COP 3530: Computer Science III
Summer 2005

Dynamic Programming – Part 3

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
CSB 242, (407)823-2790

Course Webpage:
http://www.cs.ucf.edu/courses/cop3530/sum2005

School of Computer Science
University of Central Florida

Longest Common Subsequence Problem

- Before we look at graph applications for dynamic programming, we'll consider the problem of determining how close two character strings are to one another.
- For example, a spell checker might compare a text string created on a word processor with pattern strings from a stored dictionary. If there is no exact match between the text string and any pattern string, then the spell checker offers several alternative pattern strings that are fairly close, in some sense, to the text string.
- Another example would be a forensic scientist who needs to compare two DNA strings to measure how closely they match.

Measuring the Closeness of Two Strings

- The closeness of two strings can be measured in several ways.
- One method is to use the edit distance, which is commonly used by search engines to find approximate matchings for a user-entered text string for which an exact match cannot be found. The edit distance is also used by spell checkers.
- Roughly speaking, the edit distance between two strings is the minimum number of changes that need to be made (adding, deleting, or changing characters) to transform one string into another.

Measuring the Closeness of Two Strings (cont.)

- Another measure of closeness is the longest common subsequence (LCS) contained in a text string and a particular pattern string.
- Computing either the LCS or the edit distance is an optimization problem that satisfies the Principle of Optimality and can thus be solved using dynamic programming.
- However, the solution to the LCS problem is easier to understand than is the solution to the edit distance problem, because it has a simpler recurrence relation.

The Longest Common Subsequence Problem

- Suppose $T = T_0T_1\ldots T_{n-1}$ is a text string that we want to compare to some pattern string $P = P_0P_1\ldots P_{m-1}$, where we assume that the characters in each string are drawn from some fixed alphabet *A.*
- A subsequence of T is a string of the form $T_{i1}T_{i2}\ldots T_{ik}$ where $0 \leq i1 < i2 < \ldots < ik \leq n-1$.
- Note that a substring of T is a special case of a subsequence of T in which the subscripts making up the subsequence increase by 1.
- For example, consider the pattern string “Cincinnati” and the text string “Cincinatti” The LCS has length 9, whereas the edit distance is two, since it takes two changes to transform the text string into the pattern string.

The Longest Common Subsequence Problem (cont.)

- The longest common subsequence problem is:
- Given two sequences $T = T_0T_1\ldots T_{n-1}$ and $P = P_0P_1\ldots P_{m-1}$, assuming that sequences are stored in arrays `T[0:n-1]` and `P[0:m-1]`, respectively.
- For integers i and j, the `LCS[i,j]` is defined to be the length of the longest common subsequence of the substrings `T[0:i-1]` and `P[0,j-1]`, so that `LCS[n,m]` is the length of the longest common subsequence of T and P. `LCS[i,j] = 0` if either `i = 0` or `j = 0` (corresponding to an empty string.) These conditions yield our recurrence for the LCS problem which is:

$$LCS[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS[i-1, j-1] + 1 & \text{if } T[i-1] = P[j-1] \\ \max\{LCS[i, j-1], LCS[i-1, j]\} & \text{if } T[i-1] \neq P[j-1] \end{cases}$$

Verification of the Recurrence

- To verify the recurrence given on the previous page, note first that if `T[i-1]≠ P[j-1]`, then a longest common subsequence of `T[0:i-1]` and `P[0:j-1]` might end in `T[i-1]` or `P[j-1]`, but certainly not both.
  - In other words, if `T[i-1]≠ P[j-1]`, then a longest common subsequence of `T[0:i-1]` and `P[0:j-1]` must be drawn from either the pair `T[0:i-2]` and `P[0:j-1]`or from the pair `T[0:i-1]` and `P[0:j-2]`.
- In addition, such a longest common subsequence must be a longest common subsequence of the pair of substrings from which it is drawn (that is, the principle of optimality holds). This verifies that:

$$LCS[i, j] = \max\{LCS[i, j-1], LCS[i-1, j]\} \quad \text{if } T[i-1] \neq P[j-1]$$

Verification of the Recurrence (cont.)

- On the other hand, if `T[i-1]= P[j-1] = C`, then a longest common subsequence must end at either `T[i-1]` in `T[0:i-1]` or at `P[j-1]` in `P[0:j-1]` or both; otherwise, by adding the common value *C* to a given subsequence, we would increase the length of the subsequence by 1. Also, if the last term of a longest common subsequence ends at an index *k* < *i-1* in `T[0:i-1]` (so that `T[k] = C`), then clearly we achieve an equivalent longest common subsequence by swapping `T[i-1]` for `T[k]` in the subsequence.
- By a similar argument involving `P[0:j-1]`, when `T[i-1]= P[j-1]`, we can assume without loss of generality that a longest common subsequence in `T[0:i-1]`and `P[0:j-1]`ends at `T[i-1]` and `P[j-1]= P[j-1]`. However, then removing these endpoints from the subsequence clearly must result in a longest common subsequence in `T[0:i-2]`and `P[0:j-2]`respectively (the principle of optimality again holds). So it follows that:

$$LCS[i,j] = LCS[i-1,j-1]+1 \qquad \text{if } T[i-1] = P[j-1]$$

Dynamic Programming Algorithm for LCS Problem

```
Algorithm LongestCommonSubSequence(T[0:n-1], P[0:m-1], LCS[0:n,0:m])
//Determines LCS for text and pattern strings T and P using dynamic programming
//Input: Strings T[0:n-1], P[0:m-1]
//Output: LCS[0:n,0:m] where LCS[i,j] is the length of the longest common subseq
for i ← 0 to n do  //initialize boundary conditions
   LCS[i, 0] ← 0
for j ← 0 to m do  //initialize boundary conditions
   LCS[0,j] ← 0
for i ← 1 to n do  //compute the row index by i of LCS[0:n,0:m]
      for j ← 1 to m do  //compute LCS[i,j] using recurrence
          if T[i-1] = P[j-1]
                LCS[i,j] ←  LCS[i-1, j-1] + 1
           else  LCS[i,j] ← max{ LCS[i, j-1], LCS[i-1, j]}
return LCS[0:n,0:m]
```


Evaluation of the Algorithm

- It is easy to see that the algorithm on the previous page has a complexity of O(nm), which is a rather dramatic improvement over the exponential complexity $\Theta(2^n m)$ of the brute force algorithm.
  - The brute force algorithm would examine each of the 2^n subsequences of `T[0:n-1]`and determine the longest subsequence that also occurs in `P[0:m-1]`.
- The following few pages illustrate this algorithm using T[0:7] = “usbeeune” and P[0:10] = “subsequence”. The LCS matrix has dimensions LCS[0:8, 0:11].

EXAMPLE – Initial Solution Matrix

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0											
2	0											
3	0											
4	0											
5	0											
6	0											
7	0											
8	0											

EXAMPLE – Calculations For Row 1

- The algorithm fills the LCS matrix row by row.
- Some of the calculations for the first row are:
- LCS[1,1]: T[0] ≠ P[0] so LCS[1,1] = max{ LCS[1,0], LCS[0,1]} = 0
- LCS[1,2]: T[0] = P[1] so LCS[1,2] = LCS[0,1] + 1 = 0 + 1 = 1
- LCS[1,3]: T[0] ≠ P[2] so LCS[1,3] = max{ LCS[1,2], LCS[0,3]} = 1
- LCS[1,4]: T[0] ≠ P[3] so LCS[1,4] = max{ LCS[1,3], LCS[0,4]} = 1
- LCS[1,5]: T[0] ≠ P[4] so LCS[1,5] = max{ LCS[1,4], LCS[0,5]} = 1
- LCS[1,6]: T[0] ≠ P[5] so LCS[1,6] = max{ LCS[1,5], LCS[0,6]} = 1

LCS Matrix – Row 1 completed

	0	1/s	2/u	3/b	4/s	5/e	6/q	7/u	8/e	9/n	10/c	11e/
0	0	0	0	0	0	0	0	0	0	0	0	0
1/u	0	0	1	1	1	1	1	1	1	1	1	1
2/s	0											
3/b	0											
4/e	0											
5/e	0											
6/u	0											
7/n	0											
8/e	0											

EXAMPLE – Calculations for Row 2

- Some of the calculations for the second row are:
- LCS[2,1]: T[1] = P[0] so LCS[2,1] = LCS[1,0] + 1 = 0 + 1 = 1
- LCS[2,2]: T[1] ≠ P[1] so LCS[2,2] = max{ LCS[2,1], LCS[1,2]} = 1
- LCS[2,3]: T[1] ≠ P[2] so LCS[2,3] = max{ LCS[2,2], LCS[1,3]} = 1
- LCS[2,4]: T[1] = P[3] so LCS[2,4] = LCS[1,3] + 1 = 2
- LCS[2,5]: T[1] ≠ P[4] so LCS[2,5] = max{ LCS[2,4], LCS[1,5]} = 2
- LCS[2,6]: T[1] ≠ P[5] so LCS[2,6] = max{ LCS[2,5], LCS[1,6]} = 2

LCS Matrix – Row 2 completed

	0	1/s	2/u	3/b	4/s	5/e	6/q	7/u	8/e	9/n	10/c	11/e
0	0	0	0	0	0	0	0	0	0	0	0	0
1/u	0	0	1	1	1	1	1	1	1	1	1	1
2/s	0	1	1	1	2	2	2	2	2	2	2	2
3/b	0											
4/e	0											
5/e	0											
6/u	0											
7/n	0											
8/e	0											

EXAMPLE – Calculations for Row 3

- Some of the calculations for the third row are:
- LCS[3,1]: T[2] ≠ P[0] so LCS[3,1] = max{ LCS[3,0], LCS[2,1]} = 1
- LCS[3,2]: T[2] ≠ P[1] so LCS[3,2] = max{ LCS[3,1], LCS[2,2]} = 1
- LCS[3,3]: T[2] = P[2] so LCS[3,3] = LCS[2,2] + 1 = 2
- LCS[3,4]: T[2] ≠ P[3] so LCS[3,4] = max{ LCS[3,3], LCS[2,4]} = 2
- LCS[3,5]: T[2] ≠ P[4] so LCS[3,5] = max{ LCS[3,4], LCS[2,5]} = 2
- LCS[3,6]: T[2] ≠ P[5] so LCS[3,6] = max{ LCS[3,5], LCS[1,6]} = 2

LCS Matrix – Row 3 completed

	0	1/s	2/u	3/b	4/s	5/e	6/q	7/u	8/e	9/n	10/c	11/e
0	0	0	0	0	0	0	0	0	0	0	0	0
1/u	0	0	1	1	1	1	1	1	1	1	1	1
2/s	0	1	1	1	2	2	2	2	2	2	2	2
3/b	0	1	1	2	2	2	2	2	2	2	2	2
4/e	0											
5/e	0											
6/u	0											
7/n	0											
8/e	0											

EXAMPLE – Calculations for Row 4

- Some of the calculations for the fourth row are:
- LCS[4,1]: T[3] ≠ P[0] so LCS[4,1] = max{ LCS[4,0], LCS[3,1]} = 1
- LCS[4,2]: T[3] ≠ P[1] so LCS[4,2] = max{ LCS[4,1], LCS[3,2]} = 1
- LCS[4,3]: T[3] ≠ P[2] so LCS[4,3] = max{ LCS[4,2], LCS[3,3]} = 2
- LCS[4,4]: T[3] ≠ P[3] so LCS[4,4] = max{ LCS[4,3], LCS[3,4]} = 2
- LCS[4,5]: T[3] = P[4] so LCS[4,5] = LCS[3,4] + 1 = 3
- LCS[4,6]: T[3] ≠ P[5] so LCS[4,6] = max{ LCS[4,5], LCS[3,6]} = 3
- LCS[4,7]: T[3] ≠ P[6] so LCS[4,7] = max{ LCS[4,6], LCS[3,7]} = 3
- LCS[4,8]: T[3] = P[7] so LCS[4,8] = LCS[3,7] + 1 = 3

LCS Matrix – Row 4 completed

	0	1/s	2/u	3/b	4/s	5/e	6/q	7/u	8/e	9/n	10/c	11/e
0	0	0	0	0	0	0	0	0	0	0	0	0
1/u	0	0	1	1	1	1	1	1	1	1	1	1
2/s	0	1	1	1	2	2	2	2	2	2	2	2
3/b	0	1	1	2	2	2	2	2	2	2	2	2
4/e	0	1	1	2	2	3	3	3	3	3	3	3
5/e	0											
6/u	0											
7/n	0											
8/e	0											

EXAMPLE – Calculations for Row 5

- Some of the calculations for the fifth row are:
- LCS[5,1]: T[4] ≠ P[0] so LCS[5,1] = max{ LCS[5,0], LCS[4,1]} = 1
- LCS[5,2]: T[4] ≠ P[1] so LCS[5,2] = max{ LCS[5,1], LCS[4,2]} = 1
- LCS[5,3]: T[4] ≠ P[2] so LCS[5,3] = max{ LCS[5,2], LCS[4,3]} = 2
- LCS[5,4]: T[4] ≠ P[3] so LCS[5,4] = max{ LCS[5,3], LCS[4,4]} = 2
- LCS[5,5]: T[4] = P[4] so LCS[5,5] = LCS[4,4] + 1 = 3
- LCS[5,6]: T[4] ≠ P[5] so LCS[5,6] = max{ LCS[5,5], LCS[4,6]} = 3
- LCS[5,7]: T[4] ≠ P[6] so LCS[5,7] = max{ LCS[5,6], LCS[4,7]} = 3
- LCS[5,8]: T[4] = P[7] so LCS[5,8] = LCS[4,7] + 1 = 4
- LCS[5,9]: T[4] ≠ P[8] so LCS[5,9] = max{ LCS[5,8], LCS[4,9]} = 4
- LCS[5,10]: T[4] ≠ P[9] so LCS[5,10] = max{ LCS[5,9], LCS[4,10]} = 4
- LCS[5,11]: T[4] = P[10] so LCS[5,11] = LCS[4,10] + 1 = 4

LCS Matrix – Row 5 completed

	0	1/s	2/u	3/b	4/s	5/e	6/q	7/u	8/e	9/n	10/c	11/e
0	0	0	0	0	0	0	0	0	0	0	0	0
1/u	0	0	1	1	1	1	1	1	1	1	1	1
2/s	0	1	1	1	2	2	2	2	2	2	2	2
3/b	0	1	1	2	2	2	2	2	2	2	2	2
4/e	0	1	1	2	2	3	3	3	3	3	3	3
5/e	0	1	1	2	2	3	3	3	4	4	4	4
6/u	0											
7/n	0											
8/e	0											

EXAMPLE – Calculations for Row 6

- Some of the calculations for the sixth row are:
- LCS[6,1]: T[5] ≠ P[0] so LCS[6,1] = max{ LCS[6,0], LCS[5,1]} = 1
- LCS[6,2]: T[5] = P[1] so LCS[6,2] = LCS[5,1] + 1 = 2
- LCS[6,3]: T[5] ≠ P[2] so LCS[6,3] = max{ LCS[6,2], LCS[5,3]} = 2
- LCS[6,4]: T[5] ≠ P[3] so LCS[6,4] = max{ LCS[6,3], LCS[5,4]} = 2
- LCS[6,5]: T[5] ≠ P[4] so LCS[6,5] = max{ LCS[6,4], LCS[5,5]} = 3
- LCS[6,6]: T[5] ≠ P[5] so LCS[6,6] = max{ LCS[6,5], LCS[5,6]} = 3
- LCS[6,7]: T[5] = P[6] so LCS[6,7] = LCS[5,6] + 1 = 4
- LCS[6,8]: T[5] ≠ P[7] so LCS[6,8] = max{ LCS[6,7], LCS[5,8]} = 4
- LCS[6,9]: T[5] ≠ P[8] so LCS[6,9] = max{ LCS[6,8], LCS[5,9]} = 4
- LCS[6,10]: T[5] ≠ P[9] so LCS[6,10] = max{LCS[6,9], LCS[5,10]} = 4
- LCS[6,11]: T[5] ≠ P[10] so LCS[6,11] = max{LCS[6,10], LCS[5,11]} = 4

LCS Matrix – Row 6 completed

	0	1/s	2/u	3/b	4/s	5/e	6/q	7/u	8/e	9/n	10/c	11/e
0	0	0	0	0	0	0	0	0	0	0	0	0
1/u	0	0	1	1	1	1	1	1	1	1	1	1
2/s	0	1	1	1	2	2	2	2	2	2	2	2
3/b	0	1	1	2	2	2	2	2	2	2	2	2
4/e	0	1	1	2	2	3	3	3	3	3	3	3
5/e	0	1	1	2	2	3	3	3	4	4	4	4
6/u	0	1	2	2	2	3	3	4	4	4	4	4
7/n	0											
8/e	0											

EXAMPLE – Calculations for Row 7

- Some of the calculations for the seventh row are:
- LCS[7,1]: T[6] ≠ P[0] so LCS[7,1] = max{ LCS[7,0], LCS[6,1]} = 1
- LCS[7,2]: T[6] = P[1] so LCS[7,2] = max{ LCS[7,1], LCS[6,2]} = 2
- LCS[7,3]: T[6] ≠ P[2] so LCS[7,3] = max{ LCS[7,2], LCS[6,3]} = 2
- LCS[7,4]: T[6] ≠ P[3] so LCS[7,4] = max{ LCS[7,3], LCS[6,4]} = 2
- LCS[7,5]: T[6] ≠ P[4] so LCS[7,5] = max{ LCS[7,4], LCS[6,5]} = 3
- LCS[7,6]: T[6] ≠ P[5] so LCS[7,6] = max{ LCS[7,5], LCS[6,6]} = 3
- LCS[7,7]: T[6] = P[6] so LCS[7,7] = max{ LCS[7,6], LCS[6,7]} = 4
- LCS[7,8]: T[6] ≠ P[7] so LCS[7,8] = max{ LCS[7,7], LCS[6,8]} = 4
- LCS[7,9]: T[6] = P[8] so LCS[7,9] = LCS[6,8] + 1 = 5
- LCS[7,10]: T[6] ≠ P[9] so LCS[7,10] = max{LCS[7,9], LCS[6,10]} = 5
- LCS[7,11]: T[6] ≠ P[10] so LCS[7,11] = max{LCS[7,10], LCS[6,11]} = 5

LCS Matrix – Row 7 completed

	0	1/s	2/u	3/b	4/s	5/e	6/q	7/u	8/e	9/n	10/c	11/e
0	0	0	0	0	0	0	0	0	0	0	0	0
1/u	0	0	1	1	1	1	1	1	1	1	1	1
2/s	0	1	1	1	2	2	2	2	2	2	2	2
3/b	0	1	1	2	2	2	2	2	2	2	2	2
4/e	0	1	1	2	2	3	3	3	3	3	3	3
5/e	0	1	1	2	2	3	3	3	4	4	4	4
6/u	0	1	2	2	2	3	3	4	4	4	4	4
7/n	0	1	2	2	2	3	3	4	4	5	5	5
8/e	0											

EXAMPLE – Calculations for Row 8

- The calculations for the eighth row are:
- LCS[8,1]: T[7] ≠ P[0] so LCS[8,1] = max{ LCS[8,0], LCS[7,1]} = 1
- LCS[8,2]: T[7] ≠ P[1] so LCS[8,2] = max{ LCS[8,1], LCS[7,2]} = 2
- LCS[8,3]: T[7] ≠ P[2] so LCS[8,3] = max{ LCS[8,2], LCS[7,3]} = 2
- LCS[8,4]: T[7] ≠ P[3] so LCS[8,4] = max{ LCS[8,3], LCS[7,4]} = 2
- LCS[8,5]: T[7] = P[4] so LCS[8,5] = LCS[7,4] + 1 = 3
- LCS[8,6]: T[7] ≠ P[5] so LCS[8,6] = max{ LCS[8,5], LCS[7,6]} = 3
- LCS[8,7]: T[7] ≠ P[6] so LCS[8,7] = max{ LCS[8,6], LCS[7,7]} = 4
- LCS[8,8]: T[7] = P[7] so LCS[8,8] = LCS[7,7] + 1 = 5
- LCS[8,9]: T[7] ≠ P[8] so LCS[8,9] = max{ LCS[8,8], LCS[7,9]} = 5
- LCS[8,10]: T[7] ≠ P[9] so LCS[8,10] = max{LCS[8,9], LCS[7,10]} = 5
- LCS[8,11]: T[7] = P[10] so LCS[8,11] = LCS[7,10] + 1 = 6

LCS Matrix – Row 8 completed

	0	1/s	2/u	3/b	4/s	5/e	6/q	7/u	8/e	9/n	10/c	11/e
0	0	0	0	0	0	0	0	0	0	0	0	0
1/u	0	0	1	1	1	1	1	1	1	1	1	1
2/s	0	1	1	1				2	2	2	2	2
3/b	0	1	1	2				2	2	2	2	2
4/e	0	1	1	2	2	3	3	3	3	3	3	3
5/e	0	1	1	2	2	3	3	3	4	4	4	4
6/u	0	1	2	2	2	3	3	4	4	4	4	4
7/n	0	1	2	2	2	3	3	4	4	5	5	5
8/e	0	1	2	2	2	3	3	4	5	5	5	**6**

Final result indicates the longest common subsequence for these two strings has length 6.

LCS Matrix – Row 8 completed

	0	1/s	2/u	3/b	4/s	5/e	6/q	7/u	8/e	9/n	10/c	11/e
0	0				0	0	0	0	0	0	0	0
1/u	0				1	1	1	1	1	1	1	1
2/s	0				2	2	2	2	2	2	2	2
3/b	0				2	2	2	2	2	2	2	2
4/e	0				2	3	3	3	3	3	3	3
5/e	0				2	3	3	3	4	4	4	4
6/u	0				2	3	3	4	4	4	4	4
7/n	0				2	3	3	4	4	5	5	5
8/e	0	1	2	2	2	3	3	4	5	5	5	6

Notice the principle of optimality holds at any point in the table. For example, Looking at LCS[2,10] = 2, indicates that the longest common subsequence through the first 2 characters in T and the first 10 characters in P is 2 (i.e., "u..s"). For LCS[5,8] = 4, indicating the longest common subsequence through the first 5 characters in T and first 8 characters in P is 4 (i.e., " u..s..e..e")

Determining The LCS

- The LCS algorithm determines the longest common subsequence of `T[0:n-1]` and `P[0:m-1]` but does not output the actual subsequence itself.
- In some cases, as was the case with the optimal binary search tree, we often would like to know the actual longest common subsequence. In other cases, such as DNA matching and spell checking, just knowing the length of the longest common subsequence is more important than actually knowing the sequence.
- Nevertheless, it is interesting that a longest common subsequence can be determined from just the LCS matrix without the need to maintain any additional information.

Determining The LCS (cont.)

- One way that you can generate the longest common subsequence is to start at the bottom-right corner (*n,m*) of the LCS matrix and work backwards through the matrix to build the subsequence in reverse order (i.e., starting with the right-end of the subsequence).
- The moves through the matrix are dictated by looking at how we get the value which is assigned to a given cell when we used the recurrence to build the LCS matrix.
- More precisely, if we are currently in position LCS[i,j], and T[i-1] = P[j-1], then this common value is appended to the beginning of the string already generated (beginning with a null string), and we move to position LCS[i-1, j-1].

Determining The LCS (cont.)

- On the other hand, if T[i-1] $\neq$ P[j-1], then we move to position LCS[i-1,j] or LCS[i,j-1], depending on whether LCS[i-1,j] > LCS[i,j-1].
- If LCS[i-1,j] = LCS[i, j-1], then either move can be made.

```
In LCS[i,j] and T[i-1] = P[j-1] then move to LCS[i-1,j-1]
In LCS[i,j] and T[i-1] ≠ P[j-1] then
    if LCS[i-1,j] > LCS[i,j-1] move to LCS[i-1,j]  //move-up
    if LCS[i-1,j] < LCS[i,j-1] move to LCS[i,j-1] //move-left
    if LCS[i-1,j] = LCS[i,j-1] move to either LCS[i-1,j] or LCS[i, j-1]
```

Determining The LCS (cont.)

- Shown below are the first few calculations necessary to determine the longest common subsequence using the LCS matrix and the move-left method.

In LCS[8,11], T[7] = P[10] (e = e) so move to LCS[7,10]

In LCS[7,10], T[6] ≠ P[9] (n ≠ c), LCS[6,10] < LCS[7,9] so move to LCS[7,9]

In LCS[7,9], T[6] = P[8] (n = n) so move to LCS[6,8]

In LCS[6,8], T[5] ≠ P[7] (u ≠ e), LCS[5,8] = LCS[6,7] so move to LCS[6,7]

In LCS[6,7], T[5] = P[6] so move to LCS[5,6]

In LCS[5,6], T[4] ≠ P[5] (e ≠ q), LCS[4,6] = LCS[5,5], so move to LCS[5,5]

LCS Using Move-Left Method

	0	1/s	2/u	3/b	4/s	5/e	6/q	7/u	8/e	9/n	10/c	11/e
0	0	0	0	0	0	0	0	0	0	0	0	0
1/u	0	0	1	1	1	1	1					
2/s	0	**1**	1	1	2	2	2	2	2	2	2	2
3/b	0	1	1	**2**	2	2	2	2	2	2	2	2
4/e	0	1	1	2	2	**3**	3	3	3	3	3	3
5/e	0	1	1	2	2	3	3	3	4	4	4	4
6/u	0	1	2	2	2	3	3	**4**	4	4	4	4
7/n	0	1	2	2	2	3	3	4	4	**5**	5	5
8/e	0	1	2	2	2	3	3	4	5	5	5	**6**

LCS = “sbeune” as determined by move-left rule.

LCS Using Move-Up Method

	0	1/s	2/u	3/b	4/s	5/e	6/q	7/u	8/e	9/n	10/c	11/e
0	0	0	0	0	0	0	0	0	0	0	0	0
1/u	0	0	**1**	1	1	1	1	1				
2/s	0	1	1	1	**2**	2	2	2	2	2	2	2
3/b	0	1	1	2	2	2	2	2	2	2	2	2
4/e	0	1	1	2	2	**3**	3	3	3	3	3	3
5/e	0	1	1	2	2	3	3	3	4	4	4	4
6/u	0	1	2	2	2	3	3	**4**	4	4	4	4
7/n	0	1	2	2	2	3	3	4	4	**5**	5	5
8/e	0	1	2	2	2	3	3	4	5	5	5	**6**

LCS = “useune” using move-up method