**Answer (a). :** [[Reference Source]](http://www-cs-students.stanford.edu/~csilvers/proof/node4.html#:~:text=To%20prove%20a%20theorem%20of%20the%20form%20A%20IF%20AND,proofs%20in%20your%20own%20proof.)

Let **X** and **Y** be two input strings, of lengths **m** and **n**, respectively.

Given, **X[m] = Y[n],** To prove: **X** appears in **Y** iff **X[1 ... (m-1)]** appears in **Y[1...(n-1)]**.

We physically break an **IF & ONLY IF** into two proofs: **‘forward’** & **‘backward’ :**

* **‘forward’ ( :** In this direction, we assume that ‘**X** appears in **Y’** (i.e. *LHS*) is correct, and try to prove that the *RHS* is incorrect.

Let some **X[i] ∉** **Y[1...(n-1)]**. [ 1 i (m-1) ].

Now, since **X[m] = Y[n] (given), &** there exists such an element **X[i]** which does not match with any of the elements in **Y[1...(n-1)]**, so ***X*** *will cease to “appear” in* ***Y*.**

This contradicts our assumption that **X** appears in **Y,** hence such an element **X[i]** cannot exist.

Therefore, **X[1 ... (m-1)]** must appear in **Y[1...(n-1)].**

* **‘backward’ ( ):** In this direction, we assume that **‘X[1 ... (m-1)]** appears in **Y[1...(n-1)]’** (i.e. *RHS*) is correct, and try to prove that the *LHS* is incorrect.

Let **X[m] = Y[n] = K.** We append **K** to both **X[1 ... (m-1)] &** **Y[1...(n-1)]** carefully,preserving their actual order**!**

Now, since **X[1 ... (m-1)]** already appears in **Y[1...(n-1)],** and we are appending the same element to both arrays. Then **X[1 ... m]** must also appear in **Y[1...n].**

We know that **A↔B** is equivalent to **(A→B) (B→A).**

The **‘forward’** case shows the **(A→B)** case, while the **‘backward’** case shows the **(B→A)** case. The above two cases are enough to prove that the above biconditional also holds.

**Answer (b). :**

Let **X** and **Y** be two input strings, of lengths **m** and **n**, respectively.

Given, **X[m] != Y[n],** To prove: **X** appears in **Y** iff **X[1...m]** appears in **Y[1...(n-1)]**.

Similar to the earlier solution, we consider the two cases: **‘forward’** & **‘backward’ :**

* **‘forward’ ( :** In this direction, we assume that ‘**X** appears in **Y’** (i.e. *LHS*) is correct, and try to prove that the *RHS* is incorrect.

Now, since **X[m] != Y[n] ;** if the array **X[1...m]** does not appear in the trimmed **Y[1...(n-1)],** then it will also not be able to appear in **Y[1...n].** This contradicts our assumption that **X** appears in **Y.**

Hence, **X[1...m]** will always appear in **Y[1...(n-1)]**.

* **‘backward’ ( ):** In this direction, we assume that **‘X[1...m]** appears in **Y[1...(n-1)]’** (i.e. *RHS*) is correct, and try to prove that the *LHS* is incorrect.

Now, since **X[m] != Y[n],** even if we append **Y[n]** to **Y[1...(n-1)], X[1...m]** will still appear in **Y[1...(n-1)].** Thus, *there is no way to make* **X[1...m]** *not appear in* **Y[1...n]***,* as it is already appearing in **Y[1...(n-1)].**

Hence, **X[1 ... m]** will always appear in **Y[1…(n-1)].**

Likewise, the **‘forward’** case shows the **(A→B)** case, while the **‘backward’** case shows the **(B→A)** case. The above two cases are enough to prove that the above biconditional also holds.

**Answer (c). :**

**DP solution** for checking whether **X[1...m]** appears at least twice in string **Y[1…n]** as **2** disjoint subsequences **P.** We define a new problem **Q** to solve **P.**

**1. Specify a problem Q (which will eventually solve P):**

The problem of finding **X** at least twice in **Y,** can be solved through a new problem **Q:** finding disjoint occurrences of **X1** **&** **X2** once in **Y,** where **X1 = X2 = X.**

Solving **Q** (finding exactly 2 occurrences) will solve **P** (finding at least 2 occurrences).

Given **X[1...m]** & **Y[1…n],** the function **checkTwiceSS** returns whether **X** appears twice in **Y** or not. We have also assumed a **3-D** **Memoization** table of dimensions (**n+1 \* m+1 \* m+1)**, which will be filled from left to right (i.e. from **1** to either **n** or **m**) in each dimension.

**2. Give a recurrence expression/formula or recursive algorithm for solving Q :**

**checkTwiceSS(Y[1…n], X1[1...m], X2[1...m])** = 0 **;** when **|Y|=0**

=1 ; when **|X1|=0** & **|X2|=0**

= **checkTwiceSS (Y[2…n], X1[2...m], X2[1...m])** if Y[1]==X1[1]

OR

**checkTwiceSS (Y[2…n], X1[1...m], X2[2...m])** if Y[1]==X2[1]

OR

**checkTwiceSS (Y[2…n], X1[1...m], X2[1...m])**

otherwise (i.e.Y[1] is neither equal to X1[1]or X2[1]. )

**3. Prove correctness of recurrence relation:**

The **checkTwiceSS** function will return 0, when **Y** exhausts i.e. when **|Y|=0 .**

This reflects the case when **X1** **or** **X2** does not appear disjointly in **Y.**

The **checkTwiceSS** function will return 1, when **X1** **&** **X2** exhausts i.e. when **|X1|=0** & **|X2|=0**

This reflects the case when **X1** **&** **X2** appears disjointly in **Y.**

The **checkTwiceSS** function also checks whether the first element of **Y** is equal to the first element of **X1** **or** **X2 or** it is not equal to any one of them at all.Accordingly, these 3 cases follow:

* When **Y[1]==X1[1] :**

This implies that **X1** can appear in **Y** iff **X1[2 ... m]** appears in **Y[2...n]**. *[proved above]*

Hence, it is wiser to trim the above 2 arrays before further checking for “appearance”.

**X2** however remains unaltered.

Hence, the function is recursively called as **checkTwiceSS (Y[2…n], X1[2...m], X2[1...m]).**

* When **Y[1]==X2[1] :**

This implies that **X2** can appear in **Y** iff **X2[2 ... m]** appears in **Y[2...n]**. *[proved above]*

Hence, it is wiser to trim the above 2 arrays before further checking for “appearance”.

**X1** however remains unaltered.

Hence, the function is recursively called as **checkTwiceSS (Y[2…n], X1[1...m], X2[2...m]).**

* When **(Y[1] != X1[1]) AND (Y[1] != X2[1]) :**

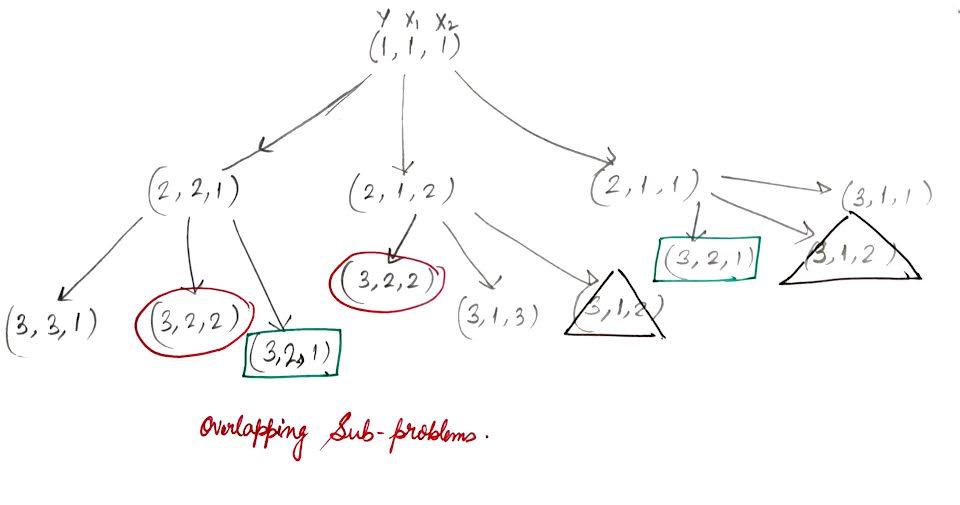
This implies that **X1** can appear in **Y** iff **X1[1 ... m]** appears in **Y[2...n]** ; and **X2** can appear in **Y** iff **X2[1 ... m]** appears in **Y[2...n].**  *[proved above]*

Hence, it is wiser to only trim the array **Y** before further checking for “appearance”.

**X1**& **X2** however remain unaltered, as their entirety is required for checking.

Hence, the function is recursively called as **checkTwiceSS (Y[2…n], X1[1...m], X2[1...m]).**

Building on the above checks, an exhaustive search will be done on **Y** to find the disjoint occurrences of **X1** & **X2**. At each step, the problem is broken into 3 subproblems with input reduced by a size of 1.

****

From the recurrence tree, it is evident that there will be overlapping subproblems.

**4. Describe a memoization data structure:**

The **Memoization** table **M** is a **3-D** array of size **(n+1)\*(m+1)\*(m+1)**, and any index of **M** say **(i,j,k)**tells when **i** elements are left in **Y**, **j** elements are left in **X1** , **k** elements are left in **X2**.

**Initialization of array:**

M[ i ][ 0 ] [ 0 ] = 1 // implies **|X1|=0** & **|X2|=0** i.e. **X1** & **X2** appears disjointly in **Y.**

M[ 0 ][ j ] [ k ] = 0 // implies **|Y|=0** i.e. **X1** & **X2** do not appear in **Y.**

M[ 0 ] [ 0 ] [ 0 ] = 1 // implies if all the 3 arrays are empty i.e. an empty string always appears twice in another empty string.

**5. Give an algorithm/ordering for solving P for all values.**

**Given**

for i in 1…n:

for j in 1…m:

for k in 1…m:

a=0,b=0,c=0

if Y[ i ] = X1[ j ]:

a = M[ i - 1] [ j -1 ] M [ k ] // Y & X1 are trimmed by 1

else if Y[ i ] = X2[ j ]:

b = M[ i - 1] [ j ] M [ k -1 ] // Y & X2 are trimmed by 1

else:

c = M[ i - 1] [ j ] M [ k ] // Only Y is trimmed by 1

M[ i ][ j ] [ k ] = a OR b OR c // OR represents logical OR

**6. How to solve problem P from values**

If **M[n][m][m]** contains **1**, it implies that **X1 & X2** appears once in **Y** as two disjoint subsequences. This solves the problem **Q**.

Since we assumed **X1 = X2 = X** ; therefore, **X** must appear at least twice in **Y** as two disjoint subsequences. Thus, **P** is also solved.

If **M[n][m][m]** contains **0**, it implies that **X1 & X2** do not appear in **Y** as two disjoint subsequences. Hence, **X** can never appear twice in **Y** as two disjoint subsequences. Thus, **P** is also solved.

**7. What is space and time complexity for solving problems?**

Since the above problem uses a **3-D** array for **Memoization** of size **(n+1)\*(m+1)\*(m+1);**

Hence, Space complexity =

As filling the Memoization table requires traversing over all **n \* m \* m** elements of **M**.

So, the time complexity of ***filling the table*** would be

To check X whether appears at least twice in Y as two disjoint subsequences it requires to check index **M[ n ][ m ][ m ]**, which requires constant i.e. **O( 1 )** time.

Time complexity

Time complexity