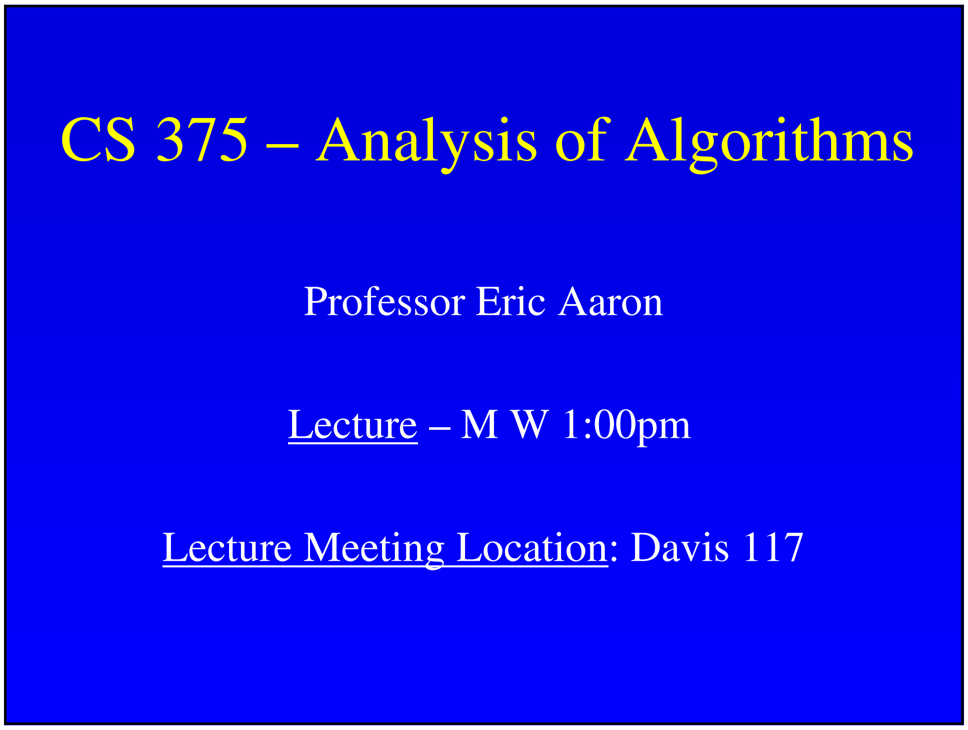
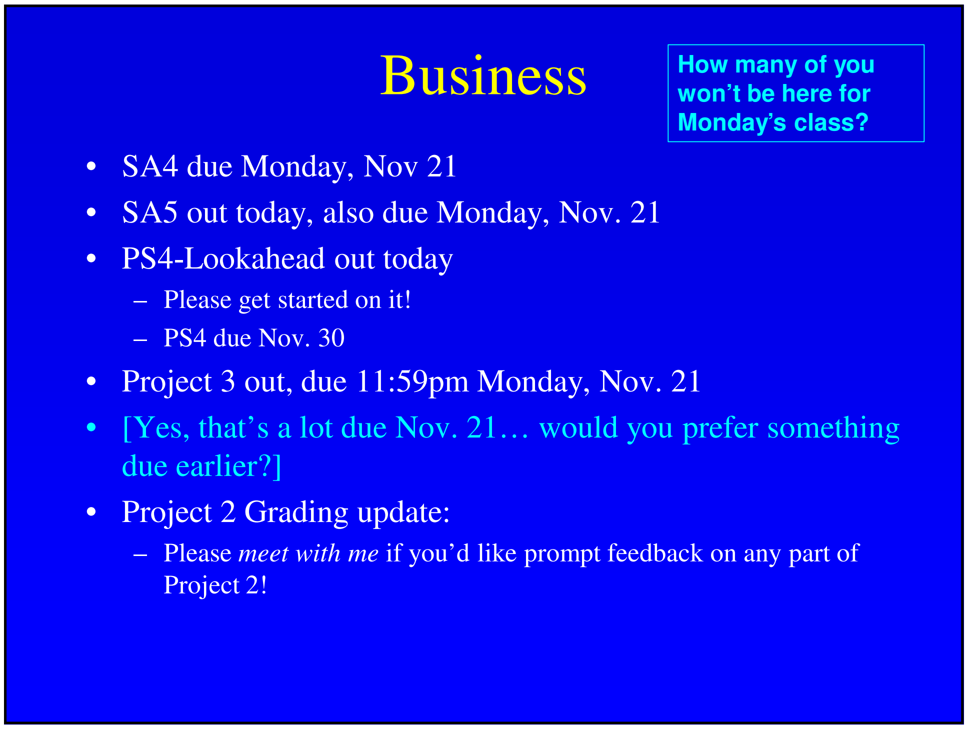


CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

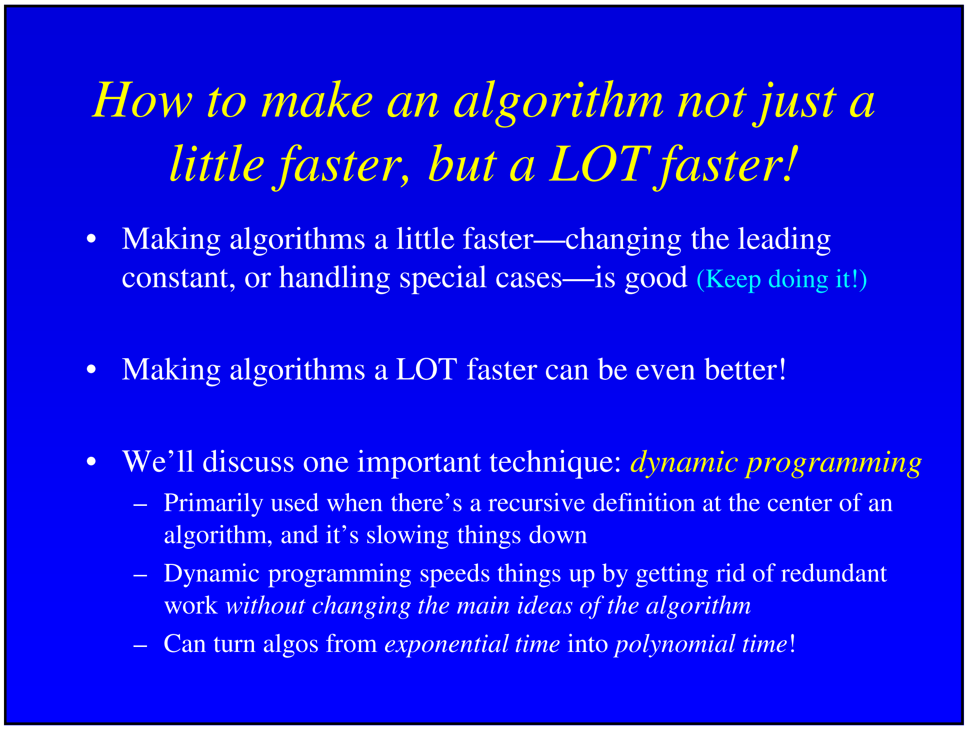
Lecture Meeting Location: Davis 117



Business

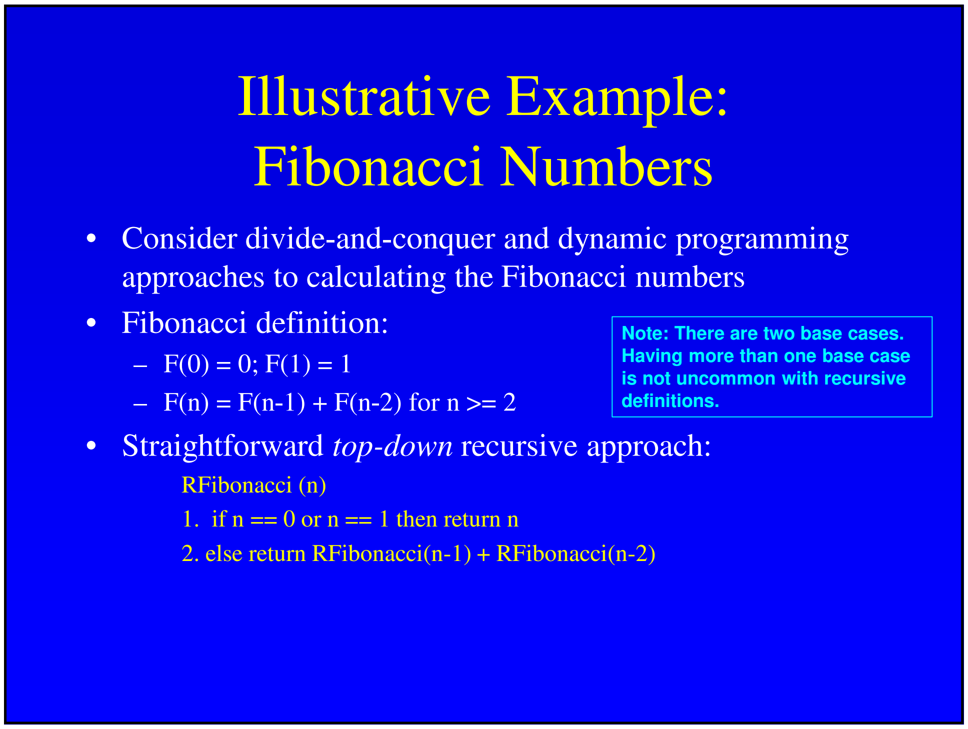
How many of you won't be here for Monday's class?

- SA4 due Monday, Nov 21
- SA5 out today, also due Monday, Nov. 21
- PS4-Lookahead out today
  - Please get started on it!
  - PS4 due Nov. 30
- Project 3 out, due 11:59pm Monday, Nov. 21
- [Yes, that's a lot due Nov. 21… would you prefer something due earlier?]
- Project 2 Grading update:
  - Please *meet with me* if you'd like prompt feedback on any part of Project 2!



How to make an algorithm not just a little faster, but a LOT faster!

- Making algorithms a little faster—changing the leading constant, or handling special cases—is good (Keep doing it!)
- Making algorithms a LOT faster can be even better!
- We'll discuss one important technique: *dynamic programming*
  - Primarily used when there's a recursive definition at the center of an algorithm, and it's slowing things down
  - Dynamic programming speeds things up by getting rid of redundant work *without changing the main ideas of the algorithm*
  - Can turn algos from *exponential time* into *polynomial time*!



Illustrative Example: Fibonacci Numbers

- Consider divide-and-conquer and dynamic programming approaches to calculating the Fibonacci numbers
- Fibonacci definition:
  - F(0) = 0; F(1) = 1
  - F(n) = F(n-1) + F(n-2) for n >= 2
- Straightforward *top-down* recursive approach:

```
RFibonacci (n)
1.  if n == 0 or n == 1 then return n
2. else return RFibonacci(n-1) + RFibonacci(n-2)
```

Note: There are two base cases. Having more than one base case is not uncommon with recursive definitions.

Even More Illustrative Example: Fibonacci Numbers

- Straightforward *top-down* recursive approach:

```
RFibonacci (n)
1.  if n == 0 or n == 1 then return n
2.  else return RFibonacci(n-1) + RFibonacci(n-2)
```

- Where are its inefficiencies?
- One possible dynamic programming approach: a *memoized* recursive approach—uses an auxiliary table to store results:

```
MFibonacci(n)
1. …
```

This is an exponential algorithm! See CLRS pg. 59 for some of the math behind that, and see me for the rest!

How would we write a memoized recursive Fibonacci function?

Even More Illustrative Example: Fibonacci Numbers

- Memoized version of Fibonacci method

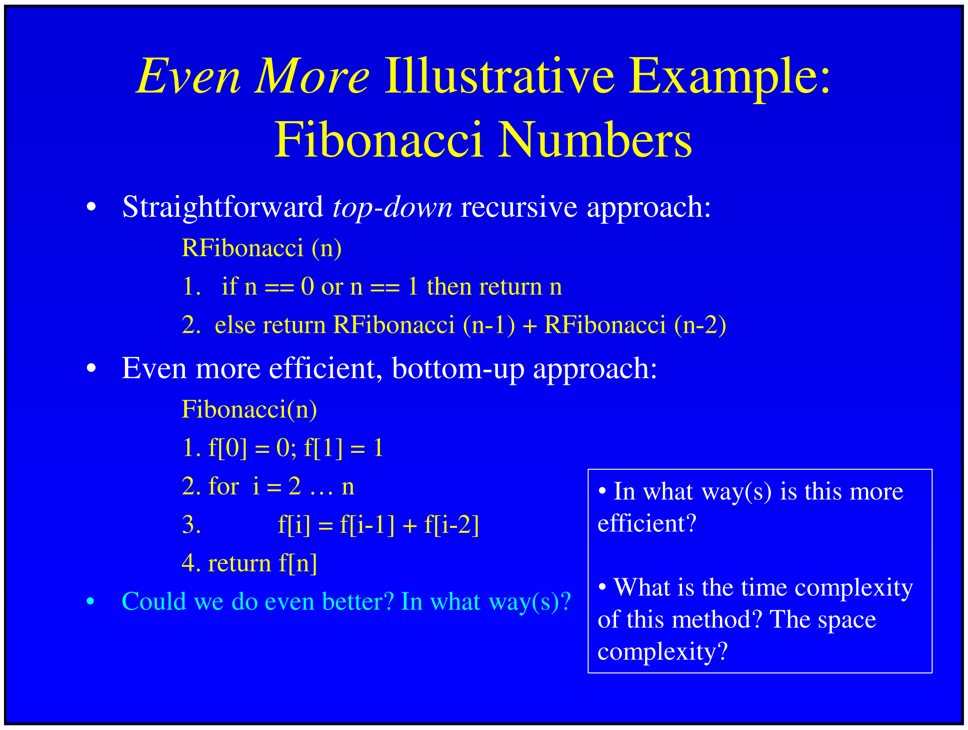
```
MFibonacci(n)

1.  let F[0..n] be a new array
2.  F[0] = 0
3.  if n > 0
4.    F[1] = 1
5.    for j = 2..n
6.      F[j] = -1 //or other sentinel
7.  return MFib-Aux(F,n)
```

```
MFib-Aux(F,n)

1.  if F[n] >= 0
2.    return F[n]
3.  else
4.    F[n] = MFib-Aux(F,n-1) +
              MFib-Aux(F,n-2)
5.    return F[n]
```

- Example of a time-memory (or time-space) trade-off
- What's the time complexity?
- What would a *bottom-up* Fibonacci approach look like?



Even More Illustrative Example: Fibonacci Numbers

- Straightforward *top-down* recursive approach:

```
RFibonacci (n)
1.  if n == 0 or n == 1 then return n
2.  else return RFibonacci (n-1) + RFibonacci (n-2)
```

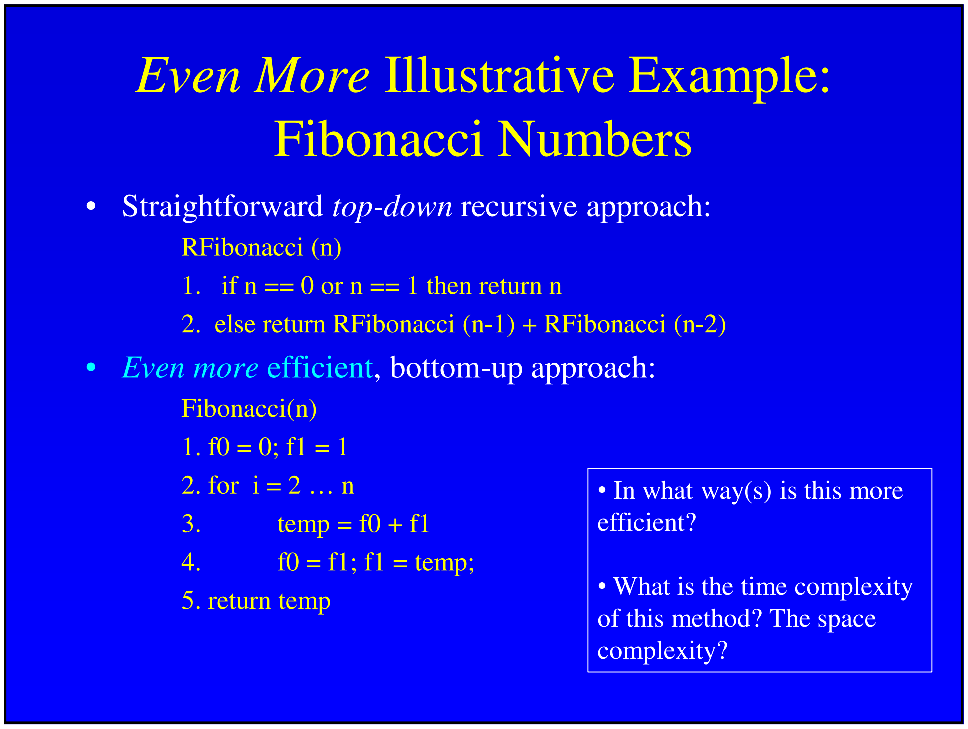
- Even more efficient, bottom-up approach:

```
Fibonacci(n)
1. f[0] = 0; f[1] = 1
2. for  i = 2 … n
3.         f[i] = f[i-1] + f[i-2]
4. return f[n]
```

- Could we do even better? In what way(s)?

• In what way(s) is this more efficient?

• What is the time complexity of this method? The space complexity?



Even More Illustrative Example: Fibonacci Numbers

- Straightforward *top-down* recursive approach:

```
RFibonacci (n)
1.  if n == 0 or n == 1 then return n
2.  else return RFibonacci (n-1) + RFibonacci (n-2)
```

- *Even more* efficient, bottom-up approach:

```
Fibonacci(n)
1. f0 = 0; f1 = 1
2. for  i = 2 … n
3.         temp = f0 + f1
4.         f0 = f1; f1 = temp;
5. return temp
```

• In what way(s) is this more efficient?

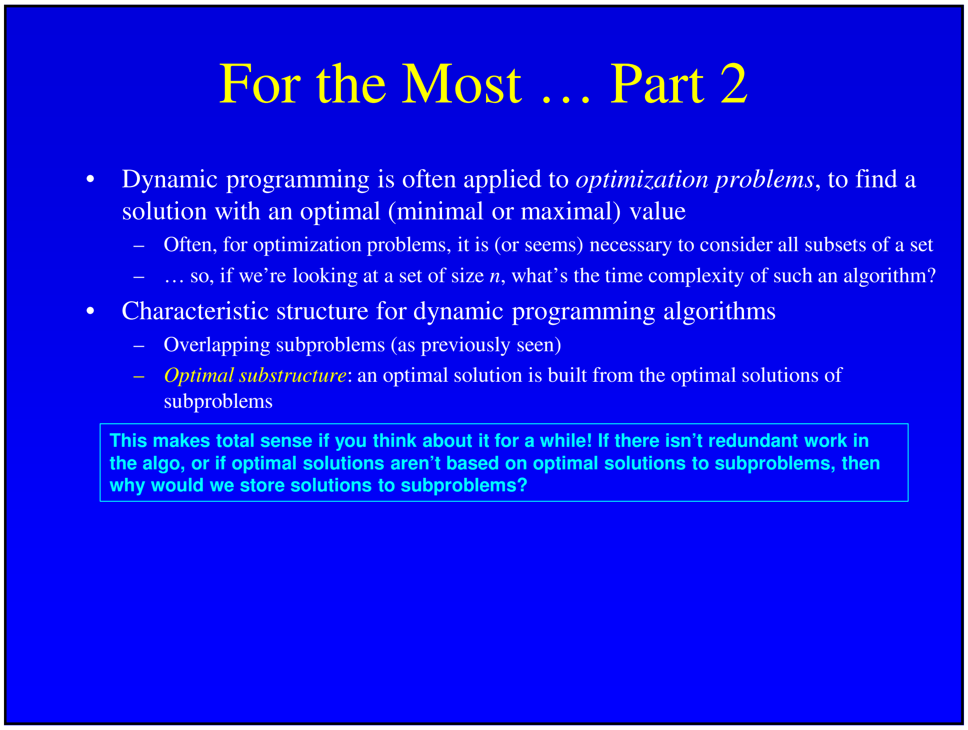
• What is the time complexity of this method? The space complexity?

Dynamic Programming

- The inefficient Fibonacci algorithms had a lot of recursive calls
  - With a lot of redundant work—the same calculations were done many times (e.g., calculate *Fib(2)* as part of calculating *Fib(5)*)
- *Dynamic programming* techniques can be more efficient for problems that have *overlapping sub-problems*
  - Divide-and-conquer methods recursively solve sub-problems, then combine sub-solutions into a solution
  - When there are overlapping (repeating) sub-problems, some of this work can be redundant…
  - Dynamic programming methods can solve each sub-problem once, store results in a table (O(1) lookup), and use the table for later solutions

For the Most Part…

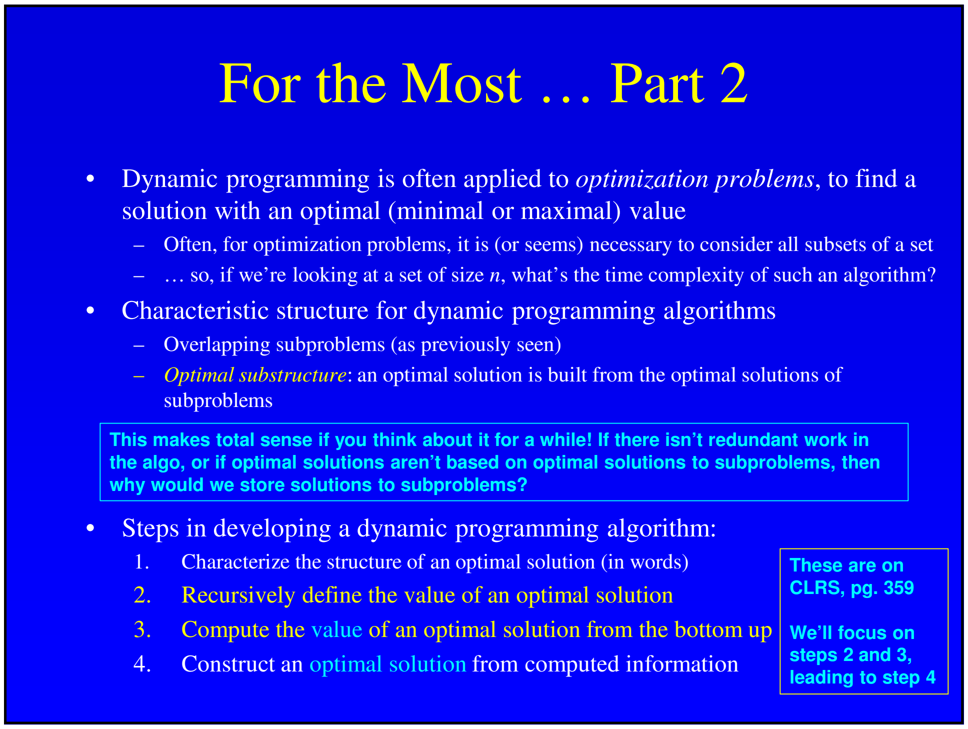
- The efficient Fibonacci methods used a characteristic technique of dynamic programming:
  - Results stored in a table (or similar), used to improve efficiency
- Dynamic programming solutions can be either top-down or bottom-up…
  - But most of the time, in practice, when people talk about a dynamic programming solution, they mean a bottom-up solution
- In general, when looking for a dynamic programming solution:
  - Try recursive, top-down approach with overlapping sub-problems
  - (Consider a memoized version)
  - Then, try bottom-up, iterative approach based on sub-problems
  - (Then, try to improve on space complexity of bottom-up method)



For the Most … Part 2

- Dynamic programming is often applied to *optimization problems*, to find a solution with an optimal (minimal or maximal) value
  - Often, for optimization problems, it is (or seems) necessary to consider all subsets of a set
  - … so, if we're looking at a set of size *n*, what's the time complexity of such an algorithm?
- Characteristic structure for dynamic programming algorithms
  - Overlapping subproblems (as previously seen)
  - *Optimal substructure*: an optimal solution is built from the optimal solutions of subproblems

This makes total sense if you think about it for a while! If there isn't redundant work in the algo, or if optimal solutions aren't based on optimal solutions to subproblems, then why would we store solutions to subproblems?



For the Most … Part 2

- Dynamic programming is often applied to *optimization problems*, to find a solution with an optimal (minimal or maximal) value
  - Often, for optimization problems, it is (or seems) necessary to consider all subsets of a set
  - … so, if we're looking at a set of size *n*, what's the time complexity of such an algorithm?
- Characteristic structure for dynamic programming algorithms
  - Overlapping subproblems (as previously seen)
  - *Optimal substructure*: an optimal solution is built from the optimal solutions of subproblems

This makes total sense if you think about it for a while! If there isn't redundant work in the algo, or if optimal solutions aren't based on optimal solutions to subproblems, then why would we store solutions to subproblems?

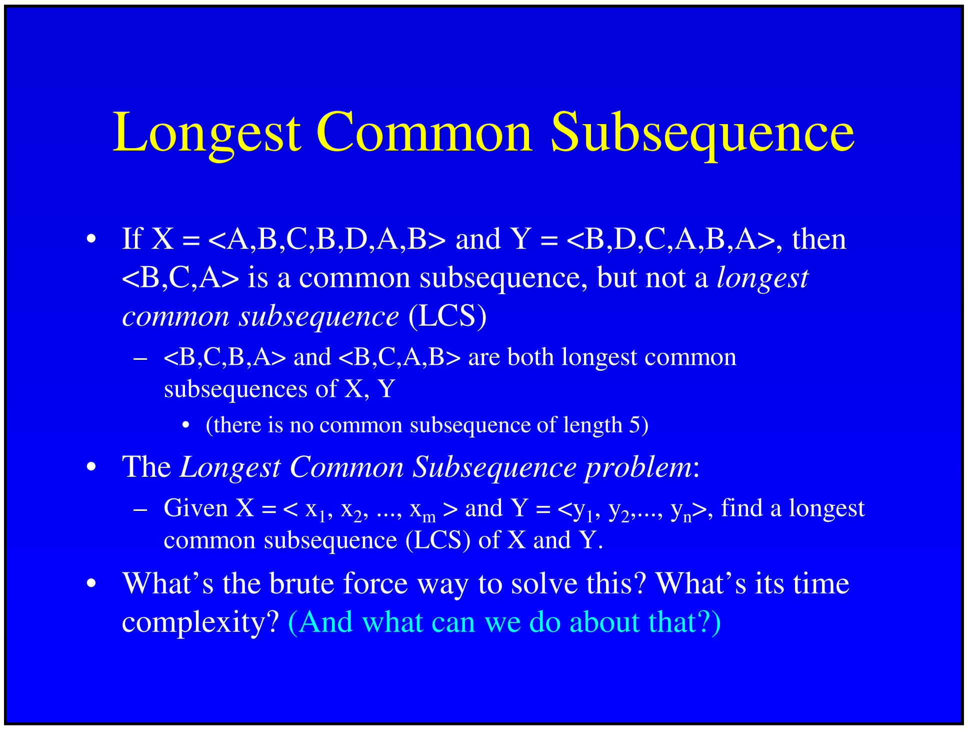
- Steps in developing a dynamic programming algorithm:
  1. Characterize the structure of an optimal solution (in words)
  2. Recursively define the value of an optimal solution
  3. Compute the value of an optimal solution from the bottom up
  4. Construct an optimal solution from computed information

These are on CLRS, pg. 359

We'll focus on steps 2 and 3, leading to step 4

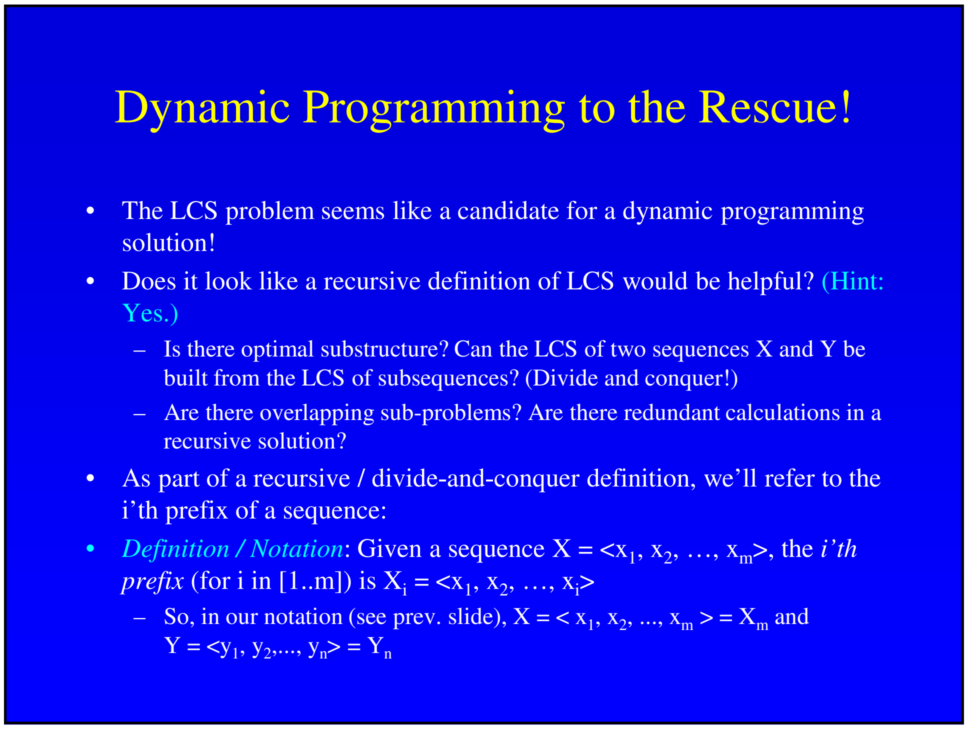
Sequences and Subsequences

- Another category of problems involves sequences and *subsequences*
- Definition:
  - 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 indices of X s.t. for all j in [1..k], $x_{i[j]} = z_j$
  - (i.e., elements of X, preserving order)
- Example / Definition:
  - If X = <A,B,A,C,A,B> and Y = <A,B,B,A> …
  - then <A,A>, <A,B>, <B,A>, <B,B>, <A,B,A>, <A,B,B> (and others) are all subsequences of both X and Y—i.e., they are *common subsequences*



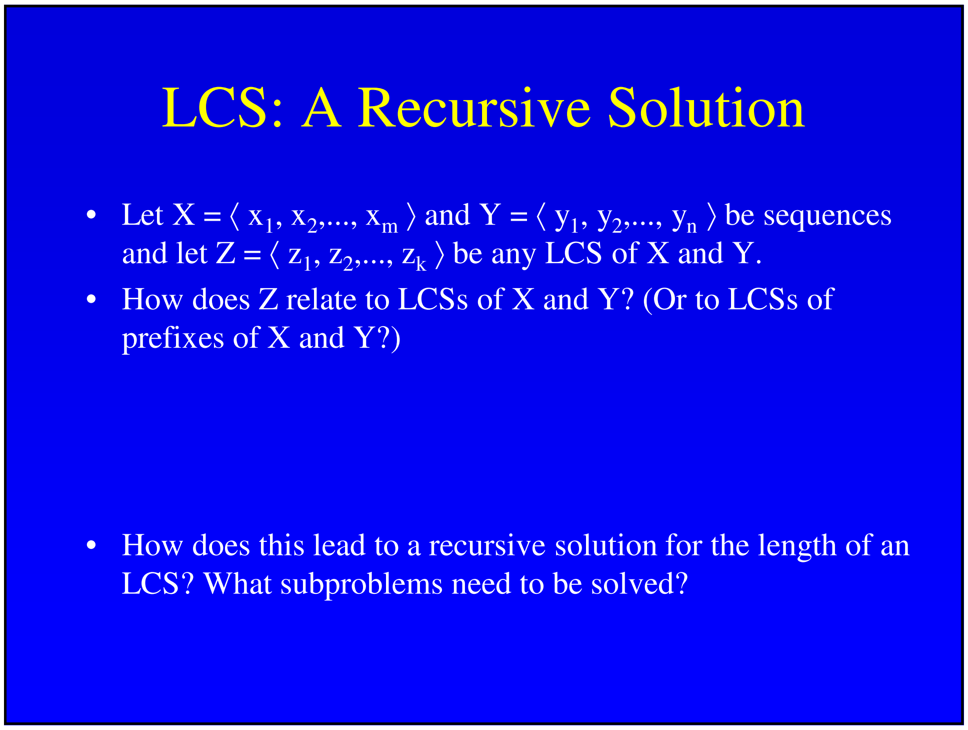
Longest Common Subsequence

- If X = <A,B,C,B,D,A,B> and Y = <B,D,C,A,B,A>, then <B,C,A> is a common subsequence, but not a *longest common subsequence* (LCS)
  - <B,C,B,A> and <B,C,A,B> are both longest common subsequences of X, Y
    - (there is no common subsequence of length 5)
- The *Longest Common Subsequence problem*:
  - Given X = < x_1, x_2, ..., x_m > and Y = < y_1, y_2,..., y_n >, find a longest common subsequence (LCS) of X and Y.
- What's the brute force way to solve this? What's its time complexity? (And what can we do about that?)



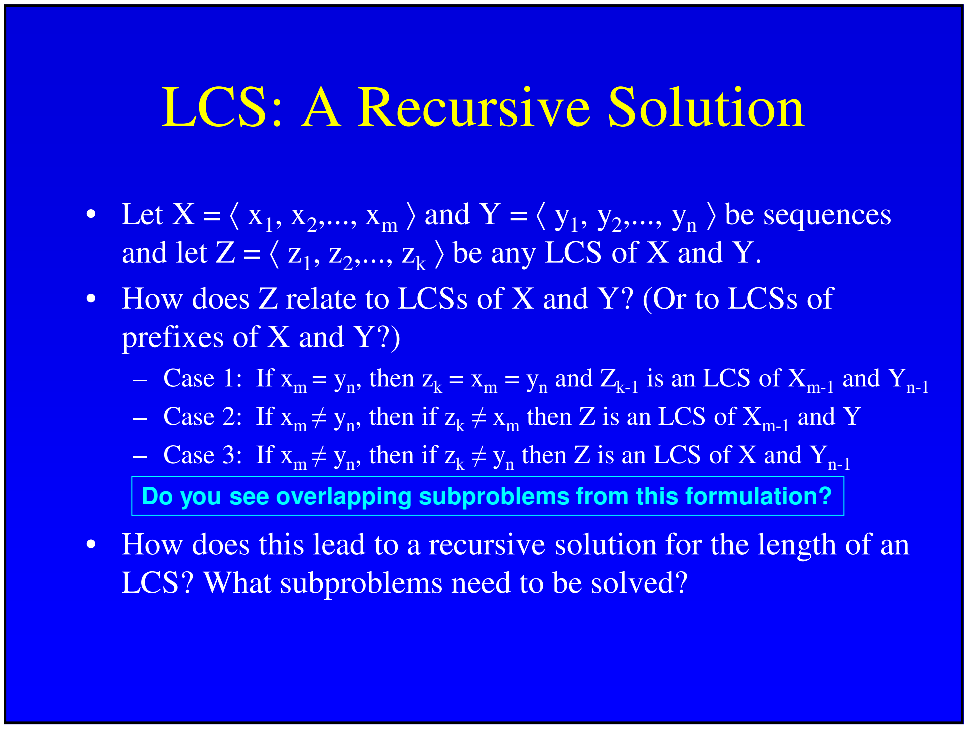
Dynamic Programming to the Rescue!

- The LCS problem seems like a candidate for a dynamic programming solution!
- Does it look like a recursive definition of LCS would be helpful? (Hint: Yes.)
  - Is there optimal substructure? Can the LCS of two sequences X and Y be built from the LCS of subsequences? (Divide and conquer!)
  - Are there overlapping sub-problems? Are there redundant calculations in a recursive solution?
- As part of a recursive / divide-and-conquer definition, we'll refer to the i'th prefix of a sequence:
- *Definition / Notation*: Given a sequence X = < x_1, x_2, …, x_m >, the *i'th prefix* (for i in [1..m]) is X_i = < x_1, x_2, …, x_i >
  - So, in our notation (see prev. slide), X = < x_1, x_2, ..., x_m > = X_m and Y = < y_1, y_2,..., y_n > = Y_n



LCS: A Recursive Solution

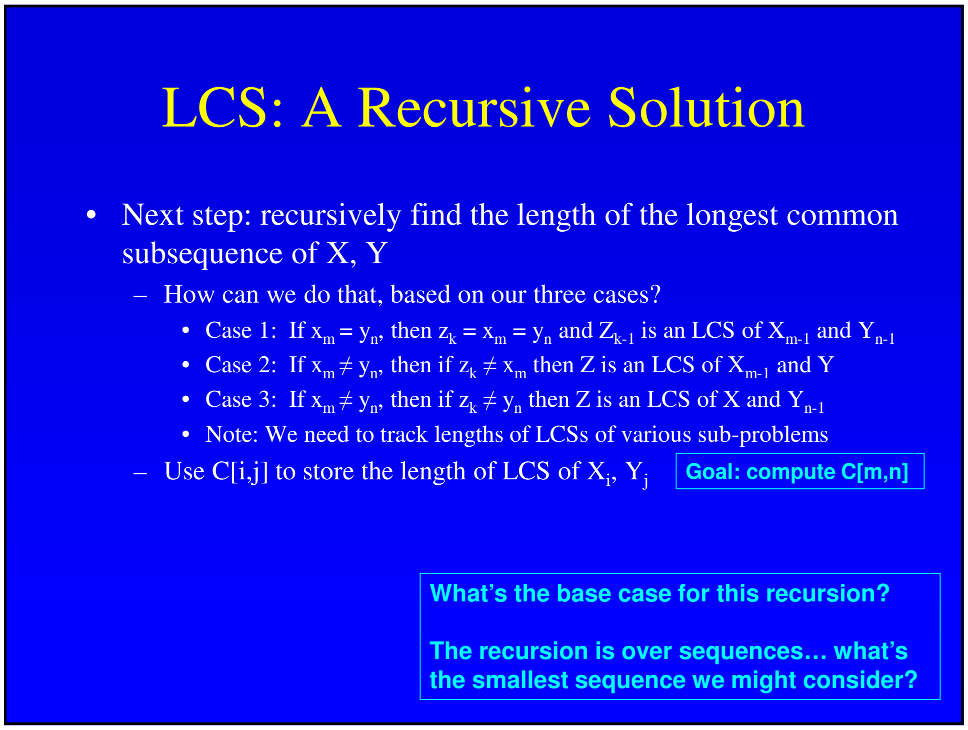
- Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.
- How does Z relate to LCSs of X and Y? (Or to LCSs of prefixes of X and Y?)
- How does this lead to a recursive solution for the length of an LCS? What subproblems need to be solved?



LCS: A Recursive Solution

- Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.
- How does Z relate to LCSs of X and Y? (Or to LCSs of prefixes of X and Y?)
  - Case 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}
  - Case 2: If $x_m \neq y_n$, then if $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y
  - Case 3: If $x_m \neq y_n$, then if $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}

  Do you see overlapping subproblems from this formulation?
- How does this lead to a recursive solution for the length of an LCS? What subproblems need to be solved?



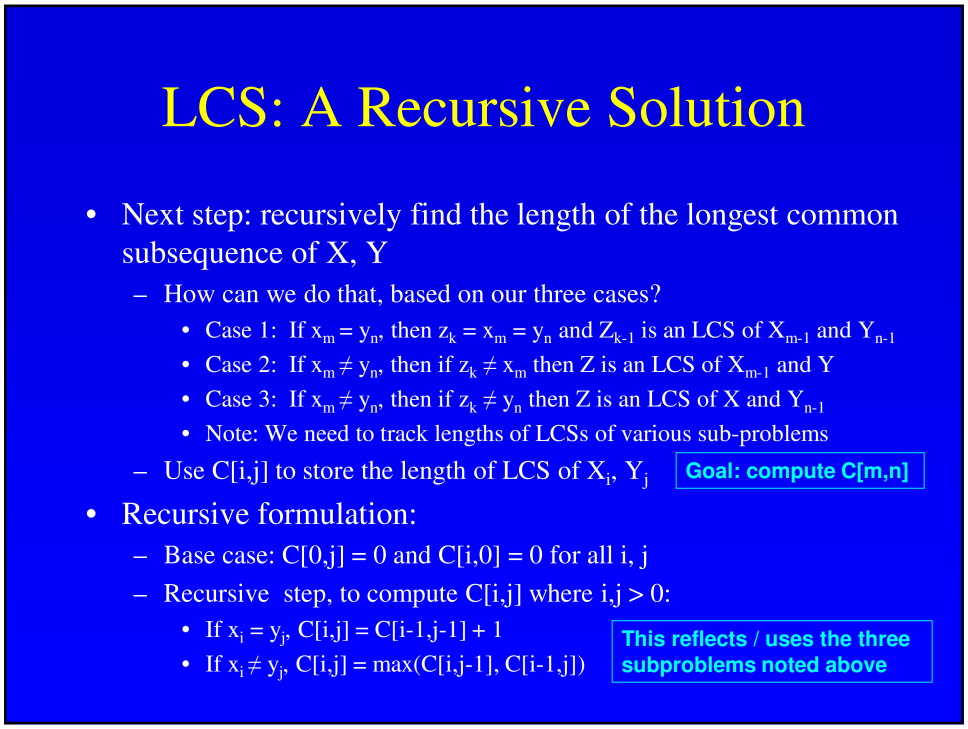
LCS: A Recursive Solution

- Next step: recursively find the length of the longest common subsequence of X, Y
  - How can we do that, based on our three cases?
    - Case 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}
    - Case 2: If $x_m \neq y_n$, then if $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y
    - Case 3: If $x_m \neq y_n$, then if $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}
    - Note: We need to track lengths of LCSs of various sub-problems
  - Use C[i,j] to store the length of LCS of X_i, Y_j

Goal: compute C[m,n]

What's the base case for this recursion?

The recursion is over sequences… what's the smallest sequence we might consider?



LCS: A Recursive Solution

- Next step: recursively find the length of the longest common subsequence of X, Y
  - How can we do that, based on our three cases?
    - Case 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}
    - Case 2: If $x_m \neq y_n$, then if $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y
    - Case 3: If $x_m \neq y_n$, then if $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}
    - Note: We need to track lengths of LCSs of various sub-problems
  - Use C[i,j] to store the length of LCS of X_i, Y_j

Goal: compute C[m,n]

- Recursive formulation:
  - Base case: C[0,j] = 0 and C[i,0] = 0 for all i, j
  - Recursive step, to compute C[i,j] where i,j > 0:
    - If $x_i = y_j$, C[i,j] = C[i-1,j-1] + 1
    - If $x_i \neq y_j$, C[i,j] = max(C[i,j-1], C[i-1,j])

This reflects / uses the three subproblems noted above

LCS: A Recursive Solution

- Recursively find the length of the LCS of X, Y
  - Base case: C[0,j] = 0 and C[i,0] = 0 for all i, j
  - Recursive step, to compute C[i,j] where i,j > 0:
    - If $x_i = y_j$, C[i,j] = C[i-1,j-1] + 1 (i.e., one less elt. of *both* X and Y)
    - If $x_i \neq y_j$, C[i,j] = max(C[i,j-1], C[i-1,j]) (i.e., one less elt. of *either* X or Y)
  - Straightforward recursive code:
    - Initialize C[i,0] = C[0,j] = 0 for i in [0..m], j in [0..n]
    - Initialize C[i,j] = NIL for i in [1..m], j in [1..n]

Initialize C, including the base cases

```
LCS(i, j) // i, j are indices – this is for a particular i, j
1.  if C[i, j] = NIL // could use other sentinel value
2.      then if $x_i = y_j$
3.              then C[i, j] ← LCS(i-1, j-1) + 1 // one less of both X,Y
4.              else C[i, j] ← max(LCS (i, j-1), LCS (i-1, j))
5.  return C[i, j]
```

For the Most… *Even More*

- Recall steps to developing a dyn. prog. algorithm:
  1. Characterize the structure of an optimal solution (in words)
  2. Recursively define the value of an optimal solution
  3. Compute the value of an optimal solution from the bottom up
  4. Construct an optimal solution from computed information
- We've done the first two. Now:
  - How to compute *the value of* an optimal solution (i.e., the length of a longest common subsequence) with bottom-up design instead of top-down recursion? (Perhaps using a table to avoid redundant work….)

    Stay with the same basic ideas, just expressed in a different design
  - What additional information would support constructing a solution—an actual longest common subsequence—from the computation of the optimal value?

Bottom-up Computation of Optimal LCS Value

- Need m-by-n matrix C to store lengths:
  - To compute C[i,j], need values of C[i-1,j-1] (when $x_i = y_j$) and C[i-1, j] and C[i, j-1] (when $x_i \neq y_j$)

Actually (m+1)-by-(n+1), to include the 0 case, too

Recall our recursive definition: Base case: C[0,j] = 0 and C[i,0] = 0 for all i, j
Recursive step, to compute C[i,j] for i,j > 0: If $x_i = y_j$, C[i,j] = C[i-1,j-1] + 1
If $x_i \neq y_j$, C[i,j] = max(C[i,j-1], C[i-1,j])

```
LCS(X, Y) // input: sequences X, Y
1. m ← length(X)
2. n ← length(Y)
3. for i ← 0 to m do  C[i, 0] ← 0  // 0 in first col of each row
4. for j ← 0 to n  do  C[0, j] ← 0 // 0 in first row of each col
5. for i ← 1 to m do
6.     for j ← 1 to n do    // process row by row
7.         if $x_i = y_j$ then C[i, j] ← C[i-1, j-1] + 1
8.         else C[i, j] ← max (C[i, j-1], C[i-1, j])
9. return C[m, n]
```

- What is the time complexity of this algorithm?

- What is the optimal length—the length of an LCS of full sequences X and Y?

Bottom-up Computation of An Optimal LCS

- To find an LCS, also store which symbols (indices of symbols) are actually part of the LCS as it's being built
  - i.e., which table elements have optimal sub-problem values
  - if $x_i = y_j$, answer came from the upper left (diagonal) of current element

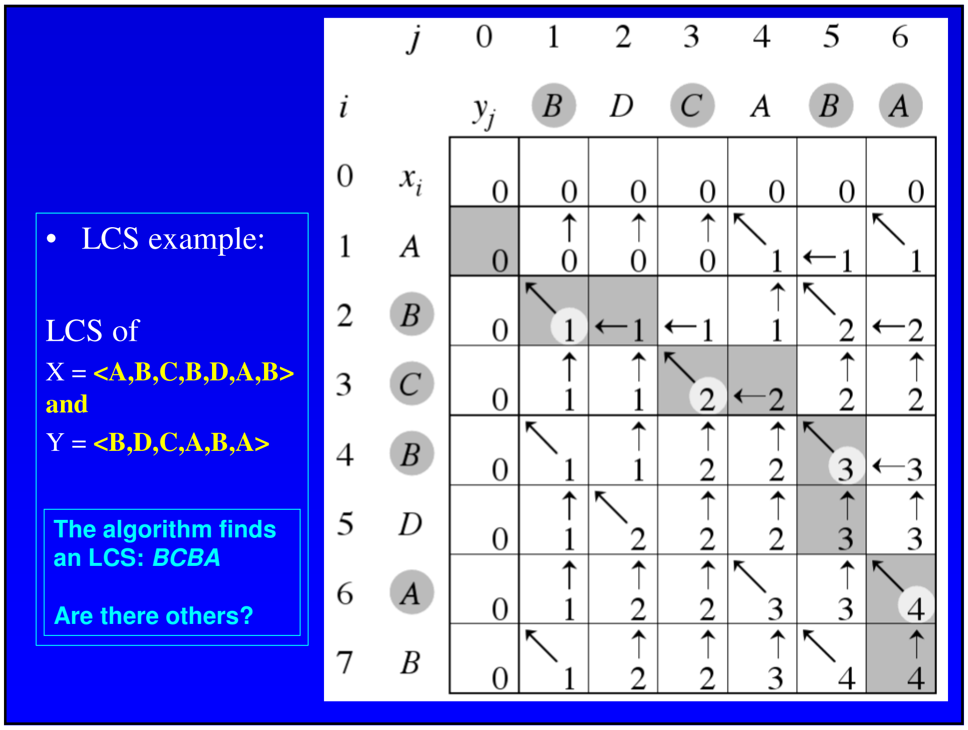
(i.e., one less elt. of *both* X and Y)

  - if $x_i \neq y_j$ the answer came from above or to the left, whichever is larger (if equal, we can choose "above", by convention)

(i.e., …*either* X or Y)

```
LCS(X, Y)
1. m ← length(X)
2. n ← length(Y)
3. for i ← 0 to m do  C[i, 0] ← 0
4. for j ← 0 to n  do  C[0, j] ← 0
5. for i ← 1 to m do
6.     for j ← 1 to n do
```

```
7.   if $x_i = y_j$ then C[i, j] = C[i-1, j-1] + 1
8.       B[i, j] ← Up&Left
9.   else
10.      if C[i - 1, j]  >= C[i, j - 1] then
11.          C[i, j] ← C[i - 1, j]
12.          B[i, j] ← Up //one less elt. of X
13.      else C[i , j] ← C[i, j - 1]
14.          B[i, j] ← Left //one less elt. of Y
```



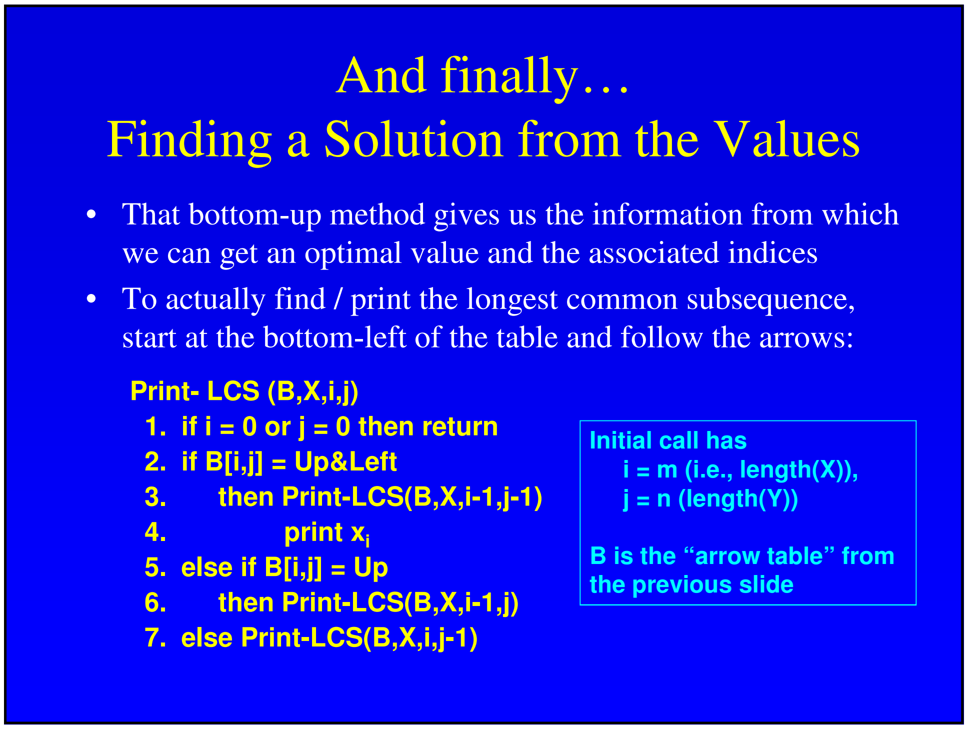
- LCS example:

LCS of
X = <A,B,C,B,D,A,B>
and
Y = <B,D,C,A,B,A>

The algorithm finds an LCS: *BCBA*

Are there others?

	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



And finally…
Finding a Solution from the Values

- That bottom-up method gives us the information from which we can get an optimal value and the associated indices
- To actually find / print the longest common subsequence, start at the bottom-left of the table and follow the arrows:

```
Print- LCS (B,X,i,j)
  1. if i = 0 or j = 0 then return
  2. if B[i,j] = Up&Left
  3.      then Print-LCS(B,X,i-1,j-1)
  4.           print $x_i$
  5. else if B[i,j] = Up
  6.      then Print-LCS(B,X,i-1,j)
  7. else Print-LCS(B,X,i,j-1)
```

Initial call has
i = m (i.e., length(X)),
j = n (length(Y))

B is the “arrow table” from the previous slide