CS4102 Algorithms

Spring 2021 – Floryan and Horton

Module 3, Day 5: Sequence Algorithms

(1) String Edit Distance

(Not in textbook. Certainly not the Shakespeare part!)

(2) Longest Common Subsequence

(CLRS 15.4)

Sequences

- Problems about the relationship between two sequences
  - Strings (sequence of characters)
  - Text (sequence of words)
  - Genetic sequences (sequence of nucleotides)
  - Etc.
- How similar are two sequences? **String Edit Distance**
- How one could be transformed to the other?
- What do they share in common? **Longest Common Subsequence**

Sequence Alignment

How could we define the *similarity* of two strings?

- Maybe character-by-character, showing places there's an "issue"?
- This is the idea of an ***alignment***
  - Mismatched characters, insertions or deletions (gaps)
- Example: *ocurrance* vs. *occurrence.* Three possible alignments are:

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	-	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

0 mismatches, 3 gaps

Edit Distance

[Levenshtien 1966, Needleman-Wunsch 1970]

- Gap penalty δ and a mismatch penalty α_{pq} for pair *p* and *q*
- Cost/distance = sum of gap and mismatch penalties

A	T	C	G	A	C	T	C	A	G	T
A	T	-	G	A	C	A	G	A	G	T

Often $\boldsymbol{\alpha_{pq}} = 2$ and $\boldsymbol{\delta}$=1

$$\boldsymbol{cost = \delta + \alpha_{TA} + \alpha_{CG}} \quad \textbf{(assuming } \boldsymbol{\alpha_{pp} = 0}\textbf{)}$$

Many applications: Spelling correction, bioinformatics, text processing and analysis, speech recognition, computer file difference (diff), birdsong analysis, …

An Interesting Application You Probably Never Considered!

- Text Processing in Early Modern English drama texts (!)
- Scholars' problems involve knowing if/how two versions of a text differ
  - A document created from one (or more) older versions
    - When they differ, which is "correct"?
  - Spelling wasn't standardized in the 1590s/early 1600s

- CS researchers in field of *digital humanities* applied sequence alignment algorithms to spelling variation, collation, …
  - E.g. someone named Horton (paper in *Research in Humanities Computing*, Vol. 2, 1994)

An Example of the Problem

Mee thought all his senses were lokt in his eye
As iewels in christall for some prince to buy
Who tendring their owne worth from where they were glast
Did poynt you to buy them along as you past

– *Love's Labours Lost* (Quarto, 1598)

Me thought all his sences were lockt in his eye
As iewels in christall for some prince to buy
Who tendring their own worth from whence they were glast
Did point out to buy them along as you past

– *Love's Labours Lost* (First Folio, 1623)

Are these different? If so, where?

An Example of the Problem

Mee thought all his **senses** were **lokt** in his eye
As iewels in christall for some prince to buy
Who tendring their **owne** worth from **where** they were glast
Did **poynt you** to buy them along as you past

– *Love's Labours Lost* (Quarto, 1598)

Me thought all his **sences** were **lockt** in his eye
As iewels in christall for some prince to buy
Who tendring their **own** worth from **whence** they were glast
Did **point out** to buy them along as you past

– *Love's Labours Lost* (First Folio, 1623)

Words marked like this are different strings. But…

An Example of the Problem

Mee thought all his **senses** were **lokt** in his eye
As iewels in christall for some prince to buy
Who tendring their **owne** worth from **where** they were glast
Did **poynt** **you** to buy them along as you past

– *Love's Labours Lost* (Quarto, 1598)

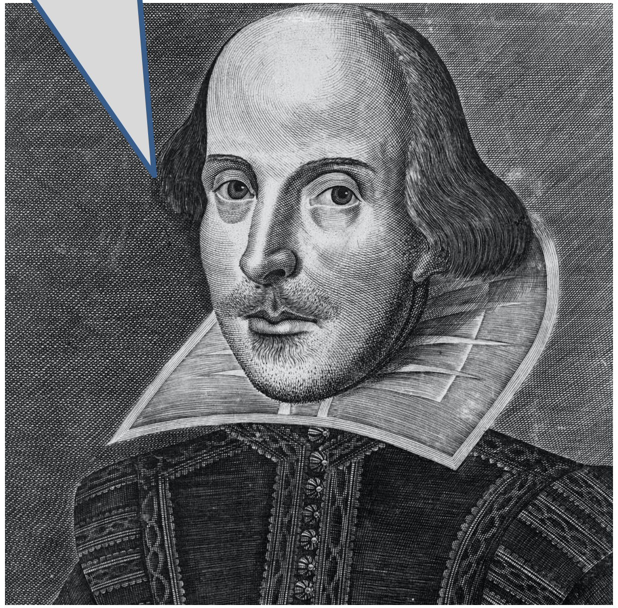
Me thought all his **sences** were **lockt** in his eye
As iewels in christall for some prince to buy
Who tendring their **own** worth from **whence** they were glast
Did **point** **out** to buy them along as you past

– *Love's Labours Lost* (First Folio, 1623)

Most scholars would say all but two are only spelling variations and not "substantive variants"

- Are two words the same?
  - Find string-edit distance
  - If higher than some threshold, consider them the same word
  - Use α_{pq} to address Early ModE spellings
    - senses/sences, poynt/point, yfaith/i'faith
  - Use a **variable gap penalty** to lessen impact of some insertions
    - fierworke/fier-worke, youle/you’l, yfaith/i’faith

OK, 21st century know-it-all, how can any of this help?



This ends our side-trip to *Hacking for Humanists*

Where were we before? Oh yeah…

[Levenshtien 1966, Needleman-Wunsch 1970]

- Gap penalty δ and a mismatch penalty α_{pq} for pair *p* and *q*
- Cost/distance = sum of gap and mismatch penalties

A	T	C	G	A	C	T	C	A	G	T
A	T	-	G	A	C	A	G	A	G	T

Often $\boldsymbol{\alpha_{pq}} = 2$ and $\boldsymbol{\delta}$=1

$$\boldsymbol{cost = \delta + \alpha_{TA} + \alpha_{CG}} \quad \textbf{(assuming } \boldsymbol{\alpha_{pp} = 0}\textbf{)}$$

Many applications: Spelling correction, bioinformatics, text processing and analysis, speech recognition, computer file difference (diff), birdsong analysis, **text processing in old-spelling texts**

Sequence Alignment: Problem Statement

- **Problem:** Given two strings X = $x_1 \ldots x_m$ and Y = $y_1 \ldots y_n$ find a min-cost alignment
- An **alignment** M is a set of ordered pairs (x_i, y_j) such that each character appears in one pair and there are no crossings
  - Two pairs (x_{i1}, y_{j1}) and (x_{i2}, y_{j2}) cross if $i_1 < i_2$ but $j_1 > j_2$
- The cost of an alignment M is:

$$cost(M) = \underbrace{\sum_{x_i, y_j \in M} \alpha_{x_i y_j}}_{\textbf{mismatches}} + \underbrace{\sum_{i\,:\,x_i \text{ unmatched}} \delta + \sum_{j:\,y_j \text{ unmatched}} \delta}_{\textbf{gaps}}$$

Example: Alignment and Matching

Example: an alignment of X = CGACTA and Y = GACAGA

x_1	x_2	x_3	x_4	x_5		x_6
C	G	A	C	T	-	A
-	G	A	C	A	G	A
	y_1	y_2	y_3	y_4	y_5	y_6

$$\boldsymbol{M} = \{\, (\boldsymbol{x_2}, \boldsymbol{y_1}), (\boldsymbol{x_3}, \boldsymbol{y_2}), (\boldsymbol{x_4}, \boldsymbol{y_3}), (\boldsymbol{x_5}, \boldsymbol{y_4}), (\boldsymbol{x_6}, \boldsymbol{y_6}) \,\}$$

Note:

- M contains “matches” where there is a mismatch cost
- Items that are part of gaps are not in M. We say they don’t “match”

Dynamic Programming

- Requires Optimal Substructure
  - Solution to larger problem contains the solutions to smaller ones
- Avoid extra work due to overlapping subproblems
- Idea:
  1. Identify the recursive structure of the problem
     - What is the "last thing" done?
  2. Save the solution to each subproblem in memory
  3. Select a good order for solving subproblems
     - "Top Down": Solve each recursively
     - "Bottom Up": Iteratively solve smallest to largest

Recursive Structure

- Let *X* have size *m* and *Y* have size *n*, and M be an optimal alignment
- Either $(x_m, y_n) \in M$ or it's not. In other words, the last two symbols in each string match each other or they don't.
  - Remember: "match" may involve mismatch cost! This means: not a gap.
- For any alignment M, if $(x_m, y_n) \notin M$ then either x_m or y_n is not matched in M.
  - If both were in M but didn't match each other, we'd have a crossing
- Thus, in an optimal alignment M, at least one of these 3 is true:
  1. $(x_m, y_n) \in M$
  2. x_m is not matched
  3. y_n is not matched

Subproblems and Costs

- How to think about subproblems: ***prefix strings***
- $Opt(i, j)$ = min cost aligning prefix strings $X_i = x_1 \ldots x_i$ and $Y_j = y_1 \ldots y_j$
- Our recurrence for $Opt(i, j)$ will have 3 cases:

  1. $(x_i, y_j) \in M$. The cost will be:
     mismatch cost for (x_m, y_n), plus min-cost to align $x_1 \ldots x_{i-1}$ and $y_1 \ldots y_{j-1}$
  2. x_i is not matched. The cost will be:
     gap cost for x_i, plus min-cost to align $x_1 \ldots x_{i-1}$ and $y_1 \ldots y_j$
  3. y_j is not matched. The cost will be:
     gap cost for y_j, plus min-cost to align $x_1 \ldots x_i$ and $y_1 \ldots y_{j-1}$

(BTW, you can prove optimal substructure property on these 3 cases using an exchange argument.)

The Recurrence

$$OPT(i,j) = \begin{cases} j\delta & \text{if } i = 0 \\ i\delta & \text{if } j = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \end{cases}$$

- Simple cases: either prefix string is empty, so cost is some number of insertions (gap cost)
- Otherwise: best of the 3 cases
- Answer for the two complete strings is $Opt(m, n)$

Bottom-up Implementation

```
Edit-Distance(X, Y, δ, α)
  m = len(X), n = len(Y)
  for i = 0 to m
     Opt[i,0] = i * δ
  for j = 0 to n
     Opt[0,j] = j * δ
  for i = 1 to m
     for j = 1 to n
        Opt[i,j] = min(  α_{x_i y_j} + Opt[i-1,j-1],
                         δ + Opt[i-1,j],
                         δ + Opt[i,j-1] )
  return Opt[m,n]
```

Time complexity?

Constant time to file each cell in the table.

So $\Theta(n \cdot m)$.

Space complexity the same.

Small Example

Y = "army" → X = "air"

$\alpha_{pq} = 2$ and δ=1

		a	r	m	y
	0	1	2	3	4
a	1	0	1	2	3
i	2				
r	3				

$$\min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases}$$

Small Example

Y = “army” → X = “air”

$\alpha_{pq} = 2$ and δ=1

		Y				
			a	r	m	y
		0	1	2	3	4
X	a	1	0	1	2	3
	i	2	???			
	r	3				

$$\min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases}$$

Value for next cell i=2, j=1?

Prefix strings are X_2=“ai” and Y_1=“a”

1. $(x_2, y_1) \in M$. The cost will be: mismatch cost + Opt(1,0) = 2+1 = 3

X_2	a	i
Y_1		a

2. x_i (“i”) aligns after Y_1
The cost will be:
gap cost + Opt(1,1) = 1+0 = 1

X_2	a	i
Y_1	a	-

3. y_j (“a”) aligns after X_2
The cost will be:
gap cost + Opt(2,0) = 1+2 = 3

X_2	a	i	-
Y_1			a

Complete Solution

army → air

$\boldsymbol{\alpha_{pq}} = 2$ and $\boldsymbol{\delta}$=1

Y

X

		a	r	m	y
	0	1	2	3	4
a	1	0	1	2	3
i	2	1	2	3	4
r	3	2	1	2	3

Backtrack to Get the Edit

$\alpha_{pq} = 2$ and $\boldsymbol{\delta}$=1

Y

		a	r	m	y
	0	1	2	3	4
a	1	0	1	2	3
i	2	1	2	3	4
r	3	2	1	2	3

X

x_1	x_2	x_3	x_4	x_5
a	i	r	-	-
a	-	r	m	y
y_1		y_2	y_3	y_4

army → air

1. y != r and we came from left:
   Delete y
   How do we know we didn't come diagonally?
2. m != r and we came from left:
   Delete m
3. r == r
   Keep r and move diagonally
   (x_3, y_2) added to alignment M
4. a != i and we came from above:
   Insert i
5. a == a
   Keep a and move diagonally
   (x_1, y_1) added to alignment M

Summary on Sequence Alignment

- Model for meaning of “similar”
  - Alignment, with flexible costs for mismatches and gaps
- Recursive structure based on whether last items in each sequence are part of the “match” (perhaps with mismatch cost)
  - Remember, if not matched, part of a gap (insertion)
  - 3 cases
- Table stores partial solutions for prefixes of both strings
- Time- and space-complexity is $\Theta(n \cdot m)$
- Want to learn more?
  - Proof of correctness is shortest-path induction proof for a graph model of table
  - Another algorithm (Hirschberg) reduces space complexity to $\Theta(n + m)$

Larger Example

		s	i	m	i	l	a	r	i	t	y
	0	1	2	3	4	5	6	7	8	9	10
i	1	2	1	2	3	4	5	6	7	8	9
d	2	3	2	3	4	5	6	7	8	9	10
e	3	4	3	4	5	6	7	8	9	10	11
n	4	5	4	5	6	7	8	9	10	11	12
t	5	6	5	6	7	8	9	10	11	10	11
i	6	7	6	7	6	7	8	9	10	11	12
t	7	8	7	8	7	8	9	10	11	10	11
y	8	9	8	9	8	9	10	11	12	11	10

identity and ***similarity***

$\alpha_{pq} = 2$ and $\boldsymbol{\delta}$=1

- Alignment found is:

```
  i   d e n t i t y
s i m i l a r i t y
```

- Note (t,r) "matched" but we moved diagonally and incurred a cost
- Same for (n,a), (e,l), (d,i)

From online calculator at:
http://www.let.rug.nl/~kleiweg/lev/
Try it!

Longest Common Subsequence

Given two sequences X and Y, find the length of their longest common subsequence

Example:

X = ATCTGAT
Y = TGCATA
LCS = TCTA

X = AT C TGAT
Y = TGCAT A

Note this is an alignment where matched items are equal

Brute force: Compare every subsequence of X with Y: $\Omega(2^n)$

Applications other than bioinformatics?
Of course, Including version control!
http://cbx33.github.io/gitt/afterhours3-1.html

Dynamic Programming

- Requires Optimal Substructure
  – Solution to larger problem contains the solutions to smaller ones
- Avoid extra work due to overlapping subproblems
- Idea:
  1. Identify the recursive structure of the problem
     - What is the "last thing" done?
  2. Save the solution to each subproblem in memory
  3. Select a good order for solving subproblems
     - "Top Down": Solve each recursively
     - "Bottom Up": Iteratively solve smallest to largest

Subproblems will be defined in same way as for edit distance: prefix strings

1. Identify Recursive Structure

Let $LCS(i, j)$ = length of the LCS for the first i characters of X, first j character of Y

Find $LCS(i, j)$:

Case 1: $X[i] = Y[j]$

X = ATCTGCGT
Y = TGCATAT

$LCS(i, j) = LCS(i-1, j-1) + 1$

Case 2: $X[i] \neq Y[j]$

X = ATCTGCGA
Y = TGCATAC

$LCS(i, j) = LCS(i, j-1)$

X = ATCTGCGA
Y = TGCATAC

$LCS(i, j) = LCS(i-1, j)$

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

Dynamic Programming

- Requires Optimal Substructure
  – Solution to larger problem contains the solutions to smaller ones
- Avoid extra work due to overlapping subproblems
- Idea:
  1. Identify the recursive structure of the problem
     - What is the “last thing” done?
  2. Save the solution to each subproblem in memory
  3. Select a good order for solving subproblems
     - “Top Down”: Solve each recursively
     - “Bottom Up”: Iteratively solve smallest to largest

1. Identify Recursive Structure

Let $LCS(i,j)$ = length of the LCS for the first i characters of X, first j character of Y

Find $LCS(i,j)$:

Case 1: $X[i] = Y[j]$

X = $ATCTGCGT$

Y = $TGCATAT$

$$LCS(i,j) = LCS(i-1,j-1) + 1$$

Case 2: $X[i] \neq Y[j]$

X = $ATCTGCGA$

Y = $TGCATAC$

$$LCS(i,j) = LCS(i,j-1)$$

X = $ATCTGCGA$

Y = $TGCATAC$

$$LCS(i,j) = LCS(i-1,j)$$

Read from M[i,j] if present

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

Save to M[i,j]

Dynamic Programming

- Requires Optimal Substructure
  - Solution to larger problem contains the solutions to smaller ones
- Avoid extra work due to overlapping subproblems
- Idea:
  1. Identify the recursive structure of the problem
     - What is the "last thing" done?
  2. Save the solution to each subproblem in memory
  3. Select a good order for solving subproblems
     - "Top Down": Solve each recursively
     - "Bottom Up": Iteratively solve smallest to largest

3. Solve in a Good Order

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

$Y =$ \ $X =$		0	A 1	T 2	C 3	T 4	G 5	A 6	T 7
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
T	1	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**
G	2	**0**	**0**	**1**	**1**	**1**	**2**	**2**	**2**
C	3	**0**	**0**	**1**	**2**	**2**	**2**	**2**	**2**
A	4	**0**	**1**	**1**	**2**	**2**	**2**	**3**	**3**
T	5	**0**	**1**	**2**	**2**	**3**	**3**	**3**	**4**
A	6	**0**	**1**	**2**	**2**	**3**	**3**	**4**	**4**

To fill in cell (i,j) we need cells $(i-1, j-1), (i-1, j), (i, j-1)$
Fill from Top->Bottom, Left->Right (with any preference)

LCS Length Algorithm

```
LCS-Length(X, Y) // Y for M's rows, X for its columns
1. n = length(X)  // get the # of symbols in X
2. m  = length(Y) // get the # of symbols in Y
3. for i = 1 to m    M[i,0] = 0   // special case: Y_0
4. for j = 1 to n    M[0,j] = 0   // special case: X_0
5. for i = 1 to m                 // for all Y_i
6.      for j = 1 to n                 // for all X_j
7.           if ( X[i] == Y[j] )
8.                M[i,j] = M[i-1,j-1] + 1
9.           else M[i,j] = max( M[i-1,j], M[i,j-1] )
10. return M[m,n]   // return LCS length for Y and X
```

Run Time?

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

$Y =$	$X =$	0	A 1	T 2	C 3	T 4	G 5	A 6	T 7
	0	0	0	0	0	0	0	0	0
T	1	0	0	1	1	1	1	1	1
G	2	0	0	1	1	1	2	2	2
C	3	0	0	1	2	2	2	2	2
A	4	0	1	1	2	2	2	3	3
T	5	0	1	2	2	3	3	3	4
A	6	0	1	2	2	3	3	4	4

Run Time: $\Theta(n \cdot m)$ (for $|X| = n$, $|Y| = m$)

Reconstructing the LCS

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

$Y =$ \ $X =$		0	A 1	T 2	C 3	$\boxed{T}$ 4	$\boxed{G}$ 5	$\boxed{A}$ 6	$\boxed{T}$ 7
	0	0	0	0	0	0	0	0	0
$\boxed{T}$	1	0	0	1	1	1	1	1	1
$\boxed{G}$	2	0	0	1	1	1	2	2	2
C	3	0	0	1	2	2	2	2	2
$\boxed{A}$	4	0	1	1	2	2	2	3	3
$\boxed{T}$	5	0	1	2	2	3	3	3	4
A	6	0	1	2	2	3	3	4	4

Start from bottom right,

if symbols matched, print that symbol then go diagonally

else go to largest adjacent

Reconstructing the LCS

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

$Y =$ \ $X =$	0	A 1	$\boxed{T}$ 2	$\boxed{C}$ 3	T 4	G 5	$\boxed{A}$ 6	$\boxed{T}$ 7
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
$\boxed{T}$ 1	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**
G 2	**0**	**0**	**1**	**1**	**1**	**2**	**2**	**2**
$\boxed{C}$ 3	**0**	**0**	**1**	**2**	**2**	**2**	**2**	**2**
$\boxed{A}$ 4	**0**	**1**	**1**	**2**	**2**	**2**	**3**	**3**
$\boxed{T}$ 5	**0**	**1**	**2**	**2**	**3**	**3**	**3**	**4**
A 6	**0**	**1**	**2**	**2**	**3**	**3**	**4**	**4**

Start from bottom right,

if symbols matched, print that symbol then go diagonally

else go to largest adjacent

Reconstructing the LCS

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

	$X =$	0	A 1	$\boxed{T}$ 2	$\boxed{C}$ 3	$\boxed{T}$ 4	G 5	$\boxed{A}$ 6	T 7
$Y =$	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
$\boxed{T}$	1	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**
G	2	**0**	**0**	**1**	**1**	**1**	**2**	**2**	**2**
$\boxed{C}$	3	**0**	**0**	**1**	**2**	**2**	**2**	**2**	**2**
A	4	**0**	**1**	**1**	**2**	**2**	**2**	**3**	**3**
$\boxed{T}$	5	**0**	**1**	**2**	**2**	**3**	**3**	**3**	**4**
$\boxed{A}$	6	**0**	**1**	**2**	**2**	**3**	**3**	**4**	**4**

Start from bottom right,

if symbols matched, print that symbol then go diagonally

else go to largest adjacent

Top-Down Solution with Memoization

We need two functions; one will be recursive.

```
LCS-Length(X, Y) // Y is M's cols.
1. n = length(X)
2. m  = length(Y)
3. Create table M[m,n]
4. Assign -1 to all cells M[i,j]
// get value for entire sequences
5. return LCS-recur(X, Y, M, m, n)
```

```
LCS-recur(X, Y, M, i, j)
1. if (i == 0 || j == 0) return 0
// have we already calculated this subproblem?
2. if (M[i,j] != -1) return M[i,j]
3. if ( X[i] == Y[j] )
4.   M[i,j] = LCS-recur(X, Y, M, i-1, j-1) + 1
5. else
6.    M[i,j] = max( LCS-recur(X, Y, M, i-1, j),
                     LCS-recur(X, Y, M, i, j-1) )
7. return M[i,j]
```

Another LCS Example

Let’s 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

	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 M[5,4]

Note: In this example, X is M's rows, Y is the columns. Opposite from earlier example.

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 $\quad$ M[i,0] = 0
for j = 1 to n $\quad$ M[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])
M[i,j] = M[i-1,j-1] + 1
else M[i,j] = max(M[i-1,j], M[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])
M[i,j] = M[i-1,j-1] + 1
else M[i,j] = max(M[i-1,j], M[i,j-1])

LCS Example (4)

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**	
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X[i] == Y[j])
M[i,j] = M[i-1,j-1] + 1
else M[i,j] = max(M[i-1,j], M[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**	$\rightarrow$ **1**
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X[i] == Y[j])
M[i,j] = M[i-1,j-1] + 1
else M[i,j] = max(M[i-1,j], M[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**					

if (X[i] == Y[j])
M[i,j] = M[i-1,j-1] + 1
else M[i,j] = max(M[i-1,j], M[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])
M[i,j] = M[i-1,j-1] + 1
else M[i,j] = max(M[i-1,j], M[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])
    M[i,j] = M[i-1,j-1] + 1
else M[i,j] = max(M[i-1,j], M[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])
M[i,j] = M[i-1,j-1] + 1
else M[i,j] = max(M[i-1,j], M[i,j-1])

LCS Example (11)

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**		
4	**B**	**0**					

if (X[i] == Y[j])
  M[i,j] = M[i-1,j-1] + 1
else M[i,j] = max(M[i-1,j], M[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])
M[i,j] = M[i-1,j-1] + 1
else M[i,j] = max(M[i-1,j], M[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])
    M[i,j] = M[i-1,j-1] + 1
else M[i,j] = max(M[i-1,j], M[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])
M[i,j] = M[i-1,j-1] + 1
else M[i,j] = max(M[i-1,j], M[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])
    M[i,j] = M[i-1,j-1] + 1
else M[i,j] = max(M[i-1,j], M[i,j-1])

Practice!

- X = [G, D, V, E, G, T, A] and
  Y = [G, V, C, E, K, S, T]
- Find the LCS, show the table M
- Can you reconstruct the LCS from M?