Incrementally developing and implementing Hirschberg's longest common subseqence algorithm using Lua

Robin Snyder, robin@robinsnyder.com

Slide notes from the Lua Workshop in Reston, VA, November 29-30, 2012

Printed: 2012/12/01

1. Abstract

The longest common subsequnce (LCS) problem is a dual problem of the shortest edit distance (SED) problem. The solution to these problems are used in open source file comparison tools such as WinMerge and DiffMerge. In 1974, Hirshberg published a reasonably space and time efficient solution to these problems. This talk will cover the incremental development and implementation of Hirshberg's algorithm in Lua, including trade-offs and design decisions along the way. The final algorithm implementation can be used for customized comparsion of files, or other applications, as needed.

2. Lua investigation

Lua for:

- Creative Zen
- Logitech G13 keypad
- Delphi custom application integration
- Command line scripts

3. Subsequence

String C = $c_1c_2...c_p$ is a *subsequence* of string A = $a_1a_2...a_m$ iff there as a mapping

F: [1, 2, ..., p] to [1, 2, ..., m]

such that F(i) = k only if c_i is a_k and F is a monotone strictly increasing function (that is, (F(i) = u) and (F(j) = v) and (i < j) imply that (u < v)).

4. Common subsequence

String C is a *common subsequence* of strings A and B iff

- C is a subsequence of A and
- C is a subsequence of B.

5. Problem

Given strings $A = a_1a_2...a_m$ and $B = b_1b_2...b_n$ find string $C = c_1c_2...c_p$ such that C is a common subsequence of both A and B and p is maximized.

C is then called a *maximal common subsequence* or **Longest Common Subsequence**.

6. Alphabet

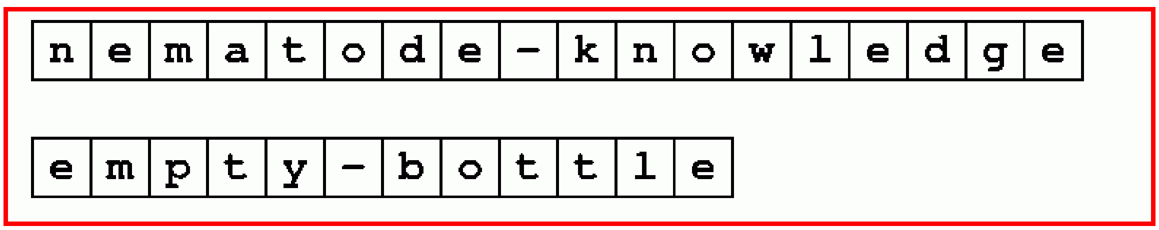
Alphabets examples:

- Characters (line comparison)
- Lines (file comparison)
- Nucleotides (DNA)

7. Example strings

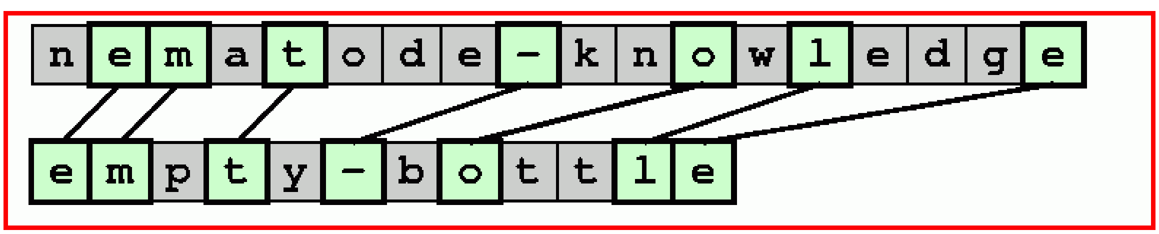
Example strings:

- a = "nematode-knowledge"
- b = "empty-bottle"



- m = string.len(a) = string.len("nematode-knowledge") = 18
- n = string.len(b) = string.len("empty-bottle") = 12

8. LCS



- No connection lines cross.
- In general there are more than one LCS (e.g., last "**e**").

9. Symbols

Symbols can be anything that can be matched.

- Letters of an alphabet
- Lines of text
- Nucleotides (in DNA)

For example purposes, letters will be used.

10. DNA

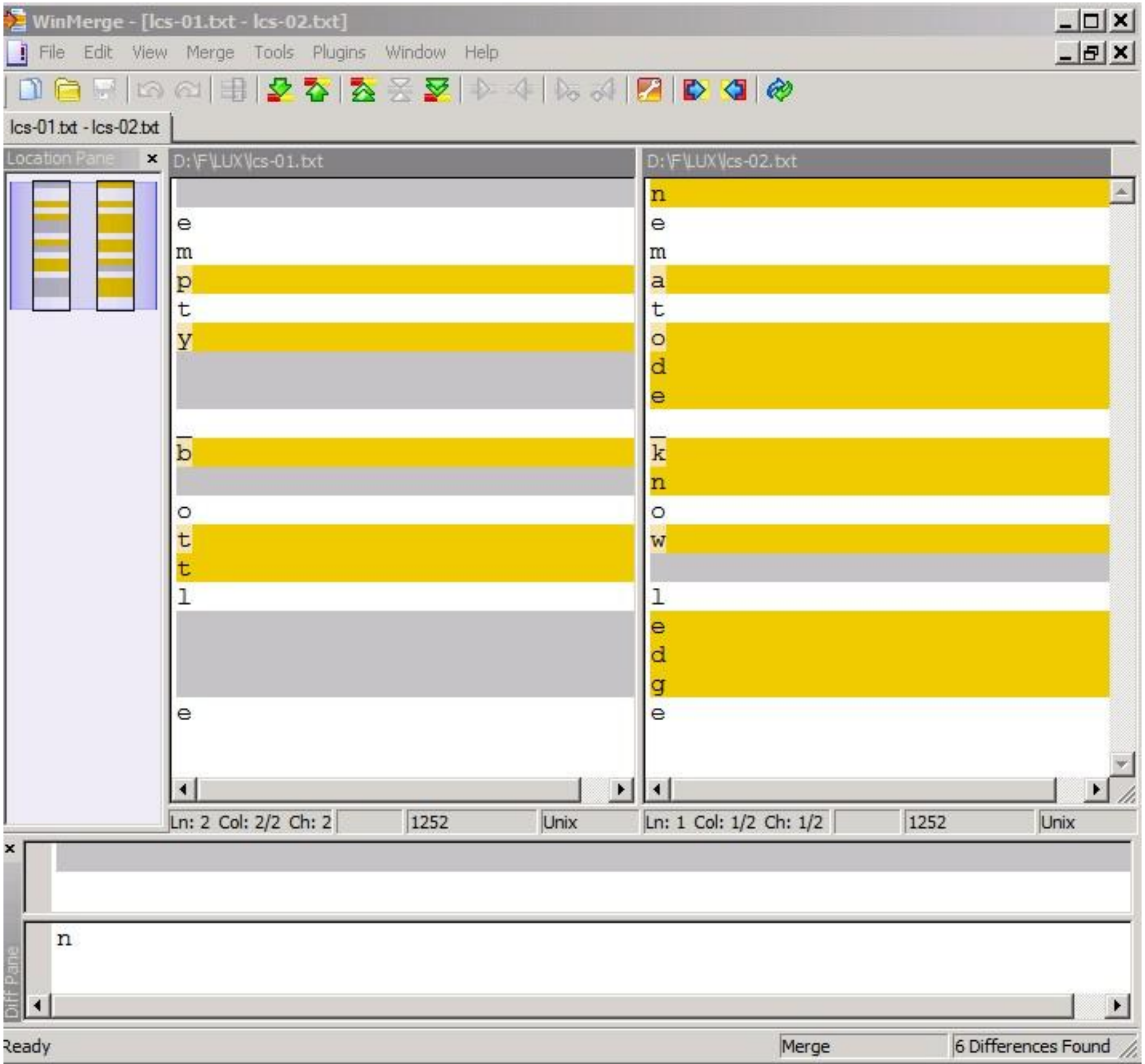
```
AGGCTATCACCTGACCTCCAGGCCGATGCCC...
TAGCTATCACGACCGCGGTCGATTTGCCCGAC...
```

11. File comparison

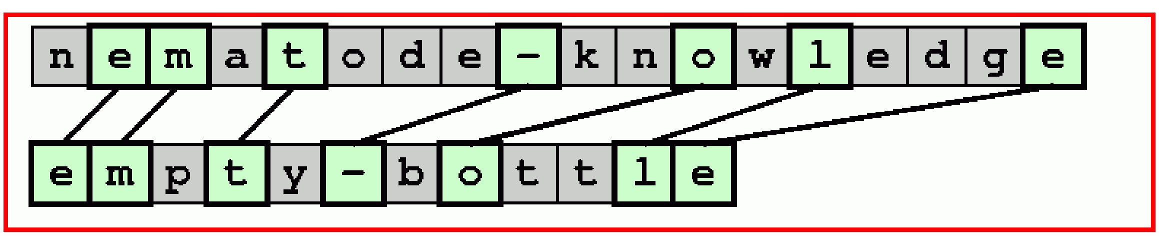
File comparison: (line oriented, useful for regression testing, etc.):

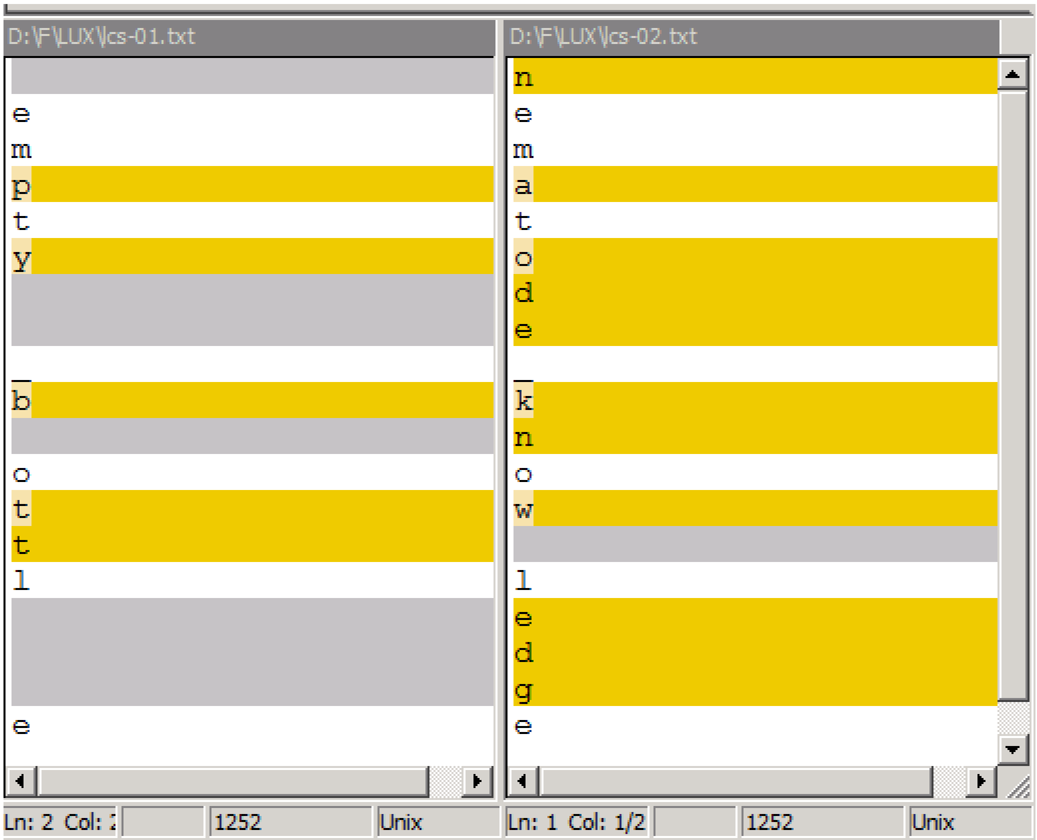
- WinMerge at http://www.winmerge.org.
- DiffMerge at http://www.sourcegear.com/difmerge.

Make each letter a line in a file.



Note: LCS can be used on individual lines to see similarities and differences within a line.





The SED (Shortest Edit Distance) is a dual problem of the LCS (Longest Common Subsequence) problem.

12. Approach

Approach:

- Top down divide and conquer (by 1) for correctness.
- Memoization (time efficiency).
- Bottom up dynamic programming (time efficiency).
- Length only (bootstrap)
- Divide and conquer (space efficiency)
- Recover solution

13. Program and output

```
a = "empty_bottle"
b = "nematode_knowledge"
print("a=[" .. a .. "]")
print("b=[" .. b .. "]")
local c = top_down_lcs1(a, b)
print(" c=[" .. c .. "]")
```

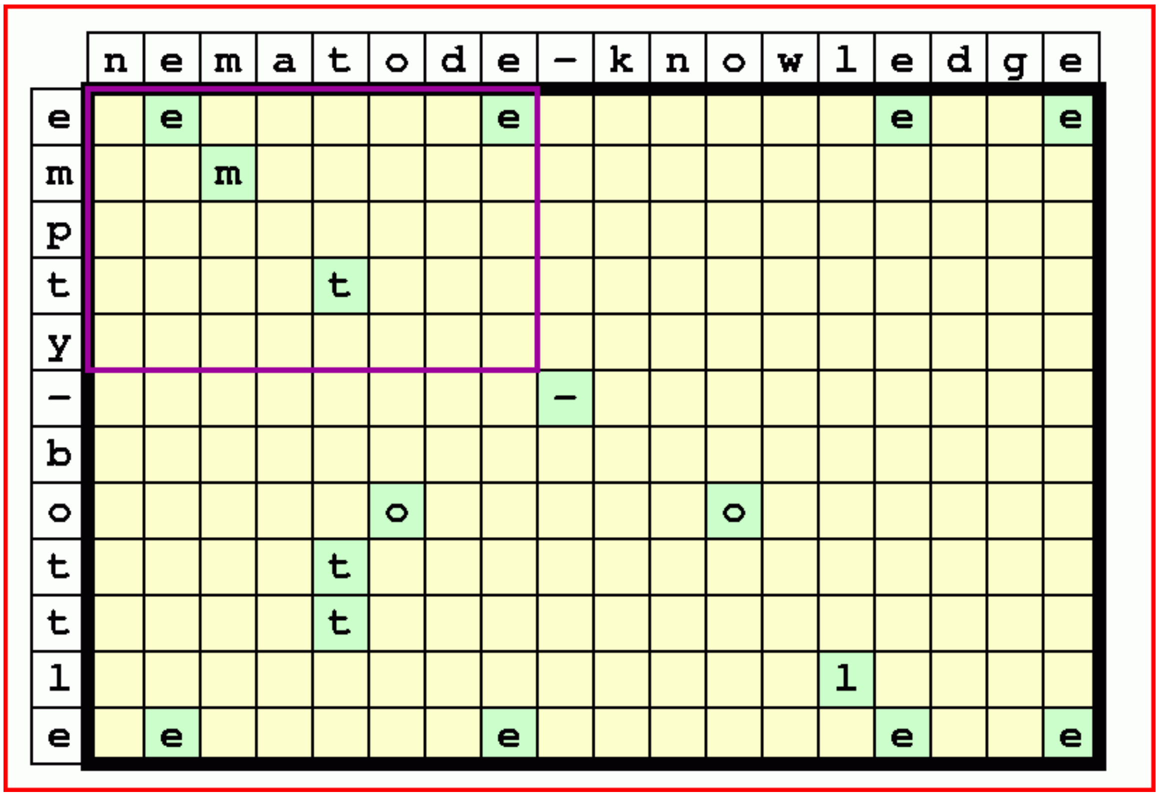
14. Output:

```
a=[empty_bottle]
b=[nematode_knowledge]
c=[emt_ole]
```

Time and space efficiency depends on the algorithm used.

15. Possible matches

- (a == "nematode-knowledge") and (m == 18)
- (b == "empty-bottle") and (n == 12)
- possible non-empty substring compares: m*n == 216



Start from the end of both strings.

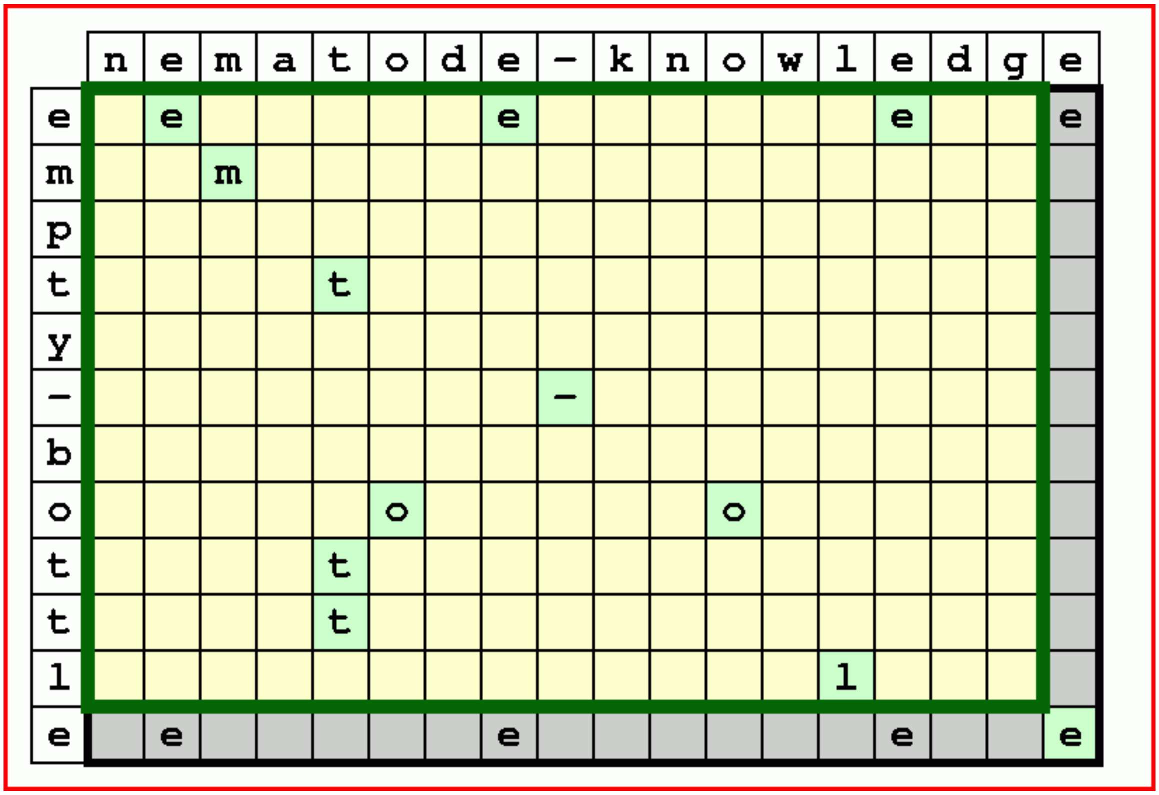
16. Compare

Compare both versions for symmetry:

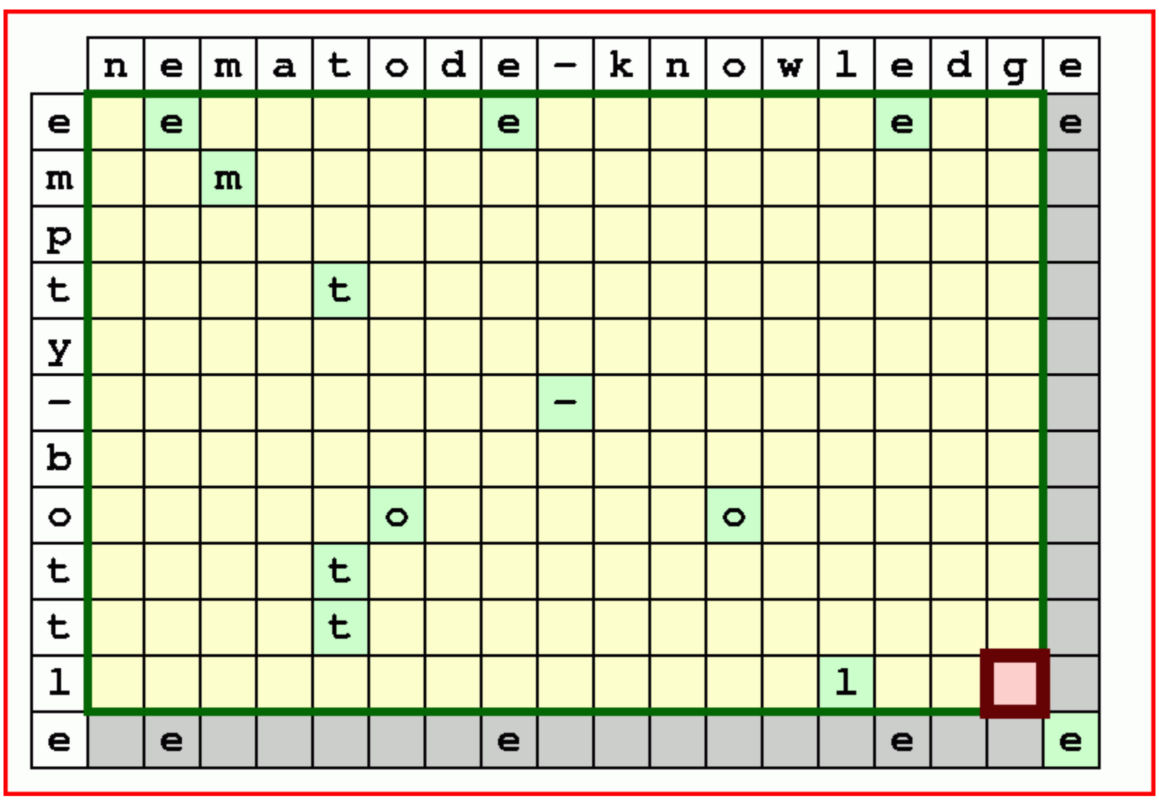
- Flip the order of the strings.
- Forward or backward in strings.
- String or reverse string.

2*2*2 = 8 approaches. All yield the same LCS.

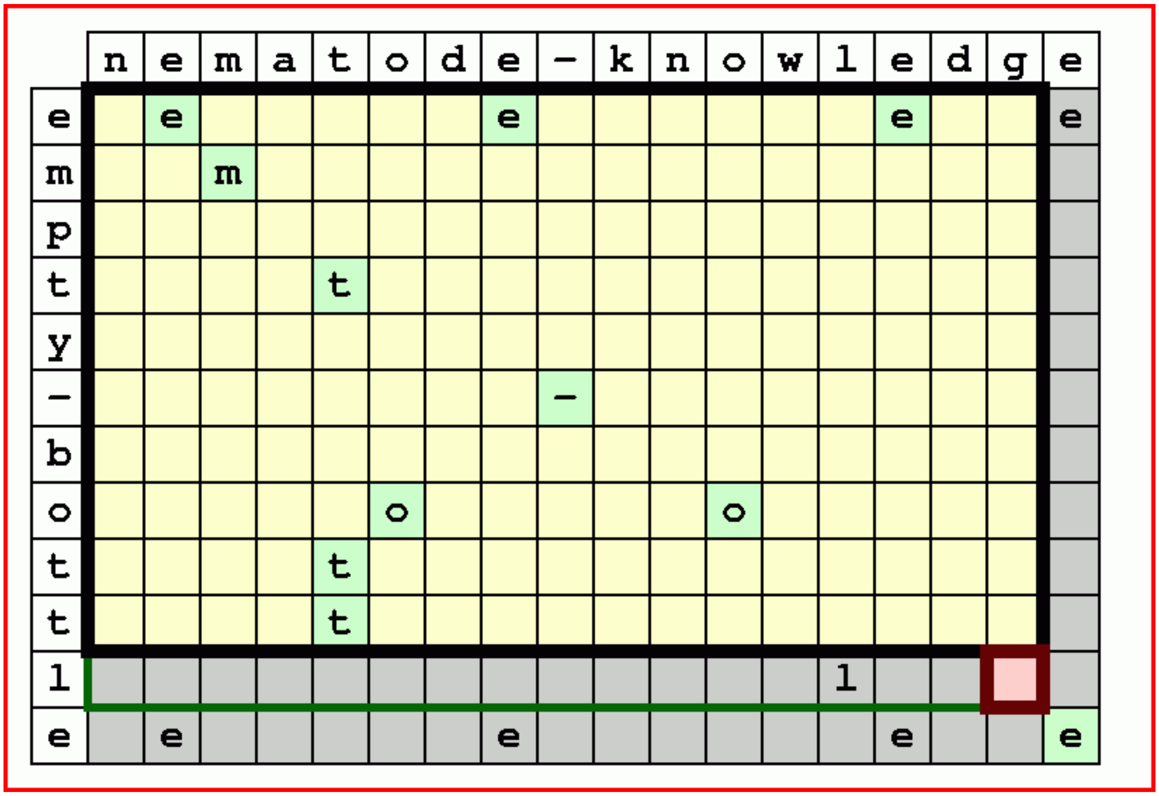
17. Match



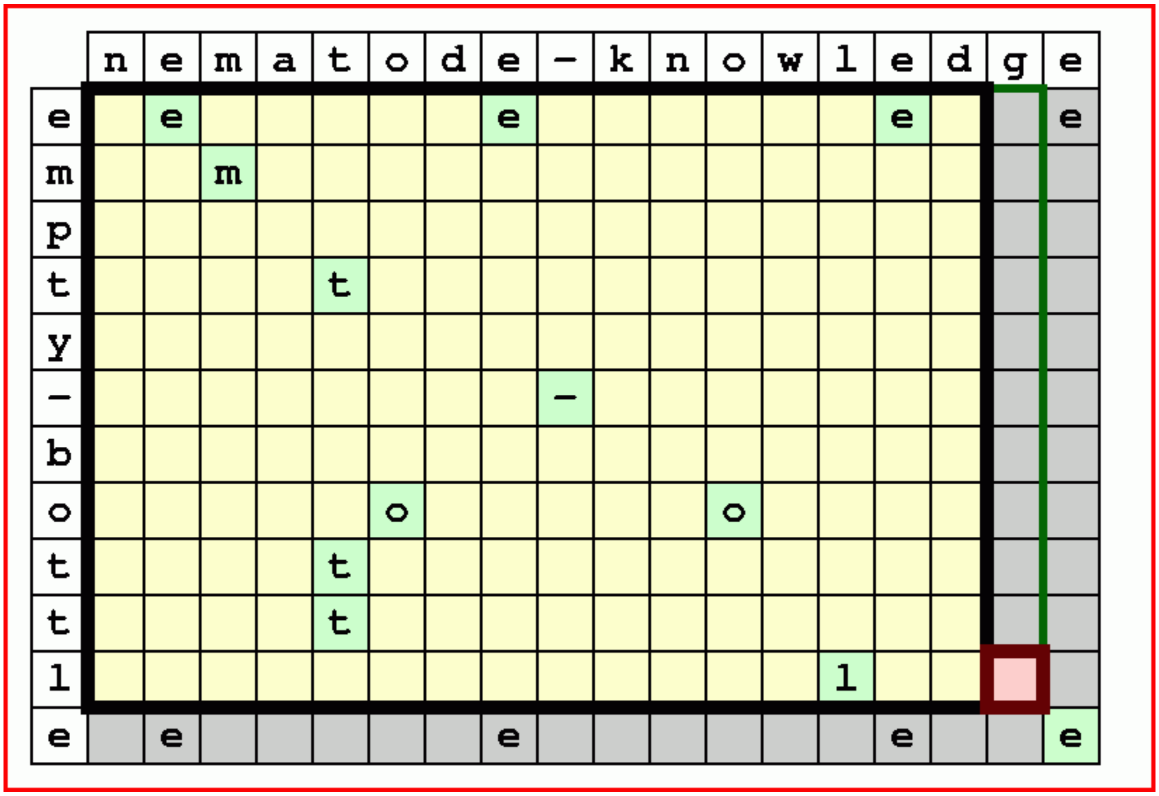
18. Non-Match (1)



19. Non-Match (2)



20. Next



21. Recursive top down backward

$a_1\ a_2\ \ldots\ a_{m-1}\ a_m$
$b_1\ b_2\ \ldots\ b_{n-1}\ x_n$

```
function lcs_1b(a, b)
   local m = #a
   local n = #b
   if (m == 0) or (n == 0) then
      return ""
   elseif string.sub(a, m, m) == string.sub(b, n, n) then
      return lcs_1b(string.sub(a, 1, m-1), string.sub(b, 1, n-1)) .. string.sub(a, m, m)
   else
      local a1 = lcs_1b(a, string.sub(b, 1, n-1))
      local b1 = lcs_1b(string.sub(a, 1, m-1), b)
      return math.max(#a1, #b1)
      end
   end
```

Time and space INEFFICIENT!!!

22. Recursive top down forward

$a_1\ a_2\ \ldots\ a_{m-1}\ a_m$
$b_1\ b_2\ \ldots\ b_{n-1}\ x_n$

```
function lcs_1f(a, b)
   local m = #a
   local n = #b
   if (m == 0) or (n == 0) then
      return ""
   elseif string.sub(a, 1, 1) == string.sub(b, 1, 1) then
      return string.sub(a, 1, 1) .. lcs_1f(string.sub(a, 2, m), string.sub(b, 2, n))
   else
      local a1 = lcs_1f(a, string.sub(b, 2, n))
      local b1 = lcs_1f(string.sub(a, 2, m), b)
```

```
      return math.max(#a1, #b1)
      end
   end
```

Time and space INEFFICIENT!!!

23. Maximum subsequence length

- String rewriting involves copies and is inefficient.
- Modify the algorithm to return the length of the maximal subsequence.
- Improve the algorithm.
- Extract the LCS from the results.

```
function lcs_2b(a, b)
   local m = #a
   local n = #b
   if (m == 0) or (n == 0) then
      return 0
   elseif string.sub(a, m, m) == string.sub(b, n, n) then
      return lcs_2b(string.sub(a, 1, m-1), string.sub(b, 1, n-1)) + 1
   else
      local a1 = lcs_2b(a, string.sub(b, 1, n-1))
      local b1 = lcs_2b(string.sub(a, 1, m-1), b)
      return math.max(a1, b1)
      end
   end
```

24. Output

```
a=[empty_bottle]
b=[nematode_knowledge]
c=[7]
```

25. Next step

- Use a list to store the string symbols.
- Pass the ending location.

26. Use a list for A and B

Use a list for A and B.

```
A = {}
setDefault(A, "")
for i=1,string.len(a) do
   A[i] = string.sub(a, i, i)
   end
B = {}
setDefault(B, "")
for j=1,string.len(b) do
   B[j] = string.sub(b, j, j)
   end
io.write("A=[")
for i,a in pairs(A) do
   io.write(a)
   end
print("]")
io.write("B=[")
for j,b in pairs(B) do
   io.write(b)
   end
print("]")
```

27. Modified code

```
function lcs_3b(A, i, B, j)
   if (i == 0) or (j == 0) then
      return 0
   elseif A[i] == B[j] then
      return lcs_3b(A, i-1, B, j-1) + 1
   else
      local a1 = lcs_3b(A, i, B, j-1)
      local b1 = lcs_3b(A, i-1, B, j)
      return math.max(a1, b1)
      end
   end
```

28. Call

```
c = lcs_3b(A, #A, B, #B)
print("c=[" .. c .. "]")
```

29. Observation

Observation: L(i, j) is a maximal possible length common subsequence of A_{1i} and B_{1j}.

Initialization of L, the Length matrix.

```
L = {}
for i=1,#A do
   L[i] = {}
   for j=1,#B do
      L[i][j] = -1
      end
   end
```

For convenience, L is initially defined as -1 everywhere (explicitly or via default metatable method).

30. Initial L matrix

```
 L=  n   e   m   a   t   o   d   e   _   k   n   o   w   l   e   d   g   e
  e -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
  m -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
  p -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
  t -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
  y -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
  _ -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
  b -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
  o -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
  t -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
  t -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
  l -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
  e -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
```

	n	e	m	a	t	o	d	e	_	k	n	o	w	l	e	d	g	e
e																		
m																		
p																		
t																		
y																		
_																		
b																		
o																		
t																		
t																		
l																		
e																		

31. Compute the L matrix

```
function lcs_4b(A, i, B, j, L)
   local p
   if (i == 0) or (j == 0) then
      p = 0
   else
      if A[i] == B[j] then
         p = lcs_4b(A, i-1, B, j-1, L) + 1
      else
         local a1 = lcs_4b(A, i, B, j-1, L)
         local b1 = lcs_4b(A, i-1, B, j, L)
         p = math.max(a1, b1)
         end
      L[i][j] = p
      end
   return p
   end
```

The L matrix is computed.

32. Computed L matrix

```
 L=  n  e  m  a  t  o  d  e  _  k  n  o  w  l  e  d  g  e
  e  0  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 -1
  m  0  1  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2 -1
  p  0  1  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2 -1
  t  0  1  2  2  3  3  3  3  3  3  3  3  3  3  3  3  3 -1
  y  0  1  2  2  3  3  3  3  3  3  3  3  3  3  3  3  3 -1
  _  0  1  2  2  3  3  3  3  4  4  4  4  4  4  4  4  4 -1
  b  0  1  2  2  3  3  3  3  4  4  4  4  4  4  4  4  4 -1
  o  0  1  2  2 -1  4  4  4  4  4  4  5  5  5  5  5  5 -1
  t  0  1  2  2  3  4  4  4  4  4  4  5  5  5  5  5  5 -1
  t -1 -1 -1 -1  3  4  4  4  4  4  4  5  5  5  5  5  5 -1
  l -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  6  6  6  6 -1
  e -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  7
```

	n	e	m	a	t	o	d	e	_	k	n	o	w	l	e	d	g	e
e	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
m	0	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	
p	0	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	
t	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	
y	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	
_	0	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	4	
b	0	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	4	
o	0	1	2	2		4	4	4	4	4	4	5	5	5	5	5	5	
t	0	1	2	2	3	4	4	4	4	4	4	5	5	5	5	5	5	
t					3	4	4	4	4	4	4	5	5	5	5	5	5	
l														6	6	6	6	
e																		7

33. Recover the LCS: approach

	n	e	m	a	t	o	d	e	_	k	n	o	w	l	e	d	g	e
e	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
m	0	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	
p	0	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	
t	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	
y	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	
_	0	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	4	
b	0	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	4	
o	0	1	2	2		4	4	4	4	4	4	5	5	5	5	5	5	
t	0	1	2	2	3	4	4	4	4	4	4	5	5	5	5	5	5	
t					3	4	4	4	4	4	4	5	5	5	5	5	5	
l														6	6	6	6	
e																		7

34. Recover the LCS: code

To recover the LCS from L, backtrack through the matrix.

```
function path_extract1(L, A, i, B, j)
   if (i == 0) or (j == 0) then
      return ""
   elseif A[i] == B[j] then
      return path_extract1(L, A, i-1, B, j-1) .. A[i]
   else
      local x1, x2
      if j == 1 then
         x1 = -1
      else
         x1 = L[i][j-1]
         end
      if i == 1 then
         x2 = -1
      else
         x2 = L[i-1][j]
         end
      if x1 > x2 then
         return path_extract1(L, A, i, B, j-1)
      else
         return path_extract1(L, A, i-1, B, j)
         end
      end
   end
```

35. Call the extraction

Call as follows.

```
p = lcs_6b(A, 1, #A, B, 1, #B, L)
print("p=[" .. p .. "]")
c = path_extract1(L, A, #A, B, #B)
print("c=[" .. c .. "]")
```

This is time efficient but space inefficient!

36. Efficiency

The recursive solution is very inefficient.

Solution: Memoization.

```
function lcs_5b(A, i, B, j, L)
   local p
   if (i == 0) or (j == 0) then
      p = 0
   else
      p = L[i][j]
      if p < 0 then
         if A[i] == B[j] then
            p = lcs_5b(A, i-1, B, j-1, L) + 1
         else
            local a1 = lcs_5b(A, i, B, j-1, L)
            local b1 = lcs_5b(A, i-1, B, j, L)
            p = math.max(a1, b1)
            end
         L[i][j] = p
         end
      end
   return p
   end
```

The same L matrix is computed.

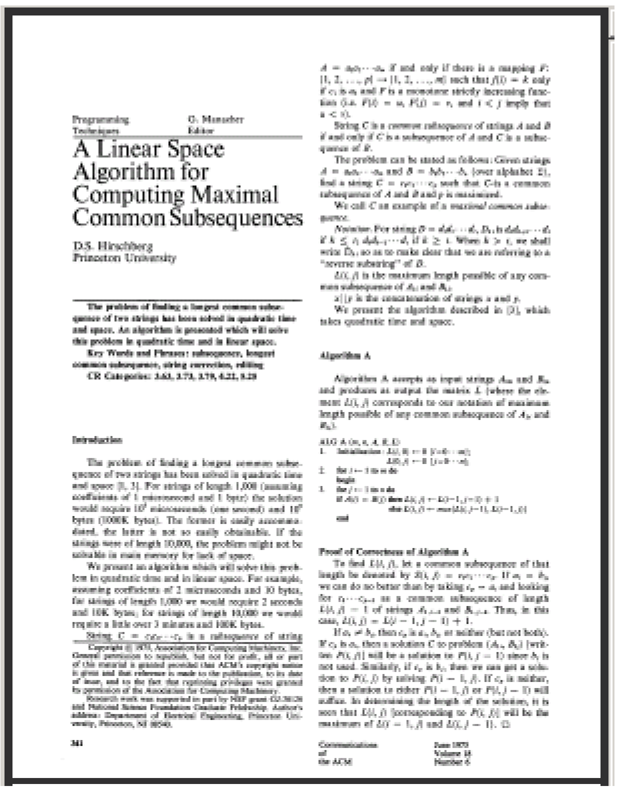
37. Add the start and stop indices

```
function lcs_6b(A, i1, i2, B, j1, j2, L)
   local p2
   if (i2 < i1) or (j2 < j1) then
      p = 0
   else
      p = L[i2][j2]
      if p < 0 then
         if A[i2] == B[j2] then
            p = lcs_6b(A, i1, i2-1, B, j1, j2-1, L) + 1
         else
            local a1 = lcs_6b(A, i1, i2, B, j1, j2-1, L)
            local b1 = lcs_6b(A, i1, i2-1, B, j1, j2, L)
            p = math.max(a1, b1)
            end
         L[i2][j2] = p
         end
      end
   return p
   end
```

The same L matrix is computed.

38. Source

Accessible 3-page paper with which to get started.


Programming Techniques — G. Manacher, Editor

A Linear Space Algorithm for Computing Maximal Common Subsequences

D.S. Hirschberg
Princeton University

The problem of finding a longest common subsequence of two strings has been solved in quadratic time and space. An algorithm is presented which will solve this problem in quadratic time and in linear space.

Key Words and Phrases: subsequence, longest common subsequence, string correction, editing

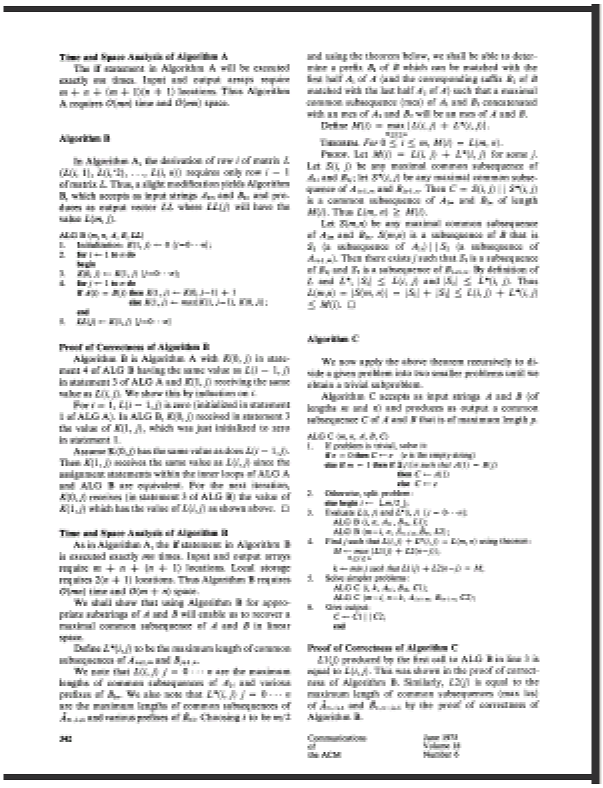
Introduction

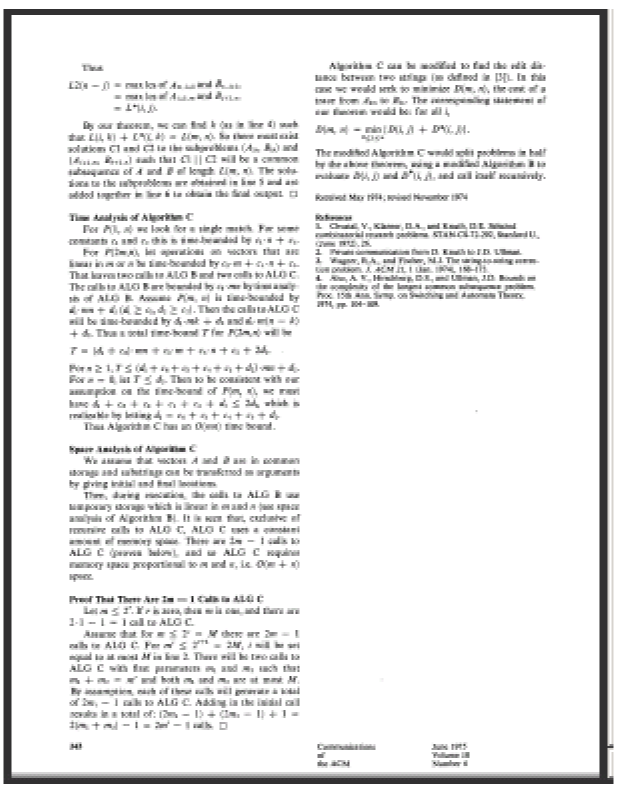
Algorithm A

Proof of Correctness of Algorithm A

341

Communications of the ACM — June 1975 Volume 18 Number 6





- **A linear space algorithm for computing maximal common subsequences**
- D. S. Hirshberg, Princeton University

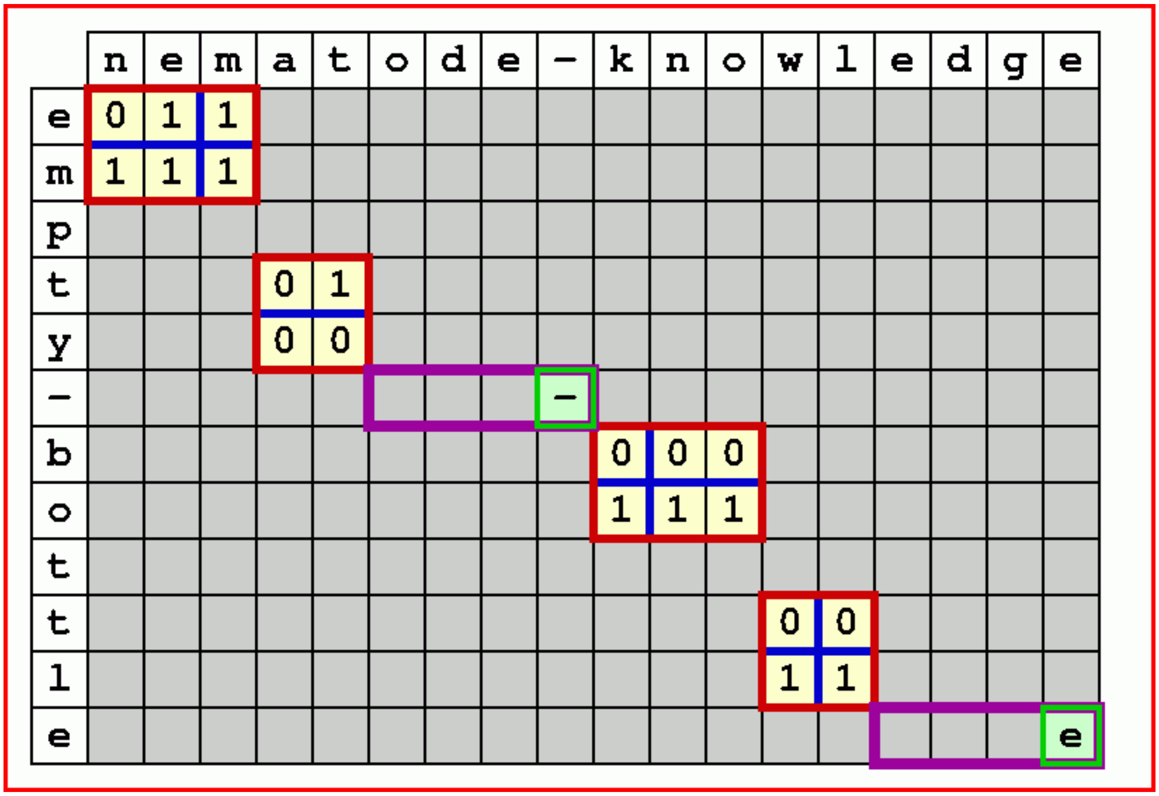
- 1975.

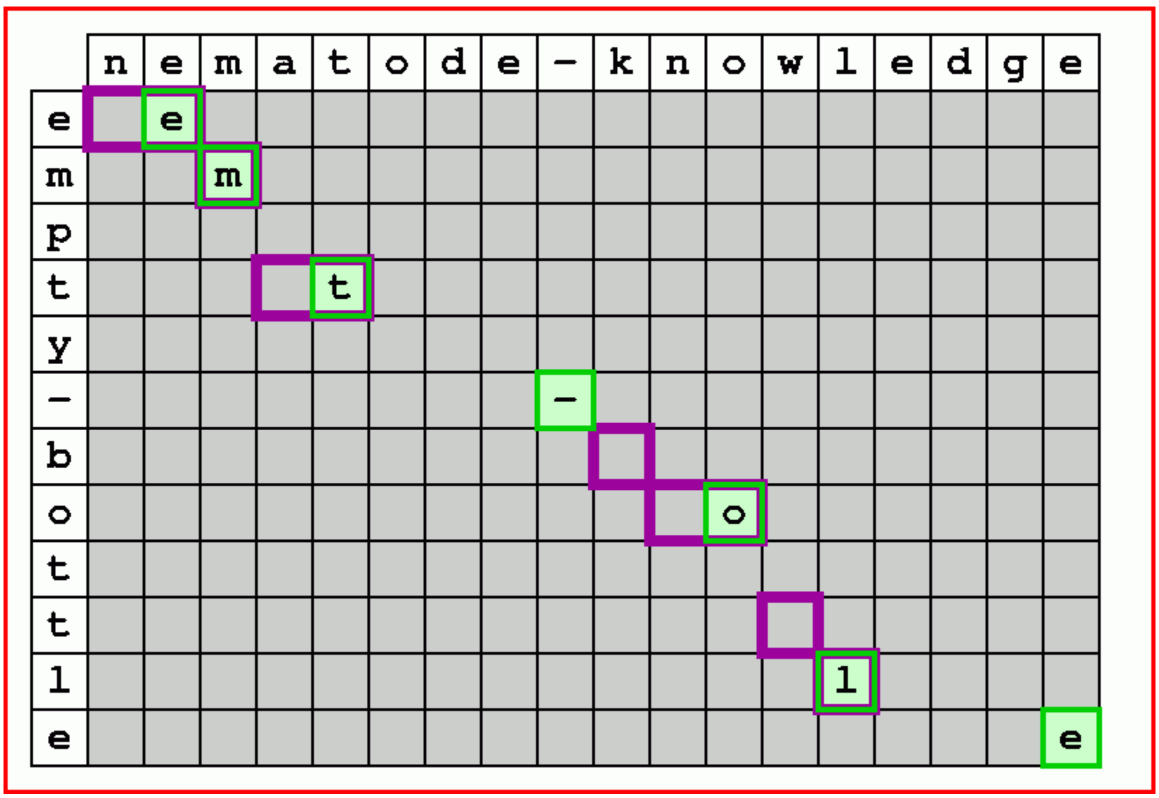
39. Hirshberg Approach

	n	e	m	a	t	o	d	e	-	k	n	o	w	l	e	d	g	e
e	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
m	0	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
p	0	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
t	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3
y	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3
-	0	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	4	4
b	3	3	3	3	3	3	3	3	3	3	3	3	2	2	1	1	1	1
o	3	3	3	3	3	3	3	3	3	3	3	3	2	2	1	1	1	1
t	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1
t	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1
l	2	2	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
e	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

	n	e	m	a	t	o	d	e	-	k	n	o	w	l	e	d	g	e
e	0	1	1	1	1	1	1	1	1									
m	0	1	2	2	2	2	2	2	2									
p	0	1	2	2	2	2	2	2	2									
t	2	2	2	2	2	1	1	1	1									
y	1	1	1	1	1	1	1	1	1									
-	1	1	1	1	1	1	1	1	1									
b										0	0	0	0	0	0	0	0	0
o										0	0	1	1	1	1	1	1	1
t										0	0	1	1	1	1	1	1	1
t										2	2	2	2	2	1	1	1	1
l										2	2	2	2	2	1	1	1	1
e										1	1	1	1	1	1	1	1	1

	n	e	m	a	t	o	d	e	-	k	n	o	w	l	e	d	g	e
e	0	1	1															
m	0	1	2															
p	0	0	0															
t				0	1	1	1	1	1									
y				0	1	1	1	1	1									
-				1	1	1	1	1	1									
b										0	0	0						
o										0	0	1						
t										0	0	0						
t													0	0	0	0	0	0
l													0	1	1	1	1	1
e													1	1	1	1	1	1





40. Hirshberg (full L, rows)

```
function dpa_traverse_4(L, A, B, i1, i3, j1, j3, dx)
   for i=i1,i3,dx do
      for j=j1,j3,dx do
         if A[i] == B[j] then
            if (i == i1) or (j == j1) then
               L[i][j] = 1
            else
               L[i][j] = 1 + L[i-dx][j-dx]
               end
         else
            local y1, y2
            if i == i1 then
               y1 = 0
            else
               y1 = L[i-dx][j]
               end
            if j == j1 then
               y2 = 0
            else
               y2 = L[i][j-dx]
               end
            L[i][j] = math.max(y1, y2)
            end
         end
      end
   end
```

41. Hirshberg (full L, main)

```
function lcs_hirschberg_4(L, A, B, i1, i3, j1, j3)
   if j1 > j3 then
      for i=i1,i3 do
         extractPut1(A[i]," ",1)
         end
   elseif i1 == i3 then
      local j2 = 0
      for j=j3,j1,-1 do
         if (A[i1] == B[j]) and (j2 == 0) then
            j2 = j
            extractPut1(A[i1],B[j],1)
         else
            extractPut1(" ",B[j],1)
            end
         end
      if j2 == 0 then
         extractPut1(A[i1]," ",1)
         end
   else
      local i2 = math.floor((i1+i3)/2)
      dpa_traverse_4(L, A, B, i1, i2, j1, j3, 1)
      dpa_traverse_4(L, A, B, i3, i2+1, j3, j1, -1)
      local j2 = j1-1
      local k1 = 0
      for j=j1,j3 do
         local k
         k = L[i2][j] + L[i2+1][j]
         if k > k1 then
            k1 = k
            j2 = j
            end
         end
      lcs_hirschberg_4(L, A, B, i1, i2, j1, j2)
      lcs_hirschberg_4(L, A, B, i2+1, i3, j2+1, j3)
      end
   end
```