Lecture # 10

Dynamic Programming

Fibonacci Sequence

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . . where each number is the sum of the two preceding numbers.
- More formally: The Fibonacci numbers F_n are defined as follows:

$$F0 = 0$$

$$F1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

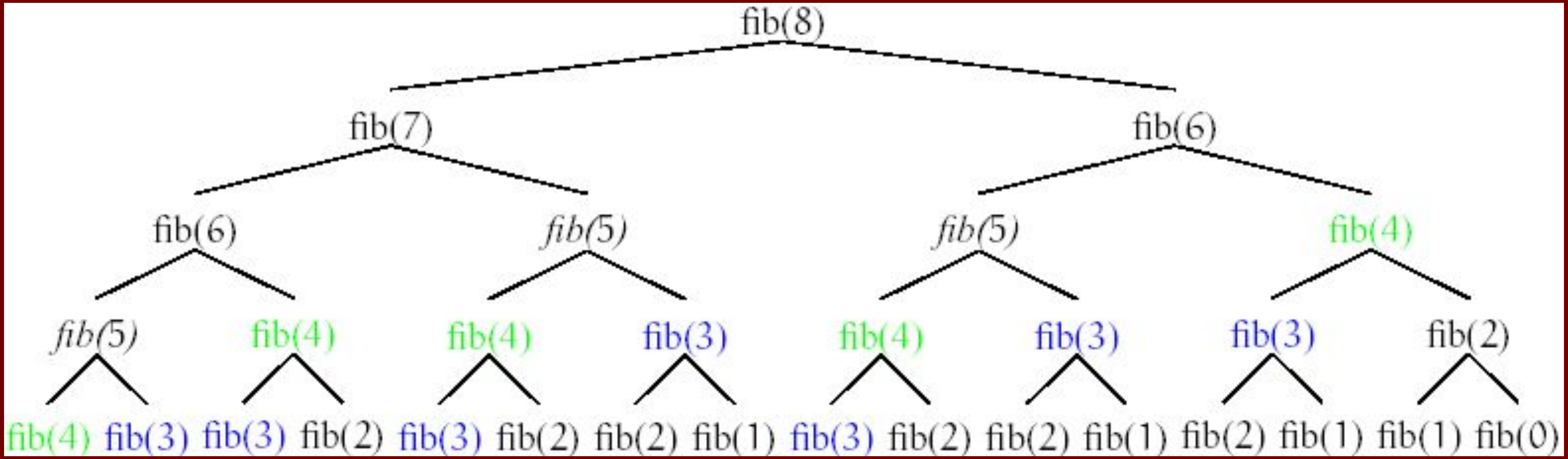
Fibonacci Sequence

- The recursive definition of Fibonacci numbers gives us a recursive algorithm for computing them:

```
FIB(n)
  if (n < 2)
     then return n
     else return FIB(n − 1) + FIB(n − 2)
```

Fibonacci Sequence

- Following Figure shows four levels of recursion for the call fib(8):



Fibonacci Sequence

- A single recursive call to fib(n) results in one recursive call to fib(n - 1) and one recursive call to fib(n-2).
- For each call, we're recomputing the same fibonacci number from scratch.
- We can avoid this unnecessary repetition by *writing down the results of recursive calls and looking them up again if we need them later*.
- This process is called ***memoization*** .

Fibonacci Sequence

- Here is the algorithm with ***memoization*** .

```
MEMOFIB(n)
if (n < 2)
    then return n
if (F[n] is undefined)
    then F[n] ← MEMOFIB(n − 1) + MEMOFIB(n − 2)
return F[n]
```

- If we trace through the recursive calls to MEMOFIB, we find that array F[] gets filled from bottom up. I.e., first F[2], then F[3], and so on, up to F[n].
- We can replace recursion with a simple for-loop that just fills up the array F[ ] in that order.

Fibonacci Sequence

- This gives us our first explicit *dynamic programming* algorithm.

```
ITERFIB(n)
    F[0] ← 0
    F[1] ← 1
    for i ← 2 to n
    do
        F[i] ← F[i − 1] + F[i − 2]
    return F[n]
```

```
Fib2(n)
{   int Fn = 1, Fn1 = 1, Fn2 = 1;
    for(I = 2; I <= n; I++)
    {      Fn = Fn1 + Fn2;
           Fn2 = Fn1;
           Fn1 = Fn;
    }
    return Fn;
}
```

Dynamic Programming: Steps

- **The seven steps in the development of a dynamic programming algorithm are as follows*:***

1. *Establish* a recursive property that gives the solution to an instance of the problem.
2. **Develop a recursive algorithm as per recursive property**
3. See if same instance of the problem is being solved again an again in recursive calls
4. Develop a memoized recursive algorithm
5. See the pattern in storing the data in the memory
6. Convert the memoized recursive algorithm into iterative algorithm
7. Optimize the iterative algorithm by using the storage as required (storage optimization)

Dynamic Programming

- Dynamic programming is essentially recursion without repetition (of same sub problem). Developing a dynamic programming algorithm generally involves two separate steps:
- Formulate problem recursively. Write down a formula for the whole problem.
- Build solution to recurrence from bottom up. Write an algorithm that starts with base cases and works its way up to the final solution.

- In contrast, Dynamic Programming is applicable when the sub problems are dependent i.e. when sub problems share sub-sub problems.

- Dynamic programming takes advantage of the duplication and arrange to solve each sub problem only once, saving the solution (in table or something) for later use.

- The underlying idea of dynamic programming is avoid calculating the same stuff twice, usually by keeping a table of known results of sub problems.

- Dynamic Programming is typically applied to optimization problems. In such problems there are many possible solutions. Each solution has a value, and we wish to find a solution with the optimal value.

Properties of a problem that can be solved with dynamic programming

- The problems which are candidates, to be solved using dynamic programming approach must have following elements in that.

1. <u>Optimal Substructure of the problems</u>

   The solution to the problem must be a composition of subproblem solutions

2. <u>Overlapping Sub problems</u>

   Optimal subproblems to unrelated problems can contain subproblems in common.

- In contrast, a problem for which divide and conquer approach is suitable usually generates brand new problems at each step of the recursion.

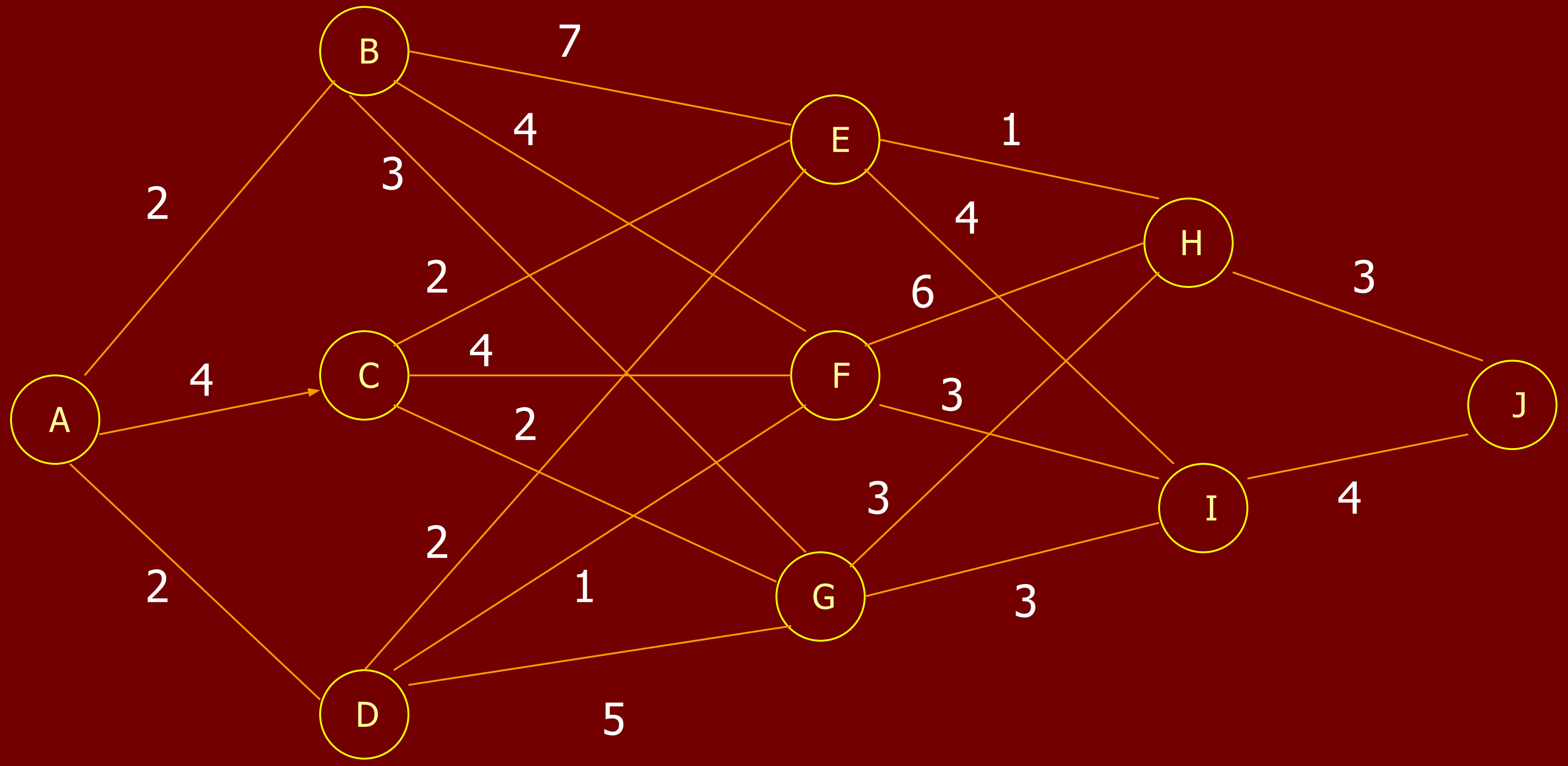
- Dynamic Programming algorithms typically take advantage of overlapping sub problems by solving each sub problem once and storing the solution in a table (may be) where it can be looked up when needed, using constant time per lookup.

Dynamic Programming Example 1:

Shortest Path Problem

- Consider the following graph. For each stage **j** and location **s**, we have

$$f_j(s) = \text{Min}_{(\text{all } z \text{ of stage } j+1)} \{ C_{sz} + f_{j+1}(z) \}$$



- We have five stages in previous figure, namely S1, S2, S3 S4 S5 and contains following nodes

- C_{sz} means cost (length) of arc s □ z

- We will be solving our problem in bottom up fashion. i.e. we will be moving from stage S5 to stage S1

- By above definition, for stage S5 we will have $f_5(J) = 0$

S4	$C_{sz} + f_5(z)$	$f_4(S4)$	Decision to go
	J		
H	3+0 = 3	3	J
I	4+0= 4	4	J

S3	$C_{sz} + f_4(z)$		$f_3(S3)$	Decision to go
	H	I		
E	1+3 = 4	4+4 = 8	4	H
F	6+3 = 9	3+4 = 7	7	I
G	3+3 = 6	3+4 = 7	6	H

S2	$C_{sz} + f_3(z)$			$f_2(S2)$	Decision to go
	E	F	G		
B	11	11	9	9	G
C	6	11	8	6	E
D	6	8	11	6	E
S1	$C_{sz} + f_2(z)$			$f_1(S1)$	Decision to go
	B	C	D		
A	11	10	8	8	D

- So from above DP approach we found that **A D E H J** is the shortest possible path.

Time complexity

- What it should be?
- stages :- m
- nodes :- n

  let us say, at every stage we have n- locations and for that we calculated values from all previous stage values.

- So we have $O(n^2)$ for every stage. And if we have m stages then for all stages it should be $O(mn^2)$.

Space complexity

- What it should be?

- Space complexity = O(n), as we calculate n – locations at every stage. So stage doesn't matter actually but the number of nodes does matter at every stage.

Dynamic Programming
Example 2:

Longest Common Subsequence
(LCS)

- *Longest common subsequence* (*LCS*) problem:
  - Given two sequences x[1..m] and y[1..n], find the longest subsequence which occurs in both
  - Ex: x = {A B C B D A B }, y = {B D C A B A}
  - {B C} and {A A} are both subsequences of both
    - *What is the LCS?*
  - Brute force (unthinking) 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?*

- Brute-force algorithm: 2^m subsequences of x to check against *n* elements of y: $O(n\,2^m)$
- We can do better: for now, let's only worry about the problem of finding the *length* of LCS
  - When finished we will see how to backtrack from this solution back to the actual LCS
- Notice LCS problem has optimal substructure property: solutions of subproblems are parts of the final solution.
- Let $X = \{ x_1, x_2, \ldots x_m \}$, we define the i^{th} prefix of X , for $i = 0,1,2\ldots.m$ as

  $X_i = \{ x_1, x_2, \ldots x_i \}$
- For example X = {A, B, C, B, D, A, B } then

  X_4 = {A, B, C, B}

Theorem (optimal substructure of an LCS)

- Let $X = \{ x_1, x_2, \ldots x_m \}$, and $Y = \{ y_1, y_2, \ldots y_n \}$ be sequences and let $Z = \{ z_1, z_2, \ldots z_k \}$ be any LCS of X and Y then

1. If $x_m = y_n$ then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m != y_n$ then $z_k != x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m != y_n$ then $z_k != y_n$ implies that Z is an LCS of X and Y_{n-1}.

- We can see the overlapping – sub problems property in the LCS problem. To find the LCS of X and Y, we may need to find the LCS's of X and Y_{n-1} and of X_{m-1} and Y. but each of these sub problems has sub-sub problems of finding of the LCS of X_{m-1} and of Y_{n-1}.

Finding LCS Length

- Lets define c[*i,j*] to *be the length of the LCS* of X_i and of Y_j . The optimal sub structure of LCS problem gives the recursive formula.
- Theorem:

c[i,j] = 0 if i=0 or j=0

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

- *What is this really saying?*

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

- **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}) ?

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: 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 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

	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]

column row

LCS Example (1)

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**					
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		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 (3)

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 (4)

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 (5)

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					
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 (6)

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				
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		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 (8)

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 (10)

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 (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

	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**					

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

	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				

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		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 (15)

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

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:

2	2
2	3

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

How to find actual LCS

- 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
		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)

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	1	2	2	3

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

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

- Read book page number 350 to 355 for details of LCS problem