DYNAMIC PROGRAMMING: LCS

Instructor: Dr Tarunpreet Bhatia
Assistant Professor, CSED
TIET

Subsequence

- A subsequence of a given sequence is just the given sequence with some elements left out.
- Formally, a given sequence $X = (x_1\ x_2.....x_m)$, another sequence $Z = (z_1\ z_2.....z_k)$ is a subsequence of X if there exists a strictly increasing sequence $(i_1\ i_2.....i_k)$ of indexes such that for all j =1, 2, …k we have $x_{i_j} = z_j$.
- Eg. Z = <B,C,D,B> is a subsequence of X = <A,B,C,B,D,A,B> with corresponding index sequence <2,3,5,7>
- Given two sequences X and Y, we say that the sequence Z is a common sequence of X and Y if Z is a subsequence of both X and Y.

Longest Common Subsequence (LCS)

Application: comparison of two DNA strings

Ex: X= {A B C B D A B }, Y= {B D C A B A}

Longest Common Subsequence:

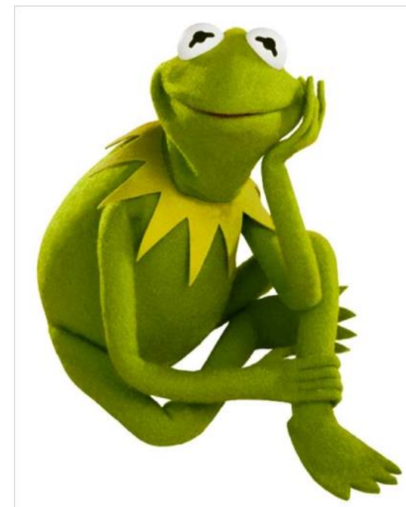
X = A **B** **C** **B** D **A** B

Y = **B** D **C** A **B** **A**

Brute force algorithm would compare each subsequence of X with the symbols in Y

Longest Common Subsequence

- How similar are these two species?

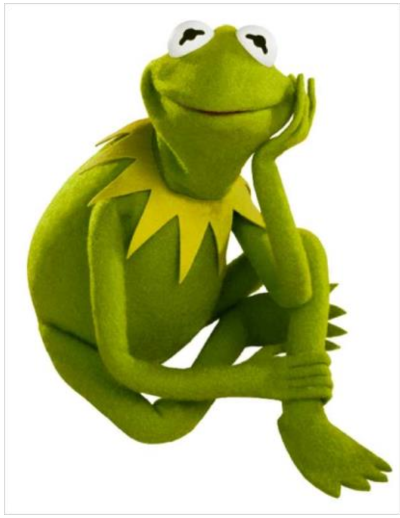


DNA:
AGCCCTAAGGGCTACCTAGCTT

DNA:
GACAGCCTACAAGCGTTAGCTTG

Longest Common Subsequence

- How similar are these two species?



DNA:
AGCCCTAAGGGCTACCTAGCTT

DNA:
GACAGCCTACAAGCGTTAGCTTG

- Pretty similar, their DNA has a long common subsequence:

AGCCTAAGCTTAGCTT

LCS Algorithm

- if $|X| = m$, $|Y| = n$, then there are 2^m subsequences of X; we must compare each with Y (n comparisons)
- So the running time of the brute-force algorithm is $O(n2^m)$
- Notice that the LCS problem has *optimal substructure*: Solutions of subproblems are parts of the final solution.
- Subproblems: "Find LCS of pairs of *prefixes* of X and Y"

LCS Algorithm

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i, Y_j to be the prefixes of X and Y of length *i* and *j* respectively.
- Given a sequence X = (x_1 x_2.....x_m) we define the i^{th} prefix of X for i=0, 1, and 2...m as X_i= (x_1 x_2.....x_i). For example: if X = (A, B, C, B, C, A, B, C) then X_4= (A, B, C, B)
- Define *c[i,j]* to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be *c[m,n]*

LCS recursive solution

$$c[i, j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- We start with $i = j = 0$ (empty substrings of x and y)
- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. $c[0,0] = 0$)
- LCS of empty string and any other string is empty, so for every i and j: $c[0, j] = c[i,0] = 0$

LCS recursive solution

$$c[i, j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- When we calculate *c[i,j]*, we consider two cases:
- **First case:** *x[i]=y[j]*: one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{j-1} , plus 1

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

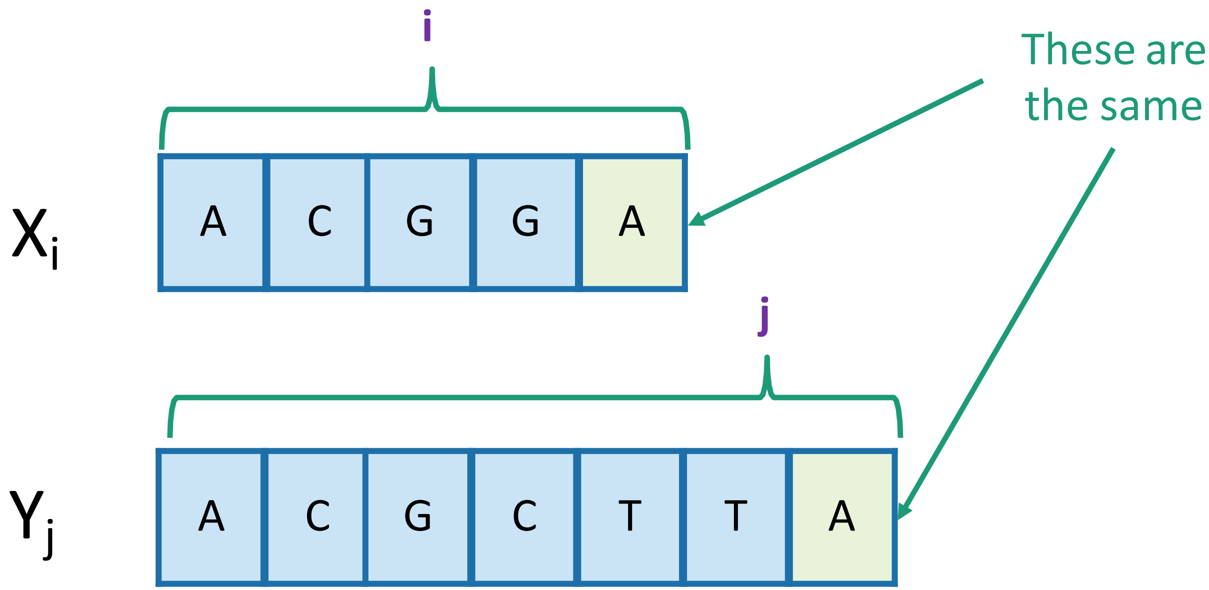
- **Second case:** *x[i] != y[j]*
- As symbols don't match, our solution is not improved, and the length of LCS(X_i , Y_j) is the same as before (i.e. maximum of LCS(X_i, Y_{j-1}) and LCS(X_{i-1},Y_j)

Why not just take the length of LCS(X_{i-1}, Y_{j-1})?

Two cases

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] = length_of_LCS(X_i, Y_j)

Case 1: X[i] = Y[j]

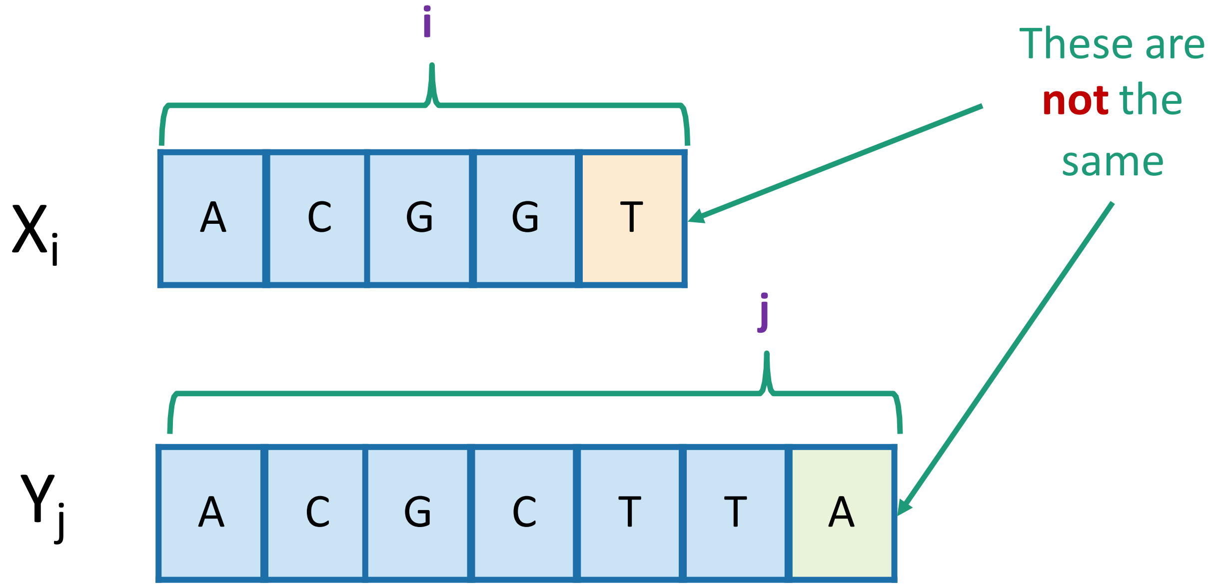


- Then C[i,j] = 1 + C[i-1,j-1].
  - because LCS(X_i,Y_j) = LCS(X_{i-1},Y_{j-1}) followed by $\boxed{A}$

Two cases

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] = length_of_LCS(X_i, Y_j)

Case 2: X[i] != Y[j]



Then C[i,j] = max{ C[i-1,j], C[i,j-1] }.

- either LCS(X_i,Y_j) = LCS(X_{i-1},Y_j) and $\boxed{\text{T}}$ is not involved,
- or LCS(X_i,Y_j) = LCS(X_i,Y_{j-1}) and $\boxed{\text{A}}$ is not involved,
- (maybe both are not involved, that's covered by the "or")

Recursive formulation of the optimal solution

Case 0

X_0	

Y_j	A	C	G	C	T	T	A

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{\, C[i, j-1], C[i-1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Case 1

X_i	A	C	G	G	A		
Y_j	A	C	G	C	T	T	A

Case 2

X_i	A	C	G	G	T		
Y_j	A	C	G	C	T	T	A

Computing the length of an LCS

LCS-LENGTH (*X*, *Y*)

```
m = length[X]
n = length[Y]
for i = 1 to m
      do c[i, 0] = 0
for j = 0 to n
      do c[0, j ] = 0
for i = 1 to m
       do for j = 1 to n
             do if x_i == y_j
                   then c[i, j ] = c[i–1, j–1] + 1
                        b[i, j ] = "↖"
                   else if c[i–1, j ] ≥ c[i, j–1]
                          then c[i, j ] = c[i– 1, j ]
                               b[i, j ] = "↑"
                          else c[i, j ] = c[i, j–1]
                               b[i, j ] = "←"
return c and b
```

$b[i, j\,]$ points to table entry whose subproblem we used in solving LCS of X_i and Y_j.

$c[m,n]$ contains the length of an LCS of X and Y.

Time: $O(mn)$

Aux. Space: $O(mn)$

Constructing an LCS

```
PRINT-LCS (b, X, i, j)
if i == 0 or j == 0
    then return
if b[i, j ] == "↖"
    then PRINT-LCS(b, X, i–1, j–1)
            print x_i
    elseif b[i, j ] == "↑"
            then PRINT-LCS(b, X, i–1, j)
else PRINT-LCS(b, X, i, j–1)
```

- Initial call is PRINT-LCS (b, X, m, n).
- Time: $O(m+n)$

LCS Example 1

We'll see how LCS algorithm works on the following example:

- X = ABCBDAB
- Y = BDCABA

What is the Longest Common Subsequence of X and Y?

LCS(X, Y) = BCBA

do if $x_i == y_j$
  then $c[i, j] = c[i-1, j-1] + 1$
    $b[i, j] =$ "↖"
  else if $c[i-1, j] \geq c[i, j-1]$
    then $c[i, j] = c[i-1, j]$
      $b[i, j] =$ "↑"
    else $c[i, j] = c[i, j-1]$
      $b[i, j] =$ "←"

	0	1 B	2 D	3 C	4 A	5 B	6 A
0							
1 A							
2 B							
3 C							
4 B							
5 D							
6 A							
7 B							

LCS Example 1
Given X=ABCBDAB and Y= BDCABA

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Length of the LCS: $c[7, 6] = 4$

LCS: <B, C, B, A>

Try this!!

Consider two strings A=*"qpqrr"* and B=*"pqprqrp"*. Let X be the length of the longest common subsequence between A and B and let Y be the number of such longest common subsequences between A and B. Then X+10Y= ___.

Question

- What are the values stored in the cells (A1, A2, A3, A4) for finding LCS using dynamic programming for strings DFGHP & DGUHP?

LCS	0	D	F	G	H	P
0	0	0	0	0	0	0
D	0	1	1	1	1	1
G	0	1	1	2	2	2
U	0	1	1	2	2	2
H	0	1	1	2	A1	3
P	0	1	1	A2	A3	A4

LCS Example 2

We'll see how LCS algorithm works on the following example:

- X = ABCB
- Y = BDCAB

What is the Longest Common Subsequence of X and Y?

LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

LCS Example 2

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,4]

LCS Example 2

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**					
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

for i = 1 to m $\qquad$ c[i,0] = 0
for j = 1 to n $\qquad$ c[0,j] = 0

LCS Example 2

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**				
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example 2

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**		
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example 2

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example 2

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example 2

ABCB
BDCAB

i		j	0	1	2	3	4	5
		Yj		**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**				
3	**C**		**0**					
4	**B**		**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example 2

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	
3	C	0					
4	B	0					

if (X_i == Y_j)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example 2

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example 2

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1			
4	B		0					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example 2

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**		
4	**B**	**0**					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example 2

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example 2

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**	**1**				

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example 2

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example 2

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	3

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Try this!

How will you find a longest palindromic subsequence in a given string using dynamic programming. Example: String A = " AABCDEBAZ"; Longest Palindromic subsequence: ABCBA or ABDBA or ABEBA

Take a string and its reverse as another string and then apply same LCS.

Solution

String A = " AABCDEBAZ"

Try this!

- Space optimized LCS

Approach 1: Using two arrays

- If we observe the previous 2-D solution of the problem, we are only using the adjacent indexes in the table to build the solution in a bottom-up manner. In other words, we are using LCS[i-1][j-1], LCS[i][j-1] and LCS[i-1][j] to fill the position LCS[i][j]. So there are two basic observations:
  - We are filling the entries of the table in a row-wise fashion.
  - To fill the current row, we only need the value stored in the previous row.
- So there is no need to store all rows in our DP matrix and we can just store two rows at a time. The time complexity of the above solution is O(m.n), where m and n are the length of given strings X and Y, respectively. The auxiliary space required by the program is O(n), which is independent of the length of the first string m.
- However, if the second string's length is much larger than the first string's length, then the space complexity would be huge. We can optimize the space complexity to O(min(m, n)) by passing a smaller string as a second argument to the LCSLength function.

```
//LCS of substring `X[0..m-1]` and `Y[0..n-1]`
int LCSLength(string X, string Y)
{
    int m = X.length(), n = Y.length();
    int curr[n + 1], prev[n + 1];
    for (int i = 0; i <= m; i++)
    {
        for (int j = 0; j <= n; j++)
        {
            if (i == 0 || j == 0) {
                curr[j] = 0;}
            else {
                // if the current character of `X` and `Y` matches
                if (X[i - 1] == Y[j - 1]) {
                    curr[j] = prev[j - 1] + 1;}
                // otherwise, if the current character of `X` and `Y` don't match
                else {
                    curr[j] = max(prev[j], curr[j - 1]);
                }}}
        // replace contents of the previous array with the current array
        for (int i = 0; i <= n; i++) {
            prev[i] = curr[i];
        }
    }
    return curr[n];
}
```

Approach 2: (Using one array)

- We can further optimize the code to use only a single array and a temporary variable.