert all characters of $b_0 \ldots b_j$ into a_0 so $DP_i = i$
2. If j=0: it means the second string is empty, only option is to remove all characters of first string so $DP_j = j$

1. otherwise if $a_i = b_j$: it means no change is needed so the edit distance of DP_{ij} would be as the same edit distance $DP_{i-1\, j-1}$. so $DP_{ij} = DP_{i-1\, j-1}$
2. $o.w$: when a_i and b_j are not the same, we need to whether insert, remove, or replace the letter so $DP_i = min(DP_{ij-1} + 1,\ DP_{i-1j} + 1,\ DP_{i-1j-1} + 2)$:

- insert: $DP_{ij} = \ DP_{ij-1} + 1$ we insert b_j into $a_0 \ldots a_i$
- remove: $DP_{ij} = \ DP_{i-1j} + 1$ we delete a_i from $a_0 \ldots a_i$
- replace: $DP_{ij} = \ DP_{i-1j-1} + 2$ we replace a_i with b_j

```
def edit_cost(a, b, m, n):
    dp·=·[[0·for·x·in·range(n·+·1)]·for·x·in·range(m·+·1)]
    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0:
                dp[i][j] = j
            elif j == 0:
                dp[i][j] = i
            elif a[i-1] == b[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] =  min(dp[i][j-1]+1, dp[i-1][j]+1,dp[i-1][j-1]+2)
    return dp[m][n],dp
```

in order to find the actions, we start from DP_{mn} and go up untill we reach DP_{00} when we are at state DP_{ij}:

1. if i,j=0: it means we have finished converting the stings.
2. if j=0: it means we are in the first column, and we want to reach b_0. so we delete a_j and go to state DP_{i-1j}. so the action is to delte a_j from the j position
3. if i=0: we are in the fist row, and our first string is a_0 so we insert b_j to the string and go to DP_{ij-1}.so the action is to insert b_i from the i position
4. o.w:

- if a_i = b_j are the same, we do not do any actions and go to state DP_{i-1j-1}
- o.w: we first calculate $min(DP_{ij-1}+1,\ DP_{i-1j}+1,\ DP_{i-1j-1}+2)$:
- 1. if $DP_{i-1j-1}+2$ is the minimum, we replace a_i with b_j and go to DP_{i-1j-1} state.
- 2. if $DP_{ij-1}+1$ is the minimum, we insert b_j and go to DP_{ij-1} state.
- 3. if $DP_{i-1j}+1$ is the minimum, we delte a_i and go to DP_{i-1j} state.

when we go to the previous state, we exacute the algorithm untill we reach BP_{00}

```
def find_action(m,n,dp,a,b):
    if m==0 and n==0:print("Done")
    elif m>0 and n==0:print("Delete",a[m-1]);find_action(m-1,n,dp,a,b);
    elif m==0 and n>0:print("Insert",b[n-1]);find_action(m,n-1,dp,a,b);
    elif a[m-1]==b[n-1]:find_action(m-1,n-1,dp,a,b)
    elif (dp[m-1][n-1]+2)<=(dp[m-1][n]+1) and (dp[m-1][n-1]+2)<=(dp[m][n-1]+1):
        print("Replace",a[m-1],"with",b[n-1])
        find_action(m-1,n-1,dp,a,b)
    elif (dp[m][n-1]+1)<(dp[m-1][n-1]+2) and (dp[m][n-1]+1)<=(dp[m-1][n]+1):
        print("Insert",b[n-1])
        find_action(m,n-1,dp,a,b)
    else:
        print("Delete",a[m-1])
        find_action(m-1,n,dp,a,b)
```

```
a="index"
b="inside"
min_cost,dp=edit_cost(a, b,len(a),len(b))
print("minimum cost to turn index into inside : "+str(min_cost))
find_action(len(a),len(b),dp,a,b)
```

```
minimum cost to turn index into inside : 3
Delete x
Insert i
Insert s
Done
```

```
a = "sunday"
```

```
b = "saturday"
min_cost,dp=edit_cost(a, b,len(a),len(b))
print("minimum cost to turn sunday to saturday : "+str(min_cost))
find_action(len(a),len(b),dp,a,b)
```

```
minimum cost to turn sunday to saturday : 4
Replace n with r
Insert t
Insert a
Done
```

B)

we consider words A and B as arrays of characters. $A = a_0 a_1 \dots a_n$, $B = b_0 b_1 \dots b_m$ where $a_0 = b_0 = \emptyset$.

we first create a two dimensional array $DP_{(n+1)*(m+1)}$ to store the longest common string of the substrings.

DP_{ij} denotes the length of the longest common string of $a_0 a_1 \dots a_i$ and $b_0 \dots b_j$.

for each $0 \leq i \leq n$, we loop through $0 \leq j \leq m$,since we iterate i and j in an increasing order, when we want to calculate DP_{ij}, if $r + t < i + j$, we already have the value of DP_{rt}

$DP_{ij} =$

1. if i=0: it means we want to find $LCS(\emptyset, b_0 .. b_j)$ which is 0 so $DP_{0j} = 0$
2. if j=0: it means we want to find $LCS(a_0 .. a_i, \emptyset)$ which is 0 so $DP_{i0} = 0$
3. o.w:

- if a_i = $b_j = x$:it means we want to find $LCS(a_0 .. a_{i-1} x, b_0 .. b_{j-1} x)$. by removing x from both strings, we just need to find $LCS(a_0 .. a_{i-1}, b_0 .. b_{j-1})$ and increament it by 1(becouse the last character x is common) so $DP_{ij} = DP_{i-1j-1} + 1$
- if $a_i \neq b_j$:since a_i and b_j are not equal, first we rmove a_i and calculate $l_1 = LCS(a_0 .. a_{i-1}, b_0 .. b_j)$ and then remove b_j and calculate $l_2 = LCS(a_0 .. a_i, b_0 .. b_{j-1})$. since we want the length of the longest common string, we choose the maximum of (l_1, l_2). so $DP_{i,j} = max(DP_{i-1,j}, DP_{i,j-1})$

so DP_{nm} would be the length of longest common string of A and B

since we first initialized DP as zeros, and the first column and value are all 0, we do not iterate the first row and column in the code.

```
def lcs(a , b):
    n = len(a)
    m = len(b)
    dp = [[0]*(m+1) for i in range(n+1)]
    for i in range(1,n+1):
        for j in range(1,m+1):
```

```
        for j in range(1,m+1):
            if i == 0 or j == 0 :
                dp[i][j] = 0
            elif a[i-1] == b[j-1]:
                dp[i][j] = dp[i-1][j-1]+1
            else:
                dp[i][j] = max(dp[i-1][j] , dp[i][j-1])
    return dp[n][m]
```

```
s1 = "abdacbab"
s2 = "acebfca"
print ("Length of the longest common string of abdacbab and acebfca: ", lcs(s1, s2) )
```

```
Length of the longest common string of abdacbab and acebfca:  4
```

```
s1 = "hello"
s2 = "what"
print ("Length of the longest common string of hello and what: ", lcs(s1, s2) )
```

```
Length of the longest common string of hello and what:  1
```