Dynamic Programming:
Example 1: Longest Common Subsequance

We look at sequences of characters (strings)

e.g. *x="ABCA"*

Def: A **subsequence** of *x* is an sequence obtained from *x* by possibly deleting some of its characters (but without changing their order

Examples:
"ABC", *"ACA"*, *"AA"*, *"ABCA"*

Def A **prefix** of *x*, denoted *x[1..m]*, is the sequence of the first *m* characters of *x*

Examples:
x[1..4]="ABCA" *x[1..3]="ABC"* *x[1..2]="AB"*
x[1..1]="A" *x[1..0]=""*

F(n){
if n=1 || n=2
return 1
else
reture f(n-1)+f(n-2)

F(20)
F(19) F(18)
F(18) ...

Memorization : doesn't compute a value that is already computed

Use Array c[1, ... , n] c(n)

Linux Example

file1	file2
aa	aa
bb	cc
cc	

diff (edit distance LCS)

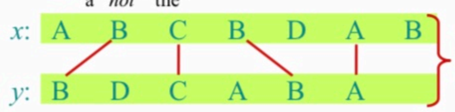
segments lines should not cross

鸡你太美

Longest Common Subsequence (LCS)

- Given two sequences $x[1 \,..\, m]$ and $y[1 \,..\, n]$, find a longest subsequence common to them both.

"a" *not* "the"



Different phrasing: Find a set of a maximum number of segments, such that
- Each segment connects a character of x to an identical character of y,
- Each character is used at most once
- Segments do not intersect.

Brute Force

Brute-force LCS algorithm

Checking every subsequence of x whether it is also a subsequence of y.

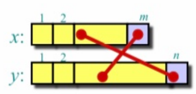
Analysis
- Checking = $\Theta(m+n)$ time per subsequence.
- 2^m subsequences of x

Worst-case running time = $\Theta((m+n)2^m)$
= exponential time.

Recursive formulation

Observation:
It is impossible that
x[m] is matched to an element in *y[1..n-1]* and simultaneously
y[n] is matched to an element in *x[1..m-1]*
(since it must create a pair of crossing segments).

Conclusion – *either x[m] is matched to y[n], or one at least of them is unmatched in* ***OPT.***
*{****OPT*** *– the optimal solution}*



LCS ∈ Textbook
edit distance ≈∈ Textbook
Frechet dtw ∉

Towards a better algorithm

Simplification:
1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Notation: Denote the length of a sequence s by $|s|$.

Strategy: Consider ***prefixes*** of x and y.
- Define $c[i, j] = |\mathrm{LCS}(x[1 \,..\, i], y[1 \,..\, j])|$.
- Then, $c[m, n] = |\mathrm{LCS}(x, y)|$.

S = "ABCD" |s| = 4
LCS("$\underbrace{\text{abc}}_{i}$", "$\underbrace{\text{adbdcd}}_{j}$")

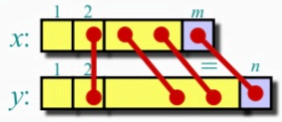
c(3, 6) LCS("$\underbrace{\text{abc}}_{3}$", "$\underbrace{\text{adbdcd}}_{6}$") = 3
c(1, 1) = 1 c(0, 6) = 0
c(3, 3) = 2 LCS("abc", "adbdcd")

Recursive formaula

Lets just consider the last character of of x and of y

Case (I): $x[m] = y[n]$. Claim: $c[m, n]=c[m-1,n-1]+1$.

Proof.



We claim that there is a max matching that matches $x[m]$ to $y[n]$.

Indeed, if $x[m]$ is matched to $y[k]$ (for $k<m$) then $y[n]$ is unmatched (otherwise we have two crossing segments). Hence we can obtain another matching of the same cardinality by matching $x[m]$ to $y[n]$.

This implies that we can find an optimal matching of LCS($x[1..m-1]$ to y[$1..n-1$], and add the segment ($x[m]$,$y[n]$).

So $c[m,n]=c[m-1,n-1]+1$

ABCD | Consider optimal solution

DABC

ABD   C[3,6]

ABDrrD   "ABD"

x = ABC

y = ABC

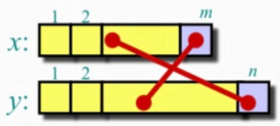
c[3,3] = 1 + c[2,2]

= 1 + 1 + c[1,1]

Recursive formulation-cont

Case (II): $x[m] \neq y[n]$ Claim: $c[m,n]=\max\{c[m,n-1], c[m-1,n]\}$

Recall - in LCS($x[1..m]$, $y[1..n]$) it cannot be that **both** $x[m]$ and $y[n]$ are both matched.



If $x[m]$ is unmatched in OPT then

LCS($x[1..m]$, $y[1..n]$)= LCS($x[1..m-1]$, $y[1..n]$)

If $y[j]$ is unmatched in OPT then

LCS($x[1..m]$, $y[1..n]$)= LCS($x[1..m]$, $y[1..n-1]$)

So $c[m,n]= \max\{c[m-1, n], c[m, n-1]\}$

C[4,5]

max { C[3,5]   ABCd

ABCff

max { C[3,4]

C[2,5] }

C[3,4] = max{ C[3,3], C[2,4] }

Start from either's end and see which one gives longer segment.

ABC

ADBD

C[3,4]

= max C[2,4) C[3,3]

1+ C(1 3)   max

C[2,3

C(3,2)

Recursive algorithm for LCS

LCS(x, y, i, j)

  if (i==0 **or** j=0) return 0

  if $x[i] = y[j]$

    then return LCS(x, y, i–1, j–1) + 1

    else return max{ LCS(x, y, i–1, j),

LCS(x, y, i, j–1)}

To call the function LCS(x, y, m,n)

Worst-case: $x[i] \neq y[j]$, for all i,j in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

LCS: Dynamic-programming algorithm

LCS(X,Y)="BCBA"

X=B D C A B A

Y=A B C B D A B

		1 A	2 B	3 C	4 B	5 D	6 A	7 B
X=	0	0	0	0	0	0	0	0
1 B	0	0	1	1	1	1	1	1
2 D	0	0	1	1	1	2	2	2
3 C	0	0	1	2	2	2	2	2
4 A	0	1	1	2	2	2	3	3
5 B	0	1	2	2	3	3	3	4
6 A	0	1	2	2	3	3	4	4

Reconstruction $z=LCS(x,y)$

IDEA: Compute the table bottom-up. Fill z backward.

Observation: $c[i;j]\geq c[i-1;j]$ and $c[i;j] \geq c[i;j-1]$

Proof Sketch: We use a longer prefix, so there are more chars to be match.

LCS Reconstruction:

```
Set i=m;  j=n;  k=c[i;j]
While(k>0){
  if (c[i;j]>c[i-1;j] and c[i;j]>c[i;j-1] ) {
        z[k] = x[i] ;
        i--; j-- ; k-- ;
  }else // c[i;j]=c[i-1;j] or c[i;j]=c[i-1;j]
  if (c[i;j]==c[i;j-1]) j-- ;
  else i-- ;
}
```

$LCS(x,y)$="BCBA"

x=B D C A B A

y=A B C B D A B

		1 A	2 B	3 C	4 B	5 D	6 A	7 B
	0	0	0	0	0	0	0	0
1 B	0	0	1	1	1	1	1	1
2 D	0	0	1	1	1	2	2	2
3 C	0	0	1	2	2	2	2	2
4 A	0	1	1	2	2	2	3	3
5 B	0	1	2	2	3	3	3	4
6 A	0	1	2	2	3	3	4	4

x = ABCD

y = ABBD

Memoization algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

```
LCS(x, y)
  for i=0 to m   c[i, 0] = 0
  for j=0 to n   c[0, j] = 0

  for i=1 to m
    for j=1 to n
      if (x[i] = y[j] )
        then c[i, j] ← c[ i–1, j–1]  + 1
        else c[i, j] ← max{ c[ i–1, j],  c[i, j–1] }
```

Time = $\Theta(mn)$ = constant work per table entry.
Space = $\Theta(mn)$.

$y =$ ABCD

			1	2	3	4
		y	A	B	C	D
	x	0	0	0	0	0
1	A	0	1	1	1	1
2	D	0	1	1	1	2
3	B	0	1	2	2	2
4	D	0	1	2	2	3

A B C | D
$\underbrace{\text{A D B}}_{2}$ | D

D