Algorithms for the Longest Common Subsequence Problem

DANIEL S. HIRSCHBERG

Princeton University, Princeton, New Jersey

ABSTRACT Two algorithms are presented that solve the longest common subsequence problem The first algorithm is applicable in the general case and requires $O(pn + n \log n)$ time where p is the length of the longest common subsequence The second algorithm requires time bounded by $O(p(m + 1 - p)\log n)$ In the common special case where p is close to m, this algorithm takes much less time than n^2

KEY WORDS AND PHRASES: subsequence, common subsequence, algorithm

CR CATEGORIES 3 73, 3 79, 5 25, 5 39

Introduction

We start by defining conventions and terminology that will be used throughout this paper.

String $C = c_1c_2 \cdots c_p$ is a *subsequence* of string $A = a_1a_2 \cdots a_m$ if there is a mapping $F:\{1, 2, \ldots, p\} \rightarrow \{1, 2, \ldots, m\}$ such that $F(i) = k$ only if $c_i = a_k$ and F is a monotone strictly increasing function (i.e. $F(i) = u$, $F(j) = v$, and $i < j$ imply that $u < v$). C can be formed by deleting $m - p$ (not necessarily adjacent) symbols from A. For example, "course" is a subsequence of "computer science."

String C is a *common subsequence* of strings A and B if C is a subsequence of A and also a subsequence of B.

String C is a *longest common subsequence* (abbreviated LCS) of string A and B if C is a common subsequence of A and B of maximal length, i.e. there is no common subsequence of A and B that has greater length.

Throughout this paper, we assume that A and B are strings of lengths m and n, $m \leq n$, that have an LCS C of (unknown) length p.

We assume that the symbols that may appear in these strings come from some alphabet of size t. A symbol can be stored in memory by using $\log t$ bits, which we assume will fit in one word of memory. Symbols can be compared ($a \leq b$?) in one time unit.

The number of different symbols that actually appear in string B is defined to be s (which must be less than n and t).

The longest common subsequence problem has been solved by using a recursion relationship on the length of the solution [7, 12, 16, 21]. These are generally applicable algorithms that take $O(mn)$ time for any input strings of lengths m and n even though the lower bound on time of $O(mn)$ need not apply to all inputs [2]. We present algorithms that, depending on the nature of the input, may not require quadratic time to recover an LCS. The first algorithm is applicable in the general case and requires $O(pn + n \log n)$ time. The second algorithm requires time bounded by $O((m + 1 - p)p \log n)$. In the common special case where p is close to m, this algorithm takes time

This research was supported by a National Science Foundation graduate fellowship and by the National Science Foundation under Grant GJ-35570.

Author's present address: Department of Electrical Engineering, Rice University, Houston, TX 77001

much less than n^2. We conclude with references to other algorithms for the LCS problem that may be of interest.

pn Algorithm

We present in this section algorithm $ALGD$, which will find an LCS in time $O(pn + n \log n)$ where p is the length of the LCS. Thus this algorithm may be preferred for applications where the expected length of an LCS is small relative to the lengths of the input strings.

Some preliminary definitions are as follows:

We represent the concatenation of strings X and Y by $X \| Y$.

A_{1i} represents the string $a_1 a_2 \cdots a_i$ (elements 1 through i of string A). Similarly, the prefix of length j of string B is represented by B_{1j}.

We define $L(i, j)$ to be the length of the LCS of prefixes of lengths i and j of strings A and B, i e. the length of the LCS of A_{1i} and B_{1j}.

$\langle i, j \rangle$ represents the positions of a_i and b_j, the ith element of string A and the jth element of string B. We refer to i (j) as the i-value (j-value) of $\langle 1, j \rangle$.

We define $\{\langle 0, 0 \rangle\}$ to be the set of 0-*candidates*, and we define $\langle i, j \rangle$ to be a k-*candidate* (for $k \geq 1$) if $a_i = b_j$ and there exist i' and j' such that $i' < i, j' < j$, and $\langle i', j' \rangle$ is a $(k-1)$-candidate. We say that $\langle i', j' \rangle$ *generates* $\langle i, j \rangle$.

Define $a_0 = b_0 =$ \$ where \$ is some symbol that does not appear in strings A or B.

LEMMA 1. *For $k \geq 1$, $\langle i, j \rangle$ is a k-candidate iff $L(i, j) \geq k$ and $a_i = b_j$. Thus there is a common subsequence of length k of A_{1i} and B_{1j}.*

PROOF. By induction on k. $\langle i, j \rangle$ is a 1-candidate iff $a_i = b_j$ (by definition), in which case $L(i, j)$ necessarily is at least 1. Thus the lemma is true for $k = 1$. Assume it is true for $k - 1$. Consider k. If $\langle i, j \rangle$ is a k-candidate then there exist $i' < i$ and $j' < j$ such that $\langle i', j' \rangle$ is a $(k-1)$-candidate. By assumption, there is a common subsequence $D' = d_1 d_2 \cdots d_{k-1}$ of $A_{1i'}$ and $B_{1j'}$. Since $a_i = b_j$ ($\langle i, j \rangle$ is a k-candidate), $D = D' \| a_i$ is a common subsequence of length k of A_{1i} and B_{1j}. Thus $L(i, j) \geq k$.

Conversely, if $L(i, j) \geq k$ and $a_i = b_j$, then there exist $i' < i$ and $j' < j$ such that $a_{i'} = b_{j'}$ and $L(i', j') = L(i, j) - 1 \geq k - 1$. $\langle i', j' \rangle$ is a $(k-1)$-candidate (by inductive hypothesis) and thus $\langle i, j \rangle$ is a k-candidate. □

The length of an LCS is p, the maximum value of k such that there exists a k-candidate. As we shall see, to recover an LCS, it suffices to maintain the sequence of a 0-candidate, 1-candidate, ... , $(p-1)$-candidate, and a p-candidate such that in this sequence each i-candidate can generate the $(i+1)$-candidate for $0 \leq i < p$.

Rule. Let $x = \langle x_1, x_2 \rangle$ and $y = \langle y_1, y_2 \rangle$ be two k-candidates. If $x_1 \geq y_1$ and $x_2 \geq y_2$, then we say that y *rules out* x (x is a superfluous k-candidate) since any $(k+1)$-candidate that could be generated by x can also be generated by y. Thus, from the set of k-candidates, we need consider only those that are minimal under the usual vector ordering. Note that if x and y are minimal elements then $x_1 < y_1$ iff $x_2 > y_2$.

LEMMA 2. *Let the set of k-candidates be $\{\langle i_r, j_r \rangle\}$ ($r = 1, 2, \ldots$). We can rule out candidates so that (after renumbering) $i_1 < i_2 < \cdots$ and $j_1 > j_2 > \cdots$.*

PROOF. Any two k-candidates $\langle i, j \rangle$ and $\langle i', j' \rangle$ satisfy one of the following (without loss of generality, $i \leq i'$):

(1) $i < i'$, $j \leq j'$.
(2) $i < i'$, $j > j'$.
(3) $i = i'$, $j \leq j'$.
(4) $i = i'$, $j > j'$.

In cases (1) and (3) $\langle i', j' \rangle$ can be ruled out; in case (4) $\langle i, j \rangle$ can be ruled out; and case (2) satisfies the statement of the lemma. Thus any set of k-candidates which cannot be reduced by further application of the rule will satisfy the condition stated in the lemma. □

The set of k-candidates, reduced by application of the rule so as to satisfy the statement of Lemma 2, are the minimal elements of the set of k-candidates (since no

element can rule out a minimal element) and will be called the set of *minimal k-candidates*. By Lemma 2, there is at most one minimal k-candidate for each i-value.

We note that if $\langle i, j\rangle$ is a minimal k-candidate then $L(i, j) = k$ and $\langle i, j\rangle$ is the k-candidate with i-value i having smallest j-value j such that $L(i, j) = k$.

LEMMA 3. *For $k \geq 1$, $\langle i, j\rangle$ is a minimal k-candidate iff j is the minimum value such that $b_j = a_i$ and $low < j < high$, where high is the minimum j-value of all k-candidates whose i-value is less than i (no upper limit if there are no such k-candidates) and low is the minimum j-value of all $(k-1)$-candidates whose i-value is less than i.*

PROOF. Assume that $\langle i, j\rangle$ is a minimal k-candidate. If $j \geq high$ then there is a k-candidate $\langle i', j'\rangle$ such that $i' < i$ and $j' = high \leq j$. $\langle i, j\rangle$ would be ruled out by $\langle i', j'\rangle$ and thus would not be minimal.

If $j \leq low$, then there is no $(k-1)$-candidate that can generate $\langle i, j\rangle$. $\langle i, j\rangle$ would not be a k-candidate.

$b_j = a_i$ is required by the definition of k-candidate and $low < j < high$ has just been shown. If j and j' both satisfy these constraints, $j < j'$, then $\langle i, j'\rangle$ is ruled out by $\langle i, j\rangle$. Thus, for a particular i, j must be the minimum j-value of all k-candidates satisfying these constraints.

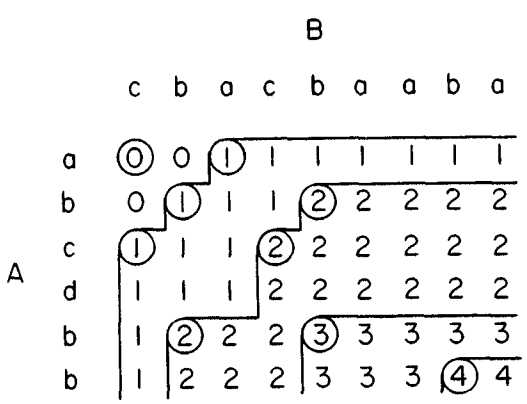
The *if* of the lemma has thus been shown.

The converse is easily shown: If $\langle i, j\rangle$ is not a k-candidate, then either $a_i \neq b_j$ or there is no $(k-1)$-candidate that can generate $\langle i, j\rangle$. That is, the j-value of all $(k-1)$-candidates with i-value less than i is greater than or equal to j. This is equivalent to $j \leq low$.

If $\langle i, j\rangle$ is a k-candidate but is not minimal, say $\langle i', j'\rangle$ rules out $\langle i, j\rangle$, then $i' \leq i$ and $j' \leq j$. If $i' < i$, then clearly $j < high$ is violated. Otherwise, $i' = i$. In this case $j' > low$ since $\langle i', j'\rangle$ must be generated from a $(k-1)$-candidate and $b_{j'} = a_i$ since $\langle i', j\rangle$ is a k-candidate. Also $j' < j < high$. Thus j' satisfies all the constraints and j is not the minimum value that does so, a contradiction. □

We present algorithm *ALGD*, which, using the results of Lemma 3, obtains an LCS C of length p of input strings A and B in time $O(pn + n \log n)$.

The algorithm is based on an efficient representation of the L matrix. Since L is nondecreasing in both arguments, we may draw contours in its matrix as shown in the following example:



The entire matrix is specified by its contours. The contours are described by sets of minimal k-candidates. The contour between L-values of $k-1$ and k is defined by the set of minimal k-candidates whose elements are positioned at the convex corners of the contour.

To keep track of the minimal k-candidates, we use the matrix D. $D[k, i]$ is the j-value of the unique minimal k-candidate having i-value of i or 0 if there is no such minimal k-candidate. Thus $D[k, i]$ describes the contours by giving the number of the first column of row i that is in region k (if that number is different from $D[k, i-1]$).

lowcheck is the smallest i-value of a $(k-1)$-candidate. *FLAG* has value 1 iff there are any k-candidates.

$NB[\theta]$ is the number of times symbol θ occurs in string B. $PB[\theta, 1], \ldots, PB[\theta, NB[\theta]]$ is the ordered list, smallest first, of positions in B in which symbol θ occurs.

If t, the size of the symbol alphabet, is not large compared to n, then we may index an array by the bit representation of a symbol. Otherwise, if $t \gg n$, then we construct a balanced binary search tree which provides a mapping from symbols that appear in string B to the integers 1 through s (there are s different symbols that appear in B). Whenever string element a_i appears as an array subscript (as in $N[a_i]$), it should be understood that we are indexing N by the integer s_i which has been obtained (during initialization for *ALGD*) from traversing the search tree just described. If a_i does not appear in B, then the integer s_i is zero. An equivalent assumption is followed for subscript b_j in step 1.

```
ALGD(m, n, A, B, C, p)
 1. NB[θ] ← 0 for θ = 1, . , s
    PB[θ, 0] ← 0 for θ = 1, . , s
    PB[0, 0] ← 0; PB[0, 1] ← 0
    for j ← 1 step 1 until n do
    begin
      NB[b_j] ← NB[b_j] + 1
      PB[b_j, NB[b_j]] ← j
    end
 2  D[0, i] ← 0 for i = 0, .. , m
    lowcheck ← 0
 3. for k ← 1 step 1 do
    begin
 4.   N[θ] ← NB[θ] for θ = 1,    , s
      N[0] ← 1
      FLAG ← 0
      low ← D[k − 1, lowcheck]
      high ← n + 1
 5.   for i ← lowcheck + 1 step 1 until m do
      begin
 6.     while PB[a_i, N[a_i] − 1] > low do N[a_i] ← N[a_i] − 1
 7      if high > PB[a_i, N[a_i]] > low
          then begin
            high ← PB[a_i, N[a_i]]
            D[k, i] ← high
            if FLAG = 0 then {lowcheck ← i, FLAG ← 1}
          end
          else D[k, i] ← 0
 8      if D[k − 1, i] > 0 then low ← D[k − 1, i]
      end loop of step 5
 9.   if FLAG = 0 then go to step 10
    end loop of step 3
10  p ← k − 1
    k ← p
    for i ← m + 1 step − 1 until 0 do
    if D[k, i] > 0 then
    begin
      c_k ← a_i
      k ← k − 1
    end
```

The loop of step 3 evaluates the set of minimal k-candidates for $k = 1, 2, \ldots$. The loop of step 5 evaluates the set of minimal k-candidates, smallest i-value first, and fills in the D array accordingly (in the example given previously this is left-to-right) while scanning the chains of occurrences of a given character in B with largest j-value first (right-to-left). For each i, i can be the i-value of a minimal k-candidate if there is a j satisfying the constraints of Lemma 3. This is tested by determining the minimum j-value of symbol a_i that is greater than *low*. If that value is less than *high*, then $\langle i, j \rangle$ is a minimal k-candidate.

There can be no k-candidate with i-value less than or equal to *lowcheck*, so the loop of step 5 begins at *lowcheck* + 1. *lowcheck* is set, in step 7, when the first minimal k-candidate (that having smallest i-value of all k-candidates) is determined.

LEMMA 4. *ALGD evaluates the correct values of high and low (as defined in Lemma 3) for determining whether each k-candidate $\langle i, j \rangle$ is minimal.*

PROOF. *high* is supposed to be the minimum j-value of all k-candidates with i-value less than i *high* is initialized at $n + 1$ (i.e. does not limit) in step 4, before any k-candidates have been generated. Thereafter, if any k-candidates are found to be minimal (in step 7), then, since the j-values of minimal k-candidates decrease as the i-values increase, the minimum j-value of all minimal k-candidates with i-value less than i will be the j-value of the minimal k-candidate with greatest i-value less than i (i.e. the last one found, since we generate minimal k-candidates in order of increasing i-value). The j-values of ruled-out (nonminimal) k-candidates cannot be smaller than the j-value of the last minimal k-candidate *high* is updated to the most recent j-value each time a new minimal k-candidate is found in step 7. Thus *high* has value as defined in Lemma 3.

low is supposed to be the minimum j-value of all $(k - 1)$-candidates whose i-value is less than i. Again, since j-values decrease as i-values increase, *low* should be the j-value of the $(k - 1)$-candidate whose i-value is as great as possible but less than i. *low* is initialized in step 4 to be the j-value of the first (lowest i-value) $(k - 1)$-candidate. As i increases, if there was a minimal $(k - 1)$-candidate with i-value of i, then the minimum permissible j-value will decrease and *low* is updated (in step 8) for the next iteration. □

LEMMA 5. *ALGD correctly determines the set of minimal k-candidates.*

PROOF. By Lemma 4, *high* and *low* are computed correctly. We must show that in the loop of steps 5–8 $D[k, i]$ gets the minimum j-value (0 if none) such that $b_j = a_i$ and $low < j < high$.

The j-values of successive minimal k-candidates decrease in value since their i-values increase. In looking for $D[k, i]$ we look for a match for symbol a_i in string B, and we can restrict our attention to occurrences (j-values) of symbol a_i in string B that are before (less than) the last occurrence (j-value) that was examined. Step 6 does that. $PB[a_i, \cdot]$ is the ordered list of j-values of symbol a_i and $N[a_i]$ points to the smallest j-values (in PB) of symbol a_i that has been examined. Initially, in step 4, $N[a_i]$ points to the last occurrence of symbol a_i. If the last-examined j-value of a_i is greater than *low*, step 6 sets $N[a_i]$ to point to the lowest j-value of a_i that is greater than *low*. If the last-examined j-value of a_i is not greater than *low*, then there can be no minimal k-candidate for this value of i since the minimum j-value that is greater than *low* either violates the high constraint or results in a candidate that can be ruled out. In this case step 6 does nothing, the test in step 7 fails, and $D[k, i]$ is set to zero. □

THEOREM 1. *ALGD correctly computes the LCS of strings A and B.*

PROOF. By Lemma 5, *ALGD* correctly determines the set of minimal k-candidates. Thus, if there are any k-candidates, at least one is minimal. If $\langle i, j \rangle$ is the pth match in an LCS which is of length p, then, by Lemma 1, $\langle i, j \rangle$ is a p-candidate. Thus there is at least one minimal p-candidate (and there are no $(p + 1)$-candidates). Step 10 of *ALGD* recovers a common subsequence of length p by recovering a sequence of (i-values of) minimal candidates such that the minimal k-candidate generated the minimal $(k + 1)$-candidate. □

THEOREM 2. *Assuming that symbols can be compared in one time unit, ALGD requires time of $O(pn + n \log s)$, where s is the number of different symbols that appear in string B.*

PROOF. Step 1 can be done in time $O(n \log s)$. Step 2 can be done in time $O(m)$. Step 3 executes steps 4–9 p times. Step 4 takes time $O(s)$ per execution, $s \leq n$, for total time less than or equal to $O(pn)$. Step 5 executes steps 6–8 at most m times, a total of at most pm times. The **while** loop in step 6 is executed at most n times within the loop of step 5 since the $N[\theta]$ are not increased within this loop (each position of B is examined at most once for each value of k). The total time in step 6 is therefore $O(pn)$

Steps 7 and 8 are done in constant time. Total time is $O(pm)$. Step 9 is done in constant time. Total time is $O(p)$. Step 10 is done in time $O(m)$. Total execution time is thus as stated above. □

Note that for $p \geq O(\log s)$, *ALGD* requires time $O(pn)$.

pe log n Algorithm

We now consider a special case that often occurs in applications such as determining the discrepancies between two files, one of which was obtained by making minor alterations to the other (and we wish to recover those alterations). We assume that there is an LCS of length at least $m - \epsilon$ (for some given ϵ).

If C is an LCS of A and B, there will be at most ϵ elements of A that do not appear in C. The position of each such element will be called a *skipped position.* Thus there are at most ϵ skipped positions. We define e to be $\epsilon + 1$.

If $\langle i, j\rangle$ is a minimal k-candidate that can be an element in an LCS (that is, $a_i = b_j$ is the kth element of an LCS), then $k \leq i \leq k + \epsilon$ (otherwise more than ϵ positions in A would be skipped). We shall call such candidates *feasible k-candidates.* Let $h = i - k$. Then $0 \leq h \leq \epsilon$ and h is the number of positions in A that have been skipped thus far (through a_{k+h}). By Lemma 2, there is at most one feasible k-candidate with i-value of i.

Let the feasible k-candidate pairs (i-value and j-value) be held in arrays F and G, e.g. $\langle h + k, j\rangle$ would be described by $F[h] = h + k$, $G[h] = j$. If there is no feasible k-candidate with i-value $h + k$, let $F[h] = F[h - 1]$, $G[h] = G[h - 1]$, and define $F[-1] = 0$, $G[-1] = n + 1$. By this construction and by Lemma 2, F is a nondecreasing sequence and G is a nonincreasing sequence.

Define $NEXTB(\theta, j)$ to be the minimum $r > j$ such that $b_r = \theta$. If there is no such r, then $NEXTB(\theta, j)$ is defined to be $n + 1$.

LEMMA 6. *If $\langle i, j\rangle$ is a feasible k-candidate, then $j = NEXTB(a_i, G[h])$, where $h = i - k$ and where $G[h]$ is the value associated with the set of feasible $(k - 1)$-candidates.*

PROOF. Let $\langle i, j\rangle$ be a feasible k-candidate. By definition of k-candidate, there must exist $i' < i$ and $j' < j$ such that $\langle i', j'\rangle$ is a feasible $(k - 1)$-candidate. By Lemma 3, j is the minimum (over possible j') of $NEXTB(a_i, j')$. But $j'' < j'$ implies that $NEXTB(\theta, j'') \leq NEXTB(\theta, j')$. Therefore $j = NEXTB(a_i,$ min possible $j')$. Since j-values of minimal k-candidates decrease as their i-values increase, the minimum possible j' is the j-value of the feasible $(k - 1)$-candidate whose i-value is as large as possible but less than $i = h + k$, i.e. not more than $h + (k - 1)$. $G[h]$ is precisely that j-value. So we conclude that $j = NEXTB(a_i, G[h])$. □

In order to be able to recover an LCS, we shall keep track (for each feasible k-candidate) of which h positions in A have been skipped. A straightforward method, keeping values of $F[h]$ for all h and k, requires space of $O(pc)$. We shall use a data structure that requires only $O(e^2 + n)$ space without changing the order of magnitude of time requirements.

Let there be an array *KEEP* whose elements are triples such that

$$KEEP[x] = \langle aa[x], nskip[x], pt[x]\rangle.$$

P is an array of size e such that, after the set of feasible k-candidates has been determined, $x = P[h]$ will be the index of the element of *KEEP* that has information enabling recovery of a common subsequence that has $a_{F[h]} = b_{G[h]}$ as its kth element. $F[h] = h + k$, and thus precisely h of the elements $a_i, \ldots, a_{F[h]}$ will not appear in the common subsequence. To recover the common subsequence, it is sufficient to recover these h skipped positions. If $x = 0$, then no positions were skipped, and if $x < 0$, then there is no common subsequence to be recovered.

The method of recovery is as follows:

If x is zero, there are no more skipped positions to be recovered.

Otherwise, $aa[x]$ is the largest index of a skipped position in string A. $nskip[x]$ is the number of consecutive positions ending in $aa[x]$, all of which are skipped positions.

If all of the skipped A-positions have been recovered, then $pt[x]$ is zero.

Otherwise, $pt[x]$ is the index of $KEEP$ that has information enabling recovery of the skipped A-positions having indices smaller than $aa[x] - nskip[x] + 1$.

Example. If positions 2, 5, 6, 7, 9, 10 in string A correspond to a common subsequence of length 6 (of $A_{1,10}$), then $h = 4$ and $KEEP[P[4]]$ will enable recovery of positions 1, 3, 4, 8: $aa[P[4]] = 8$, $nskip[P[4]] = 1$, $pt[P[4]] = y$ (another index of $KEEP$). $aa[y] = 4$, $nskip[y] = 2$ (positions 3 and 4 have been skipped), $pt[y] = z$. $aa[z] = 1$, $nskip[z] = 1$, $pt[z] = 0$ (all skipped positions have been recovered).

Reference counts are kept for each element of $KEEP$. Spaces in the $KEEP$ array are maintained by garbage collection functions $GETSPACE$ which provides an available space and $PUTSPACE$ which places a newly available space (i.e. one whose reference count drops to zero) on the garbage linked list. See [10] for implementation techniques.

We now present $ALGE$, which uses Lemma 6 in order to solve the LCS problem in time $O(pe \log n)$:

```
ALGE (m, n, A, B, C, p, ε)
 1  F[h], G[h] ← 0 for h = 0, .. , ε
    P[0] ← 0; P[h] ← −1 for h = 1,    , ε
 2  for k ← 1 step 1 while there were candidates found in the last pass do
    begin
 3    imax ← 0
      jmin ← n + 1
 4    for h ← 0 step 1 until ε do
      begin
 5      i ← h + k
        j ← NEXTB(a_i, G[h])
        if j ≥ jmin
 6      then begin
          F[h] ← imax
          G[h] ← jmin
          NEWP[h] ← −1
        end
 7      else begin
          nskip ← (i − 1) − F[h]
          if nskip = 0
          then NEWP[h] ← P[h]
          else begin
            NEWP[h] ← GETSPACE
            KEEP[NEWP[h]] ← ⟨i − 1, nskip, P[h − nskip]⟩
          end
 8        imax ← i
          jmin ← j
          F[h] ← i
          G[h] ← j
        end
 9    end loop of step 4
10.   if no k-candidates were found then goto step 13
      for i ← 0 step 1 until ε do
      begin
11      REMOVE(P[i])
        P[i] ← NEWP[i]
      end loop of step 10
12  end loop of step 2
13  x ← min h such that P[h] ≥ 0, −1 if none such
    p ← k − 1
    if x < 0 OR p < m − ε then {print "NO", goto step 15}
14. RECOVER
15  END of ALGE

SUBROUTINE RECOVER
 1  SKIP[x + 1] ← 0
    lastmatch ← F[x]
    y ← P[x]
```

```
2  while y ≠ do
   begin
     count ← nskip[y]
     position ← aa[y]
3    while count > 0 do
     begin
       SKIP[x] ← position
       x ← x − 1
       position ← position − 1
       count ← count − 1
     end loop of step 3
     y ← pt[y]
   end loop of step 2
4. x ← 1
   k ← 1
   for ι ← 1 step 1 until lastmatch do
   if ι = SKIP[x] then x ← x + 1
   else begin
     c_k ← a_ι
     k ← k + 1
   end
5  END OF RECOVER
```

The loop of step 2 evaluates sets of feasible k-candidates for $k = 1, 2, \ldots$. The loop of step 4 evaluates whether there is a feasible k-candidate having precisely h skipped positions, for $h = 0, 1, \ldots, \epsilon$, by using Lemma 6 to determine the j-value for a particular ι-value and then checking, by using Lemma 2, whether $\langle i, j \rangle$ is minimal. *imax* is the maximum ι-value of feasible k-candidates generated thus far (i.e. with i-values less than the current value of i); *jmin* is the corresponding j-value (which is the minimum j-value of feasible k-candidates generated thus far). If $\langle i, j \rangle$ is a feasible k-candidate, then it is stored in the F and G arrays and information *will be* stored in $P[h]$, enabling recovery of any additional skipped positions that occur between i and $F[h]$ as well as the skipped positions occurring before $F[h]$ ($\langle F[h], G[h] \rangle$ is a $(k-1)$-candidate that can generate $\langle i, j \rangle$). The h skipped positions corresponding to $\langle F[h], G[h] \rangle$ are recoverable by accessing $KEEP[P[h]]$. In general there may be more than one feasible k-candidate that will be generated by $\langle F[h], G[h] \rangle$. Thus we must not destroy $P[h]$ until all required references to $KEEP[P[h]]$ are made. For this reason, new values for the P array are stored in the *NEWP* array. When we no longer need the old values of P (after the inner loop of steps 4–9), we can then replace them with the new values, being careful to decrement reference counts of *KEEP* elements that were pointed to by the old P array

Function $REMOVE(x)$ decrements the reference count of $KEEP[x]$ (unless $x \leq 0$, in which case nothing is done), and, if $KEEP[x]$ now has reference count zero, then a call will be made to $REMOVE(pt[x])$ after $KEEP[x]$ has been put on the garbage linked list by using *PUTSPACE*.

Implementation of NEXTB

The following should be done before using *ALGE*:

```
1  Sort the symbols in A and then construct a balanced binary search tree of symbols that appear in string A
   Let there be ss such symbols (ss ≤ m).
2. for k ← 1 step 1 until ss do LAST[k] ← 0
3  for ι ← 1 step 1 until n do
   begin
     find out that b_ι = θ_k
     j ← LAST[k]
     LAST[k] ← ι
     if j ≠ 0 then NEXT[j] ← ι
     else FIRST[k] ← ι
   end loop of step 3
```

4. *start* ← 1
for $k \leftarrow 1$ **step** 1 **until** *ss* **do**
begin
Place the positions j of B such that $b_j = \theta_k$ into $N[start]$ through $N[start + nn - 1]$ where θ_k occurs nn times in string B. The first position in B at which θ_k occurs is at $FIRST[k]$. If θ_k occurs at position j, then the next occurrence of θ_k in B will be at position $NEXT[j]$ unless $LAST[k] = j$, in which case there are no more occurrences of θ_k in B.
$S[k] \leftarrow start$
$start \leftarrow start + nn$
end

We can find out that $a_i = \theta_k$ in time $O(\log s)$. $N[S[1]:S[k + 1] - 1]$ holds the block of positions j with $b_j = \theta_k$. This block of cells can be searched by using binary search of a linearly ordered array [11, Sec. 6.2.1]. $NEXT(a_i, j)$ can thus be executed in time $O(\log n)$.

If s is very small, then the following alternate way of computing $NEXTB(\theta, j)$ may be preferred: Instead of constructing a compressed array in step 4, construct a $NEXTB$ matrix while in step 3. For each i, set $NEXTB[k, t] = i$ for $j \leq t < i$. This will result in time and space complexity (of the setup) of $O(sn)$. The function $NEXTB(\theta, j)$ can be evaluated by determining that $\theta = \theta_k$ in time $O(\log S)$ and by doing a simple table look-up.

ALGE retains k-candidates, as did *ALGD*, except for those candidates that cannot lead to a sufficiently long common subsequence because too many A-positions have already been skipped. The $(k + 1)$-candidates that can be generated by the dropped k-candidates also skip too many A-positions.

LEMMA 7. *ALGE retains all feasible k-candidates.*

PROOF. By induction on k. It is trivially true for $k = 0$ (the F and G arrays are initialized to zero in step 1). Assume that the set of feasible $(k - 1)$-candidates has been evaluated and stored in arrays F and G. *ALGE* generates the set of feasible k-candidates in order of increasing i-value. $F[h]$ is to hold $i = h + k$ if i is an i-value of a feasible k-candidate; otherwise $F[h]$ is to hold the maximum $i' < i$ such that i' is a feasible k-candidate. $G[h]$ is to hold the corresponding j-value. *imax* and *jmin* hold the last-generated feasible k-candidate, which, by Lemma 2, has the maximum i-value and minimum j-value generated thus far. Step 3 initializes them to correctly indicate that no k-candidates have yet been generated. Step 5 evaluates the j-value for a given potential k-candidate by using Lemma 6. If $j \geq jmin$ then, even though the necessary condition for feasibility has been met, $\langle i, j\rangle$ is not minimal since it would be ruled out by $\langle imax, jmin\rangle$. In this case step 6 sets $F[h]$ and $G[h]$ to *imax* and *jmin*. If $j < jmin$, then $\langle i, j\rangle$ is minimal since it cannot be ruled out by any previously generated k-candidate ($j < jmin$) and it cannot be ruled out by any future generated k-candidate (all future $i' > i$). In this case step 8 sets $F[h]$ and $G[h]$ and also updates *imax* and *jmin*. □

THEOREM 3. *ALGE correctly computes the LCS of strings A and B if the LCS is of length at least* $m - \epsilon$.

PROOF. By Lemma 7, *ALGE* correctly keeps minimal k-candidates. Thus, if there is a common subsequence of length $p \geq m - \epsilon$, then there is a minimal p-candidate which will be feasible. The data structure of *ALGE* keeps track, for each feasible k-candidate $\langle i, j\rangle$, of the $h = i - k$ positions in string A that have been skipped in the common subsequence of length k of A_{1i} and B_{1j}. $P[h]$ points to the element of *KEEP* that contains the necessary information. P is updated in step 7 when a feasible k-candidate is generated. If any additional positions are skipped (between the k-candidate $\langle i, j\rangle$ and the $(k - 1)$-candidate $\langle i', j'\rangle$ that generated $\langle i, j\rangle$), then that information is recorded in an element of *KEEP* as well as a pointer, enabling recovery of the $h - nskip$ previously skipped A-positions (of $\langle i', j'\rangle$). Subroutine *RECOVER* recovers the skipped positions of a feasible p-candidate by reversing the process in which they were stored and then computes the LCS by deleting the skipped positions from string A. □

THEOREM 4. *For* $\epsilon \leq O(n^{1/2})$, *ALGE requires space linear in n.*

PROOF. The *KEEP* array requires $O(e^2)$ space: The common subsequence implied by k-candidate $\langle h + k, j\rangle$ has h skipped A-positions, $h \leq \epsilon$, and thus can use at most h spaces in the *KEEP* array. The total number of spaces referred to by all feasible k-candidates is thus at most $\epsilon(\epsilon + 1)/2$. Adding to that the (exactly) ϵ references to get the set of feasible $(k + 1)$-candidates gives a total of no more than $(e^2 + e)/2$. Each element of array *KEEP* requires four words (*aa*, *nskip*, *pt*, and a reference counter).

The arrays and space that they use are as follows: $F[e]$, $G[e]$, $C[p]$, $P[e]$, $NEWP[e]$, $KEEP[2e^2 + 2e]$, $FIRST[ss]$, $NEXT[n]$, $LAST[ss]$, $SKIP[e]$, $S[ss]$, $N[n]$.

The *NEXTB* function requires at most $2n$ locations to store the various balanced binary search trees.

Thus a total of at most $2e^2 + 7e + 4n + p + 3ss$ locations is used. For $e \leq O(n^{1/2})$, space requirements are linear in n. □

THEOREM 5. *ALGE requires time* $O(pe \log n)$.

PROOF. Preprocessing for the *NEXTB* function requires time $O(n \log m)$. Step 1 takes time $O(e)$. Step 2 executes steps 3–12 p times. Step 3 takes constant time for a total time of $O(p)$. Step 4 executes steps 5–9 at most e times. Step 5 takes time $O(\log n)$ for a total time of $O(pe \log n)$. Steps 6–9 take constant time for a total time of $O(pe)$. Steps 10–12, excluding time spent in function *REMOVE*, take time $O(e)$ for a total time of $O(pe)$.

Subroutine *RECOVER* recovers at most ϵ skipped positions (taking time $O(e)$) and then deletes them from string A (taking time $O(m)$) for a total time of $O(m)$.

The number of references (to array *KEEP*) removed is at most the number of references inserted. There are at most pe references inserted (one per execution of step 7), and the amount of time (per reference removal) spent in function *REMOVE* is constant. Therefore the total time spent in function *REMOVE* is $O(pe)$.

Therefore the total time of execution of *ALGE* is $O(pe \log n)$. □

It is noted that step 5, requiring $O(\log n)$ time, is the bottleneck, causing total time requirements of $O(pe \log n)$. P. van Emde Boas's recent algorithm for priority queues [19] appears capable of solving the position-finding problem in time $O(\log \log n)$. If so, this would reduce the time bound of this problem to $O(pe \log \log n)$.

ALGE assumes that ϵ is known. If ϵ is not known, then set $\epsilon \leftarrow 2$ and proceed through the algorithm. If that value of ϵ is insufficient (i.e. there is no common subsequence of length $m - \epsilon$), then double the guess for ϵ and continue iteratively until a common subsequence is found.

Total time spent will be (letting k be the multiplicative coefficient of the time requirement)

$$2pk \log n + 4pk \log n + \cdots + epk \log n,$$

which is less than $2pek \log n$. Since $e < 2(m + 1 - p)$, we can recover an LCS in time $O(p(m + 1 - p)\log n)$.

Other Algorithms

The only known algorithm for the LCS problem with worst-case behavior less than quadratic is due to Paterson [14]. The algorithm has complexity $O(n^2 \log \log n / \log n)$. It uses a "Four Russians" approach (see [3] or [1, pp. 244–247]). Essentially, instead of matrix L (where $L[i, j]$ is the length of an LCS of A_{1i} and B_{1j}) being calculated one element at a time (see [7]), the matrix is broken up into boxes of some appropriate size k. The *high* sides of a box (the $2k - 1$ elements of L on the edges of the box with largest indices) are computed from L-values known for boxes adjacent to it on the *low* side and from the relevant symbols of A and B by using a look-up table which was precomputed.

The algorithm assumes a fixed alphabet size although modifications to the algorithm may be able to get around that condition.

There are $2k + 1$ elements of L adjacent to a box on the *low* side. Two adjacent L-elements can differ by either zero or one. There are thus 2^{2k} possibilities in this respect. The A- and B-values range over an alphabet of size s for each of $2k$ elements, yielding a multiplicative factor of s^{2k}, and the total number of boxes to be precomputed is therefore $2^{2k(1+\log s)}$. Each such box can be precomputed in time $O(k^2)$ for a total precomputing time of $O(k^2 2^{2k(1+\log s)})$.

There are $(n/k)^2$ boxes to be looked up, each of which will require $O(k \log k)$ time to be read, for a total time of $O(n^2 \log k/k)$.

The total execution time will therefore be $O(k^2 2^{2k(1+\log s)} + n^2 \log k/k)$. If we let $k = \log n/2(1 + \log s)$, we see that the total execution time will be $O(n^2 \log \log n/\log n)$.

Restrictions on the LCS Problem

Szymanski [17] shows that if we consider the LCS problem with the restriction that no symbol appears more than once within either input string, then this problem can be solved in time $O(n \log n)$.

In addition if one of the input strings is the string of integers $1 - n$, this problem is equivalent to finding the longest ascending subsequence in a string of distinct integers. If we assume that a comparison between integers can be done in unit time, this problem can be solved in time $O(n \log \log n)$ by using the techniques of van Emde Boas [18].

ACKNOWLEDGMENT. I would like to thank the (anonymous) referees for their many helpful suggestions which have led to a material improvement in the readability of this paper.

REFERENCES

(Note References [4–6, 8, 9, 13, 15, 20, 22, 23] are not cited in the text.)

1 AHO, A V , HOPCROFT, J E , AND ULLMAN, J D *The Design and Analysis of Computer Algorithms* Addison-Wesley, Reading, Mass , 1974
2 AHO, A V , HIRSCHBERG, D.S , AND ULLMAN, J D Bounds on the complexity of the longest common subsequence problem *J ACM 23*, 1 (Jan 1976), 1-12.
3 ARLAZAROV, V L., DINIC, E A , KRONROD, M A , AND FARADZEV, I A On economic construction of the transitive closure of a directed graph *Dokl Akad Nauk SSSR 194* (1970), 487-488 (in Russian) English transl in *Soviet Math Dokl 11*, 5 (1970), 1209-1210
4 CHVATAL, V , KLARNER, D A , AND KNUTH, D E Selected combinatorial research problems STAN-CS-72-292, Stanford U , Stanford, Calif , 1972, p 26
5 CHVATAL, V , AND SANKOFF, D Longest common subsequences of two random sequences STAN-CS-75-477, Stanford U , Stanford, Calif , Jan 1975
6 HIRSCHBERG, D S On finding maximal common subsequences TR-156, Comptr Sci Lab , Princeton U , Princeton, N.J , Aug 1974
7. HIRSCHBERG, D S A linear space algorithm for computing maximal common subsequences *Comm ACM 18*, 6 (June 1975), 341-343
8 HIRSCHBERG, D S The longest common subsequence problem Ph D Th , Princeton U , Princeton, N J , Aug 1975
9 HUNT, J W , AND SZYMANSKI, T G A fast algorithm for computing longest common subsequences *Comm ACM 20*, 5 (May 1977), 350-353.
10 KNUTH, D E *The Art of Computer Programming, Vol 1. Fundamental Algorithms* Addison-Wesley, Reading, Mass., sec. ed , 1973
11 KNUTH, D. E *The Art of Computer Programming, Vol 3· Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973
12 LOWRANCE, R , AND WAGNER, R A An extension of the string-to-string correction problem *J ACM 22*, 2 (April 1975), 177-183
13 NEEDLEMAN, S B , AND WUNSCH, C D A general method applicable to the search for similarities in the amino acid sequence of two proteins *J. Mol Biology 48* (1970), 443-453
14 PATERSON, M.S Unpublished manuscript U of Warwick, Coventry, England, 1974
15 SANKOFF, D Matching sequences under deletion/insertion constraints *Proc Nat Acad Sci USA 69*, 1 (Jan 1974), 4-6
16 SELLERS, P H An algorithm for the distance between two finite sequences *J. Combinatorial Theory, Ser A, 16* (1974), 253-258

17 SZYMANSKI, T G A special case of the maximal common subsequence problem TR-170, Comptr Sci Lab , Princeton U , Princeton, N J , Jan 1975.
18 VAN EMDE BOAS, P An $O(n \log \log n)$ on-line algorithm for the insert-extract min problem TR 74-221, Dept Comptr Sci , Cornell U , Ithaca, N Y , Dec 1974
19 VAN EMDE BOAS, P. Preserving order in a forest in less than logarithmic time Conf Rec 16th Annual Symp on the Foundations of Comptr Sci , Oct 1975, pp 75–84
20 WAGNER, R A On the complexity of the extended string-to-string correction problem Proc Seventh Annual ACM Symp on Theory of Comptng , 1975, pp 218–223
21 WAGNER, R A , AND FISCHER, M J The string-to-string correction problem *J. ACM 21*, 1 (Jan 1974), 168–173
22 WONG, C K., AND CHANDRA, A.K Bounds for the string editing problem *J ACM 23*, 1 (Jan 1976), 13–16
23 YAO, C C , AND YAO, F C On computing the rank function for a set of vectors UIUCDCS-R-75-699, Dept Comptr Sci , U of Illinois at Urbana-Champaign, Urbana, Ill , Feb 1975.

RECEIVED JUNE 1975, REVISED FEBRUARY 1977