DP Examples

- Fibonacci
- Binomial Coefficients
- Longest Common Subsequence
- Longest Increasing Subsequence
- Knapsack
- Shortest Path
- Chain Matrix Multiplication
- Edit Distance
- Rod Cutting
- Optimal BST

Longest Common Subsequence

- Given two sequences x[1..m] and y[1..n]

  $X = \langle x_1, x_2, \ldots, x_m \rangle$

  $Y = \langle y_1, y_2, \ldots, y_n \rangle$

  find a maximum length common subsequence (LCS) of X and Y

- *E.g.:*

  $X = \langle A, B, C, B, D, A, B \rangle$

- Subsequences of X:
  - A subset of elements in the sequence taken in order

  $\langle A, B, D \rangle$, $\langle B, C, D, B \rangle$, etc.

Example

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

- $\langle B, C, B, A \rangle$ and $\langle B, D, A, B \rangle$ are longest common subsequences of X and Y (length = 4)
- $\langle B, C, A \rangle$, however is not a LCS of X and Y

Longest Common Subsequence (LCS)

Application: comparison of two DNA strings

Ex: X= $\langle$A, B, C, B, D, A, B $\rangle$, Y= $\langle$B, D, C, A, B, A$\rangle$

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

Brute-Force Solution

- For every subsequence of X, check whether it's a subsequence of Y
- There are 2^m subsequences of X to check
- Each subsequence takes $\Theta(n)$ time to check
  - scan Y for first letter, from there scan for second, and so on
- Running time: $\Theta(n2^m)$

Steps in Dynamic Programming

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values bottom-up in a table.
4. Construct an optimal solution from computed values.

We’ll study these with the help of examples.

Making the choice

$X = \langle A, B, D, E \rangle$

$Y = \langle Z, B, E \rangle$

- Choice: include one element into the common sequence (E) and solve the resulting subproblem

$X = \langle A, B, D, G \rangle$

$Y = \langle Z, B, D \rangle$

- Choice: exclude an element from a string and solve the resulting subproblem

Notations

- Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$ we define the i-th prefix of X, for $i = 0, 1, 2, \ldots, m$

  $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ or $x[1,..,i]$

  $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$ or $y[1,..,j]$

- $c[i, j]$ = the length of a LCS of the sequences $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$

A Recursive Solution

Case 1: $x_i = y_j$

e.g.: $X_i = \langle A, B, D, E \rangle$

$Y_j = \langle Z, B, E \rangle$

$$c[i, j] = c[i\text{-}1, j\text{-}1] + 1$$

- Append $x_i = y_j$ to the LCS of X_{i-1} and Y_{j-1}
- Must find a LCS of X_{i-1} and Y_{j-1} $\Rightarrow$ optimal solution to a problem includes optimal solutions to subproblems

A Recursive Solution

Case 2: $x_i \neq y_j$

e.g.: $X_i = \langle A, B, D, G \rangle$

$Y_j = \langle Z, B, D \rangle$

$$c[i, j] = \max \{ c[i-1, j], c[i, j-1] \}$$

– Must solve two problems

- find a LCS of X_{i-1} and Y_j: $X_{i-1} = \langle A, B, D \rangle$ and $Y_j = \langle Z, B, D \rangle$
- find a LCS of X_i and Y_{j-1}: $X_i = \langle A, B, D, G \rangle$ and $Y_j = \langle Z, B \rangle$

- Optimal solution to a problem includes optimal solutions to subproblems

Overlapping Subproblems

- To find a LCS of X and Y
  - we may need to find the LCS between X and Y_{n-1} and that of X_{m-1} and Y
  - Both the above subproblems has the subproblem of finding the LCS of X_{m-1} and Y_{n-1}
- Subproblems share subsubproblems

Finding LCS Length

Define c[i,j] to be the length of the LCS of x[1..,*i*] and y[1.., *j*]

Theorem:

$$c[i,j] = \begin{cases} 0 & i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

LCS 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 $c[m,n]$

$$c[i,j] = \begin{cases} 0 & i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

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 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

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[4,5]

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

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

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**					
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

	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		
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(mn)

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

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)

	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**

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

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

Additional Information

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

b & c:

		0	1	2	3		n
		y_j:	A	C	D		F
0	x_i	0	0	0	0	0	0
1	A	0					
2	B	0			c[i-1,j]		
3	C	0		↖ c[i,j-1]	↑		
		0					
m	D	0					

i

j

A matrix b[i, j]:

- For a subproblem [i, j] it tells us what choice was made to obtain the optimal value
- If $x_i = y_j$
  b[i, j] = "↖"
- Else, if
  $c[i - 1, j] \geq c[i, j-1]$
  b[i, j] = " ↑ "
  else
  b[i, j] = " ← "

Example

$X = \langle A, B, C, B, D, A \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

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

If $x_i = y_j$

$b[i, j] = $ " $\nwarrow$ "

Else if

$c[i - 1, j] \geq c[i, j-1]$

$b[i, j] = $ " $\uparrow$ "

else

$b[i, j] = $ " $\leftarrow$ "

		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2	B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3	C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4	B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5	D	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
6	A	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
7	B	0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

Constructing a LCS

- Start at b[m, n] and follow the arrows
- When we encounter a "↖" in b[i, j] $\Rightarrow$ $x_i = y_j$ is an element of the LCS

		0	1	2	3	4	5	6
		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

PRINT-LCS(b, X, i, j)

Running time: $\Theta(m + n)$

1. **if** $i = 0$ or $j = 0$
2. **then return**
3. **if** $b[i, j] =$ "$\nwarrow$"
4. **then** PRINT-LCS(b, X, $i - 1$, $j - 1$)
5. print x_i
6. **elseif** $b[i, j] =$ "$\uparrow$"
7. **then** PRINT-LCS(b, X, $i - 1$, j)
8. **else** PRINT-LCS(b, X, i, $j - 1$)

Initial call: PRINT-LCS(b, X, $length[X]$, $length[Y]$)

Improving the Code

- What can we say about how each entry $c[i, j]$ is computed?
  - It depends only on $c[i-1, j-1]$, $c[i-1, j]$, and $c[i, j-1]$
  - Eliminate table b and compute in $O(1)$ which of the three values was used to compute $c[i, j]$
  - We save $\Theta(mn)$ space from table b
  - However, we do not asymptotically decrease the auxiliary space requirements: still need table c

Improving the Code

- If we only need the length of the LCS
  - LCS-LENGTH works only on two rows of c at a time
    - The row being computed and the previous row
  - We can reduce the asymptotic space requirements by storing only these two rows