

Lecture # 11

Dynamic Programming

Fibonacci Sequence

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

$$F_0 = 0$$

$$F_1 = 1$$

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

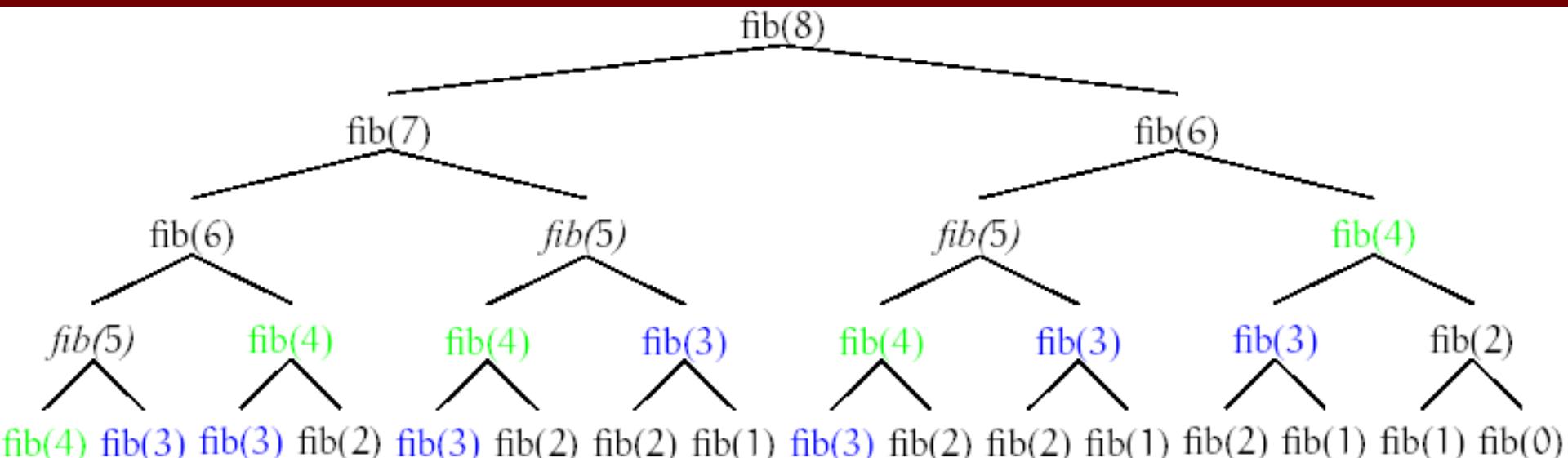
Fibonacci Sequence

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

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

Fibonacci Sequence

- Following Figure shows four levels of recursion for the call $\text{fib}(8)$:



Fibonacci Sequence

- A single recursive call to $\text{fib}(n)$ results in one recursive call to $\text{fib}(n - 1)$,
- two recursive calls to $\text{fib}(n - 2)$,
- three recursive calls to $\text{fib}(n - 3)$,
- five recursive calls to $\text{fib}(n - 4)$ and, so on....
- 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)
1  if (n < 2)
2    then return n
3  if (F[n] is undefined)
4    then F[n] ← MEMOFIB(n - 1) + MEMOFIB(n - 2)
5  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)
1   F[0] ← 0
2   F[1] ← 1
3   for i ← 2 to n
4   do
5       F[i] ← F[i - 1] + F[i - 2]
6   return F[n]
```

- This algorithm clearly takes only $O(n)$ time to compute F_n . By contrast, the original recursive algorithm takes $\Theta(\Phi^n)$, $\Phi = (1+\sqrt{5}) / 2 \approx 1.618$. ITERFIB achieves an exponential speedup over the original recursive algorithm.

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 as a simple combination of answers to smaller sub problems.
- Build solution to recurrence from bottom up. Write an algorithm that starts with base cases and works its way up to the final solution.

- Dynamic programming algorithms need to store the results of intermediate sub problems. This is often *but not always* done with some kind of table
- Dynamic Programming, like divide and conquer, solves the problems by combining the solutions to sub problems. (*programming here refers to a tabular method; not to write a computer code*).
- 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.

- The development of a Dynamic Programming algorithm can be broken into sequence of four steps.
 1. Characterize the *structure of an optimal solution*.
 2. Recursively define the *value of an optimal solution*.
 3. Compute the value of an optimal solution in a *bottom-up* fashion.
 4. *Construct an optimal solution* from computed information

For step 4, We sometimes maintains additional information during step 3 for easy construction.

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. Optimal Substructure of the problems

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

2. Overlapping Sub problems

Optimal subproblems to unrelated problems can contain subproblems in common

1. Optimal Substructure of the problems

- First step in solving in an optimization problem by **dynamic programming** is to characterize the structure of an optimal solution.
- Problem exhibits Optimal Substructure if an optimal solution to the problem contains within it optimal solutions to sub problems. (good clue to apply **dynamic programming**).
- In **dynamic programming**, we build an optimal solution to the problem from optimal solutions to sub problems

2. Overlapping Sub problems

- The space of the sub problems must be “small” in the sense that a recursive algorithm for the problem solves the same sub problems over and over, rather than always generating new sub problems.
- When a recursive algorithm revisits the same problem over and over again we say that the optimization problem has Overlapping Sub problems.

- 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.

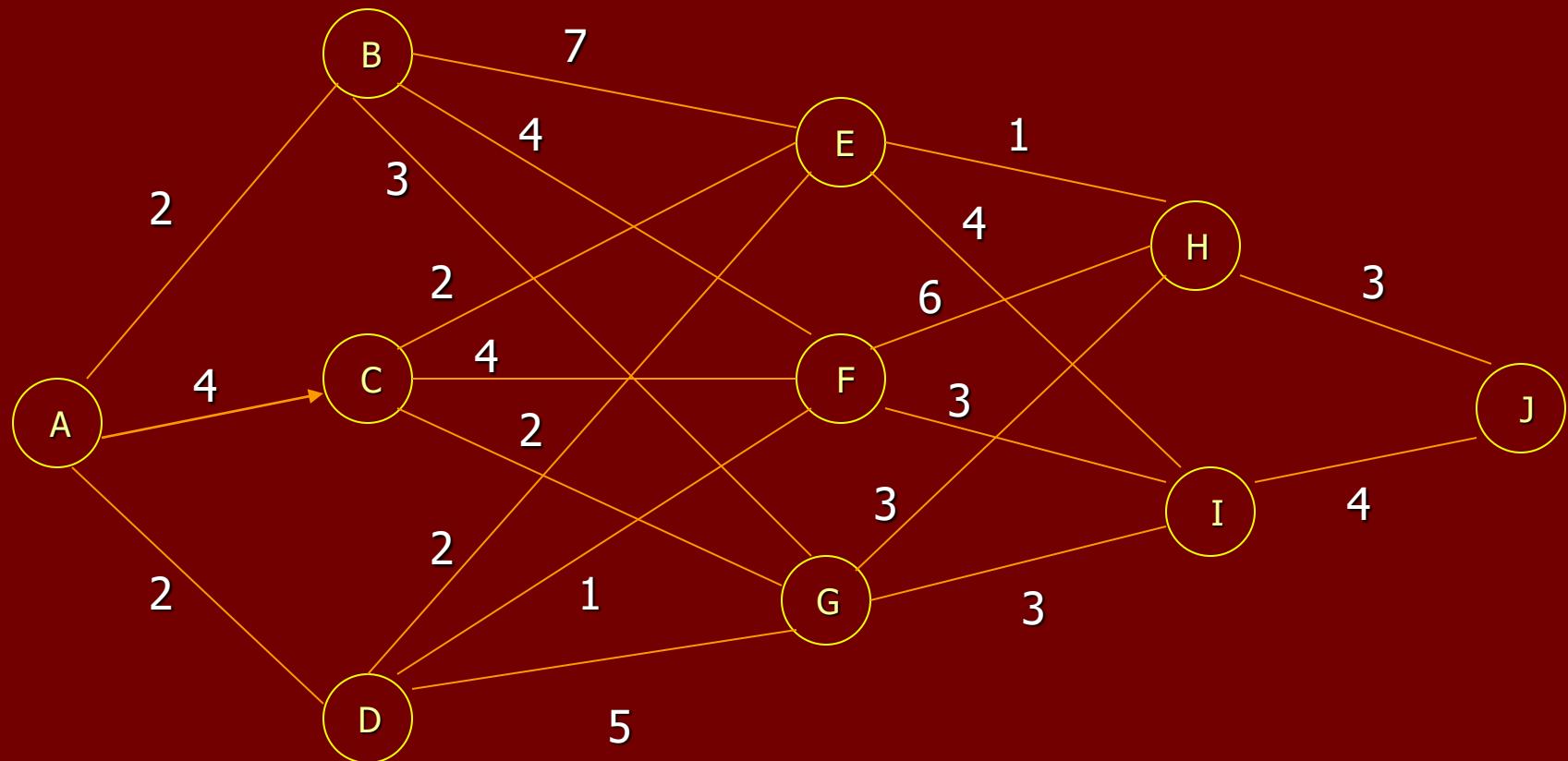
Dynamic Programming

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

S_2	$C_{sz} + f_3(z)$			$f_2(S_2)$	Decision to go
	E	F	G		
B	11	11	9	9	G
C	6	11	8	6	E
D	6	8	11	6	E

S_1	$C_{sz} + f_2(z)$			$f_1(S_1)$	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 $X = \{ x_1, x_2, \dots, x_m \}$, we define the i^{th} prefix of X , for $i = 0, 1, 2, \dots, m$ as
$$X_i = \{ x_1, x_2, \dots, x_i \}$$
- For example $X = \{A, B, C, B, D, A, B\}$ then
$$X_4 = \{A, B, C, B\}$$

Theorem (optimal substructure of an LCS)

- Let $X = \{x_1, x_2, \dots, x_m\}$, and $Y = \{y_1, y_2, \dots, y_n\}$ be sequences and let $Z = \{z_1, z_2, \dots, z_k\}$ 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 $x_m \neq y_n$ then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y .
 3. If $x_m \neq y_n$ then $z_k \neq y_n$ 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:

$$c[i,j] = 0 \quad \text{if } i=0 \text{ or } j=0$$

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

- *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_{j-1} , plus 1

LCS recursive solution

- Second case: $x[i] \neq y[j]$
- As symbols don't match, our solution is not improved, and the length of $\text{LCS}(X_i, Y_j)$ is the same as before (i.e. maximum of $\text{LCS}(X_i, Y_{j-1})$ and $\text{LCS}(X_{i-1}, Y_j)$)

Why not just take the length of $\text{LCS}(X_{i-1}, Y_{j-1})$?

LCS Length Algorithm

LCS-Length(X, Y)

1. $m = \text{length}(X)$ // get the # of symbols in X
2. $n = \text{length}(Y)$ // 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?

$$\text{LCS}(X, Y) = BCB$$

$X = A \ B \ C \ B$

$Y = \quad B \ D \ C \ A \ B$

LCS Example (0)

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

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

$$c[0,j] = 0$$

ABCB
BDCAB

LCS Example (2)

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				
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

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 ( Xi == Yj )
    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
	Yj	Y	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 ( Xi == Yj )
    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)

ABC
BDCAB

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

```

if ( Xi == Yj )
    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
	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					

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

LCS Example (7)

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	
3	C	0					
4	B	0					

```

if ( Xi == Yj )
    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
	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	Y _j	B	D	C	A	B
Xi	0	0	0	0	0	0
A	0	0	0	0	1	1
B	0	1	1	1	1	2
3	0	1	1			
B	0					
4	0					

```

if ( Xi == Yj )
    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	Y	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					

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

LCS Example (12)

ABCB
BDCAB

i	j	0	1	2	3	4	5
	Yj	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 ( Xi == Yj )
    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	Y _j	B	D	C	A	B
0	X _i	0	0	0	0	0
1	A	0	0	0	0	1
2	B	0	1	1	1	1
3	C	0	1	1	2	2
4	B	0	1			

$$\begin{aligned} & \text{if } (X_i == Y_j) \\ & \quad c[i,j] = c[i-1,j-1] + 1 \\ & \text{else } c[i,j] = \max(c[i-1,j], c[i,j-1]) \end{aligned}$$

LCS Example (14)

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	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	Y _j	B	D	C	A	B
0	X _i	0	0	0	0	0
1	A	0	0	0	0	1
2	B	0	1	1	1	2
3	C	0	1	1	2	2
4	B	0	1	1	2	3

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

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

The diagram illustrates the construction of an LCS matrix. The matrix is a grid where rows represent sequence X and columns represent sequence Y. The value at each cell (i, j) is determined by comparing the i-th character of X and the j-th character of Y. If they are equal, the value is the same as the value in the previous row and column (top-left diagonal). If they are different, the value is the maximum of the value from the previous row (left) or the previous column (top). The final value in the bottom-right corner represents the length of the Longest Common Subsequence (LCS).

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