

Incrementally developing and implementing Hirschberg's longest common subsequence 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 subsequence (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, Hirschberg published a reasonably space and time efficient solution to these problems. This talk will cover the incremental development and implementation of Hirschberg's algorithm in Lua, including trade-offs and design decisions along the way. The final algorithm implementation can be used for customized comparison 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 $C = c_1c_2\dots c_p$ is a *subsequence* of string $A = a_1a_2\dots a_m$ iff there is a mapping

$F: [1, 2, \dots, p]$ to $[1, 2, \dots, m]$

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 $A = a_1a_2\dots a_m$ and $B = b_1b_2\dots b_n$ find string $C = c_1c_2\dots c_p$ 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

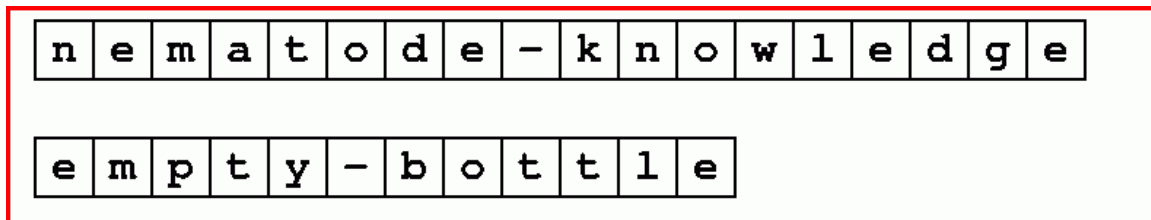
Alphabets examples:

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

7. Example strings

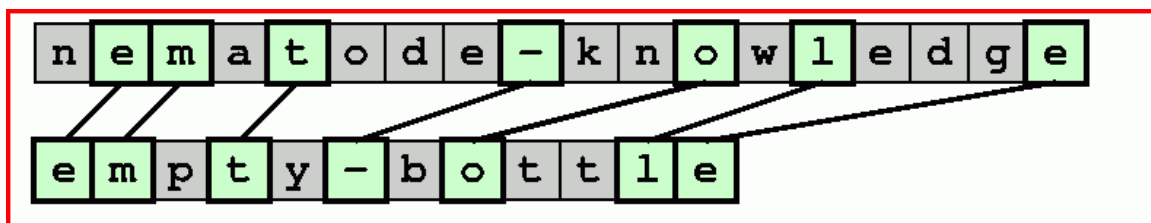
Example strings:

- $a = \text{"nematode-knowledge"}$
- $b = \text{"empty-bottle"}$



- $m = \text{string.len}(a) = \text{string.len}(\text{"nematode-knowledge"}) = 18$
- $n = \text{string.len}(b) = \text{string.len}(\text{"empty-bottle"}) = 12$

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.

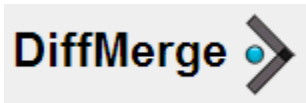
10. DNA

```
AGGCTATCACCTGACCTCCAGGCCGATGCC...  
TAGCTATCACGACCGCGGTCGATTGCCCGAC...
```

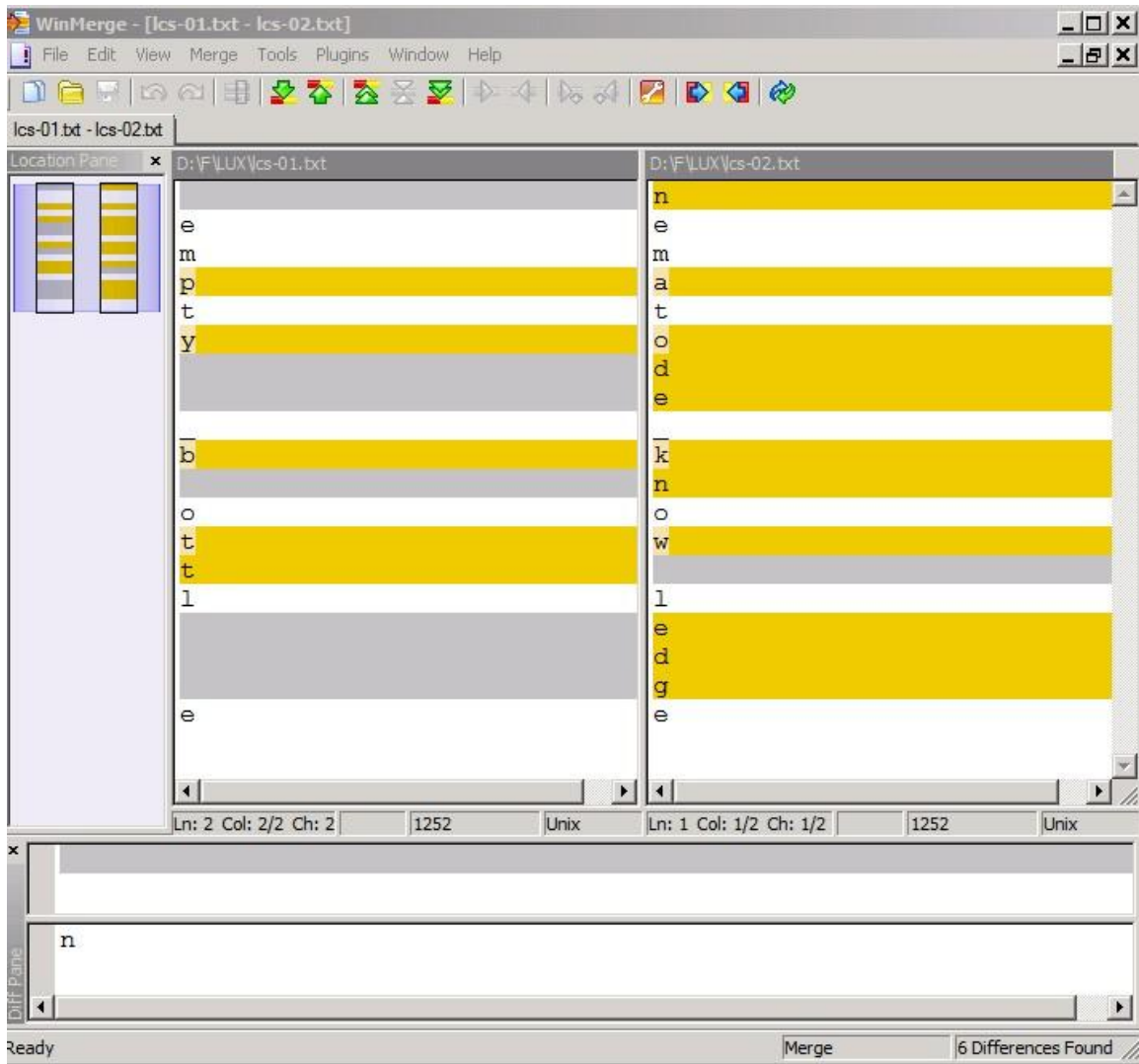
11. File comparison

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

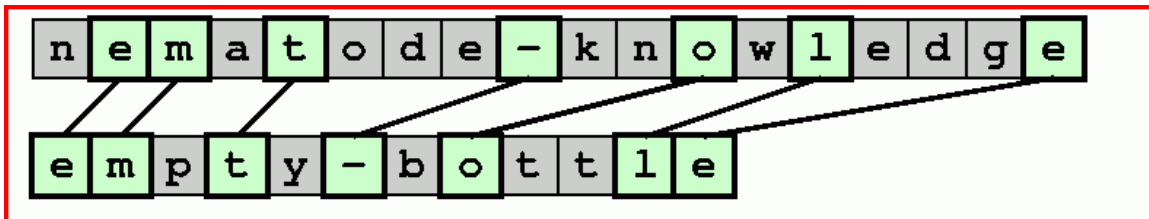
- WinMerge at <http://www.winmerge.org>.
- DiffMerge at <http://www.sourcegear.com/difmerge>.

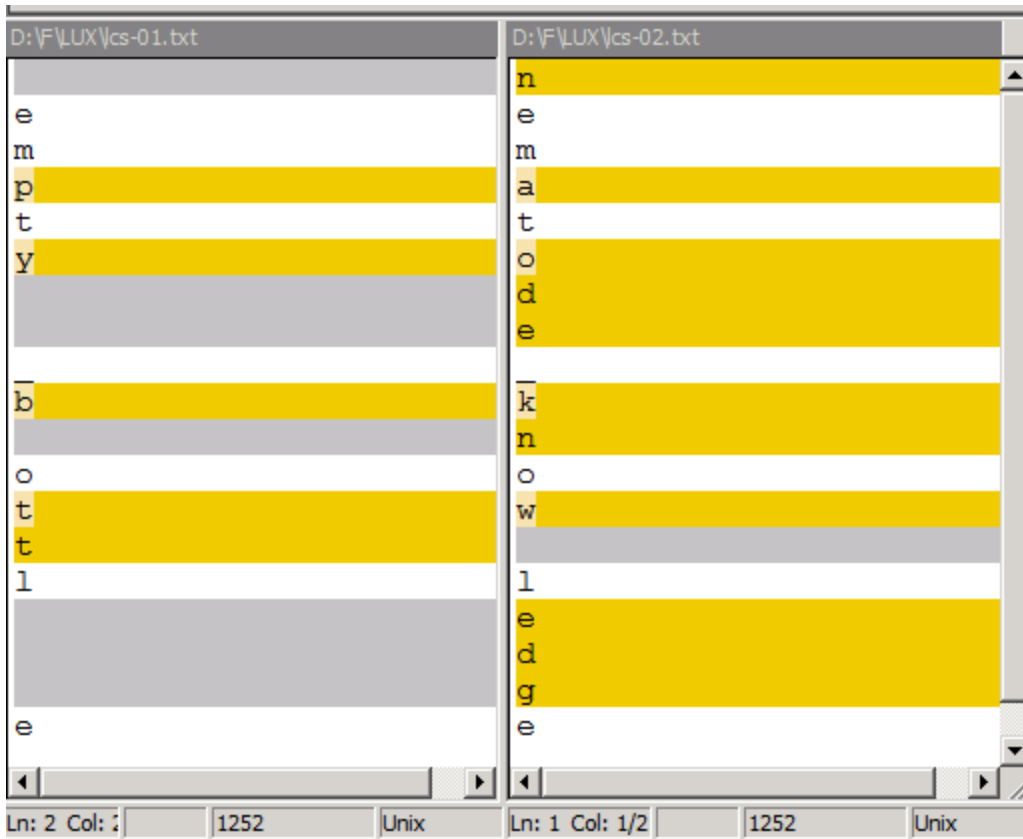


Make each letter a line in a file.



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: $m*n == 216$

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

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

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

18. Non-Match (1)

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

19. Non-Match (2)

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

20. Next

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

21. Recursive top down backward

```

a1 a2 ... am-1 am
b1 b2 ... bn-1 xn

function lcs_lb(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_lb(string.sub(a, 1, m-1), string.sub(b, 1, n-1)) .. string.sub(a, m, m)
  else
    local a1 = lcs_lb(a, string.sub(b, 1, n-1))
    local b1 = lcs_lb(string.sub(a, 1, m-1), b)
    return math.max(#a1, #b1)
  end
end

```

Time and space INEFFICIENT!!!

22. Recursive top down forward

```

a1 a2 ... am-1 am
b1 b2 ... bn-1 xn

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)
  end
end

```

```

    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
b̄ -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
```


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.

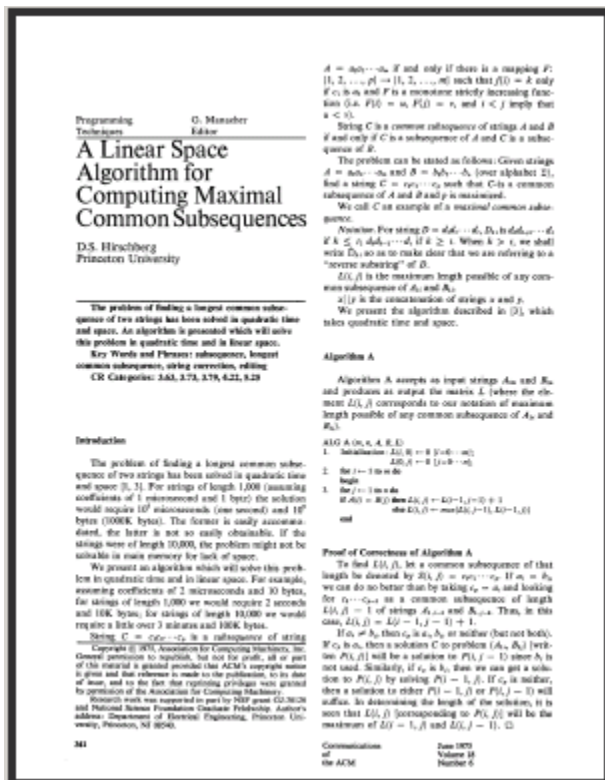
37. Add the start and stop indices

```
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
    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.



Time and Space Analysis of Algorithm A

The if statement in Algorithm A will be executed exactly one time. Input and output arrays require $m + n + (m + n) + 1$ locations. Thus Algorithm A requires $O(mn)$ time and $O(m+n)$ space.

Algorithm B

In Algorithm A, the deletion of row i of matrix L , $L(i, 1), L(i, 2), \dots, L(i, k)$ requires only row $i - 1$ of matrix L . Thus, a slight modification yields Algorithm B which accepts as input strings A_m and B_n and produces as output matrix L . When $L(i, 1)$ will have the value $L(i, 1)$.

```
ALG B (m, n, A, B, L);
1. Initialization:  $L(i, 1) = 0, i = 0 \dots m$ ;
2.  $for\ i = 1\ to\ m$ 
  begin
3.  $L(i, 1) = A(i, 1)$ ;
4.  $for\ j = 1\ to\ n$ 
  begin
5.  $L(i, j) = \max(L(i, j-1), R(i, j))$ ;
6.  $L(i, j) = \max(L(i, j), L(i, j-1) + 1)$ ;
  end
  end
```

Proof of Correctness of Algorithm B

Algorithm B is Algorithm A with $R(i, j)$ in statement 4 of ALG B having the same value as $L(i - 1, j)$ in statement 3 of ALG A and $R(i, j)$ receiving the same value as $L(i, j)$. We show this by induction on i .

For $i = 1$, $L(i - 1, j)$ is not initialized in statement 3 of ALG A; in ALG B, $R(i, j)$ received in statement 3 the value of $L(i, j)$, which was just initialized to zero in statement 1.

Assume $R(i, j)$ has the same value as does $L(i - 1, j)$. Then $R(i, j)$ receives the same value as $L(i, j)$ since the assignment statements within the inner loops of ALG A and ALG B are equivalent. For the next iteration, $R(i, j)$ receives in statement 3 of ALG B the value of $R(i, j)$ which has the value of $L(i, j)$ as shown above. \square

Time and Space Analysis of Algorithm B

As in Algorithm A, the if statement in Algorithm B is executed exactly one time. Input and output arrays require $m + n + (m + n) + 1$ locations. Local storage requires $O(m + n)$ locations. Thus Algorithm B requires $O(mn)$ time and $O(m + n)$ space.

We shall show that using Algorithm B for appropriate substrings of A and B will enable us to recover a maximal common subsequence of A and B in linear space.

Define $L^*(i, j)$ to be the maximum length of common subsequence of $A_{1..i}$ and $B_{1..j}$. We note that $L^*(i, j) = 0, i = 0 \dots m$ and various lengths of common subsequences of A_i and various prefixes of B_n . We also note that $L^*(i, j) = 0, i = 0 \dots m$ are the maximum lengths of common subsequences of $A_{1..i}$ and various prefixes of B_n . Choosing i to be $m-1$

and using the theorem below, we shall be able to determine a prefix A_i of A which can be matched with the last half A_j of A (and the corresponding suffix B_j of B matched with the last half A_i of A) such that a maximal common subsequence (mcs) of A and B concatenated with an mcs of A_i and B_j will be an mcs of A and B .

Define $M(i) = \max(L^*(i, j) + L^*(j, i))$.

Theorem. For $0 \leq i \leq m$, $M(i) = L^*(m, i)$.

Proof. Let $M(i) = L^*(i, j) + L^*(j, i)$ for some j . Let $S_1(j)$ be any maximal common subsequence of A_i and B_j in $S^*(i, j)$ and $S_2(j)$ any maximal common subsequence of $A_{i+1..m}$ and $B_{j+1..n}$. Then $C = S_1(j) \cup S_2(j)$ is a common subsequence of A_m and B_n of length $M(i)$. Thus $L^*(m, i) \geq M(i)$. Let $S(m, i)$ be any maximal common subsequence of A_m and B_n . $S(m, i)$ is a subsequence of B that is a subsequence of $A_i \cup S_1(j)$ (a subsequence of $A_{i+1..m}$). Thus there exists j such that $S_1(j)$ is a subsequence of B_j and $S_2(j)$ is a subsequence of $B_{j+1..n}$. By definition of L and L^* , $|S_1(j)| \leq L^*(i, j)$ and $|S_2(j)| \leq L^*(j, i)$. Thus $L^*(m, i) = |S(m, i)| = |S_1(j)| + |S_2(j)| \leq L^*(i, j) + L^*(j, i) \leq M(i)$. \square

Algorithm C

We now apply the above theorem recursively to divide a given problem into two smaller problems until we obtain a trivial subproblem.

Algorithm C accepts as input strings A and B (of lengths m and n) and produces as output a common subsequence C of A and B that is of maximum length p .

```
ALG C (m, n, A, B, C);
1. If problem is trivial, solve it
    $p = \max(m, n)$ ;
    $C = A$  if  $m \geq n$  else  $C = B$ ;
   exit
2. Otherwise, split problem:
    $split\ i = \lfloor m/2 \rfloor$ ;
    $split\ j = \lfloor n/2 \rfloor$ ;
   ALG B (i, A_i, B_j, L);
   ALG B (m-i+1, A_{i+1..m}, B_{j+1..n}, L);
3. Find  $i$  such that  $L^*(i, j) + L^*(j, i) = M$  using theorem;
    $M = \max(L^*(i, j) + L^*(j, i))$ ;
4.  $k = min(i, j)$  and find  $L^*(k, j)$  and  $L^*(j, k)$ ;
   ALG B (k, A_k, B_k, C1);
   ALG C (m-k+1, A_{k+1..m}, B_{k+1..n}, C2);
5. One output:
    $C = C1 \cup C2$ ;
   exit
```

Proof of Correctness of Algorithm C

$L^*(i, j)$ produced by the first call to ALG B in line 3 is equal to $L^*(i, j)$. This was shown in the proof of correctness of Algorithm B. Similarly, $L^*(j, i)$ is equal to the maximum length of common subsequences (mcs) of $A_{i+1..m}$ and $B_{j+1..n}$ by the proof of correctness of Algorithm B.

Communications of the ACM, June 1975, Volume 18, Number 6

Thus

$$L^*(i, j) = \max(\max(L^*(i, j) \text{ and } R^*(i, j)), \max(L^*(i, j) \text{ and } R^*(i, j))) = L^*(i, j).$$

By our theorem, we can find k (as in line 4) such that $L^*(i, k) + L^*(k, j) = M$. So these maximal solution C1 and C2 in the subproblems (A_k, B_k) and $(A_{k+1..m}, B_{k+1..n})$ will be a common subsequence of A and B of length M . The solutions to the subproblems are obtained in line 3 and are added together in line 5 to obtain the final output. \square

Time Analysis of Algorithm C

For $P(i, j)$ we look for a single match. For some constants c_1 and c_2 , this is time-bounded by $c_1 + c_2$. For $P(m, n)$, we operate on vectors that are time-bounded by $c_1 m + c_2 n + c_3$. That takes two calls to ALG B and two calls to ALG C. The calls to ALG B are bounded by c_1 one by time analysis of ALG B. Assume $P(m, n)$ is time-bounded by d ; we $m + n$ ($d \geq c_1 m + c_2 n$). Then the calls to ALG C will be time-bounded by $d, m \geq d$, and $d, m \leq d$. Thus a total time-bound T for $P(m, n)$ will be $T = |d + c_1 m + c_2 n + c_3 + c_1 m + c_2 n| + 2d$.

For $m \geq n$, $T \leq (d + c_1 + c_2 + c_3 + d) m + d$. For $m < n$, let $T \leq d$. Then to be consistent with our assumption on the time-bound of $P(m, n)$, we must have $d + c_1 m + c_2 n + c_3 + d \leq d$, which is resolved by letting $d = c_1 m + c_2 n + c_3 + d$. Thus Algorithm C has an $O(mn)$ time bound.

Space Analysis of Algorithm C

We assume that vectors A and B are in constant storage and subranges can be transferred in arguments by giving initial and final locations.

Then, during recursion, the calls to ALG B use temporary storage which is linear in m and n (see space analysis of Algorithm B). It is seen that, exclusive of recursive calls to ALG C, ALG C uses a constant amount of memory space. There are $2m - 1$ calls to ALG C (shown below), and so ALG C requires memory space proportional to m and n , i.e. $O(m + n)$ space.

Proof That There Are $2m - 1$ Calls to ALG C

Let $m \leq 2^r$. For r is even, then m is even, and there are $2^{r-1} - 1$ calls to ALG C.

Assume that for $m \leq 2^r = M$ there are $2m - 1$ calls to ALG C. For $m \leq 2^{r+1} = 2M$, j will be set equal to at most M in line 3. There will be two calls to ALG C with four parameters m_1 and n_1 such that $m_1 + n_1 = m$ and both m_1 and n_1 are at most M . By assumption, each of these calls will generate a total of $2m_1 - 1$ calls to ALG C. Adding in the initial call results in a total of $(2m_1 - 1) + (2m_1 - 1) + 1 = 2m_1 + m_1 - 1 = 2m - 1$ calls. \square

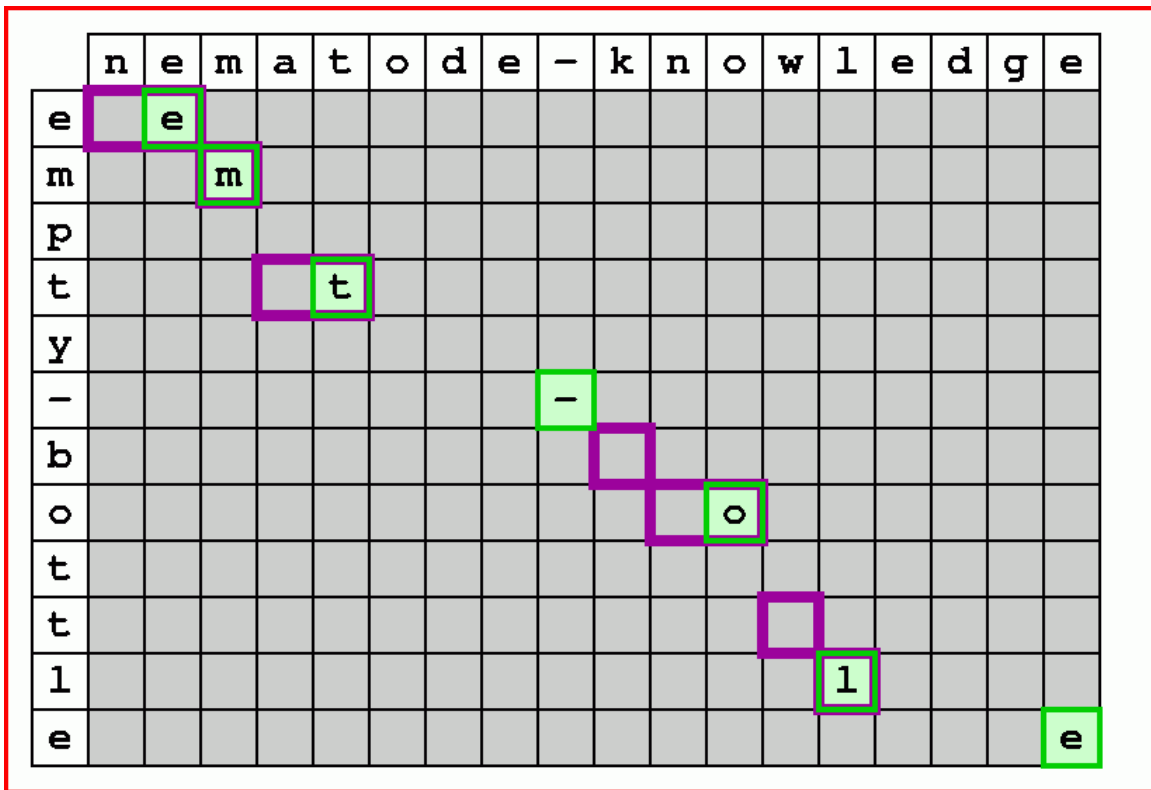
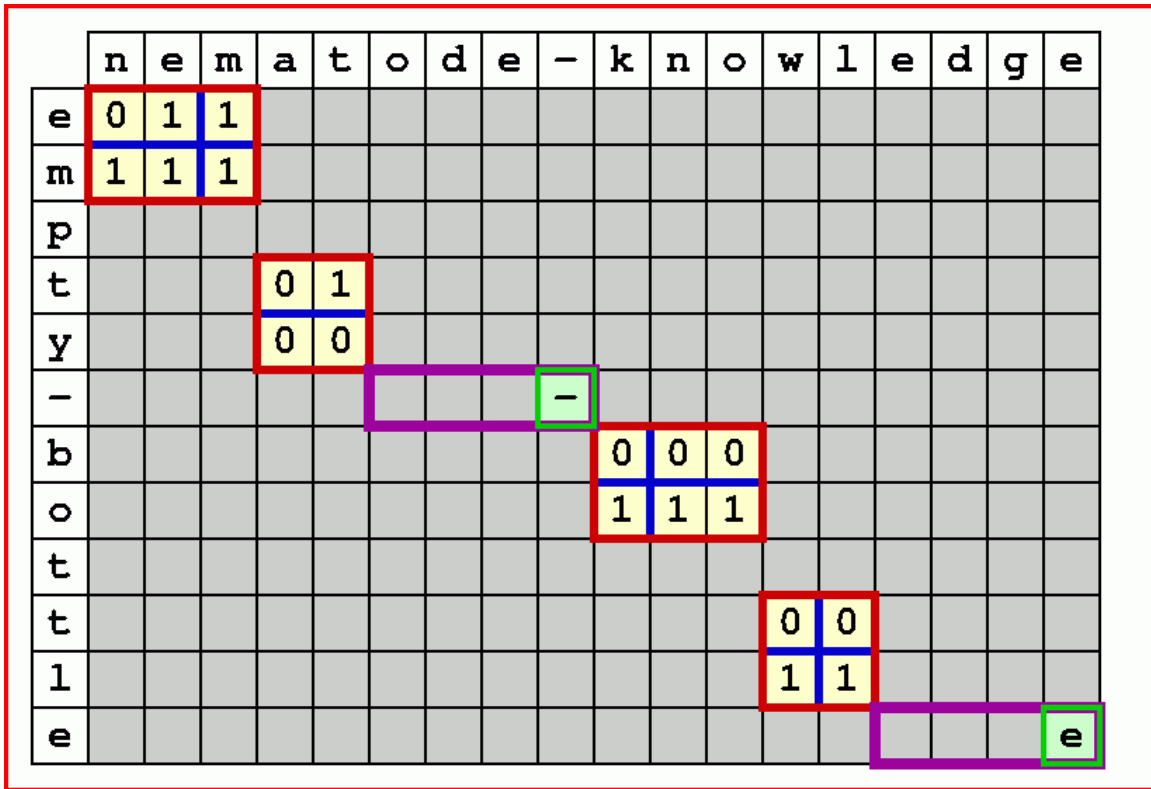
181

Communications of the ACM, July 1975, Volume 18, Number 7

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

	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

```