COMP 355
Advanced Algorithms

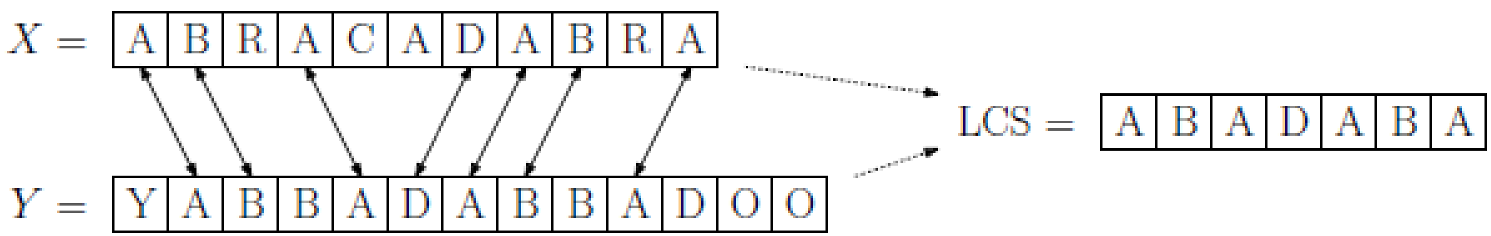
Dynamic Programming:
Finish LCS & Sequence Alignment
Section 6.6-6.7(KT)

Rhodes College

—1848—

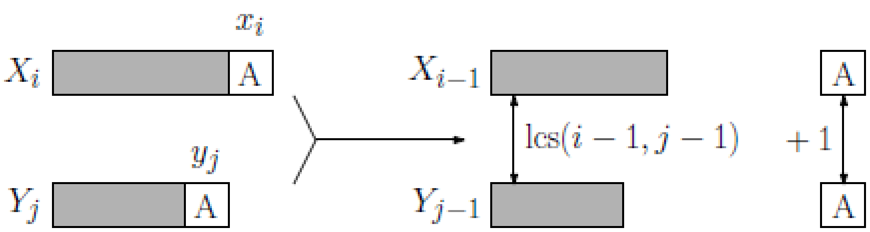
Longest Common Subsequence (LCS)

Given two sequences $X = <x_1, x_2, . . . , x_m>$ and $Z = <z_1, z_2, . . . , z_k>$, we say that Z is a subsequence of X if there is a strictly increasing sequence of k indices $<i_1, i_2, . . . , i_k>$ $(1 \leq i_1 < i_2 <. . . < i_k \leq n)$ such that $Z = <x_{i1} , x_{i2} , . . . , x_{ik} >$.
For example, let X = <ABRACADABRA> and let Z = <AADAA>, then Z is a subsequence of X.

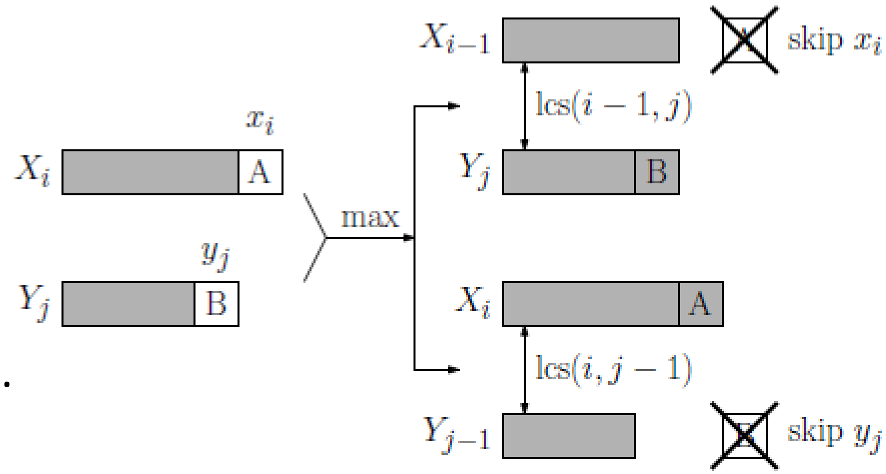


LCS Problem: Given two sequences $X = <x_1, . . . , x_m>$ and $Y = <y_1, . . . , y_n>$ determine the length of their longest common subsequence, and more generally the sequence itself.

Recursive Formulation of LCS



LCS of two strings whose last characters are equal.
if ($x_i = y_j$) then lcs(i, j) = lcs(i − 1, j − 1) + 1



The possible cases in the DP formulation of LCS.
if (x_i != y_j) then lcs(i, j) = max(lcs(i − 1, j), lcs(i, j − 1))

$$\mathrm{lcs}(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \mathrm{lcs}(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(\mathrm{lcs}(i-1, j), \mathrm{lcs}(i, j-1)) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Memoized Implemention

Memoized Longest Common Subsequence

```
memoized-lcs(i,j) {
    if (lcs[i,j] has not yet been computed) {
        if (i == 0 || j == 0)                    // basis case
            lcs[i,j] = 0
        else if (x[i] == y[j])                   // last characters match
            lcs[i,j] = memoized-lcs(i-1, j-1) + 1
        else                                     // last chars don't match
            lcs[i,j] = max(memoized-lcs(i-1, j), memoized-lcs(i, j-1))
    }
    return lcs[i,j]                              // return stored value
}
```

- The running time of the memoized version is O(mn).
- Observe that there are m+1 possible values for i, and n + 1 possible values for j.
- Each time we call memoized-lcs(i, j), if already computed - returns in O(1) time.
- Each call to memoized-lcs(i, j) generates a constant number of additional calls.
- Therefore, the time needed to compute the initial value of any entry is O(1), and all subsequent calls with the same arguments is O(1).
- Total running time is equal to the number of entries computed, which is O((m + 1)(n + 1)) = O(mn).

Bottom-up implementation

Bottom-up Longest Common Subsequence

```
bottom-up-lcs() {
    lcs = new array [0..m, 0..n]
    for (i = 0 to m) lcs[i,0] = 0          // basis cases
    for (j = 0 to n) lcs[0,j] = 0
    for (i = 1 to m) {                      // fill rest of table
        for (j = 1 to n) {
            if (x[i] == y[j])               // take x[i] (= y[j]) for LCS
                lcs[i,j] = lcs[i-1, j-1] + 1
            else
                lcs[i,j] = max(lcs[i-1, j], lcs[i, j-1])
        }
    }
    return lcs[m, n]                         // final lcs length
}
```

- Running time: O(mn)
- Space: O(mn)

Adding Hints to Reconstruct LCS

add$_{XY}$: Add $x_i (= y_j)$ to the LCS ('↖') and continue with lcs$[i-1, j-1]$

skip$_X$: Do not include x_i to the LCS ('↑') and continue with lcs$[i-1, j]$

skip$_Y$: Do not include y_j to the LCS ('←') and continue with lcs$[i, j-1]$

Bottom-up Longest Common Subsequence with Hints

```
bottom-up-lcs-with-hints() {
    lcs = new array [0..m, 0..n]                        // stores lcs lengths
    h = new array [0..m, 0..n]                          // stores hints
    for (i = 0 to m) { lcs[i,0] = 0;  h[i,0] = skipX }
    for (j = 0 to n) { lcs[0,j] = 0;  h[0,j] = skipY }
    for (i = 1 to m) {
        for (j = 1 to n) {
            if (x[i] == y[j])
                { lcs[i,j] = lcs[i-1, j-1] + 1;  h[i,j] = addXY }
            else if (lcs[i-1, j] >= lcs[i, j-1])
                { lcs[i,j] = lcs[i-1, j];  h[i,j] = skipX }
            else
                { lcs[i,j] = lcs[i, j-1];  h[i,j] = skipY }
        }
    }
    return lcs[m, n]                                    // final lcs length
}
```

Extracting the LCS


```
get-lcs-sequence() {
    LCS = new empty character sequence
    i = m; j = n                                   // start at lower right
    while(i != 0 or j != 0)                        // loop until upper left
        switch h[i,j]
            case addXY:                            // add x[i] (= y[j])
                prepend x[i] (or equivalently y[j]) to front of LCS
                i--;  j--;    break
            case skipX: i--;  break                // skip x[i]
            case skipY: j--;  break                // skip y[j]
    return LCS
}
```

LCS Example

LCS length

		0	1	2	3	4 = n
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
$m = 5$	B	0	1	2	2	3

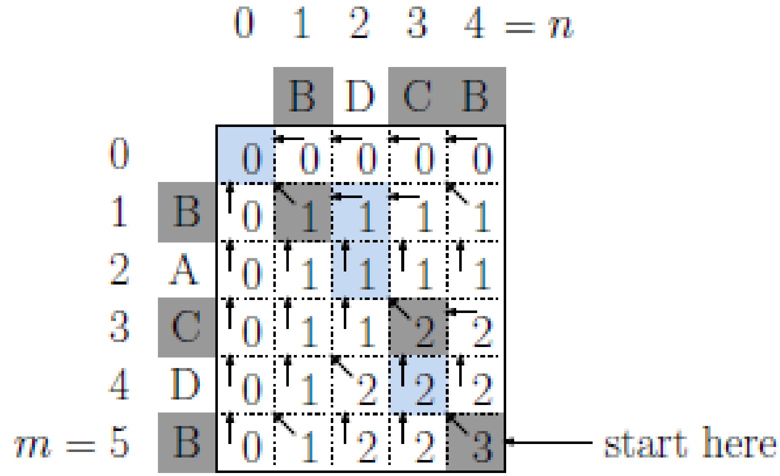
(a)

$X = \langle BACDB \rangle$

$Y = \langle BDCB \rangle$

$LCS = \langle BCB \rangle$

. . . with hints



(b)

Contents of the lcs array for the input sequences X = <BACDB> and Y = <BCDB>. The numeric table entries are the values of lcs[i, j] and the arrow entries are used in the extraction of the sequence.

How similar are two strings?

Spell correction

- The user typed "graffe"

Which is closest?

- graf
- graft
- grail
- giraffe

Computational Biology

- Align two sequences of nucleotides

```
AGGCTATCACCTGACCTCCAGGCCGATGCCC
TAGCTATCACGACCGCGGTCGATTTGCCCGAC
```

- Resulting alignment:

```
-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC
```

Also for Machine Translation, Information Extraction, Speech Recognition

Minimum Edit Distance

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

Editing Operations

- Insertion
- Deletion
- Substitution

- If each operation has cost of 1
  - Distance between these is 5
- If substitutions cost 2 (Levenshtein)
  - Distance between them is 8

Min Edit Distance Algorithm

```
"Calculate Levenshtein edit distance for strings s1 and s2."
    len1 = len(s1) # vertically
    len2 = len(s2) # horizontally
    # Allocate the table
    table = [None]*(len2+1)
    for i in range(len2+1): table[i] = [0]*(len1+1)
    # Initialize the table
    for i in range(1, len2+1): table[i][0] = i
    for i in range(1, len1+1): table[0][i] = i
    # Do dynamic programming
    for i in range(1,len2+1):
        for j in range(1,len1+1):
            if s1[j-1] == s2[i-1]:
                d = 0
            else:
                d = 2
            table[i][j] = min(table[i-1][j-1] + d,
                              table[i-1][j]+1,
                              table[i][j-1]+1)
```

The Edit Distance Table

	#	E	X	E	C	U	T	I	O	N
#	0	1	2	3	4	5	6	7	8	9
I	1									
N	2									
T	3									
E	4									
N	5									
T	6									
I	7									
O	8									
N	9									

The Edit Distance Table

	#	E	X	E	C	U	T	I	O	N
#	**0**	1	2	3	4	5	6	7	8	9
I	1									
N	2									
T	3									
E	4									
N	5									
T	6									
I	7									
O	8									
N	9									

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$

Edit Distance

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$

	#	E	X	E	C	U	T	I	O	N
#	**0**	1	2	3	4	5	6	7	8	9
I	1	2								
N	2									
T	3									
E	4									
N	5									
T	6									
I	7									
O	8									
N	9									

The Edit Distance Table

	#	E	X	E	C	U	T	I	O	N
#	**0**	1	2	3	4	5	6	7	8	9
I	1	2	3	4	5	6	7	6	7	8
N	2	3	4	5	6	7	8	7	8	7
T	3	4	5	6	7	8	7	8	9	8
E	4	3	4	5	6	7	8	9	10	9
N	5	4	5	6	7	8	9	10	11	10
T	6	5	6	7	8	9	8	9	10	11
I	7	6	7	8	9	10	9	8	9	10
O	8	7	8	9	10	11	10	9	8	9
N	9	8	9	10	11	12	11	10	9	**8**

Practice

Find the Levenshtein minimum edit distance of the words **mean** and **name**.

Hint: Levenshtein means that you should assume a substitution (mismatch) penalty of 2, and a indel (gap) penalty of 1.