Longest Common Subsequence

Review: Dynamic programming

- DP is a method for solving certain kind of problems
- DP can be applied when the solution of a problem includes solutions to subproblems
- We need to find a recursive formula for the solution
- We can recursively solve subproblems, starting from the trivial case, and save their solutions in memory
- In the end we'll get the solution of the whole problem

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

Properties of a problem that can be solved with dynamic programming

- Simple Subproblems
  – We should be able to break the original problem to smaller subproblems that have the same structure
- Optimal Substructure of the problems
  – The solution to the problem must be a composition of subproblem solutions
- Subproblem Overlap
  – Optimal subproblems to unrelated problems can contain subproblems in common

Longest Common Subsequence Problem

- Given two strings X and Y, a common subsequence is a subsequence that appears in both X and Y.
- The longest common subsequence problem is to find a longest common subsequence (lcs) of X and Y
  - subsequence: characters need not be contiguous
  - different than substring
- Can you use dynamic programming to solve the longest common subsequence problem?

Longest Common Subsequence

- *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 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 alg?*

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)

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

LCS Length Algorithm

```
LCS-Length(X, Y)
m = length(X)  // get the # of symbols in X
n  = length(Y) // get the # of symbols in Y
for i = 1 to m  c[i,0] = 0   // special case: Y_0
for j = 1 to n   c[0,j] = 0   // special case: X_0
for i = 1 to m                // for all X_i
    for j = 1 to n                     // for all Y_j
        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] )
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

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[5,4]

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

	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

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

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**	**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])

ABCB
BDCAB

LCS Example (6)

	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])

ABCB
BDCAB

LCS Example (7)

	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

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

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

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

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

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 - 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)

	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**
(this string turned out to be a palindrome)

Review: Longest Common Subsequence (LCS)

- Problem: how to find the longest pattern of characters that is common to two text strings X and Y
- Dynamic programming algorithm: solve subproblems until we get the final solution
- Subproblem: first find the LCS of *prefixes* of X and Y.
- this problem has *optimal substructure*: LCS of two prefixes is always a part of LCS of bigger strings

Review: Longest Common Subsequence (LCS) continued

- Define X_i, Y_j to be prefixes of X and Y of length i and j; m = |X|, n = |Y|
- We store the length of LCS(X_i, Y_j) in c[i,j]
- Trivial cases: LCS(X_0 , Y_j) and LCS(X_i, Y_0) is empty (so c[0,j] = c[i,0] = 0)
- Recursive formula for c[i,j]:

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

c[m,n] is the final solution

Review: Longest Common Subsequence (LCS)

- After we have filled the array *c[ ]*, we can use this data to find the characters that constitute the Longest Common Subsequence

- Algorithm runs in $O(m*n)$, which is *much* better than the brute-force algorithm: $O(n\ 2^m)$

The End