MA 515: Introduction to Algorithms & MA353 : Design and Analysis of Algorithms [3-0-0-6]

Lecture 27

http://www.iitg.ernet.in/psm/indexing_ma353/y09/index.html

Partha Sarathi Manal

psm@iitg.ernet.in

Dept. of Mathematics, IIT Guwahati

Mon 10:00-10:55 Tue 11:00-11:55 Fri 9:00-9:55

Class Room : 2101

Dynamic programming

- One of the most important algorithm tools!
- Very common interview question

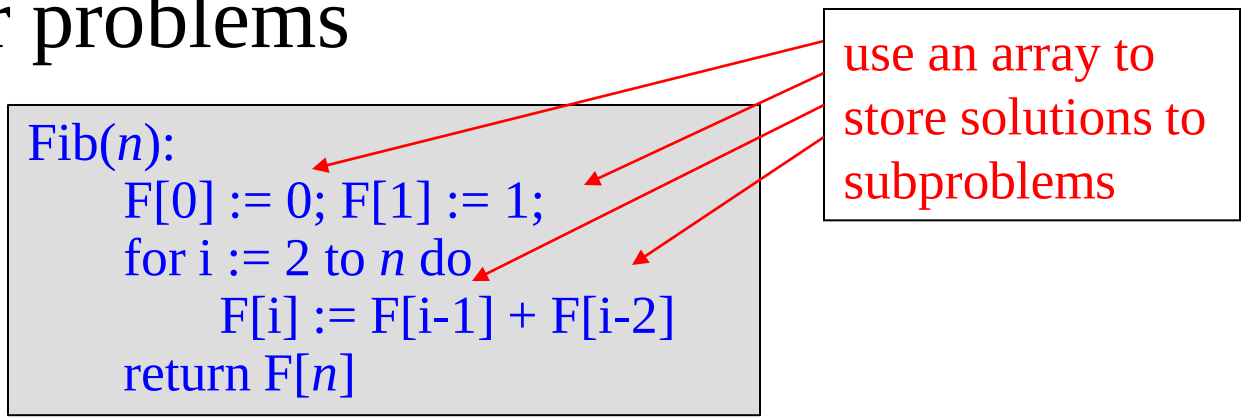
- Method for solving problems where optimal solutions can be defined in terms of optimal solutions to sub-problems AND
- the sub-problems are overlapping

Identifying a dynamic programming problem

- The solution can be defined with respect to solutions to subproblems
- The subproblems created are overlapping, that is we see the same subproblems repeated

Two main ideas for dynamic programming

- Identify a solution to the problem with respect to **smaller** subproblems
  - F(n) = F(n-1) + F(n-2)
- Bottom up: start with solutions to the smallest problems and build solutions to the larger problems



Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$

$$X = A\ B\ A\ C\ D\ A\ B\ A\ B$$

ABA?

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$

$$X = A\ B\ A\ C\ D\ A\ B\ A\ B$$

$$ABA$$

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$

$$X = A\ B\ A\ C\ D\ A\ B\ A\ B$$

ACA?

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$

$$X = A\ B\ A\ C\ D\ A\ B\ A\ B$$

$$A\ C\ A$$

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$

$$X = A\ B\ A\ C\ D\ A\ B\ A\ B$$

DCA?

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$

$$X = A\ B\ A\ C\ D\ A\ B\ A\ B$$

~~DCA~~

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$

$X = A\ B\ A\ C\ D\ A\ B\ A\ B$

AADAA?

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$

$X = A\ B\ A\ C\ D\ A\ B\ A\ B$

AADAA

LCS problem

- Given two sequences X and Y, a **common subsequence** is a subsequence that occurs in both X and Y
- Given two sequences $X = x_1, x_2, \ldots, x_m$ and $Y = y_1, y_2, \ldots, y_n$, What is the longest common subsequence?

X = A B C B D A B

Y = B D C A B A

LCS problem

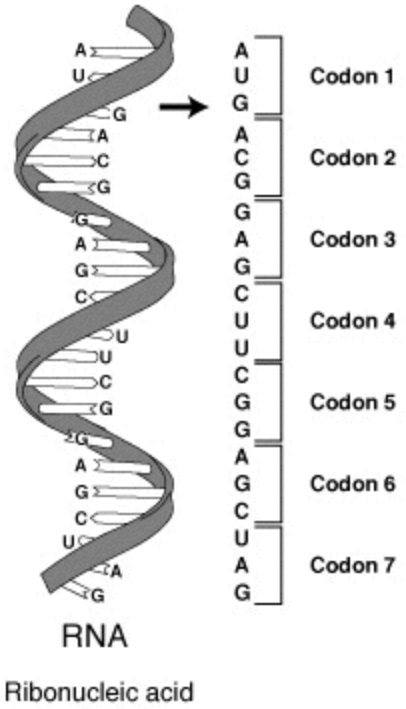
- Given two sequences X and Y, a **common subsequence** is a subsequence that occurs in both X and Y
- Given two sequences $X = x_1, x_2, \ldots, x_m$ and $Y = y_1, y_2, \ldots, y_n$, What is the longest common subsequence?

X = A B C B D A B

Y = B D C A B A

The sequences{B, D, A, B} of length 4 are the LCS of *X* and *Y*, since there is no common subsequence of length 5 or greater.

LCS problem



Application:

comparison of two DNA strings

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

LCS Algorithm

- Brute-force algorithm: For every subsequence of x, check if it's a subsequence of y
  - *How many subsequences of x are there ?*
  - *What will be the running time of the brute-force algorithm ?*

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 $O(n\ 2^m)$
- 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
- 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 $c[m,n]$
- $c[m,n]$ is the final solution.

Step 1: Define the problem with respect to subproblems

X = A B C B D A B

Y = B D C A B A

Step 1: Define the problem with respect to subproblems

X = A B C B D A ?

Y = B D C A B ?

Is the last character part of the LCS?

Step 1: Define the problem with respect to subproblems

X = A B C B D A ?

Y = B D C A B ?

Two cases: either the characters are the same or they're different

Step 1: Define the problem with respect to subproblems

X = A B C B D A A

LCS

Y = B D C A B A

The characters are part of the LCS

If they're the same

$$LCS(X, Y) = LCS(X_{m-1}, Y_{n-1}) + 1$$

Step 1: Define the problem with respect to subproblems

X = A B C B D A B

LCS

Y = B D C A B A

If they're different

$$LCS(X, Y) = LCS(X_{m-1}, Y)$$

Step 1: Define the problem with respect to subproblems

X = A B C B D A B

LCS

Y = B D C A B A

If they're different

$$LCS(X, Y) = LCS(X, Y_{n-1})$$

Step 1: Define the problem with respect to subproblems

X = A B C B D A B

Y = B D C A B A

?

X = A B C B D A B

Y = B D C A B A

If they're different

Step 1: Define the problem with respect to subproblems

X = A B C B D A B

Y = B D C A B A

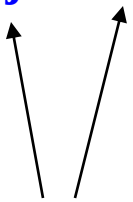
$$LCS(X,Y) = \begin{cases} 1 + LCS(X_{m-1}, Y_{n-1}) & \text{if } x_m = y_n \\ \max(LCS(X_{m-1}, Y), LCS(X, Y_{n-1})) & \text{otherwise} \end{cases}$$

Step 2: Build the solution from the bottom up

$$LCS(X,Y) = \begin{cases} LCS(X_{m-1}, Y_{n-1}) + 1 & \text{if } x_m = y_n \\ \max(LCS(X_{m-1}, Y), LCS(X, Y_{n-1})) & \text{otherwise} \end{cases}$$

What types of subproblem solutions do we need to store?

$LCS(X_j, Y_k)$



two different indices

Step 2: Build the solution from the bottom up

$$LCS(X,Y) = \begin{cases} LCS(X_{m-1}, Y_{n-1}) + 1 & if\ x_m = y_n \\ \max(LCS(X_{m-1}, Y), LCS(X, Y_{n-1})) & otherwise \end{cases}$$

What types of subproblem solutions do we need to store?

$LCS(X_i, Y_j)$

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

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_{i-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:** $x_i \neq y_j$
- 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}) ?

LCS Length 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: Y0
4. for j = 1 to n      c[0,j] = 0           // special case: X0
5. for i = 1 to m                           // for all Xi
6.       for j = 1 to n                     // for all Yj
7.             if ( Xi == Yj )
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 Example

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 (0)

ABCB
BDCAB

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[5,4]

LCS Example (1)

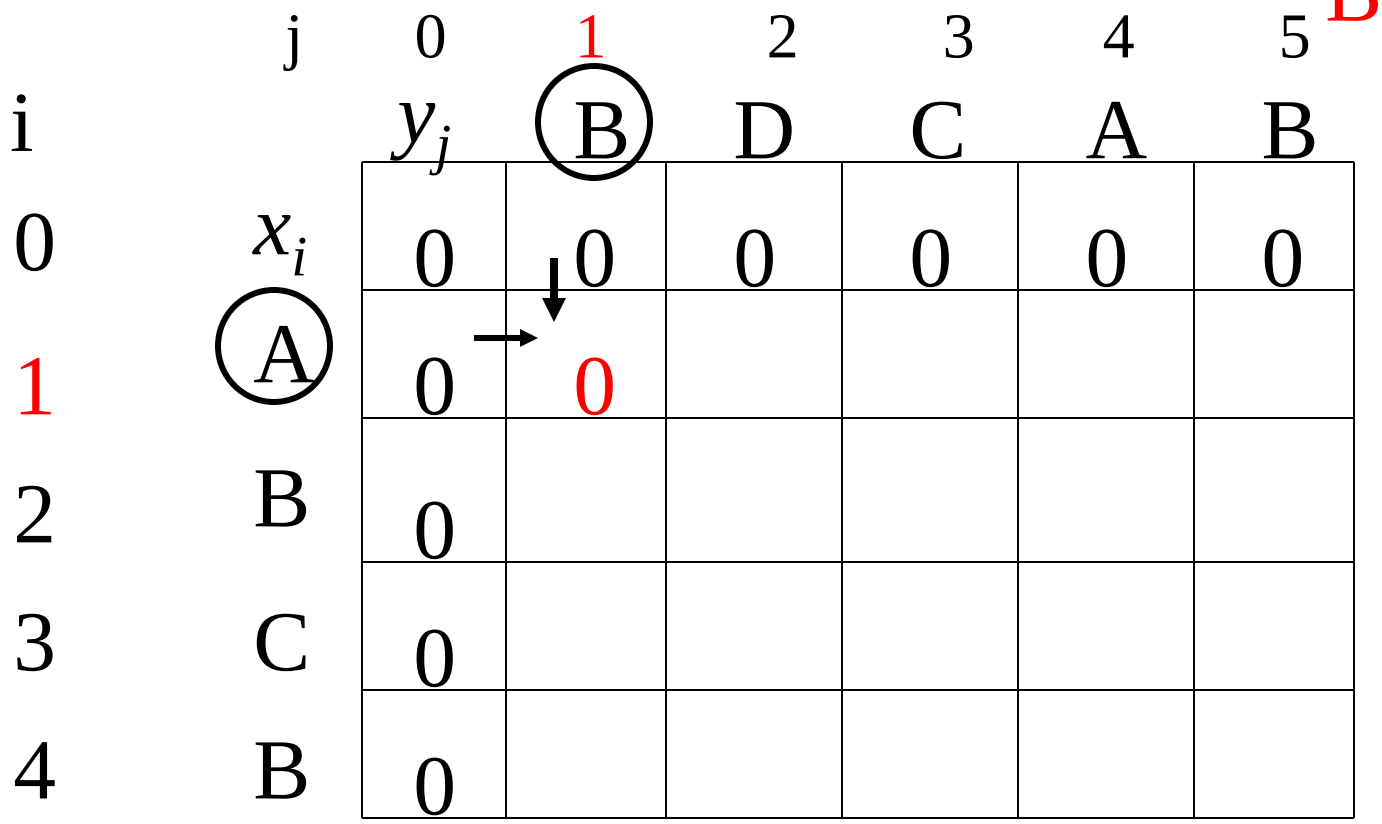
ABCB
BDCAB

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					
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



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

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		
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

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	
2	B		0					
3	C		0					
4	B		0					

if ($X_i == Y_j$)

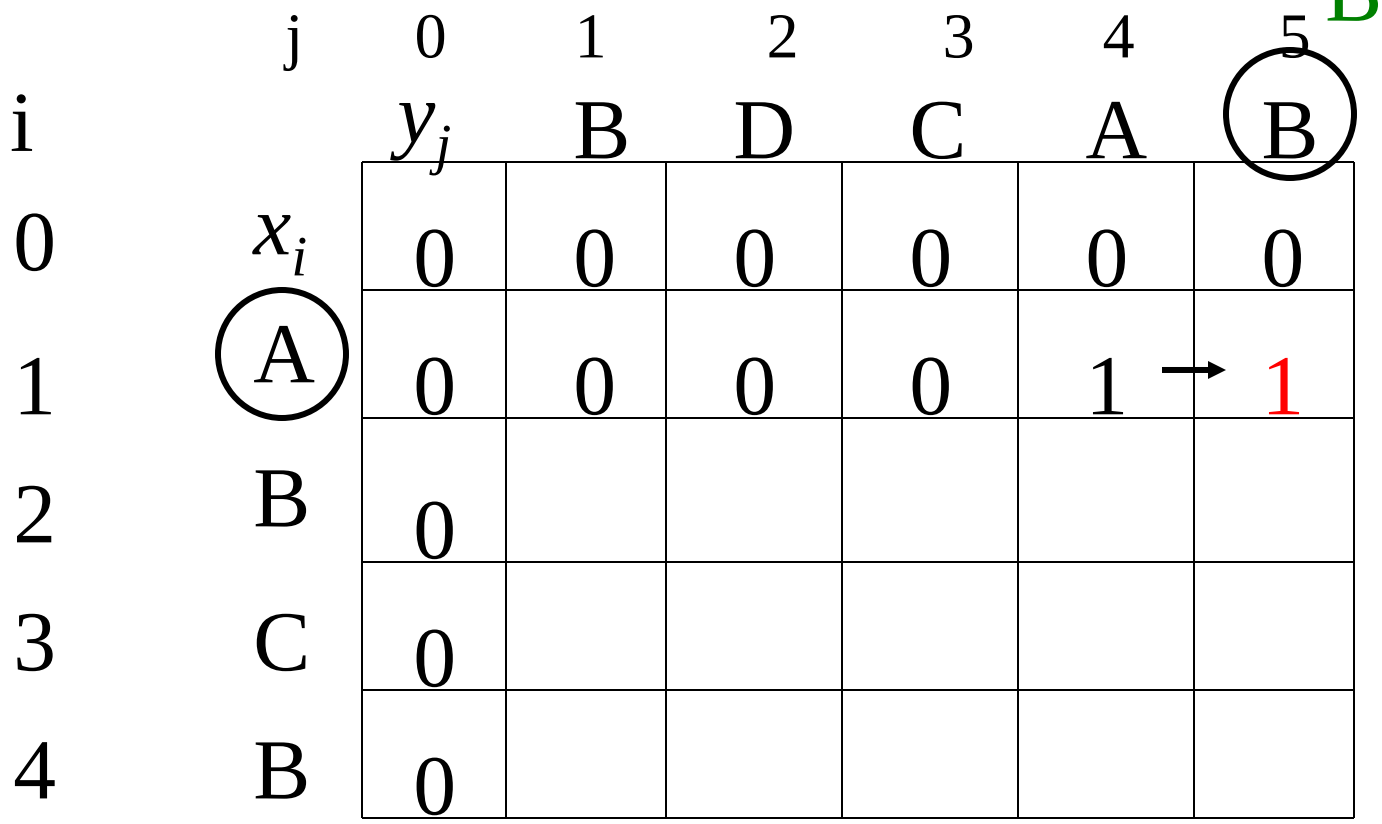
c[i,j] = c[i-1,j-1] + 1

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

LCS Example (5)

ABCB

BDCAB



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 (6)

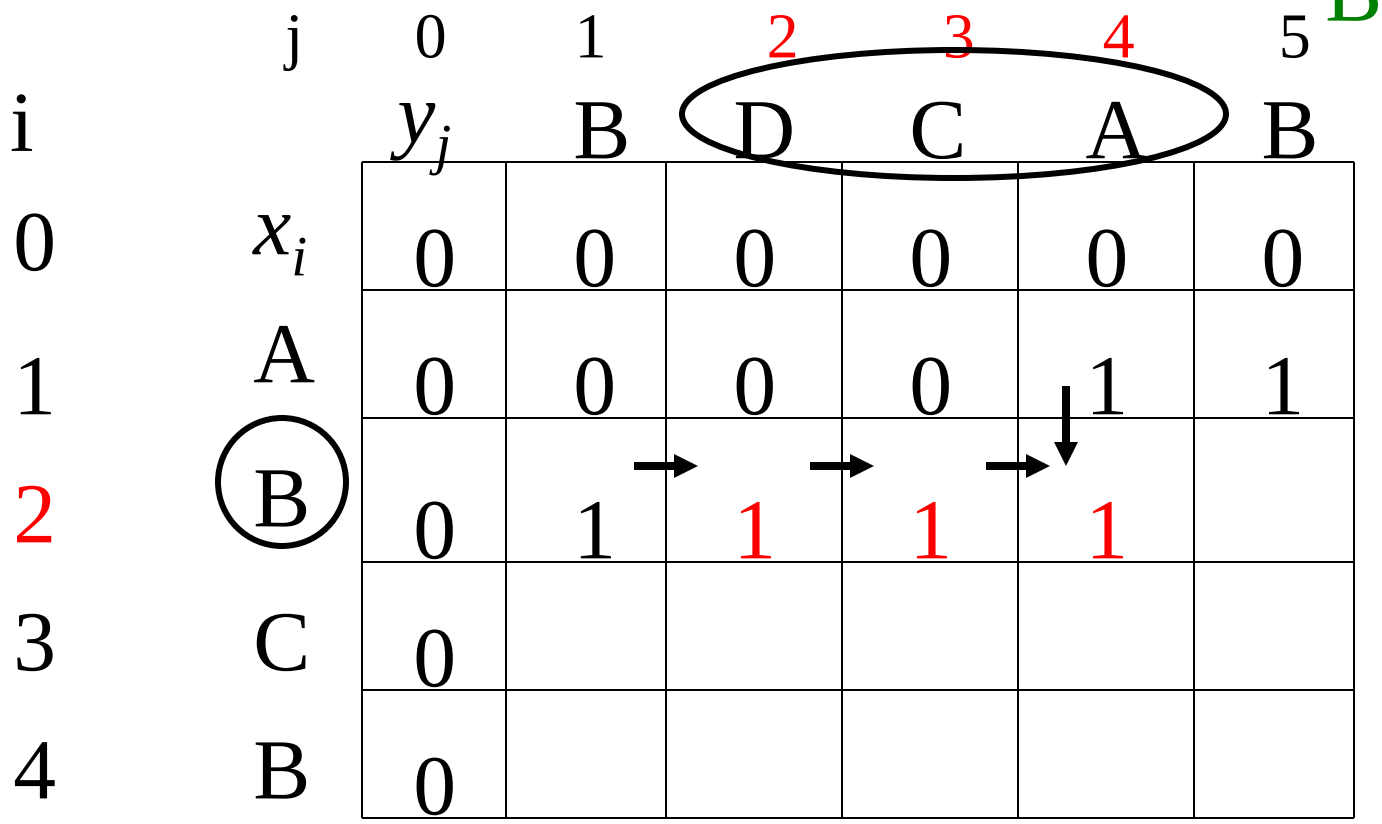
ABCB
BDCAB

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				
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



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 (8)

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					
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			
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

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 (12)

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					

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				

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

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 (15)

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	3

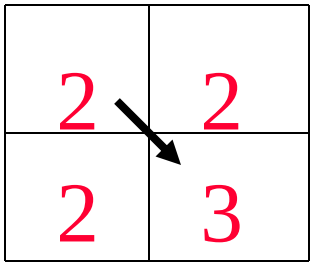
if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?
- O(*mn*)
- since each c[i,j] is calculated in constant time, and there are *mn* elements in the array

How to find actual LCS

- So far, we have just found the *length* of LCS, but not LCS itself.
- We want to modify this algorithm to make it output Longest Common Subsequence of X and Y
- Each c[i,j] depends on c[i-1,j] and c[i,j-1] or c[i-1,j-1]
- For each c[i,j] we can say how it was acquired:



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

How to find actual LCS - continued

- Remember that

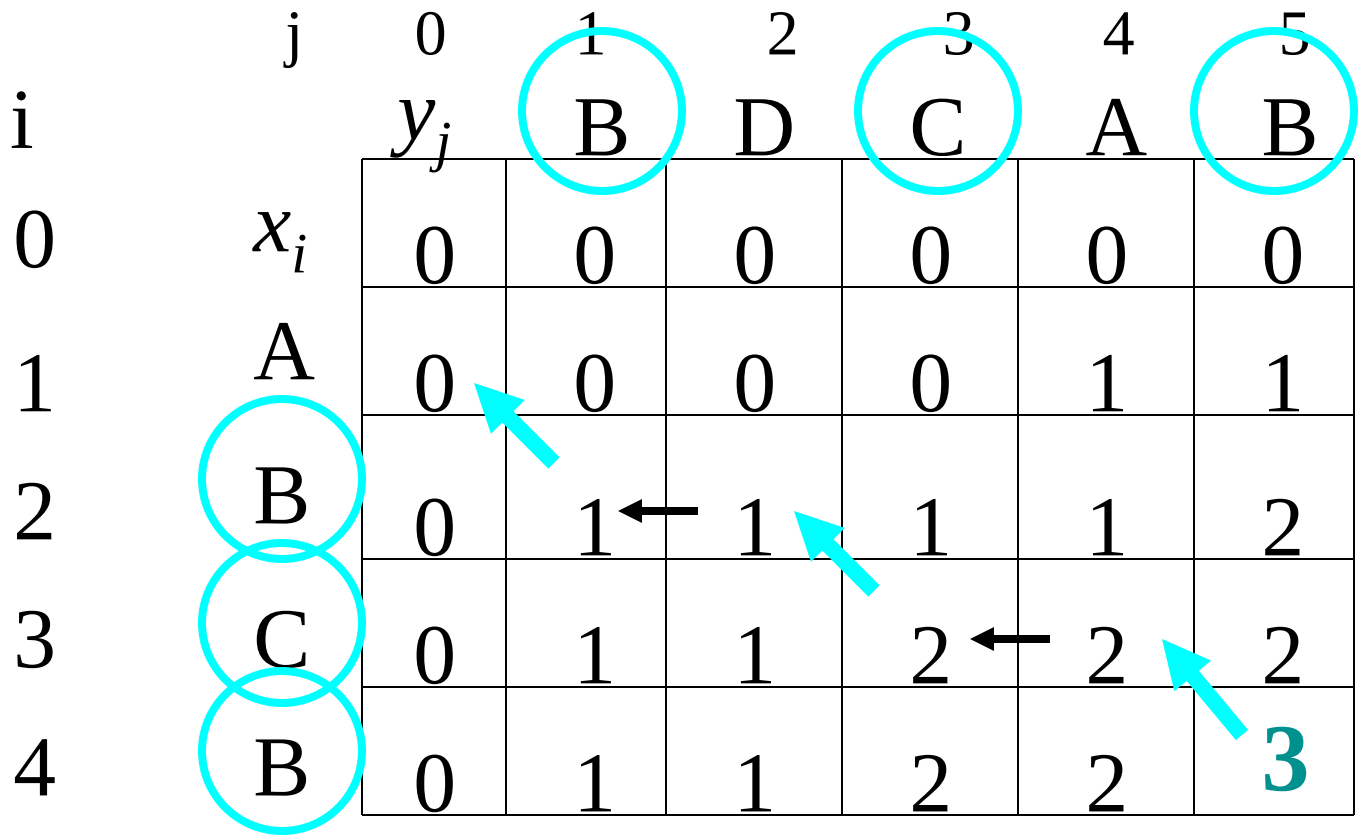
$$c[i, j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

  - So we can start from c[m,n] and go backwards
  - Whenever c[i,j] = c[i-1, j-1]+1, remember *x[i]* (because x[i] is a part of LCS)
  - When i=0 or j=0 (i.e. we reached the beginning), output remembered letters in reverse order

Finding LCS

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	**3**

Finding LCS (2)



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

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

(this string turned out to be a palindrome)

Complexity Analysis

- After we have filled the array *c*[ ], we can use this data to find the characters that constitute the Longest Common Subsequence
- Algorithm runs in *O(mn)*, which is *much* better than the brute-force algorithm: $O(n2^m)$