Longest Common Subsequence (LCS)

Chapter 15

p 390

Longest Common Subsequence

- Problem: Let $x_1x_2...x_m$ and $y_1y_2...y_n$ be two sequences over some alphabet.
  - We assume they are strings of characters

- Find a longest common subsequence (LCS) of $x_1x_2...x_m$ and $y_1y_2...y_n$

Many other string operations have the same basic structure.

Example

- $x_1x_2x_3x_4x_5x_6x_7x_8$= b a c b f f c b
- $y_1y_2y_3y_4y_5y_6y_7y_8y_9$ = d a b e a b f b c

- Longest Common Subsequence is:

  A subsequence is a set of characters that appear in left- to-right order, *but not necessarily consecutively*.

Dynamic Programming

- LCS can be solved using dynamic programming

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution bottom-up
4. Construct an optimal solution from the computed information

Step 1 Characterizing

- Optimal substructure:

  If $z = z_1z_2...z_p$ is a LCS of $x_1x_2...\underline{x_m}$ and $y_1y_2...\underline{y_n}$, then

  At least one of these most hold

  - $x_m = y_n$, and $z_1z_2...z_{p-1}$ is an LCS of $x_1x_2...x_{\mathbf{m-1}}$ and $y_1y_2...y_{\mathbf{n-1}}$,
  - $x_m \mathrel{!}= y_n$, and $z_1z_2...z_p$ is an LCS of $x_1x_2...x_{\mathbf{m-1}}$ and $y_1y_2...y_n$,
  - $x_m \mathrel{!}= y_n$, and $z_1z_2...z_p$ is an LCS of $x_1x_2...x_m$ and $y_1y_2...y_{\mathbf{n-1}}$.

Step 2: Recursive Solution

Let c_{ij} = length of LCS of $x_1x_2\ldots x_i$ and $y = y_1y_2\ldots y_j$.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ 1 + c[i-1, j-1] & \text{if } x_i = y_j, \\ \max(c[i-1,j], c[i,j-1]) & \text{if } x_i \neq y_j. \end{cases}$$

$$b[i,j] = \begin{cases} \text{“}\nwarrow\text{”} & \text{if } x_i = y_j, \\ \text{"}\uparrow\text{"} & \text{if } x_i \neq y_j \text{ and } c[i-1,j] \geq c[i,j-1], \\ \text{"}\leftarrow\text{"} & \text{if } x_i \neq y_j \text{ and } c[i-1,j] < c[i,j-1]. \end{cases}$$

We compute the $c[i,j]$ and $b[i,j]$ in order of increasing $i+j$, or alternatively in order of increasing i, and for a fixed i, in order of increasing j.

p 394

Step 3&4

LCS-LENGTH(X, Y, m, n)

```
let b[1..m, 1..n] and c[0..m, o..n] be new tables
for i = 1 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
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        else if c[i − 1, j] ≥ c[i, j − 1]
                c[i, j] = c[i − 1, j]
                b[i, j] = "↑"
            else c[i, j] = c[i, j − 1]
                b[i, j] = "←"
return c and b
```

Example

b,c matrices combined

\ j i \	0	1 d	2 a	3 b	4 e	5 a	6 b	7 f	8 b	9 c
0	0	0	0	0	0	0	0	0	0	0
1 b	0									
2 a	0									
3 c	0									
4 b	0									
5 f	0									
6 f	0									
7 c	0									
8 b	0									

```
PRINT-LCS(b, X, i, j)
  if i == 0 or j == 0
        return
  if b[i, j] == "↖"
        PRINT-LCS(b, X, i − 1, j − 1)
        print x_i
  elseif b[i, j] == "↑"
        PRINT-LCS(b, X, i − 1, j)
  else PRINT-LCS(b, X, i, j − 1)
```

String Similarity

- Edit Distance
  - Levenshtein
  - minimize changes
- Sequence Alignment
  - Needleman-Wunsch
  - maximize similarity
    - by giving weights to types of differences

http://xlinux.nist.gov/dads/HTML/Levenshtein.html

Edit Distance

- How many insertions, deletions, replacements will transform one string into another?
  - Damerau-Levenshtein includes transpositions as a special case the → teh

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j) + 1 \\ \mathrm{lev}_{a,b}(i,j-1) + 1 \\ \mathrm{lev}_{a,b}(i-1,j-1) + [a_i \neq b_j] \end{cases} & \text{otherwise.} \end{cases}$$

http://en.wikipedia.org/wiki/Levenshtein_distance

Levenshtein

Edit Distance Matrix

ATCGTT vs AGTTAC

		A	G	T	T	A	C	
	0	1	2	3	4	5	6	j
A	1							
T	2							
C	3							
G	4							
T	5							
T	6							

i

Backtracking

deletion	UP
insertion	LEFT
match/mismatch	DIAG

Sequence Alignment

- Similarity based on gaps and mismatches.
- Alignment
  - matched pairs from both strings
  - no crossings
- Generalized form of Levenshtein
  - additional parameters:
    - gap penalty, δ
    - mismatch cost ($\alpha_{x,y}$; $\alpha_{x,x} = 0$)

Kleinberg, Tardos, Algorithm Design, Pearson Addison Wesley, 2006, p 278

http://www.aw-bc.com/info/kleinberg/

Recurrence

- Two strings $x_1 \ldots x_m$ and $y_1 \ldots y_n$
- In an optimal alignment, *M*, at least one of the following is true:
  - (x_m, y_n) is in M
  - x_m is not matched
  - y_n is not matched

Recurrence

- So, for i and $j > 0$
- $opt(i,j) = \min[\alpha_{xi,yj} + opt(i-1,j-1),$
  $\delta + opt(i-1,j),$ // x_i is not matched
  $\delta + opt(i,j-1)\ ]$ // y_j is not matched
- (x_i,y_j) is in an optimal alignment M for this subproblem iff the minimum achieved is achieved by the first of these three values.

Kleinberg, Tardos, Algorithm Design, Pearson Addison Wesley, 2006, p 282

Sequence Alignment Graph

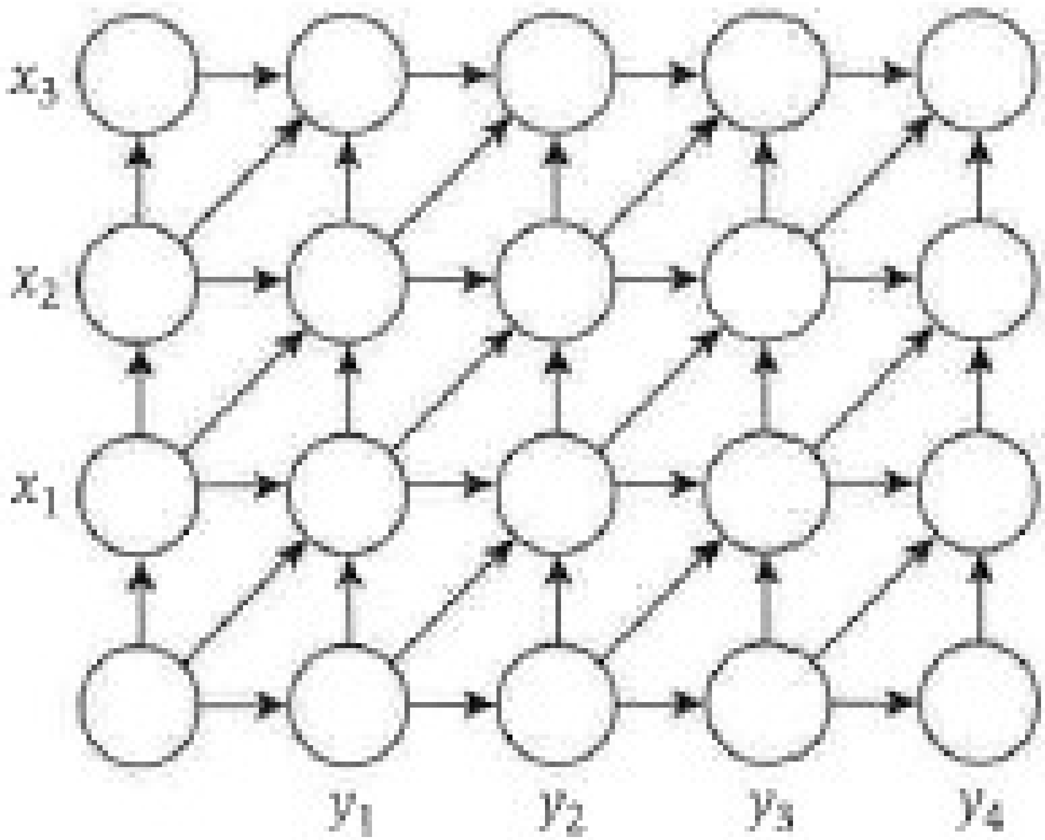


Figure 6.17 A graph-based picture of sequence alignment.

Kleinberg, Tardos, p 283

Recover Alignment

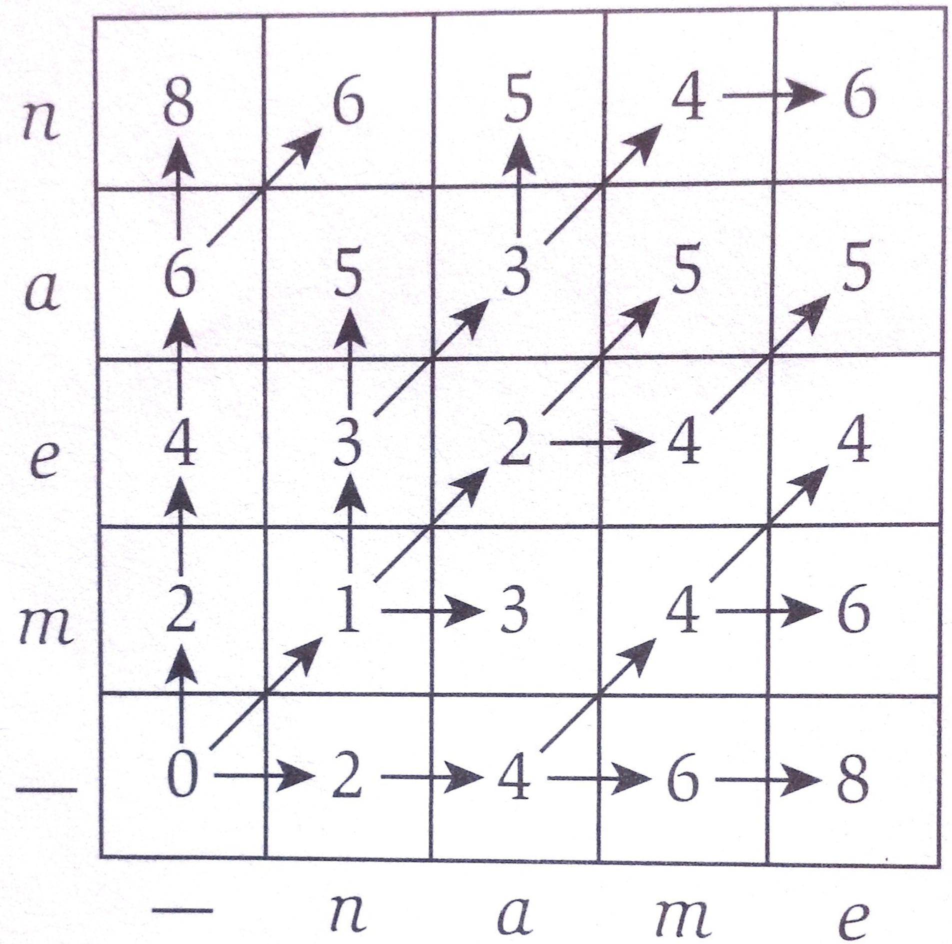


Figure 6.18 The OPT values for the problem of aligning the words *mean* to *name*.

Kleinberg, Tardos, p 284

Sequence Alignment (space efficient)

Kleinberg, Tardos, p 285

- Hirschberg - 1975
  - value: need the current and previous column

```
Space-Efficient-Alignment(X,Y)
  Array B[0...m, 0...1]
  Initialize B[i,0] = iδ for each i (just as in column 0 of A)
  For j = 1,...,n
     B[0,1] = jδ (since this corresponds to entry A[0,j])
     For i = 1,...,m
         B[i,1] = min[α_{x_i y_j} + B[i-1,0],
                          δ + B[i-1,1],  δ + B[i,0]]
     Endfor
     Move column 1 of B to column 0 to make room for next iteration:
         Update B[i,0] = B[i,1] for each i
  Endfor
```

Actual Alignment

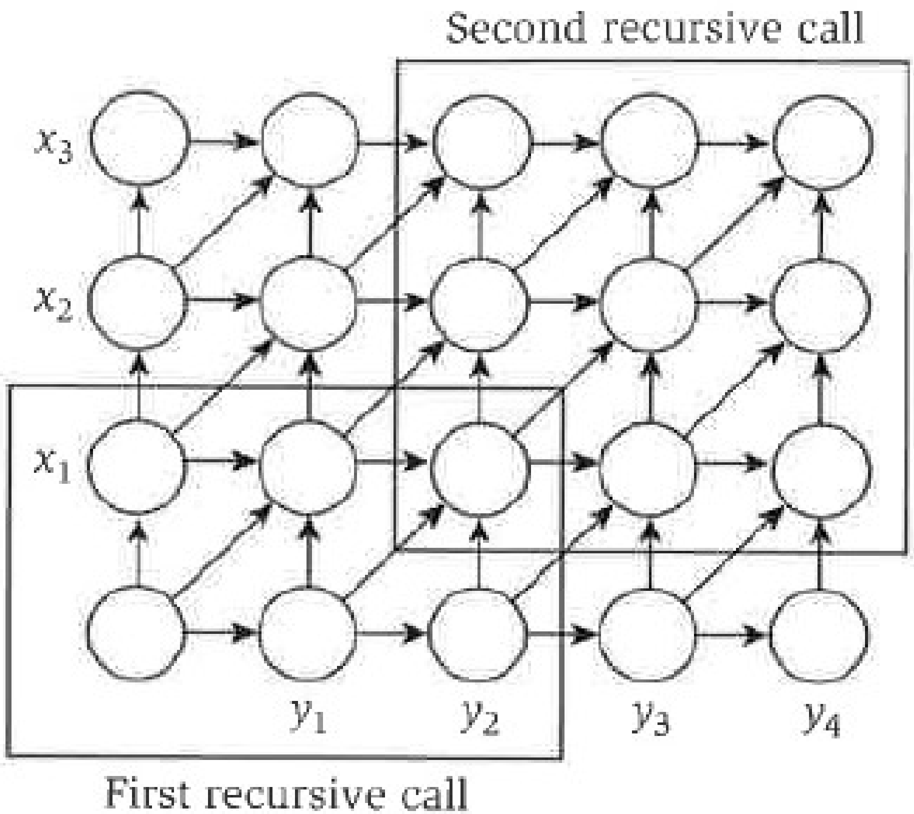
- How do we recover the actual alignment?
  - We need the entire matrix?

Algorithm

```
Divide-and-Conquer-Alignment(X,Y)
  Let m be the number of symbols in X
  Let n be the number of symbols in Y
  If m ≤ 2 or n ≤ 2 then
     Compute optimal alignment using Alignment(X,Y)
  Call Space-Efficient-Alignment(X,Y[1:n/2])
  Call Backward-Space-Efficient-Alignment(X,Y[n/2+1:n])
  Let q be the index minimizing f(q,n/2)+g(q,n/2)
  Add (q,n/2) to global list P
  Divide-and-Conquer-Alignment(X[1:q],Y[1:n/2])
  Divide-and-Conquer-Alignment(X[q+1:n],Y[n/2+1:n])
  Return P
```

Kleinberg, Tardos, p 288

6.7 Sequence Alignment in Linear Space via Divide and Conquer



Kleinberg, Tardos, p 289

String Matching / Searching

- Naive
- **Horspool**[1]
- Boyer-Moore[1]
- Rabin Karp[2]
- **Knuth Morris Pratt**[2]

Not Dynamic Programming because there are not subproblems.

But precompute a table to help you solve the problem.

[1] Levitin, Introduction to The Design and Analysis of Algorithms, 3rd edition, Pearson Addison Wesley, p 259

[2] CLRS, p 990 &1002

Naive

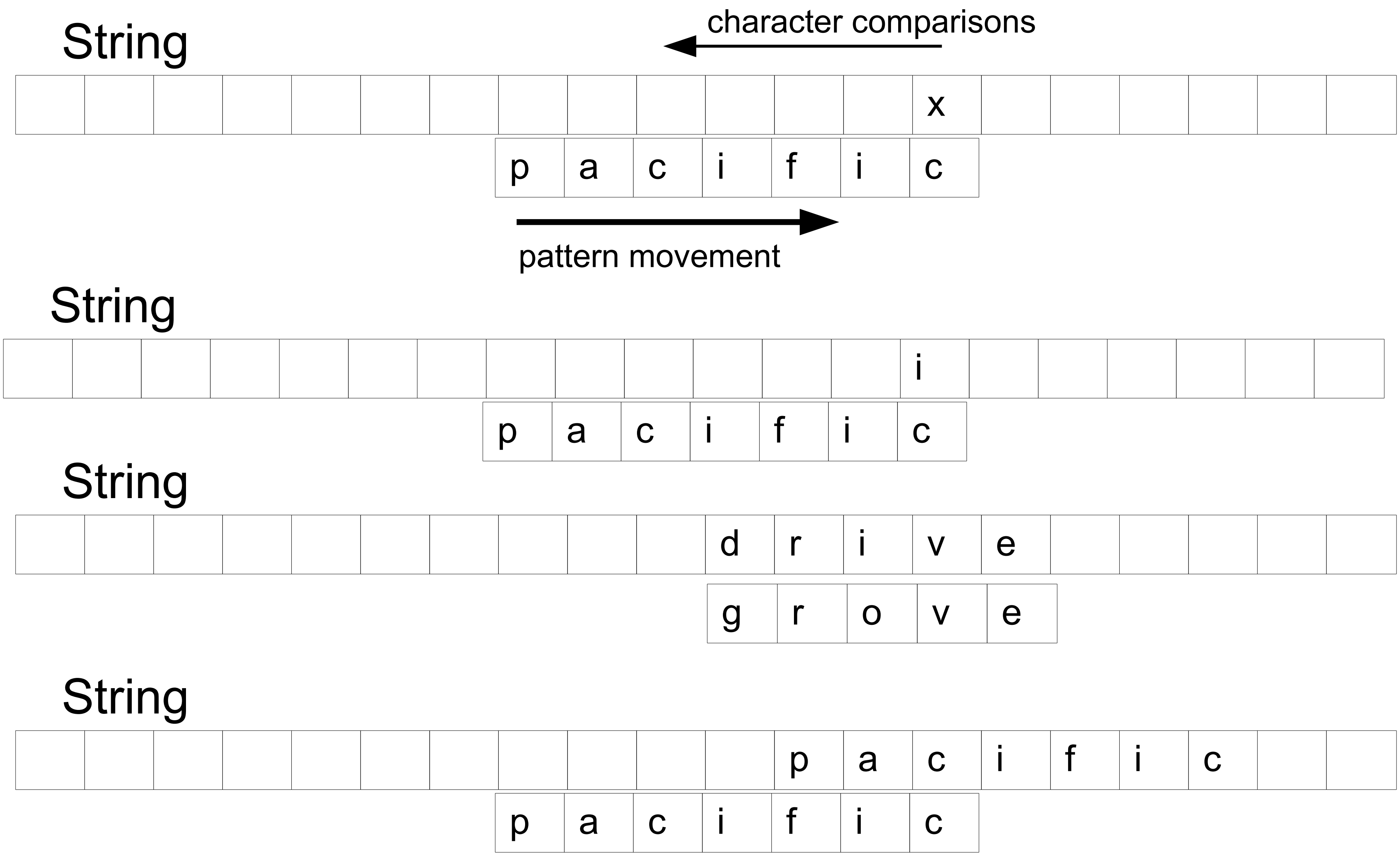
```
int naiveSearch(string, pattern)
{
  retVal = -1;
  mismatch = true:

  for( i=0;i  < string.length - pattern.length &&
       true == mismatch  ; i++)
  {
     mismatch = false;
     for( j =0;j<pattern.length && !mismatch ;j++)
     {
         if( string[i] != pattern[j] )
         {
             mismatch = true;
         }
     }
     if ( !mismatch)
     {
          retVal = i;
     }
  }
  return retVal;
}
```

Horspool

- match the pattern right to left
- on mismatch, shift the pattern smartly
    - by 1+ character
- Preprocess string to determine shifting
    - build a table for shifts for each valid character

Example

need a better string!



Shifting

- t(c) =

  the pattern's length, m, if c is not among the first m-1 characters of the pattern

  the distance from the rightmost c among the first m-1 characters of the pattern to its last character, otherwise

p	a	c	i	f	i	c

a	b	c	f	i	...	p	...	x	y	z
								7		

```
Horspool(P[0..m-1],  T[0..n-1])
        T ← ComputeShifts(P)                      ' Generate table of shifts
        i ← m – 1                                 ' Position of pattern's right end
        while i ≤ n – 1
                k ← 0                             ' Number of matched characters
                while k ≤ m -1 and P[m-1-k] = T[i-k]
                        k++
                if k = m
                        return "Match at " + (i – m + 1)
                else
                        i ← i + T[i]
        return -1                                 ' No match found
```

[1] Levitin, Introduction to The Design and Analysis of Algorithms, 3rd edition, Pearson Addison Wesley, p 262

http://www.math.uaa.alaska.edu/~afkjm/cs351/handouts/strings.ppt