

14/06/2023Q1 Longest common subsequencei/p \rightarrow $S_1 = \text{"abcde"}$ $S_2 = \text{"ace"}$ o/p \rightarrow 3 (ace is the longest common subsequence) $S = abc$ Subsequence of above string can be a, b, c, ab, bc, ac, abc, " "
 \rightarrow empty stringEx \rightarrow abc

acd

a

a

b

c

c

d

ab

ac

bc

cd

ac

ad

abc

acd

Common subsequences are a, c, and ac.
Here ac is the longest subsequence here.Approach \swarrow i text1 \rightarrow abcdetext2 \rightarrow ace \uparrow j

text1[i] == text2[j] } char matches & rest of both strings, recursion will handle.

rec(i+1, j+1)

But what if character does not match.

text1 → abcde
 ↑
 i

text2 → vine
 ↑
 j

Simply take $\max(\text{rec}(i+1, j), \text{rec}(i, j+1))$;

exclude 1st

exclude 2nd

String char

String char

The above approach is 2-pointer approach.

Code

```
int solveRec (string a, string b, int i, int j) {
    // Base case (String completely traversed)
    if (i == a.length())
        return 0;
    // Base case
    if (j == b.length())
        return 0;
    // Match
    if (a[i] == b[j])
        return 1 + solveRec (a, b, i+1, j+1);
    // Mismatch
    else {
        return 0 + max (solveRec (a, b, i, j+1),
                        solveRec (a, b, i+1, j));
    }
}
```

3

// Top-down approach (Gave TLE if & wasn't used)
int solveTopDown (string &a, string &b, int i,
int j, vector <vector <int>> &dp) {

// Base case

if (i == a.length())

return 0;

if (j == b.length())

return 0;

// Step 3: Check ans exists in dp or not.

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

return dp[i][j];

// Match case

int ans = 0;

if (a[i] == b[j])

ans = 1 + solveTopDown(a, b, i+1, j+1, dp);

// Mismatch case

else {

ans = 0 + max(solveTopDown(a, b, i, j+1, dp),

solveTopDown(a, b, i+1, j, dp));

}

// Step 2: Store ans in dp array

dp[i][j] = ans;

return dp[i][j];

}

Note -> dp array created in main() as ÷

vector<vector<int>> dp(a.length(), vector<int>(b.length(), -1));

// Bottom up approach

int solveTab(string a, string b) {

// Step 1: Create dp array

vector<vector<int>> dp(a.length() + 1,

vector<int>(b.length() + 1, 0));

// Step 2: Observe base case of top-down

// Already handled in initialization step

```
// Step 3: Reverse flow of top-down
for (int i = a.length() - 1; i >= 0; i--) {
    for (int j = b.length() - 1; j >= 0; j--) {
        // Match case
        int ans = 0;
        if (a[i] == b[j])
            ans = 1 + dp[i+1][j+1];
        // Mismatch case
        else {
            ans = 0 + max(dp[i][j+1], dp[i+1][j]);
        }
        dp[i][j] = ans;
    }
}
return dp[0][0];
```

* Space optimization possible or not?

Yes, space optimization is possible.

1) Create 2, 1D arrays

```
vector<int> curr (b.length() + 1, 0);
vector<int> next (b.length() + 1, 0);
```

2) Replace $dp[i+1][j]$ with $next[j]$ and $dp[i][j]$ with $curr[j]$

3) Shifting $\rightarrow next = curr$ as we are going upwards.

Q2 Longest palindromic subsequence

text1 = i/p string

text2 = reverse of i/p string

Apply longest common subsequence on text1 and text2 strings.

→ (Very famous question)

Q3 Edit distance

We will be given 2 words in i/p and we need to apply minimum operations such as insert, delete or replace character to make the second word.

Conversion
W1 → W2

Ex → horse
hi j

Here 1st characters are matching. Hence simply recursive call for $i+1$ and $j+1$.

Insertion case

m horse
mse
→ inserted

Hence recursive call for i and $j+1$ as now 1st character of both strings are matching.

Deletion case

horse
ose

Hence recursive call for $i+1$ and j .

Replace case

m horse
mse

Now character are matching and hence recursive call for $i+1$ and $j+1$.

Code

```
int solveRec (String &a, String &b, int i, int j) {  
    // Base case  
    if (i == a.length())  
        return b.length() - j;  
    if (j == b.length())  
        return a.length() - i;  
    // Match case  
    int ans = 0;  
    if (a[i] == b[j])  
        ans = solveRec(a, b, i+1, j+1);  
    // Mismatch → Operations to be performed  
    else {  
        int insert = 1 + solveRec(a, b, i, j+1);  
        int deleted = 1 + solveRec(a, b, i+1, j);  
        int replace = 1 + solveRec(a, b, i+1, j+1);  
        ans = min(insert, min(deleted, replace));  
        // ↑ we need to apply minimum no. of operations  
    }  
    return ans;  
}
```

3

// Top down approach

```
int solveTopDown (String &a, String &b, int i, int j) {  
    // Base case  
    if (i == a.length())  
        return b.length() - j;  
    if (j == b.length())  
        return a.length() - i;  
    // Step 3 : Check if answer already exists in dp  
    if (dp[i][j] != -1)  
        return dp[i][j];
```



```
// match case
```

```
int ans = 0;
```

```
if (a[i] == b[j]) {
```

```
    ans = solveTopDown(a, b, i+1, j+1, dp);
```

```
}
```

```
// mismatch
```

```
else {
```

```
    int insert = 1 + solveTopDown(a, b, i, j+1, dp);
```

```
    int deleted = 1 + solveTopDown(a, b, i+1, j, dp);
```

```
    int replace = 1 + solveTopDown(a, b, i+1, j+1, dp);
```

```
    ans = min(insert, min(deleted, replace));
```

```
}
```

```
// Step 2 : Store ans in dp array
```

```
dp[i][j] = ans;
```

```
return dp[i][j];
```

```
}
```

```
// Bottom up approach
```

```
int solveTab (String &a, String &b) {
```

```
    // Step 1 : Create dp array
```

```
    vector<vector<int>> dp (a.length() + 1,
```

```
        vector<int> (b.length() + 1, 0));
```

```
    // Step 2 : Observe base case of top down
```

```
    for (int j = 0; j <= b.length(); j++) {
```

```
        dp[a.length()][j] = b.length() - j;
```

```
    }
```

```
    for (int i = 0; i <= a.length(); i++) {
```

```
        dp[i][b.length()] = a.length() - i;
```

```
    }
```

```
    // Step 3 : Reverse flow of top-down
```

```
    for (int i = a.length() - 1; i >= 0; i--) {
```

```
        for (int j = b.length() - 1; j >= 0; j--) {
```

```
            // Match case
```

```
            int ans = 0;
```

```

    if (a[i] == b[j])
        ans = dp[i+1][j+1];
    // mismatch case
    else {
        int insert = 1 + dp[i][j+1];
        int deleted = 1 + dp[i+1][j];
        int replace = 1 + dp[i+1][j+1];
        ans = min(insert, min(deleted, replace));
    }
    dp[i][j] = ans;
}
}
return dp[0][0];
}

```

* Space optimization is possible or not?
Yes space optimization is possible.

1) Create 2 1D arrays

```

vector<int> curr (b.length()+1, 0);
vector<int> next (b.length()+1, 0);

```

2) Replace $dp[i+1][--]$ with $next[--]$ and $dp[i][--]$ with $curr[--]$.

3) Base case modifications

```

* for (int j = 0; j <= b.length(); j++)
    next[j] = b.length() - j;

```

→ Can do mistake here

* Now 2nd base case is tricky here as we don't have all the rows. Hence in step-3

inside outer for loop, write

$$curr[b.length() - 1] = a.length() - i;$$

4) Shifting to be done as $next = curr$ as we are going upwards.