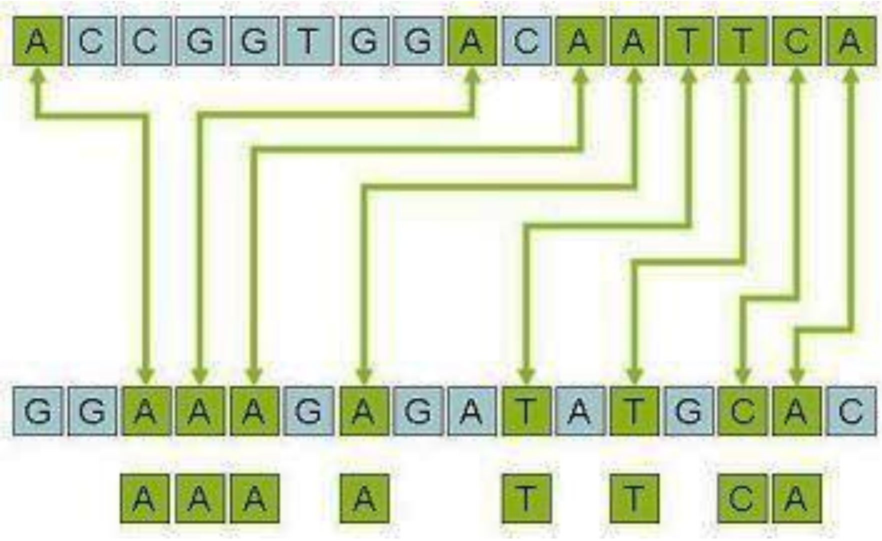
CSE- 207
Algorithms

Lecture: 14
Dynamic Programming-IV

Fahad Ahmed
Lecturer, Dept. of CSE
E-mail: fahadahmed@uap-bd.edu

Longest Common Subsequence (LCS)



Subsequence

- A **subsequence** is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

- For example, the sequence **{A, B, D}** is a subsequence of {A, B, C, D, E, F} obtained after removal of elements C, E, and F.

Sub-sequences vs. Substring

- **Subsequences** can contain consecutive elements which were not consecutive in the original sequence.

- **Substring** contains consecutive elements which were also consecutive in the original sequence.

- Example:
  - "gramm" is both subsequence and substring of "programming"
  - "gammg" is a subsequence of "programming" but not substring.
  - All substrings are subsequences but all subsequences are not substrings.

Sub-sequences vs. Substring

	Subarray	**Substring**	**Subsequence**	**Subset**
Contiguous	Yes	Yes	No	No
Elements Ordered	Yes	Yes	Yes	No

Common subsequence

- Given two sequences *X* and *Y*, a sequence *Z* is said to be a *common subsequence* of *X* and *Y*, if *Z* is a subsequence of both *X* and *Y*.
- For example, if
  - X = e c d g i   Y = a b c d e f g h i j

then Z is the common subsequence of X and Y.

  - Z = e g i ,

    c d g i, ….

- Longest Common Subsequence (LCS) will be c d g i that is 4 .

COMMON SUBSEQUENCE

- Another example, if
  - X = babace
  - Y = abdace

Z is the common subsequence of X and Y.

  - Z = bace,

    abce, ….

- There can be multiple Subsequence with **same length.**

Application

- To compare the DNA of two (or more) different organisms.
  - One reason to compare two strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are.

  - DNA is represented as strings of the small alphabet, Sigma = {A, C, G, T}.
  - DNA strings can be changed by **mutation** by insertion of a character into string.

  - LCS is widely used by revision control systems and in biometrics.

Longest Common Subsequence

- Given two sequences $x[1\,.\,.\,m]$ and $y[1\,.\,.\,n]$, find a longest subsequence common to them both.

"a" *not* "the"

x: A B C B D A B

y: B D C A B A

$$\left.\begin{array}{l} x: \text{A B C B D A B} \\ y: \text{B D C A B A} \end{array}\right\} \text{BCBA} = \text{LCS}(x, y)$$

functional notation, but not a function

Brute-force LCS algorithm

Check every subsequence of $x[1\,.\,.\,m]$ to see if it is also a subsequence of $y[1\,.\,.\,n]$.

Analysis

- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).
- Hence, the runtime would be exponential !

Towards a better algorithm: a DP strategy

- Key: optimal substructure and overlapping sub-problems
- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.

RECURSIVE ALGORITHM FOR LCS

$$
\begin{aligned}
&\text{LCS}(x, y, i, j) \\
&\quad \textbf{if } x[i] = y[j] \\
&\qquad \textbf{then } c[i, j] \leftarrow \text{LCS}(x, y, i+1, j+1) + 1 \\
&\qquad \textbf{else } c[i, j] \leftarrow \max\left\{ \begin{array}{l} \text{LCS}(x, y, i+1, j), \\ \text{LCS}(x, y, i, j+1) \end{array} \right\}
\end{aligned}
$$

Worst-case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

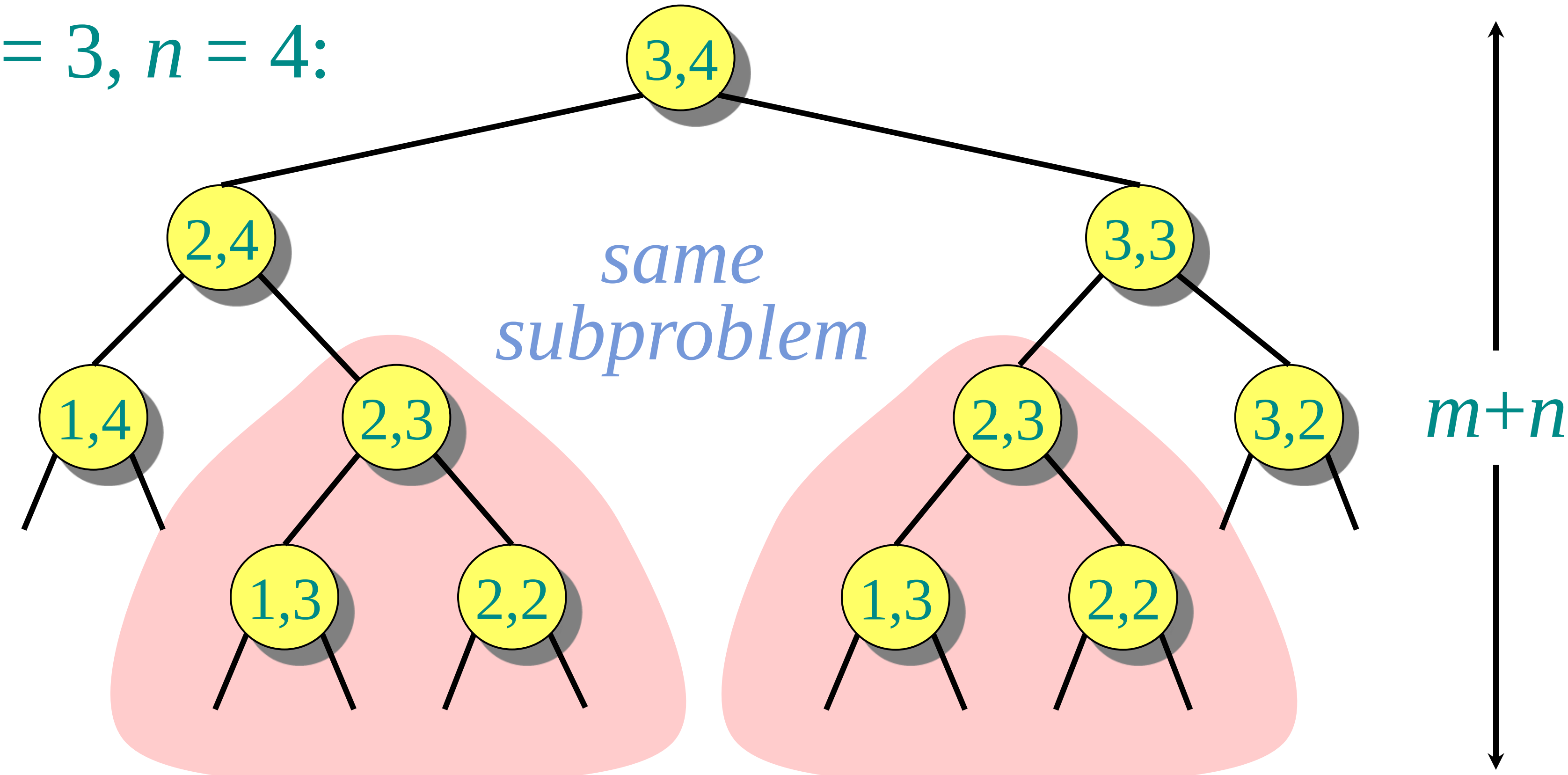
Recursive algorithm for LCS

```
int lcs(string X, string Y, int i, int j)
{
    if (i == M || j == N)
        return 0;

    if (X[i] == Y[j])
        return (lcs(X, Y, i+1, j+1) + 1);
    else
        return max(lcs(X, Y, i+1, j), lcs(X, Y, i, j+1));
}
```

Recursion tree

$m = 3, n = 4$:

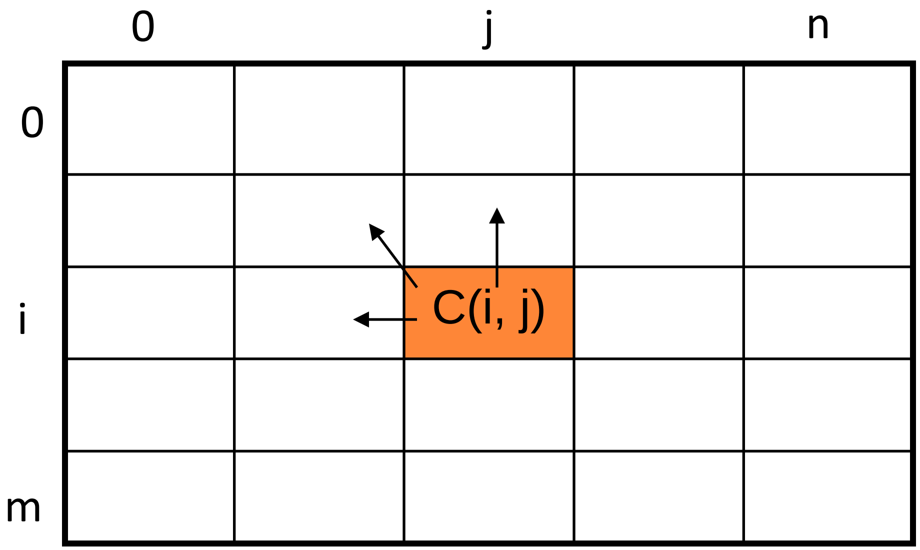


Height $= m + n \Rightarrow$ work potentially exponential, but we're solving subproblems already solved!

DP ALGORITHM

- Key: find out the correct order to solve the sub-problems
- Total number of sub-problems: $m * n$

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



DP Algorithm

```
LCS-Length(X, Y)
1. m = length(X)  // get the # of symbols in X
2. n  = length(Y) // get the # of symbols in Y
3. for i = 1 to m          c[i,0] = 0        // special case: Y[0]
4. for j = 1 to n          c[0,j] = 0        // special case: X[0]
5. for i = 1 to m                            // for all X[i]
6.        for j = 1 to n                     // for all Y[j]
7.               if ( X[i] == Y[j])
8.                      c[i,j] = c[i-1,j-1] + 1
9.               else c[i,j] = max( c[i-1,j], c[i,j-1] )
10. return c
```

LCS Solution Bottom Up : Tabulation

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

- X = ABCB
- Y = BDCAB

What is the LCS of X and Y?

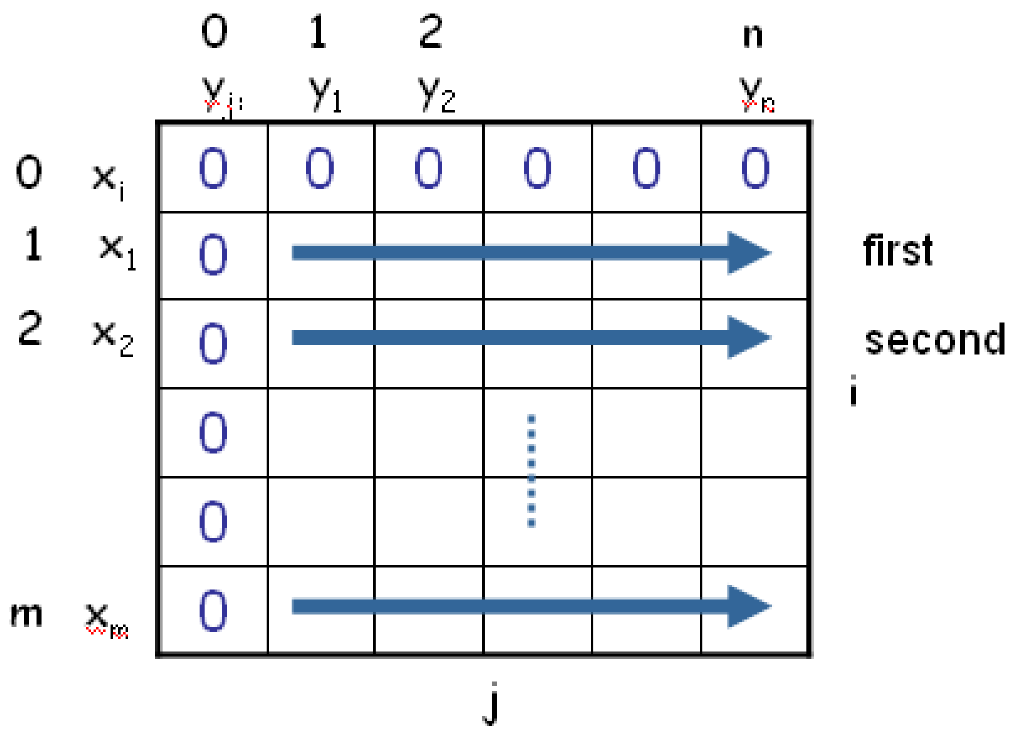
LCS(X, Y) = BCB

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

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

Computing the Length of the LCS

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } x_i \neq y_j \end{cases}$$



ABCB
BDCAB

LCS Example (0)

i		j: 0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,6]

ABCB
BDCAB

LCS Example (1)

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

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

LCS Example (2)

ABCB

BDCAB

	j	0	1	2	3	4	5
i		Y[j]	B	D	C	A	B
0	X[i]	**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 (3)

ABCB

BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**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 (4)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**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 (5)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB

BDCAB

LCS Example (6)

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**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 (7)

ABCB

BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB
BDCAB

LCS Example (8)

		j	0	1	2	3	4	5
i			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**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 (9)

ABCB
BDCAB

			j	0	1	2	3	4	5
i				Y[j]	B	D	C	A	B
0	X[i]			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 (10)

ABCB
BDCAB

i		j →	0	1	2	3	4	5
		Y[j]		**B**	**D**	**C**	**A**	**B**
0	X[i]		**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 (11)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**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 (12)

ABCB
BDCAB

j			0	1	2	3	4	5
i		Y[j]		B	D	C	A	B
0	X[i]		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 (13)

ABCB
BDCAB

i		j: 0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**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 (14)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**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])

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

Time Complexity: We can see that the time complexity of the DP and memoization approach is reduced to **O(m*n),** where **m** and **n** are the lengths of the given strings.

since each c[i,j] is calculated in constant time, and there are m*n elements in the array

How to find actual LCS

- The algorithm just found the *length* of LCS, but not LCS itself.
- How to find the actual LCS?
- For each c[i,j] we know how it was acquired:

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

- A match happens only when the first equation is taken
- So we can start from *c[m,n]* and go backwards, remember *x[i]* whenever *c[i,j] = c[i-1, j-1]+1*.

2	2
2	3

For example, here
c[i,j] = c[i-1,j-1] +1 = 2+1=3

Finding LCS

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**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**

Time for trace back: $O(m+n)$.

FINDING LCS (2)

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**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**

LCS (reversed order): **B C B**

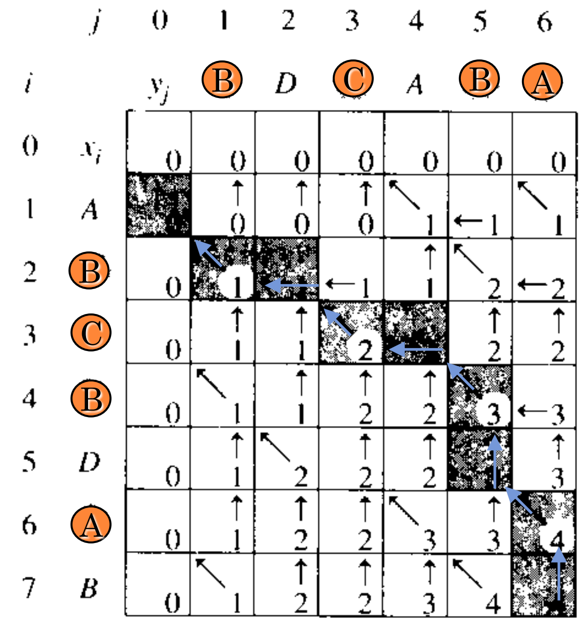
LCS (straight order): **B C B**

(this string turned out to be a palindrome)

Compute 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
```



source: textbook Cormen, et al.

Construct 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)
```

For the b table in Figure 15.6, this procedure prints "$BCBA$." The procedure takes time $O(m+n)$, since at least one of i and j is decremented in each stage of the recursion.

LCS-Length(X, Y) // dynamic programming solution

m = X.length()
n = Y.length()
for i = 1 to m do c[i,0] = 0
for j = 0 to n do c[0,j] = 0 **O(nm)**
for i = 1 to m do // row
for j = 1 to n do // cloumn
if x_i = =y_i 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] = “<”

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

First Optimal-LCS initializes row 0 and column 0

	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**			
4	***B***	**0**						
5	***D***	**0**						
6	***A***	**0**						
7	***B***	**0**						

Next each c[*i*, *j*] is computed, row by row, starting at c[1,1].
If $x_i == y_j$ then c[i, j] = c[i-1, j-1]+1
and b[i, j] = ↖

	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		
4	*B*	0						
5	*D*	0						
6	*A*	0						
7	*B*	0						

If x_i <> y_j then c[i, j] = max(c[i-1, j], c[i, j-1]) and b[i, j] points to the larger value

	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**	
4	***B***	**0**						
5	***D***	**0**						
6	***A***	**0**						
7	***B***	**0**						

if c[i-1, j] == c[i, j-1]
then b[i,j] points up

	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**

To construct the LCS, start in the bottom right-hand corner and follow the arrows. A ↖ indicates a matching character.

	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**

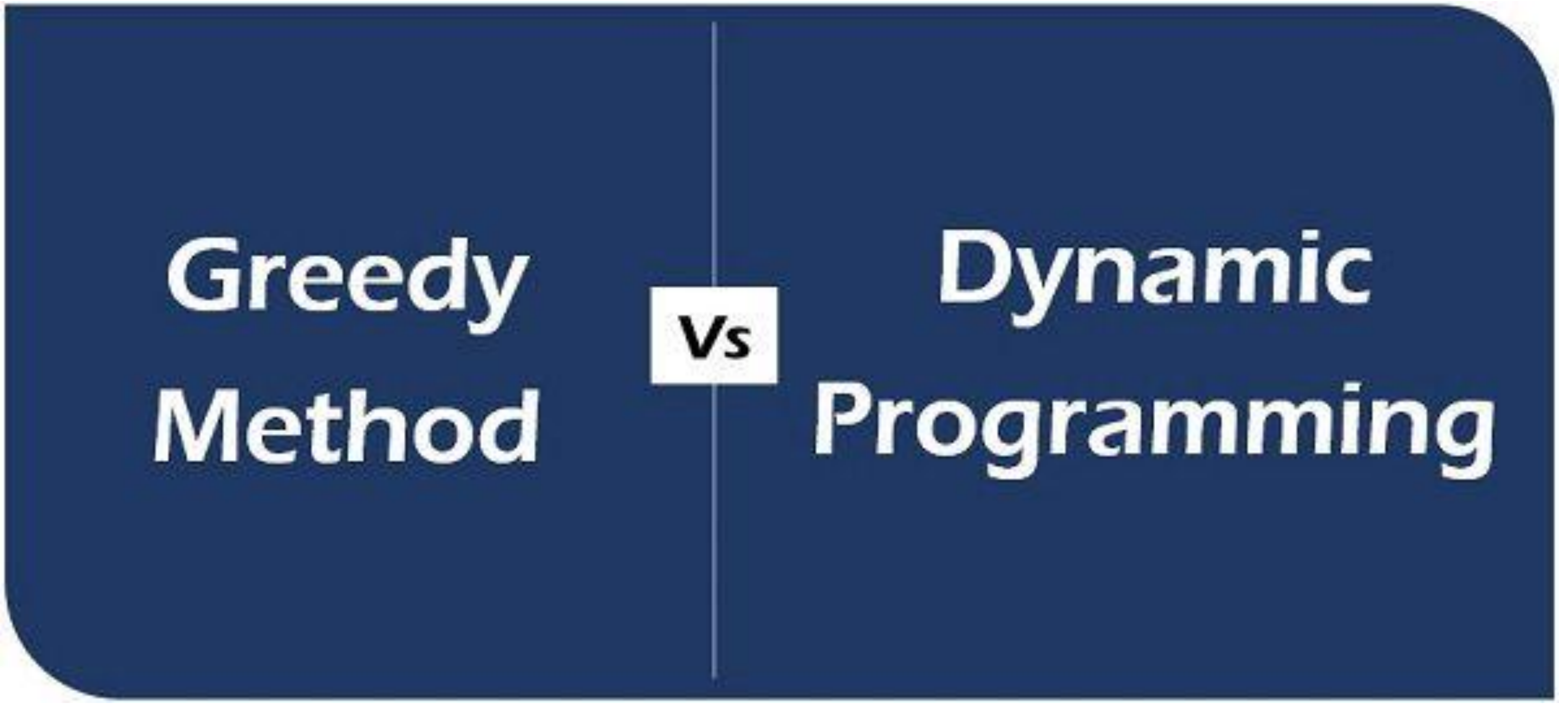
LCS: B C B A

LCS Using Memorization: Top Down

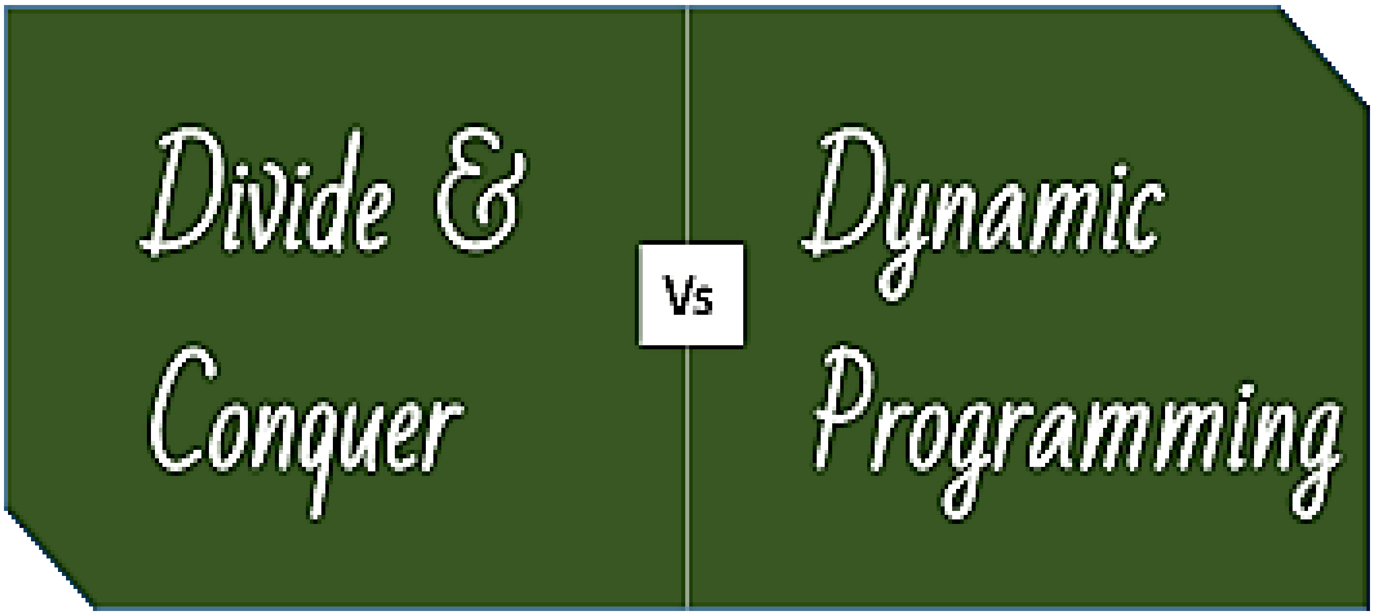
```
int lcs(string S, string T, int i, int j, int memo[][max_value])
{
    if (i == N || j == M)
        return 0;

    if (memo[i][j] != -1)
        return memo[i][j];

    if (S[i] == T[j])
        memo[i][j] = lcs(S, T, i + 1, j + 1, memo) + 1;
    else
        memo[i][j] = max(lcs(S, T, i + 1, j, memo),
                         lcs(S, T, i, j + 1, memo));
    return memo[i][j];
}
```


Greedy Method
Vs
Dynamic Programming

BASIS FOR COMPARISON	GREEDY METHOD	DYNAMIC PROGRAMMING
Basic	Generates a single decision sequence.	Many decision sequence may be generated.
Reliability	Less reliable	Highly reliable
Follows	Top-down approach.	Bottom-up approach.
Solutions	Contain a particular set of feasible set of solutions.	There is no special set of feasible set of solution.
Efficiency	More	Less
Overlapping subproblems	Cannot be handled	Chooses the optimal solution to the subproblem.
Example	Fractional knapsack, shortest path.	0/1 Knapsack


Divide & Conquer
Vs
Dynamic Programming

Divide and Conquer	Dynamic Programming
Divide and conquer is the top down approach.	Dynamic programming is bottom up approach.
Divide and conquer prefers recursion.	Dynamic programming prefers iteration.
In divide and conquer, sub problems are independent.	Sub problems of dynamic programming are dependent and overlapping.
Solutions of sub problems are not stored.	Solutions of sub problems are stored in the table.
Lots of repetition of work.	No repetition at work at all.
It splits input at a specific point.	It splits input at each and every possible point.
Less efficient due to rework.	More efficient due to the saving of solution.
Solution using divide and conquer is simple.	Sometimes, a solution using dynamic programming is complex and tricky.
Only one decision sequence is generated.	Many decision sequences may be generated.

ADDITIONAL PROBLEM BASED ON DP

1. Longest Increasing Subsequence
2. Edit Distance
3. Minimum Partition
4. Ways to Cover a Distance
5. Longest Path In Matrix
6. Subset Sum Problem
7. Optimal Strategy for a Game
8. Boolean Parenthesization Problem
9. Shortest Common Supersequence
10. Matrix Chain Multiplication
11. Partition problem
12. Word Break Problem
13. Maximal Product when Cutting Rope
14. Dice Throw Problem
15. Box Stacking
16. Egg Dropping Puzzle

Reference

- https://www.codingninjas.com/blog/2021/09/01/longest-common-subsequence/
- https://www.javatpoint.com/longest-common-subsequence

- **https://www.geeksforgeeks.org/dynamic-programming/**

Thanks to All