Dynamic Programming

Prof. S.J. Soni

Introduction

- Idea is Very Simple..

  Avoid calculating the same thing twice, usually by keeping a table of known results that fills up as subinstances are solved.

- Dynamic Programming is a *bottom-up* technique.

  We usually start with the smallest, and hence the simplest, subinstances. By combining their solutions, we obtain the answers to subinstances of increasing size, until finally we arrive at the solution of the original instance.

Calculating the binomial coefficient

- Consider the problem of calculating the binomial coefficient

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{for all integers } n, k > 0,$$

$$\frac{n!}{k!\,(n-k)!}$$

$$\binom{n}{0} = 1 \quad \text{for all integers } n \geq 0,$$

$$\binom{0}{k} = 0 \quad \text{for all integers } k > 0.$$

Calculating the binomial coefficient

Function C (n,k)

if k=0 or k=n then return 1

else return C (n-1,k-1) + C (n-1,k)

Calculate for C (5,3)

C (4,2),	C (4,3)
C (3,1), C (3,2)	C (3,2), C (3,3)
C (2,0), C (2,1), C (2,1), C(2,2)	C (2,1), C (2,2), C (2,2), C (2,3)

This algorithm calculates same values again & again, so that it is inefficient algorithm.

Solution using Dynamic Programming

Pascal's Triangle

```
0:                 1
1:               1   1
2:             1   2   1
3:           1   3   3   1
4:         1   4   6   4   1
5:       1   5  10  10   5   1
6:     1   6  15  20  15   6   1
7:   1   7  21  35  35  21   7   1
8: 1   8  28  56  70  56  28   8   1
```

Solution using Dynamic Programming

Pascal's Triangle

	0	1	2	3	4	5	…….	k-1	k
0:	1								
1:	1	1							
2:	1	2	1						
3:	1	3	3	1					
4:	1	4	6	4	1				
5:	1	5	10	10	5	1			

n-1:	C(n-1,k-1)	C(n-1,k)
n:		C(n,k)

Making Change (2) Problem

- There are several instances of problem in which greedy algorithm fails to generate answer.
- For example,

  There are coins for 1,4 and 6 units. If we have to make change for 8 units, the greedy algorithm will propose doing so using one 6-unit coin and two 1-unit coins, for a total of three coins.

  However it is clearly possible to do better than this: we can give the customer his change using just two 4-unit coins.

  Although the greedy algorithm does not find this solution, it is easily obtained using dynamic programming.

Solution using Dynamic Programming

- To solve this problem using DP, we set up a table c [1…n, 0…N], with one row for each available denomination and one column for each amount from 0 to N units.
- In this table c [i,j] will be the minimum number of coins required to pay an amount of j units, 0<= j <=N.
- c [i,j] = min (c[i-1,j] , 1+c[i, j - di])

Solution using Dynamic Programming

Amount		0	1	2	3	4	5	6	7	8
d1=1	1	0	1	2	3	4	5	6	7	8
d2=4	2	0	1	2	3	1	2	3	4	2
d3=6	3	0	1	2	3	1	2	1	2	**2**

c [i,j] = min (c[i-1,j] , 1+c[i, j - di])

c [2,1] = min (c[1,1] , 1+c[2, 1 - 4]) = min(1,Inf) =1
c [2,2] = min (c[1,2] , 1+c[2, 2 - 4]) = min(2,Inf) =2
c [2,4] = min (c[1,4] , 1+c[2, 4 - 4]) = min(4,1) =1

c [3,8] = min (c[2,8] , 1+c[3, 8 - 6]) = min(2,3) =2

Solution using Dynamic Programming

- Which Coins?

Amount		0	1	2	3	4	5	6	7	8
d1=1	1	0	1	2	3	4	5	6	7	8
d2=4	2	0	1	2	3	1	2	3	4	2
d3=6	3	0	1	2	3	1	2	1	2	2

If c[i,j]=c[i-1,j], we move up to c[i-1, j]

c[3,8] = c[2,8] , we move on to c[2,8]

Otherwise check c[i,j]=1+c[i,j-di]

so check c[2,8]=1+c[2,8-4] = 1+c[2,4]=2 *[select coin]*

Again check c[2,4]<>c[1,4] so, c[2,4]=1+c[2,4-4]=1 *[select coin]*

Knapsack Problem (2)

- We are given number of objects and knapsack.
- This time, however, we suppose that the objects may not be broken into smaller pieces, we may decide either to take an object or to leave it behind, but we may not take a fraction of an object.
- So it's called Non-fractional knapsack problem or 0/1 knapsack problem.

Knapsack Problem (2)

- Example for which greedy method cannot work.

Object	1	2	3
Weight	6	5	5
Value	8	5	5

- W=10
- Greedy method generates the value 8 which is not optimal.

Solution using Dynamic Programming

- To solve this problem using DP, we set up a table V [1…n, 0…W], with one row for each available object and one column for each weight from 0 to W.
- In this table V [i,j] will be the maximum value of the objects , 0<= j <=W & 1<= i <=n
- V [i,j] = max (V[i-1,j] , V[i-1, j - wi]+vi)

Knapsack Problem (2) Example

Objects	1	2	3	4	5
Weight	1	2	5	6	7
Value	1	6	18	22	28
Vi/Wi	1	3	3.6	3.67	4

W=11

Solution using Dynamic Programming

Weight Limit	0	1	2	3	4	5	6	7	8	9	10	11
w1=1, v1=1	0	1	1	1	1	1	1	1	1	1	1	1
w2=2 v2=6	0	1	6	7	7	7	7	7	7	7	7	7
w3=5, v3=18	0	1	6	7	7	18	19	24	25	25	25	25
w4=6, v4=22	0	1	6	7	7	18	22	24	28	29	29	40
w5=7, v5=28	0	1	6	7	7	18	22	28	29	34	35	**40**

$$V[i,j] = \max(V[i-1,j], V[i-1, j - wi]+vi)$$

$$V[5,11] = \max(V[4,11], V[4,11-7] +28) = \max(40,35)=40$$

Solution using Dynamic Programming

Weight Limit	0	1	2	3	4	5	6	7	8	9	10	11
w1=1, v1=1	0	1	1	1	1	1	1	1	1	1	1	1
w2=2 v2=6	**0**	1	6	7	7	7	7	7	7	7	7	7
w3=5, v3=18	0	1	6	7	7	**18**	19	24	25	25	25	25
w4=6, v4=22	0	1	6	7	7	18	22	24	28	29	29	40
w5=7, v5=28	0	1	6	7	7	18	22	28	29	34	35	**40**

Which Objects?

If V[i,j]=V[i-1,j], we move up to V[i-1, j]

V[5,11] = V[4,11] , we move on to V[4,11]

Otherwise check V[i,j]=V[i-1,j-wi]+vi

so check V[4,11]=V[3,11-6]+22 = V[3,5]+22=40 *[select object 4]*

now check V[3,5]=V[2,5-5]+18=V[2,0]+18=18 *[select object 3]*

Practice Examples

- Solve following knapsack problem using dynamic programming algorithm with given capacity W=5,

  Weight and Value are as follows : (2,12),(1,10),(3,20),(2,15).

- Solve the following Knapsack Problem using Dynamic Programming Method. Write the equation for solving above problem.

  n = 5, W = 100

  Object → 1 2 3 4 5

  Weight (w) → 10 20 30 40 50

  Value (v) → 20 30 66 40 60

Shortest Paths

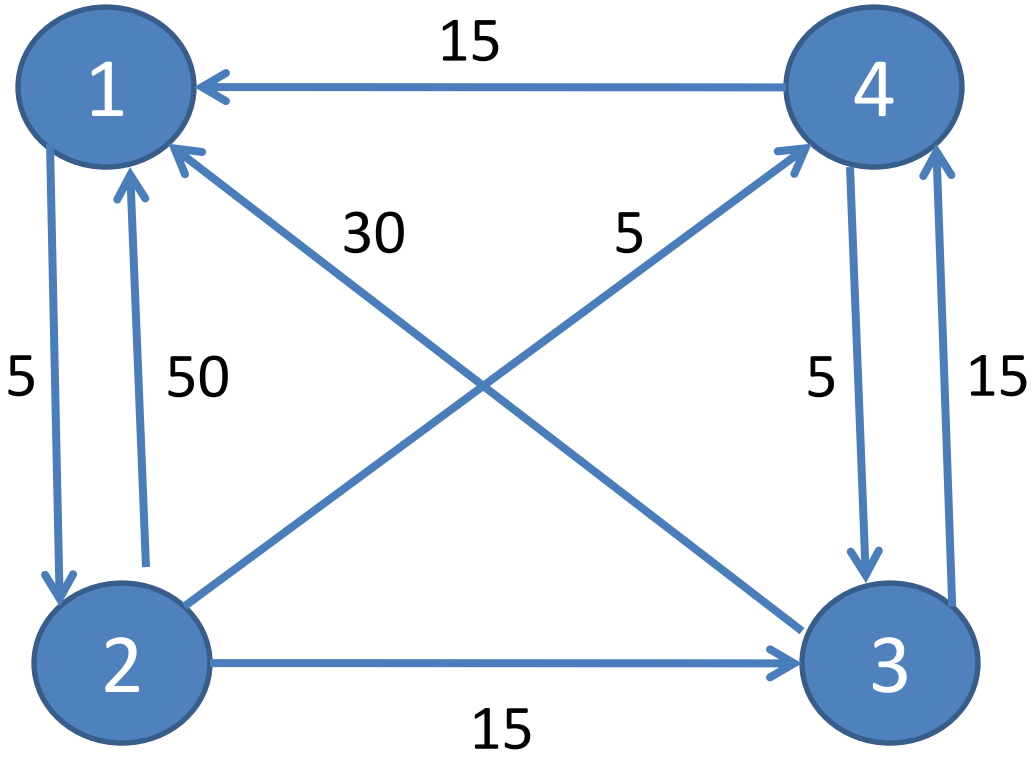
- Let G = <N,A> be a directed graph; N is a set of nodes and A is a set of edges.
- We want to calculate the length of the shortest path between **each pair of nodes.**
- Suppose the nodes of G are numbered from 1 to n, so N=[1,2,…n] and suppose matrix L gives the length of each edge, with L[i,j]=0 for i=1,2,..n.
- L[i,j] >= 0 for all i and j, and L[i,j]=∞ if the edge (i,j) does not exist.

Shortest Paths

- The principle of optimality:
  - If k is a node on the shortest path from i to j, then the part of the path from i to k, and the part from k to j, must also be optimal.
- We construct matrix D that gives the length of the shortest path between each pair of nodes.

$$D_k[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k]+D_{k-1}[k,j])$$

Floyd's Algorithm



$$D_0 = L = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$D_k[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k]+D_{k-1}[k,j])$

$D_1[3,2] = \min(D_0[3,2], D_0[3,1]+D_0[1,2]) = \min(Inf, 30+5) = 35$

Floyd's Algorithm

$$D_0 = L = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_k[i,j] = \min(D_{k-1}[i,j],\ D_{k-1}[i,k]+D_{k-1}[k,j])$$

$D_1[1,1] = \min(D_0[1,1],\ D_0[1,1]+D_0[1,1]) = \min(0,\ 0+\ 0) = 0$

$D_1[1,2] = \min(D_0[1,2],\ D_0[1,1]+D_0[1,2]) = \min(5,\ 0+\ 5) = 5$

$D_1[2,1] = \min(D_0[2,1],\ D_0[2,1]+D_0[1,1]) = \min(50,\ 50+\ 0) = 50$

$D_1[3,2] = \min(D_0[3,2],\ D_0[3,1]+D_0[1,2]) = \min(Inf,\ 30+5) = 35$

Floyd's Algorithm

$$D_0 = L = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix} \qquad D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_k[i,j] = \min(D_{k-1}[i,j],\ D_{k-1}[i,k] + D_{k-1}[k,j])$$

Practice Example

- Write the equation for finding out shortest path using Floyd's algorithm. Use Floyd's method to find shortest path for below mentions all pairs.

$$\begin{pmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{pmatrix}$$

Chained Matrix Multiplication

- Recall that the product C of a *p x q* matrix A and a *q x r* matrix B is the *p x r* matrix given by

$$C_{ij} = \sum_{k=1}^{q} a_{ik}\, b_{kj} \qquad 1<=i<=p,\ 1<=j<=r$$

It is clear that a total of *pqr* scalar multiplications are required to calculate the matrix product using this algorithm.

e.g. A(3X5) B(5X4) => C(3X4)

[3X5X4=60 multiplications]

Chained Matrix Multiplication

- Suppose now we want to calculate the product of more than two matrices.
- Matrix multiplication is **associative**, so we can compute the matrix product

$$M = M_1 M_2 \ldots M_n$$

in a number of ways, which all gives the same answer.

$$\begin{aligned} M &= (((M_1 M_2) M_3) \ldots M_n) \\ &= ((M_1 M_2) (M_3 \ldots M_n)) \\ &= ((M_1 (M_2 M_3)) \ldots M_n)) \quad \text{and so on.} \end{aligned}$$

However matrix multiplication is **not commutative**, so we are not allowed to change the order of the matrices in these arrangements.

CMM - Example

- Suppose for example, we want to calculate the product ABCD of four matrices, where

  A is 13X5, B is 5X89, C is 89X3 and D is 3X34

  Instances

  ((AB)C)D => AB=5785, (AB)C=3471, ((AB)C)D=1326

  => 5785+3471+1326 = 10582 scalar multpls

  (AB)(CD) = 54201 A((BC)D) = 4055

  (A(BC))D = 2856 A(B(CD)) = 26418

The most efficient method is almost 19 times faster than the slowest.

CMM – Example

- Number of combinations

N	1	2	3	4	5	10	….	15
T(n)	1	1	2	5	14	4832		2674440

The values of T(n) are called the *Catalan numbers*.

Chained Matrix Multiplication

- Suppose the dimensions of the matrices are given by a vector d[0..n] such that the matrix M_i, 1<= i<=n, is of dimension d_{i-1} X d_i

  We fill the table m_{ij} using the following rules for s=0,1,…, n-1.

  $s=0 : m_{ii} = 0$ $\quad i=1,2,\ldots,n$

  $s=1 : m_{i,i+1} = d_{i-1}d_id_{i+1}$ $\quad i=1,2,\ldots,n-1$

  $1<s<n: m_{i,i+s} = \min_{i<= k < i+s} (m_{ik} + m_{k+1,i+s} + d_{i-1}d_kd_{i+s})$ $\quad i=1,2,\ldots,n-s$

Chained Matrix Multiplication

- Suppose for example, we want to calculate the product ABCD of four matrices, where

  A is 13X5, B is 5X89, C is 89X3 and D is 3X34

  do d1 d2 d3 d4

  The vector d is therefore (13, 5, 89, 3, 34).

$s=1 : m_{i,i+1} = d_{i-1}d_i d_{i+1}$ $\quad i=1,2,\ldots,n-1$

For s=1, we find

m12 = d0 d1 d2 = 13X5X89 = 5785

m23 = d1 d2 d3 = 5X89X3 = 1335

m34 = d2 d3 d4 = 89X3X34 = 9078

Chained Matrix Multiplication

$1<s<n$: $m_{i,i+s} = \min_{i<= k < i+s} (m_{ik} + m_{k+1,i+s} + d_{i-1} d_k d_{i+s})$ $\quad i=1,2,\ldots,n-s$

For s=2, we obtain,

[where 1<s<n=1<2<4 and i = 1,2..,4-2, so i=1,2]

[i<=k<i+s = 1<=k<3, so k=1 & k=2]

m13 = min (m11+m23+13X5X3, // for k=1

m12+m33+13X89X3) // for k=2

= min (1530,9256) = 1530

[i<=k<i+s = 2<=k<4, so k=2 & k=3]

m24 = min (m22+m34+5X89X34, // for k=2

m23+m44+5X3X34) // for k=3

= min (24208,1845) = 1845

Chained Matrix Multiplication

$1<s<n$: $m_{i,i+s} = \min_{i<= k < i+s} (m_{ik} + m_{k+1,i+s} + d_{i-1} d_k d_{i+s})$ $\quad i=1,2,\ldots,n-s$

Finally for s=3, we obtain,

[where 1<s<n=1<3<4 and i = 1,..,4-3, so i=1 only]

[i<=k<i+s = 1<=k<4, so k=1, k=2, k=3]

m14 = min (m11+m24+13X5X34, // for k=1

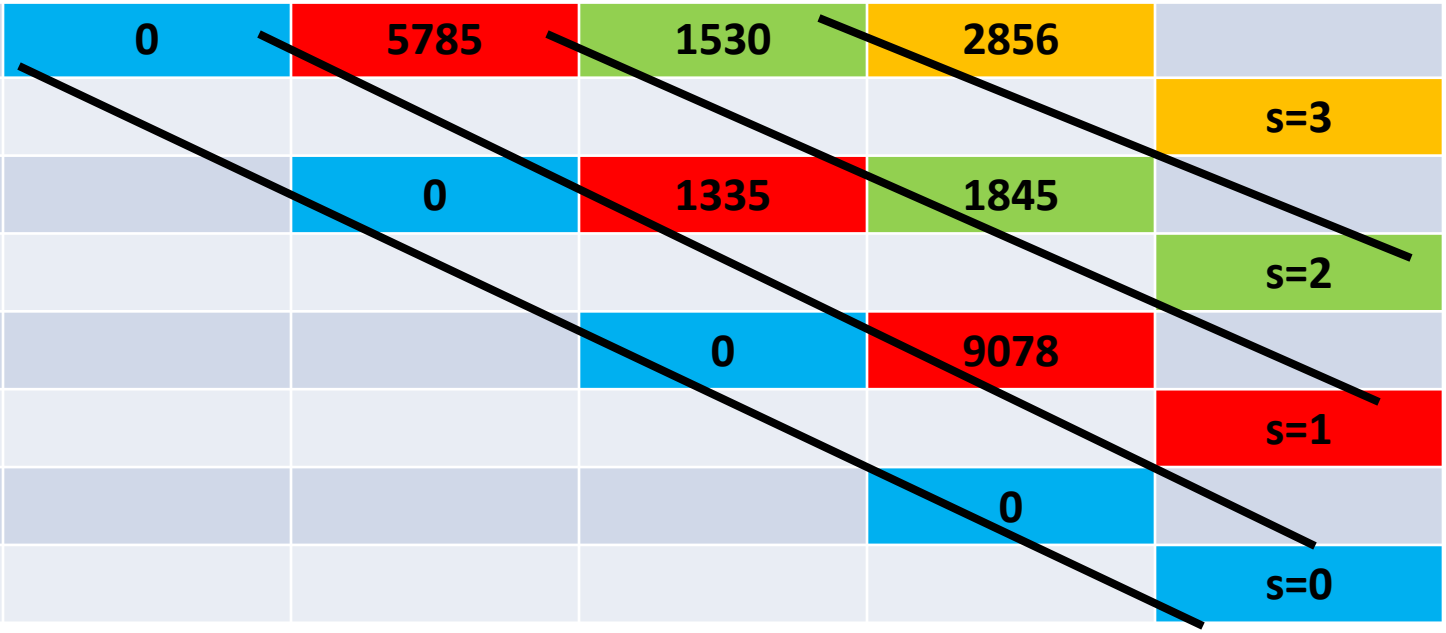
m12+m34+13X89X34, // for k=2

m13+m14+13X3X34) // for k=3

= min (4055,54201,2856) = 2856

Chained Matrix Multiplication

	j=1	2	3	4	
i=1	0	5785	1530	2856	
					s=3
2		0	1335	1845	
					s=2
3			0	9078	
					s=1
4				0	
					s=0



Solution: ((A (B C)) D)

A is 13X5, B is 5X89, C is 89X3 and D is 3X34

[(BC)=1335 + A(BC)=195 + (A(BC))D=1326] = 2856

GTU Practice Examples

- Given the four matrix find out optimal sequence for multiplication D=<5,4,6,2,7>

- Using algorithm find an optimal parenthesization of a matrix chain product whose sequence of dimension is (5,10,3,12,5,50,6) (use dynamic programming).

Longest Common Subsequence Problem using Dynamic Programming

Dynamic programming

- It is used, when the solution can be recursively described in terms of solutions to subproblems (*optimal substructure*)
- Algorithm finds solutions to subproblems and stores them in memory for later use
- More efficient than "*brute-force methods*", which solve the same subproblems over and over again

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

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 $c[m,n]$

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

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] != 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)

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
			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[4,5]

LCS Example (1)

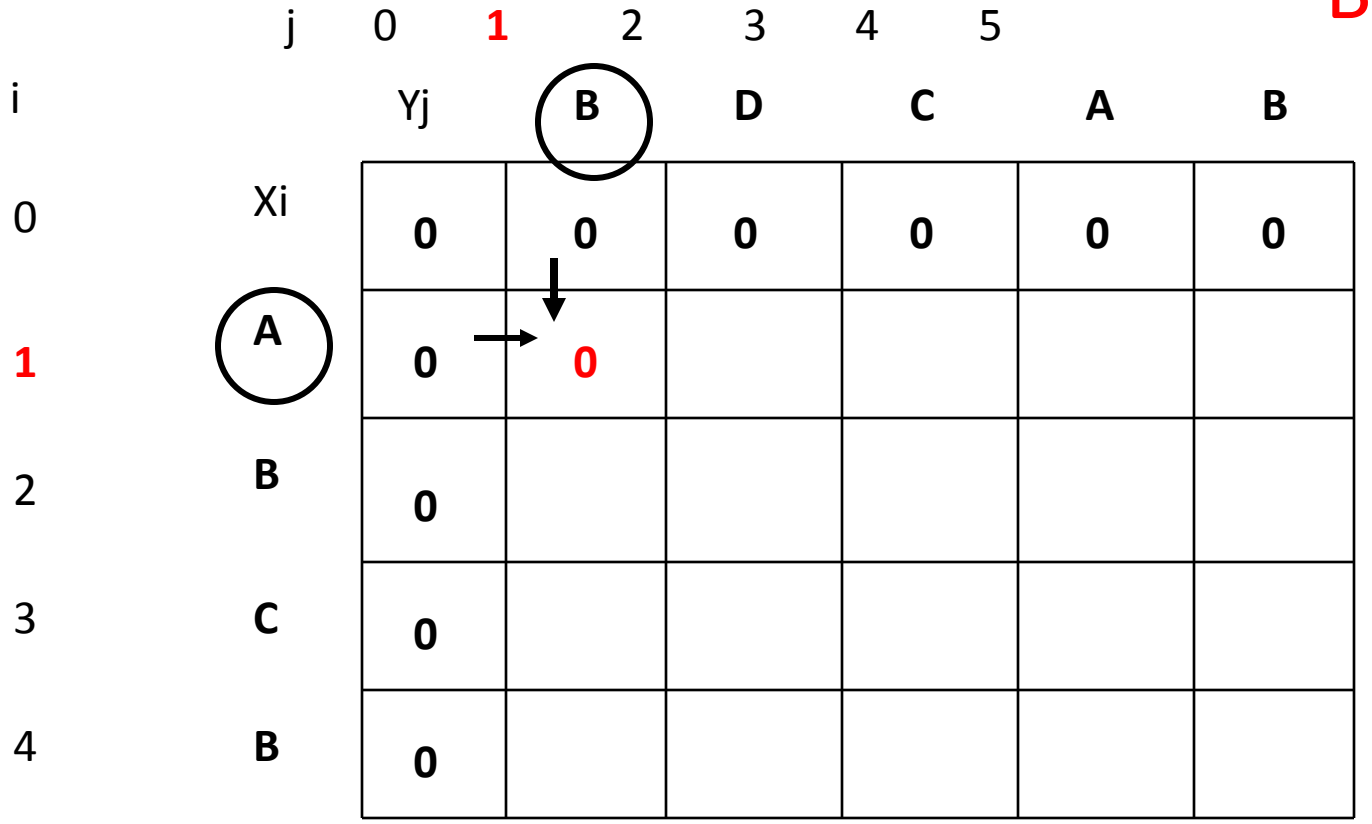
ABCB
BDCAB

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 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
			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 ($X_i == Y_j$)

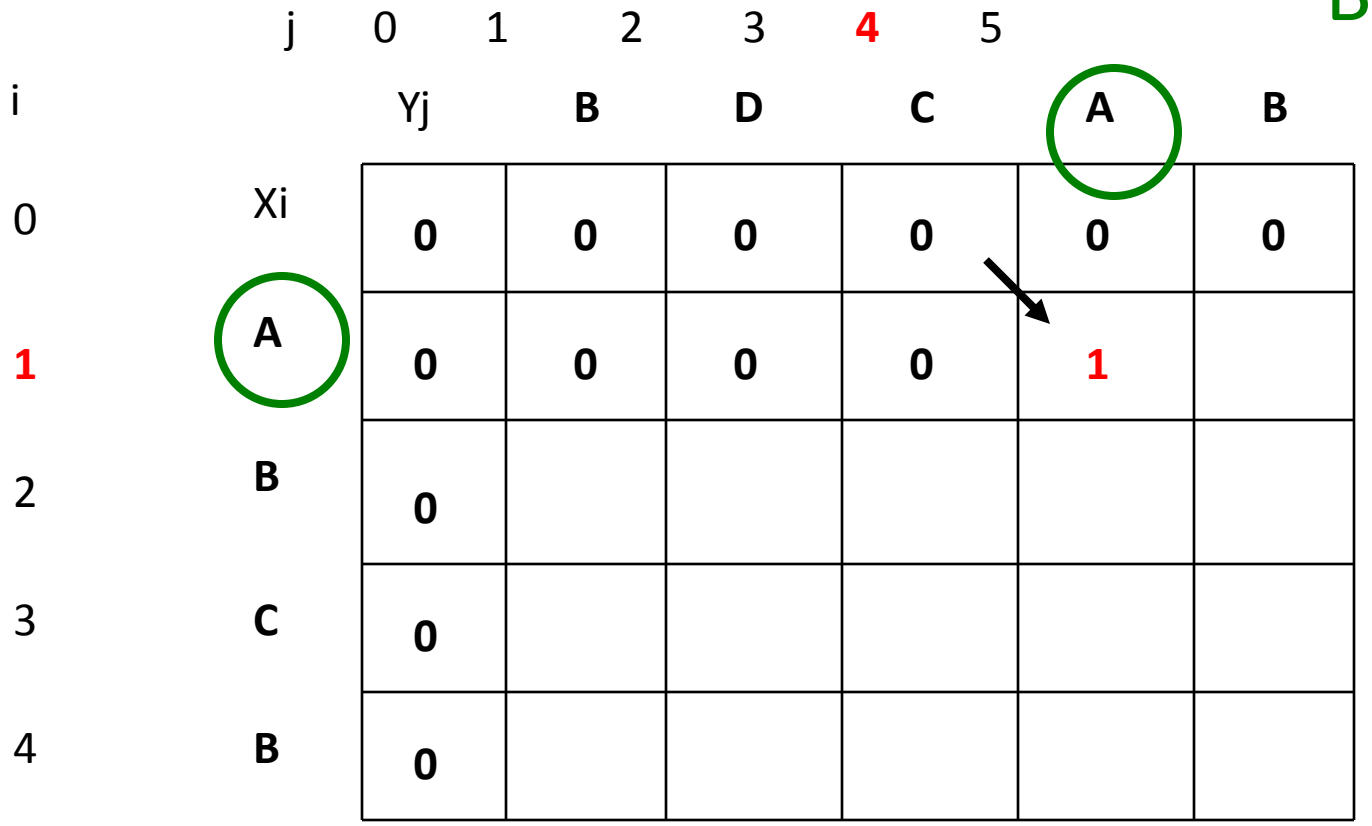
c[i,j] = c[i-1,j-1] + 1

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

LCS Example (4)

ABCB

BDCAB



if ($X_i == Y_j$)

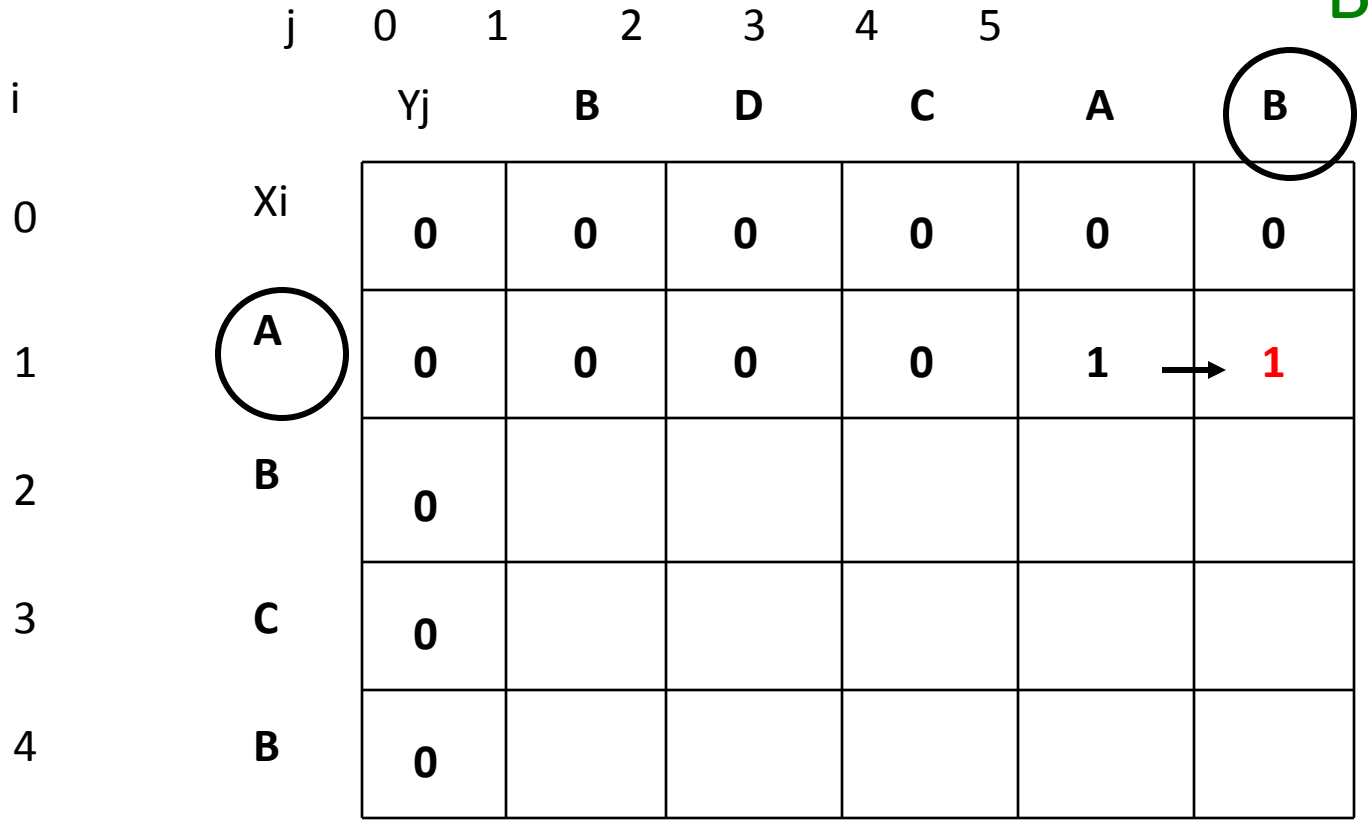
c[i,j] = c[i-1,j-1] + 1

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

LCS Example (5)

ABCB

BDCAB



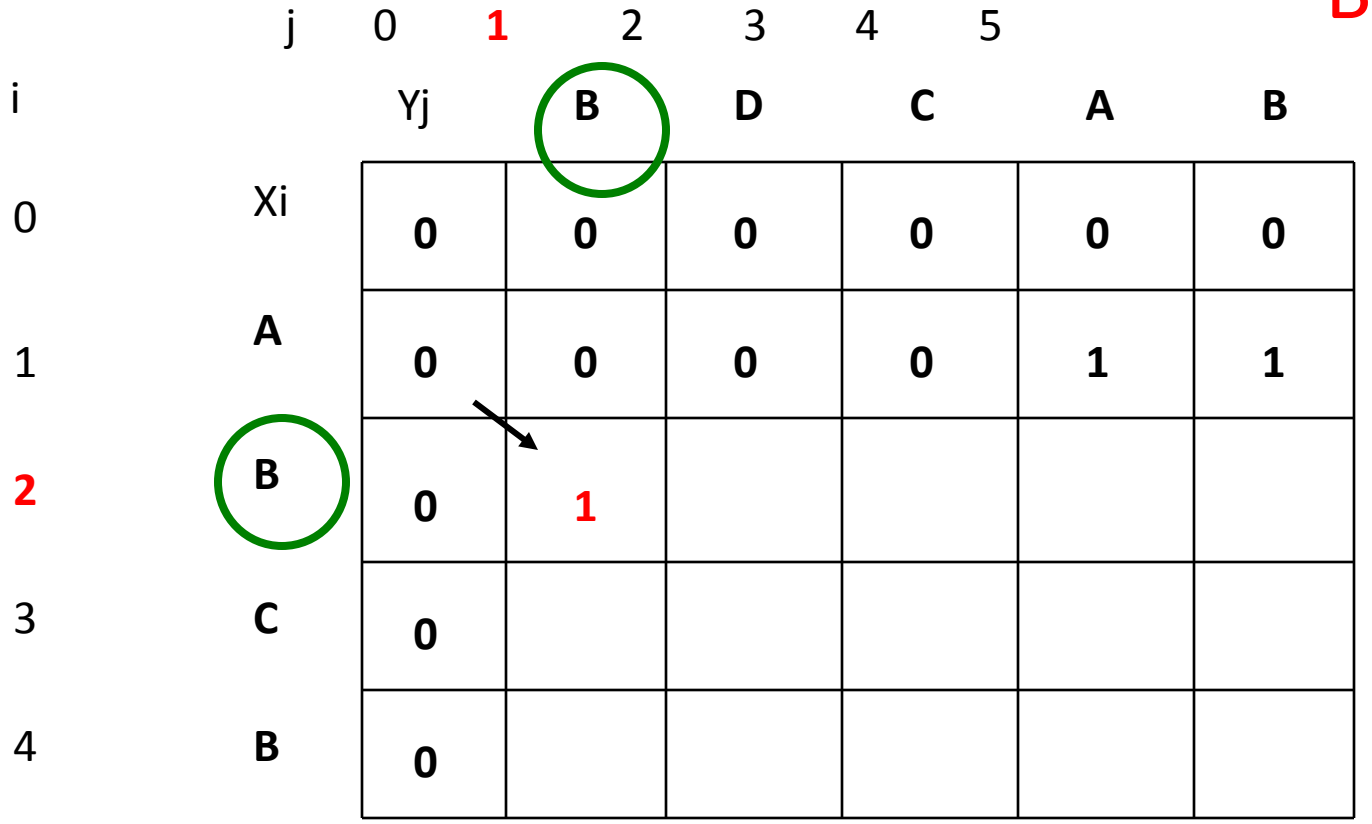
if ($X_i == Y_j$)

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

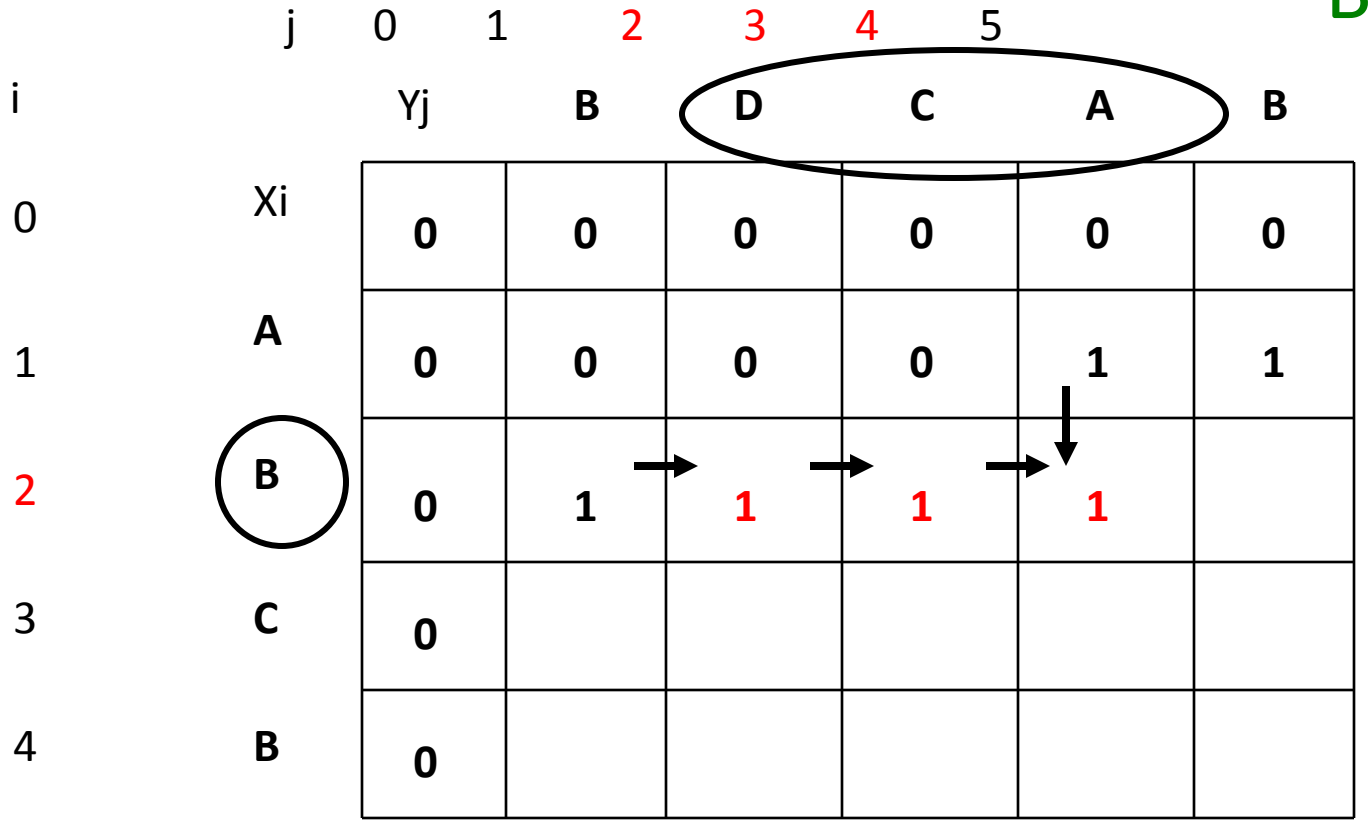


if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (7)

ABCB

BDCAB



if ($X_i == Y_j$)

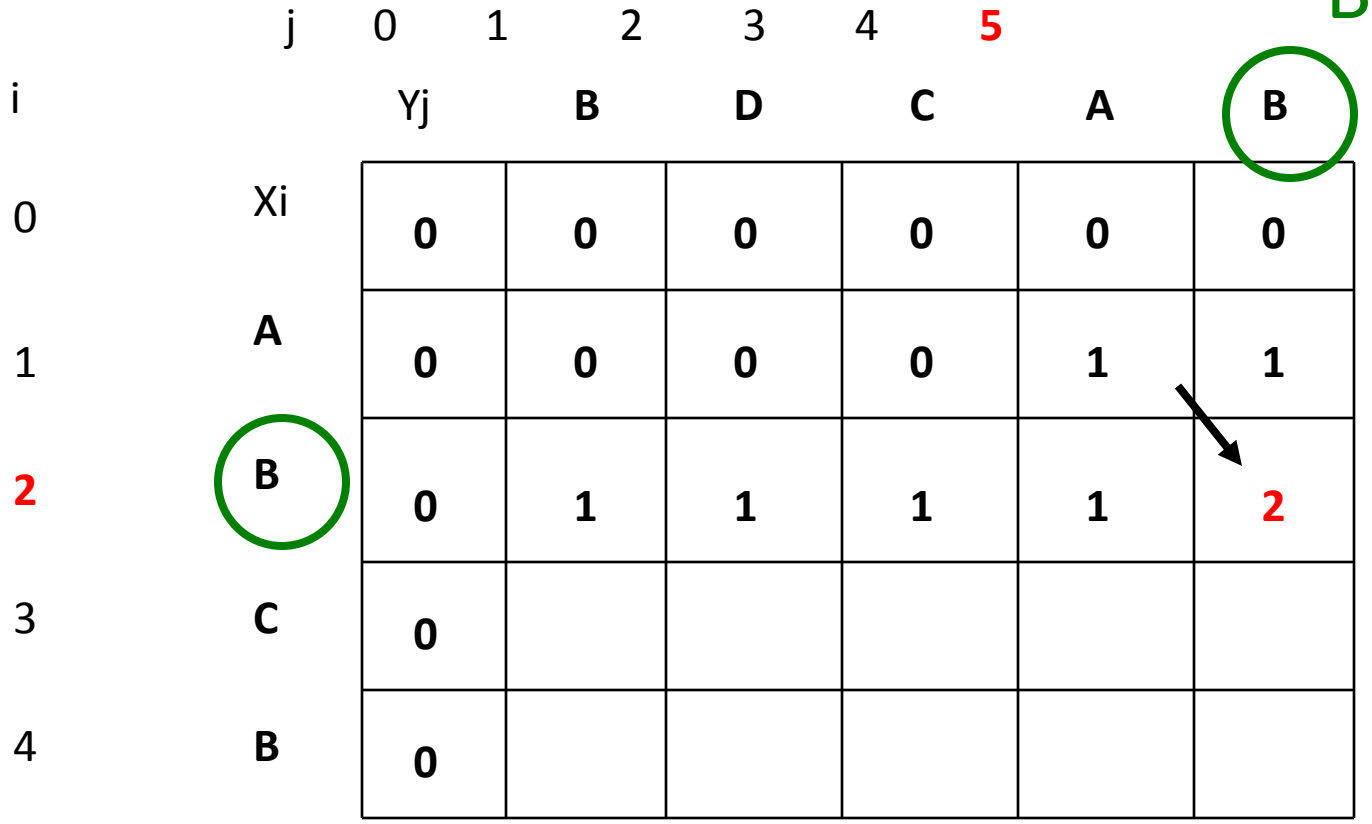
c[i,j] = c[i-1,j-1] + 1

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

LCS Example (8)

ABCB

BDCAB



if ($X_i == Y_j$)

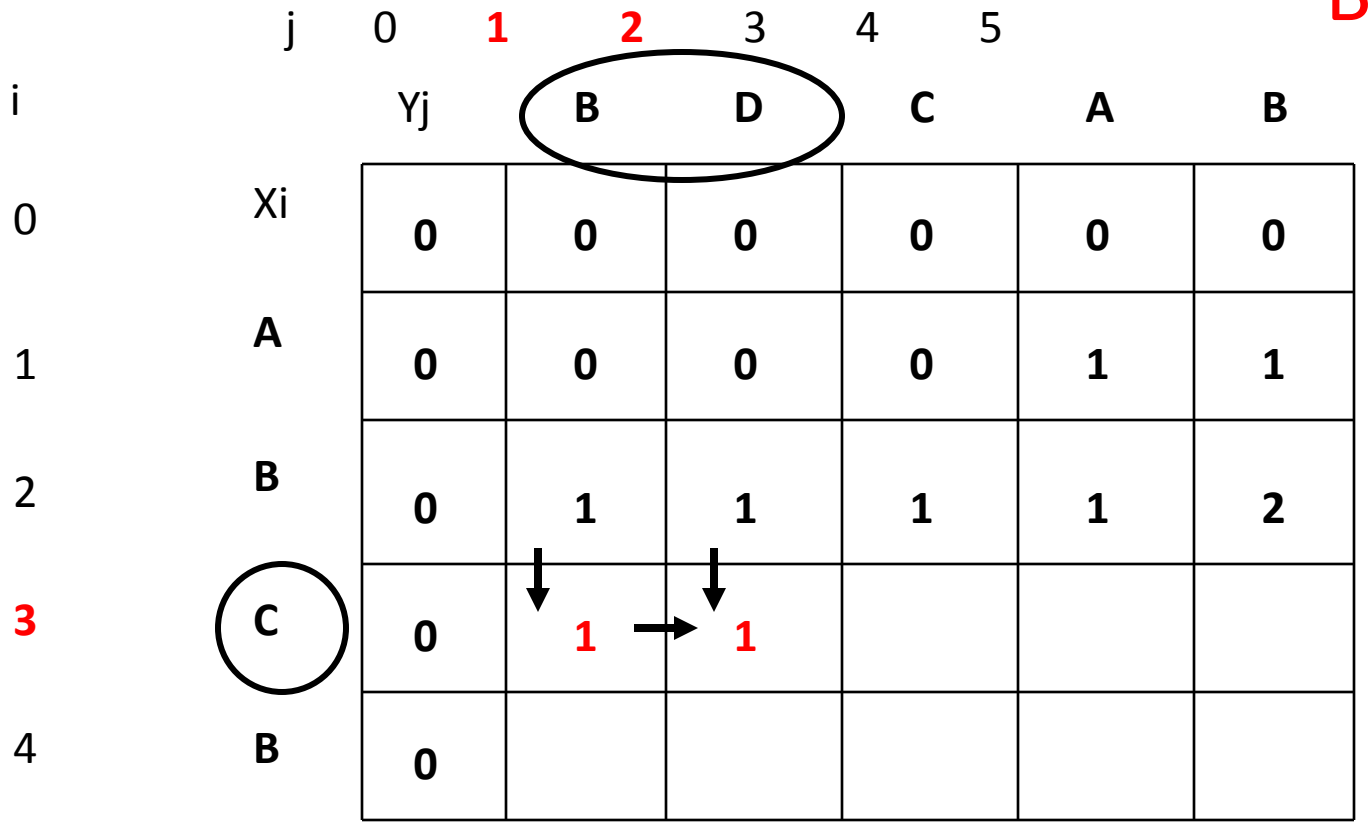
c[i,j] = c[i-1,j-1] + 1

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

LCS Example (10)

ABCB

BDCAB



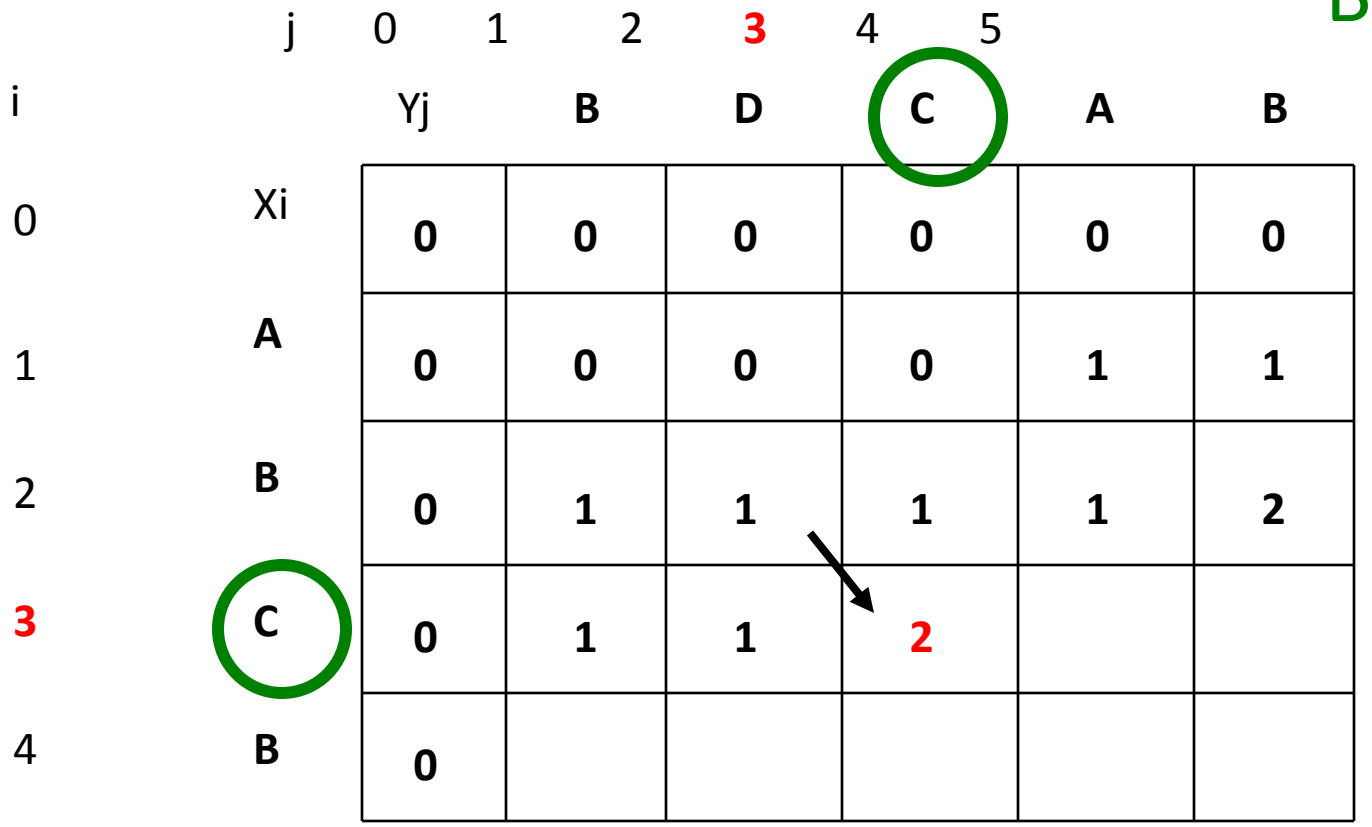
if ($X_i == Y_j$)

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



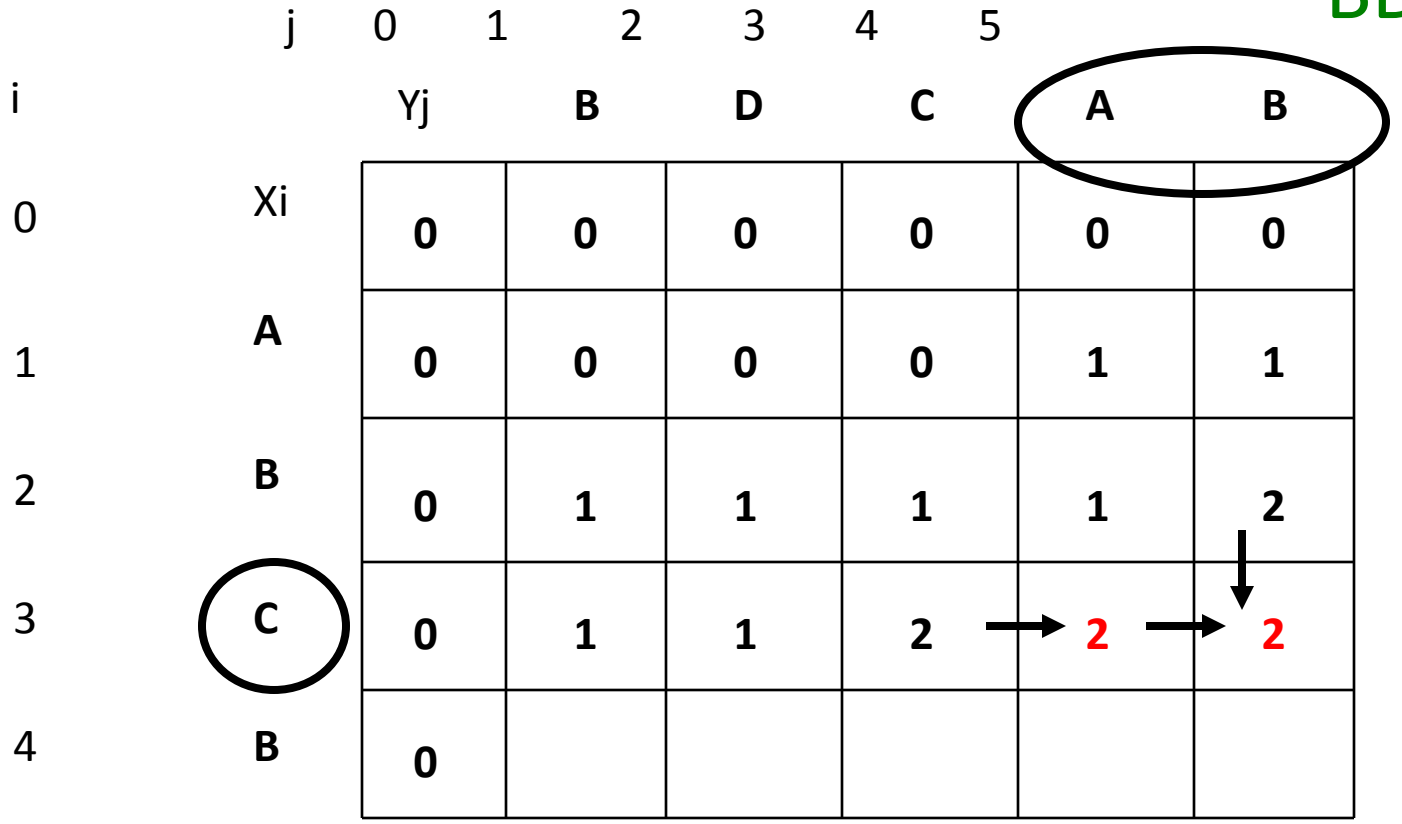
i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2		
4	B		0					

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 (12)

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 (13)

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 ($X_i == Y_j$)

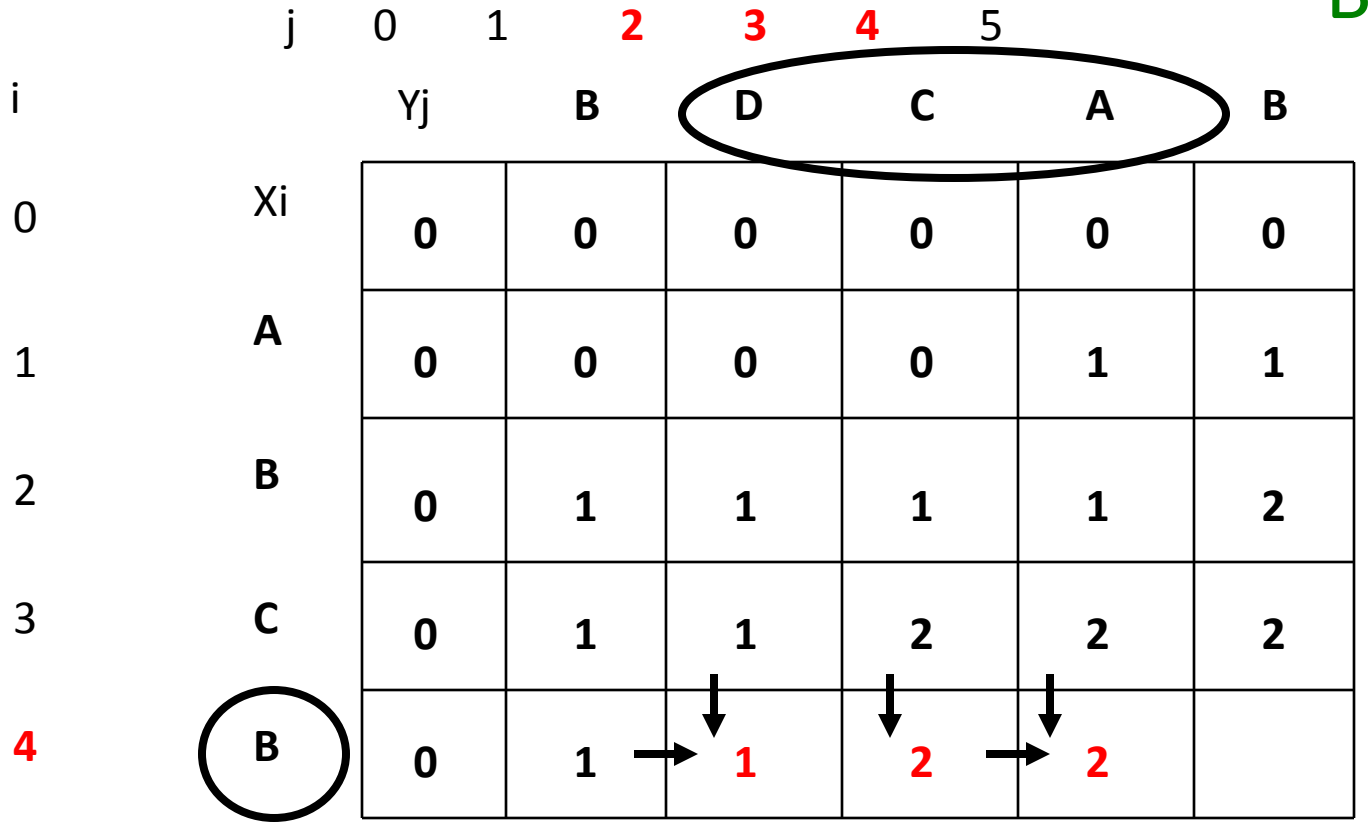
c[i,j] = c[i-1,j-1] + 1

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

LCS Example (14)

ABCB

BDCAB



if ($X_i == Y_j$)

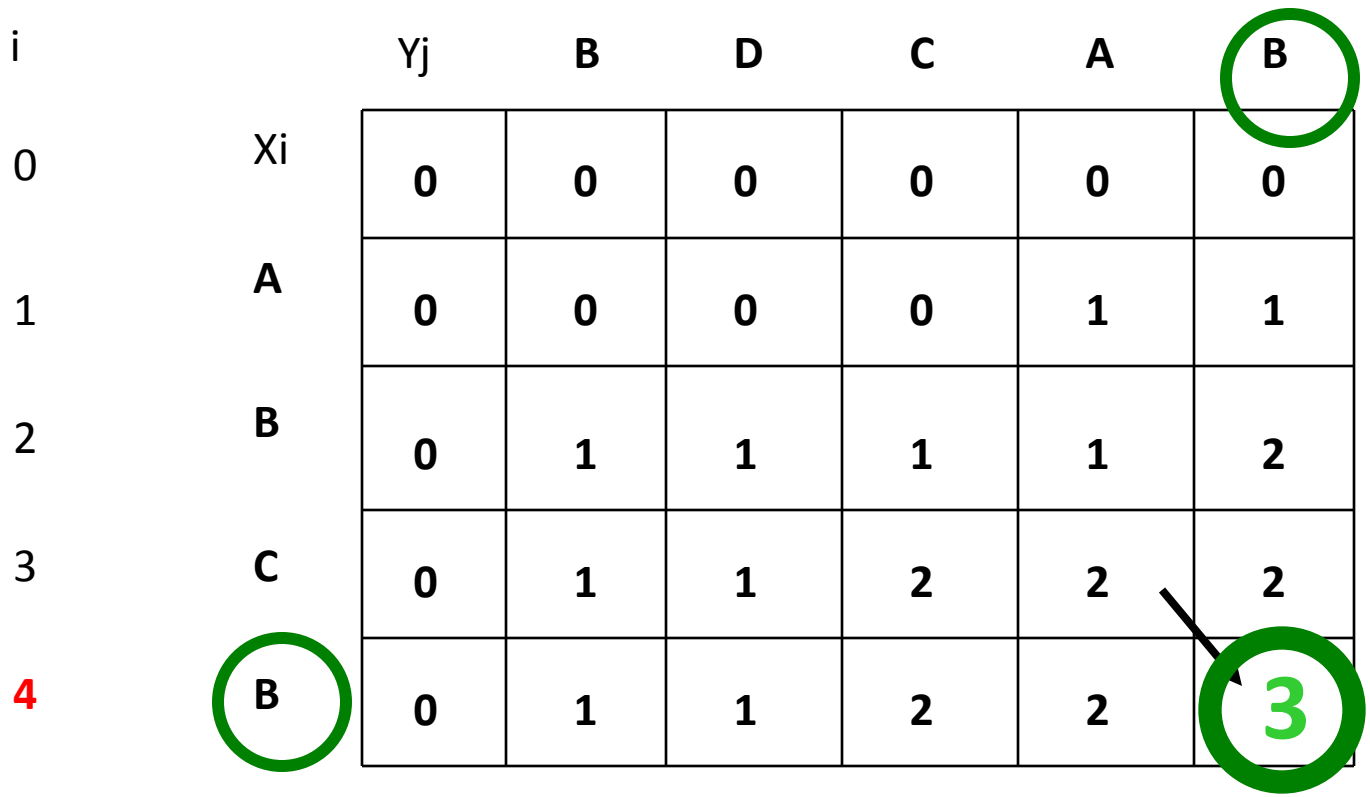
c[i,j] = c[i-1,j-1] + 1

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

LCS Example (15)

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 Algorithm Running Time

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

O(m*n)

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

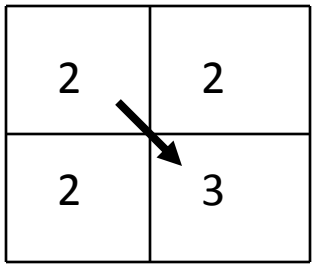
How to find actual LCS

- 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

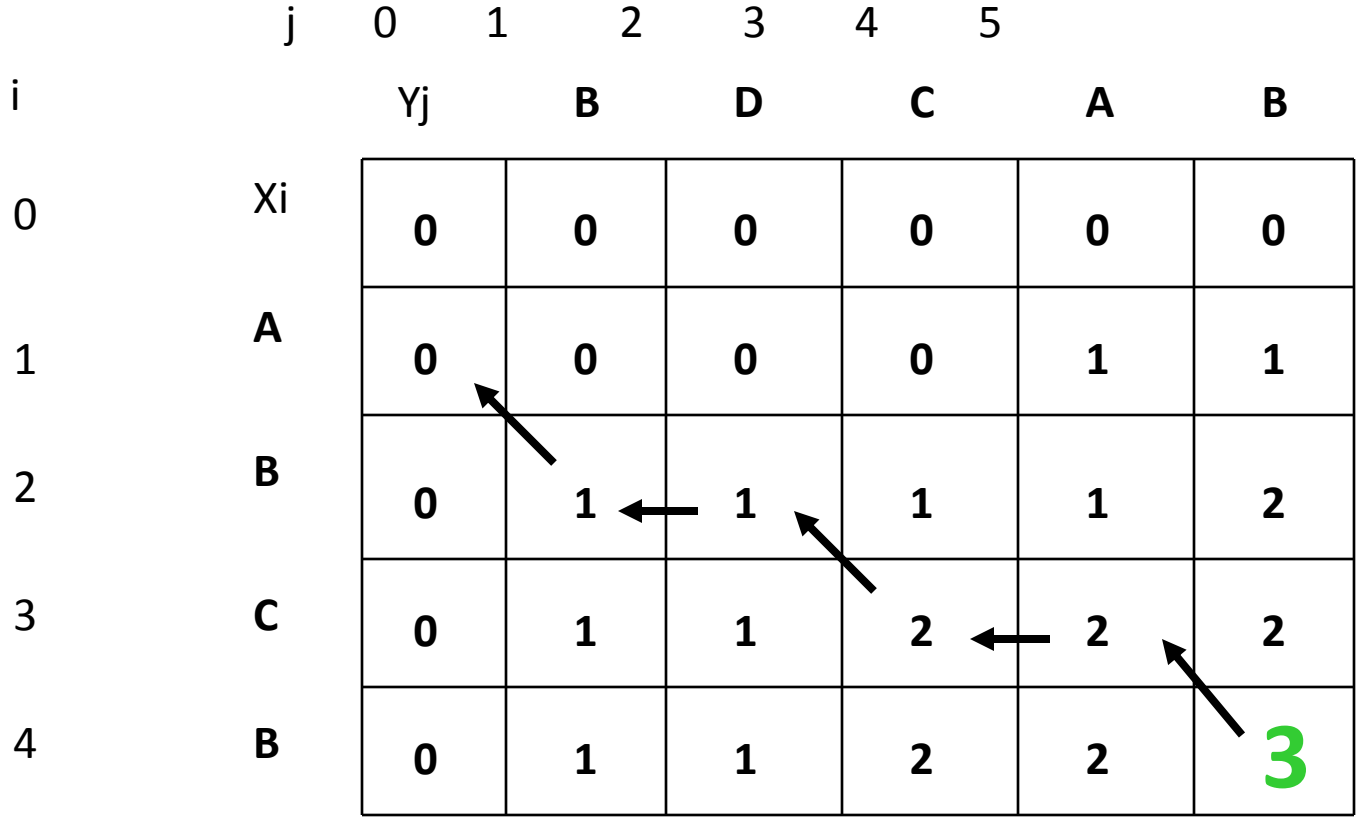
How to find actual LCS - continued

- Remember that

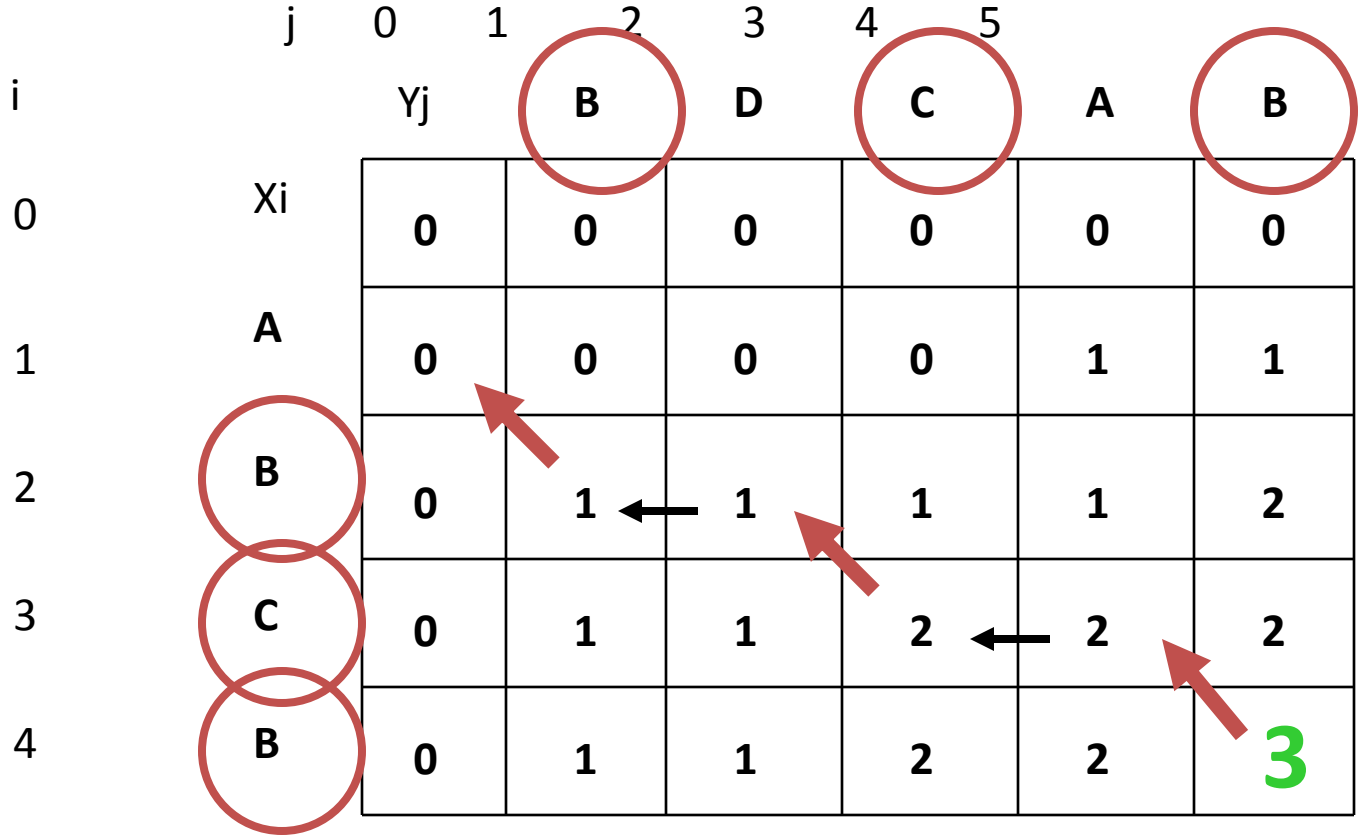
$$c[i, j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- 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


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

Finding LCS (2)



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

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

(this string turned out to be a palindrome)

GTU Practice Examples

- Given two sequences of characters, P=<MLNOM> Q=<MNOM> Obtain the longest common subsequence.

- Find Longest Common Subsequence using Dynamic Programming Technique with illustration X={A,B,C,B,D,A,B} Y={B,D,C,A,B,A}

- Using algorithm determine an Longest Common Sequence of (A,B,C,D,B,A,C,D,F) and (C,B,A,F)(use dynamic programming).

- Explain how to find out Longest Common Subsequence of two strings using Dynamic Programming method.Find any one Longest Common Subsequence of given two strings using Dynamic Programming. S1=abbacdcba S2=bcdbbcaac