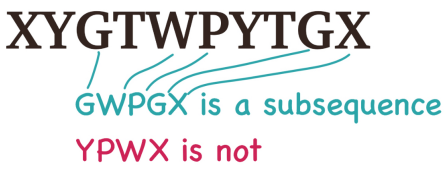
Dynamic Programming 2

The problem of comparing two strings in order to determine if they have similar characteristics is a problem that has applications across biology, computer security, and artificial intelligence. In biology for example, we study DNA sequences and search for sections that may have similarities.

In this lecture we study the problem of comparing strings based on their subsequences. A subsequence of a string A is a sequence of characters that exists in A in that order. Some characters of A can be "skipped" by the subsequence. The string $A = XYGTWPYTGX$ has many possible subsequences, and one such possible subsequence is shown in blue in the figure below. Notice that the characters of the subsequence appear in A and in the same order. The sequence $YPWX$ does not correspond to a subsequence of A.



The goal of this lecture is to provide an efficient algorithm for finding the **longest common subsequence** between two strings. In the example below, both strings contain the same subsequence $YTWPY$, which has length 5. If there is no other longer subsequence, then this is in fact the **longest common subsequence** between the two strings A and B.

We shall assume that the two strings in question can be represented as $A = (a_1, a_2, \ldots, a_m)$ and $B = (b_1, b_2, \ldots, b_n)$, where the values a_i and b_j represent the individual characters of the strings. The lengths of the two initial strings do not have to be the same, therefore we let m be the length of A and n be the length of B.

Let's first think about why a brute-force algorithm would not be efficient. How many possible subsequences does a string have? Each character in the string can be either *included* or *not included* in the subsequence. So the number of possible subsequences for string A is 2^m, and the number of possible subsequences for string B is 2^n. Any brute-force procedure that tries to enumerate all the possible subsequences of each string would take exponential time. That does not even include the time to comparing the subsequences!

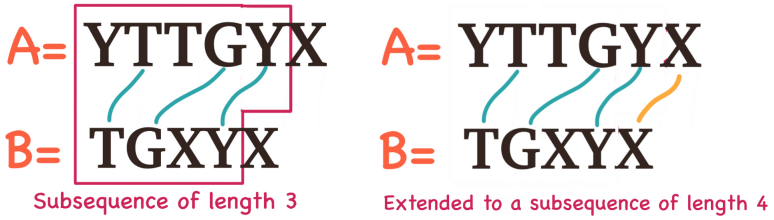
The longest common subsequence problem

In this section we present an efficient dynamic programming algorithm for the longest common subsequence problem. Recall that the dynamic programming technique relies on the ability to define the optimum problem in terms of the optimum sub-problems. Therefore we begin by trying to identify a relationship between the longest common subsequence of two strings A and B, and a an optimal *sub*problem.

Given the strings A and B, we can focus on the last characters of each string and decide whether or not that character *is* or *is not* included in the longest common subsequence.

Case 1: Same last character

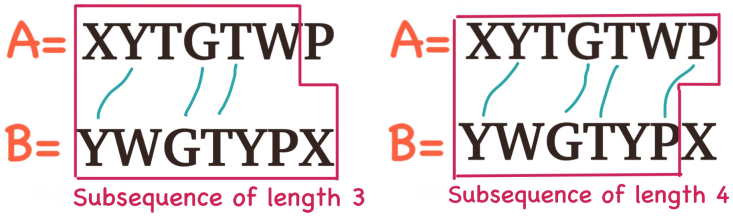
The example below shows two strings A and B that have the same last character, X.



Suppose X were *not* part of the longest common subsequence. The longest subsequence would then involve characters from the first parts of the strings, up to the second last character, as shown on the left above. In this case, we could just extend this subsequence by the character X and get a sequence that is one unit longer. Therefore it *must* be that the longest common subsequence includes the character X.

Case 2: Different last characters

If the last characters of A and B are different, then the longest common subsequence cannot contain the last character from *both* strings. In the example below, the last character P from A and the last X from B cannot both be included in the longest common subsequence. Instead, the longest subsequence is the longest common subsequence that we can make *either* without the last P *or* without the last X. The maximum of these two options represents the longest common subsequence in this case. From the figure below, the optimal value is a subsequence of length 4.



These two cases above represent the key to the dynamic programming solution.

The dynamic programming solution

The above two cases indicate that the optimal solution to the larger problem (the entire strings) can be found if we know the optimal values of smaller subproblems (shorter strings). The dynamic programming technique relies on the fact that we can **store the values of the subproblems** so they can be referenced by the algorithm when needed.

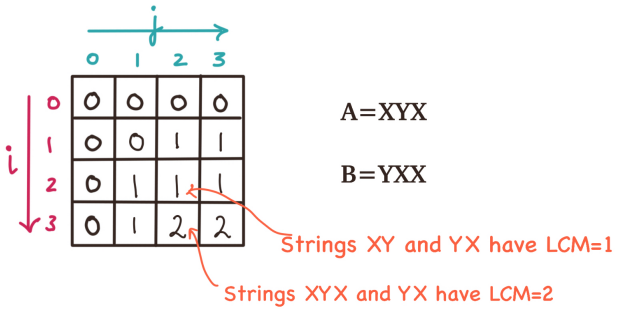
Our smaller subproblems in this case are based on finding the longest common subsequence within shorter sections of A and B. For example, notice in case one above, we referenced the problem of the longest common subsequence between the substring of A which was $(a_1, a_2, \ldots, a_{m-1})$ and the substring of B which was $(b_1, b_2, \ldots, b_{n-1})$. Therefore a dynamic programming table which sotred the optimal value for *substrings* of A and B would enable us to perform the look-ups that we need.

Since there are two strings, and therefore two possible substring lengths, we use a **double array,** $C\,[i, j]$ to store the optimal subproblem values. This double array starts at 0 because if either $i = 0$ or $j = 0$ then the string has length 0, which means the subsequence length is 0.

The Table: $C[0\ldots m, 0\ldots n]$

- Given strings $A = (a_1, a_2, \ldots, a_m)$ and $B = (b_1, b_2, \ldots, b_n)$
- $C[i, j]$ stores the length of the maximum common subsequence between the strings $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_j)$.
- $C[0, j] = 0$ for all j and $C[i, 0] = 0$ for all i
- The last entry in the table, $C[m, n]$, is the length of the longest common subsequence between A and B

An example of the table C for the simple strings $A = XYX$ and $B = YXX$ is shown below:



The final value for the longest common subsequence between A and B is stored in $C[3, 3]$. Therefore the longest common subsequence for these strings is 2.

It remains to show how a dynamic programming algorithm can fill the entries of the table efficiently, and in such a way that each look-up to the table is to values that have previously been stored. The key is to use **Case 1 and Case 2** from above, which define the relationship between the optimal solution and the smaller subproblems.

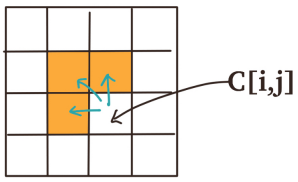
Recursively defining $C[i, j]$

Given strings $A = (a_1, a_2, \ldots, a_m)$ and $B = (b_1, b_2, \ldots, b_n)$ we define $C[i, j]$ in the following recursive way:

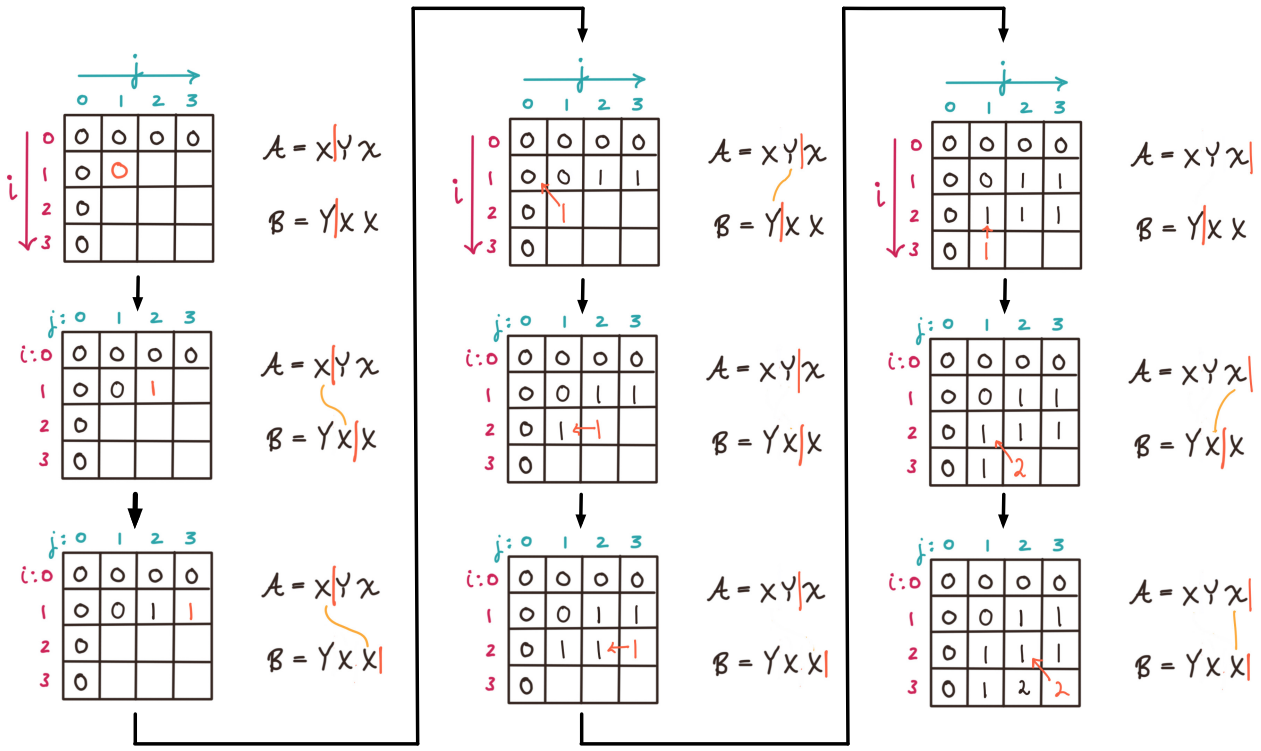
- **Case 1:** If $a_i = b_j$ then
$$C[i, j] = 1 + C[i - 1, j - 1]$$
- **Case 2:**. If $a_i \neq b_j$ then
$$C[i, j] = \max\{C[i - 1, j], C[i, j - 1]\}$$

The final value: $C[m, n]$ represents the longest common subsequence between A and B.

From above it is clear that the entry for $C[i, j]$ depends on $C[i - 1, j - 1]$ or $C[i - 1, j]$ or $C[i, j - 1]$. Therefore the table must be filled in in such a way that all three of these entries are stored *before* $C[i, j]$ is computed. We can visualize this dependence as:



If we fill up the table row by row, from left to right, then each look-up in the table will be for values that are already filled in.



The dynamic programming algorithm fills in this table row by row and left to right:

LCS(A,B)

Step 1: Initialize the first column of the table: for $i = 0$ to m, $C[i, 0] = 0$.
Initialize the first row of the table: for all $j = 0$ to n, $C[0, j] = 0$

Step 2: Loop through the table row by row from left to right.

```
for i = 1 to m
    for j = 1 to n
        if A[i] = B[j]
            C[i, j] = C[i − 1, j − 1] + 1
        else if C[i − 1, j] > C[i, j − 1]
            C[i, j] = C[i − 1, j]
        else C[i, j] = C[i, j − 1]
return C[m, n]
```

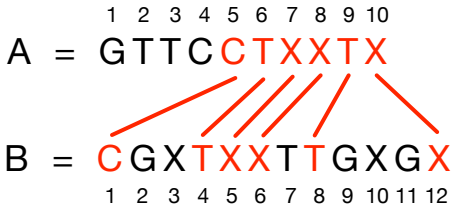
Runtime: The algorithm consists of a nested for-loop that iterates through a table of size mn. It performs a constant number of operations per table entry. Therefore the runtime is $O(mn)$.

Reconstructing the subsequence:

The final entry in $C[m, n]$ is the *length* of the longest common subsequence. But this value does not tell us what that common subsequence actually *is*. In order to trace back the original subsequence, we need to reconstruct the choices that were made during LCS(A,B).

In order to describe how this reconstruction is possible, we examine a larger example based on the strings $A = GTTCCTXXTX$ and $B = CGXTXXTTGXGX$. The completed dynamic programming table is shown below:

	j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8	j = 9	j = 10	j = 11	j = 12
i = 0	0	0	0	0	0	0	0	0	0	0	0	0	0
i = 1	0	0	1	1	1	1	1	1	1	1	1	1	1
i = 2	0	0	1	1	2	2	2	2	2	2	2	2	2
i = 3	0	0	1	1	2	2	2	3	3	3	3	3	3
i = 4	0	1	1	1	2	2	2	3	3	3	3	3	3
i = 5	0	1	1	1	2	2	2	3	3	3	3	3	3
i = 6	0	1	1	1	2	2	2	3	4	4	4	4	4
i = 7	0	1	1	2	2	3	3	3	4	4	5	5	5
i = 8	0	1	1	2	2	3	4	4	4	4	5	5	6
i = 9	0	1	1	2	3	3	4	5	5	5	5	5	6
i = 10	0	1	1	2	3	4	4	5	5	5	6	6	6



From the above table we can see that the longest common subsequence has length $C[10, 12] = 6$. How was this very last entry computed? Notice that the last characters of the strings A and B are the same, and so $C[10, 12] = 1 + C[9, 11]$. Therefore the last entry of the table was computed by *extending* the subsequence from $C[9, 11]$ to *include* the last character X. Therefore the last character in our subsequence is X. We denote this "choice" made by the algorithm by using the color red in the above table, which indicates that the value 6 was made by incrementing the value 5 by 1.

Continuing in this way, consider the entry $C[9, 11]$ in the table. The characters $A[9]$ and $B[11]$ are not the same, and thus the algorithm selects the maximum of $C[8, 11]$ and $C[9, 10]$. In this case $C[9, 10]$ is chosen by default since there is a tie. This situation indicates that no matching character was found, and therefore the longest common subsequence depends on previous values in the table. This step is shown in blue in the above table.

This pattern can be used to trace the characters of subsequence back through the table C.

```
PRINT-LCS(A,B,C)
    i = m, j = n
    while i > 0 and j > 0
        if A[i] = B[j]
            output A[i]
            i = i − 1, j = j − 1
        else if C[i − 1, j] > C[i, j − 1]
            i = i − 1
        else j = j − 1
```

This outputs the characters in reverse order, from back to front. The runtime depends on the maximum of m, n, and therefore it runs in time $O(\max\{m, n\})$.