Equation (22) may be proved by inspection of (18), while (21) may be demonstrated by expanding the logarithm in (17) into a Taylor series and retaining only the first two terms. Equations (21) and (22) show that as the number of accesses becomes immaterial, one should use very small resident and overflow records to reduce the total storage volume. Substituting (21) and (22) into eq. (13), we get

$$V^* \sim R \cdot s/a + R \cdot s \cdot (a - 1)/a = R \cdot s, \quad (23)$$

which is the absolute minimal storage volume needed, without any "overhead" added by the storage method.

Appendix

PROPERTY 1. For all $a > 1$,

$$p^*(a) < q^*(a). \quad (A1)$$

PROOF. Instead of (A1), we prove the equivalent

$$\exp(q^*/s)/\exp(p^*/s) > 1. \quad (A2)$$

Let us introduce the following notation:

$$b = a + (a^2 - 1)^{\frac{1}{2}}, \quad c = 2(a - 1). \quad (A3)$$

Then it is easily seen that

$$q^*/s = 2/(b - 1), \quad (A4)$$
$$p^*/s = \ln(b/c). \quad (A5)$$

Substituting (A4) and (A5) into (A2) and expanding the numerator as a Taylor series, retaining the first three terms, we get

$$\frac{\exp(q^*/s)}{\exp(p^*/s)} = \frac{\exp(2/(b-1))}{b/c} > \frac{1 + 2/(b-1) + 2/(b-1)^2}{b/c} = \frac{(b^2+1)c}{(b-1)^2 b} \quad (A6)$$

From definition (A3), we get the following identity:

$$(b - 1)^2 = b \cdot c. \quad (A7)$$

Substituting (A7) into (A6), we finally have

$$\frac{\exp(q^*/s)}{\exp(p^*/s)} > \frac{(b^2+1)\cdot c}{b^2 \cdot c} = \frac{b^2+1}{b^2} > 1.$$

Received February 1974; revised March 1975

References

1. Benner, F.H. On designing generalized file records for management information systems. Proc. AFIPS 1967 FJCC, Vol. 31, AFIPS Press, Montvale, N.J., pp. 291–303.
2. *Burroughs DISK FORTE Users Manual.* Burroughs Corp., Detroit, Mich., 1973.
3. Collmeyer, A.J., and Shemer, J.E. Analysis of retrieval performance for selected file organization techniques. Proc. AFIPS 1970 FJCC, Vol. 37, pp. 201–210.
4. Olle, T.W. Generalized systems for storing structured variable length data and retrieving information. In *Mechanized Information Storage, Retrieval and Dissemination,* K. Samuelson, Ed., Rome, 1968.
5. Wilde, D.J., and Beightler, C.S. *Foundations of Optimization.* Prentice-Hall, Englewood Cliffs, N.J., 1967.

Programming Techniques

G. Manacher, S.L. Graham Editors

A Fast Algorithm for Computing Longest Common Subsequences

James W. Hunt
Stanford University
Thomas G. Szymanski
Princeton University

Previously published algorithms for finding the longest common subsequence of two sequences of length n have had a best-case running time of $O(n^2)$. An algorithm for this problem is presented which has a running time of $O((r + n) \log n)$, where r is the total number of ordered pairs of positions at which the two sequences match. Thus in the worst case the algorithm has a running time of $O(n^2 \log n)$. However, for those applications where most positions of one sequence match relatively few positions in the other sequence, a running time of $O(n \log n)$ can be expected.

Key Words and Phrases: Longest common subsequence, efficient algorithms
CR Categories: 3.73, 3.63, 5.25

The work of the first author was partially supported by Bell Laboratories' Cooperative Research Fellowship Program. The work of the second author was partially supported by NSF Grants GJ-35570 and DCR74-21939.

Author's addresses: J.W. Hunt, Department of Electrical Engineering, Stanford University, Stanford CA 94305; T.G. Szymanski, Dept. of Electrical Engineering and Computer Science, Princeton University, Brackett Hall, Engineering Quadrangle, Princeton, NJ 98540.

Introduction

Many algorithms [1, 4, 6] for finding the longest common subsequence of two sequences of length n have appeared in the literature. These algorithms all have a worst-case (as well as a best-case) running time of $O(n^2)$.[1]

A more relevant parameter for this problem is r, the total number of matching pairs of positions within the sequences in question. We shall present an $O((r+n)\log n)$ algorithm for the longest common subsequence problem. In the worst case this is of course $O(n^2 \log n)$. However, for a large number of applications, we can expect r to be close to n. In these situations our algorithm will exhibit an $O(n \log n)$ behavior. Typical of such applications are the following:

(1) Finding the longest ascending subsequence of a permutation of the integers from 1 to n [3].

(2) Finding a maximum cardinality linearly ordered subset of some finite collection of vectors in 2-space [7].

(3) Finding the edit distance between two files in which the individual lines of the files are considered to be atomic. The longest common subsequence of these files, considered as sequences, represents that common "core" which does *not* have to be changed if we desire to edit one file into the other.

Thus in the general case our algorithm will not take much longer than the algorithms of [1, 4, 6], whereas in many common applications, our algorithm will perform substantially better.

Let A be a finite sequence of elements chosen from some alphabet. We denote the length of A by $|A|$. $A[i]$ is the ith element of A and $A[i{:}j]$ denotes the sequence $A[i], A[i+1], \cdots, A[j]$.

If U and V are finite sequences, then U is said to be a *subsequence* of V if there exists a monotonically increasing sequence of integers $r_1, r_2, \cdots, r_{|U|}$ such that $U[i] = V[r_i]$ for $1 \leq i \leq |U|$. U is a *common subsequence* of A and B if U is a subsequence of both A and B. A *longest common subsequence* is a common subsequence of greatest possible length.

Throughout this paper A and B will be used to denote the sequences in question. For ease of presentation, we shall assume both sequences have the same length which will be denoted by n. The number of elements in the set $\{(i, j)$ such that $A[i] = B[j]\}$ will be denoted by r.

Preliminary Results

The key data structure needed by our algorithm is an array of "threshhold values" $T_{i,k}$ defined by $T_{i,k}$ = the smallest j such that $A[1{:}i]$ and $B[1{:}j]$ contain a common subsequence of length k. For example, given sequences $A = abcbdda$, $B = badbabd$ we have $T_{5,1} = 1$, $T_{5,2} = 3$, $T_{5,3} = 6$, $T_{5,4} = 7$, $T_{5,5}$ = undefined.

Each $T_{i,k}$ may thus be considered as a pointer which tells us how much of the B sequence is needed to produce a common subsequence of length k with the first i elements of A.

Note that each row of the T array is strictly increasing; that is,

LEMMA 1. *If $T_{i,1}, T_{i,2}, \cdots, T_{i,p}$ are defined, then $T_{i,1} < T_{i,2} < \cdots < T_{i,p}$.*

PROOF. Consider the common subsequence of length k contained in $A[1{:}i]$ and $B[1{:}T_{i,k}]$. Clearly $B[T_{i,k}]$ is the last member of this common subsequence or else $T_{i,k}$ would not be minimal. Therefore $A[1{:}i]$ and $B[1{:}T_{i,k}-1]$ contain a common subsequence of length $k-1$, that is, $T_{i,k-1} \leq T_{i,k} - 1$. □

This linear ordering is of paramount importance in the efficient implementation of our algorithm.

Suppose that we have computed $T_{i,k}$ for all values of k and wish to compute $T_{i+1,k}$ for all values of k. We first show $T_{i+1,k}$ must lie in a specific range of values.

LEMMA 2. $T_{i,k-1} < T_{i+1,k} \leq T_{i,k}$.

PROOF. If $A[1{:}i]$ and $B[1{:}T_{i,k}]$ have a common subsequence of length k, then certainly $A[1{:}i+1]$ and $B[1{:}T_{i,k}]$ do also. Thus $T_{i+1,k} \leq T_{i,k}$.

By definition, $A[1{:}i+1]$ and $B[1{:}T_{i+1,k}]$ have a common subsequence of length k. Deleting the last element from each of these sequences can remove at most one element from this common subsequence. thus $A[1{:}i]$ and $B[1{:}T_{i+1,k}-1]$ have a common subsequence of length $k-1$. Accordingly $T_{i,k-1} \leq T_{i+1,k} - 1$ and $T_{i,k-1} < T_{i+1,k}$. □

The following rule suffices to compute $T_{i+1,k}$ from $T_{i,k-1}$ and $T_{i,k}$.

LEMMA 3.

$$T_{i+1,k} = \begin{cases} \text{smallest } j \text{ such that } A[i+1] = B[i] \\ \qquad \text{and } T_{i,k-1} < j \leq T_{i,k} \\ T_{i,k} \text{ if no such } j \text{ exists.} \end{cases}$$

PROOF.

Case 1. No such j exists. By the minimality of $T_{i+1,k}$, any common subsequence of the sequences $A[1{:}i+1]$ and $B[1{:}T_{i+1,k}]$ must have $B[T_{i+1,k}]$ as its last element. Moreover, by Lemma 2 and the premise of this case, $B[T_{i+1,k}]$ does not match $A[i+1]$. Therefor the same common subsequence of length k is also contained in $A[1{:}i]$ and $B[1{:}T_{i+1,k}]$. Thus $T_{i,k} \leq T_{i+1,k}$ and by Lemma 2, $T_{i,k}$ must equal $T_{i+1,k}$.

Case 2. There exists a minimal j for which $A[i+1] = B[j]$ and $T_{i,k-1} < j \leq T_{i,k}$. Certainly $A[1{:}i+1]$ and $B[1{:}j]$ contain a common subsequence of length k, namely the length $k-1$ common subsequence of

[1] An unpublished result of Michael Paterson shows how to construct an $O(n^2/\log n)$ algorithm for the longest common subsequence problem for sequences over a finite alphabet, and an $O((n^2 \log\log n)/\log n)$ algorithm for sequences over an infinite ordered alphabet. All results of this paper apply to the case of the infinite ordered alphabet.

$A[1{:}i]$ and $B[T_{i,k-1}]$ with the pair $A[i+1]$, $B[j]$ "tacked" onto the end. Thus $T_{i+1,k} \leq j$.

Assume temporarily that $T_{i+1,k} < j$. Since Lemma 2 guarantees that $T_{i,k-1} < T_{i+1,k}$ we can conclude that the last element of the length k common subsequence of $A[1{:}i+1]$ and $B[1{:}T_{i+1,k}]$ does *not* match $A[i+1]$. Thus $A[1{:}i]$ and $B[1{:}T_{i+1,k}]$ also contain a common subsequence of length k which implies that $T_{i,k} \leq T_{i+1,k}$. By Lemma 2 then, $T_{i,k} = T_{i+1,k}$. However, by the above assumption and the premise of this case, $T_{i+1,k} < j \leq T_{i,k}$, implying that $T_{i,k} \neq T_{i+1,k}$. This contradiction leads us to conclude that the original assumption of $T_{i+1,k} < j$ is incorrect and hence we must have $T_{i+1,k} = j$. □

We can now present an $O(n^2 \log n)$ algorithm for determining the length of the longest common subsequence. Subsequent refinements will enable us to not only improve the running time to $O((r+n)\log n)$ but also recover the actual longest common subsequence.

Algorithm 1

```
element array A[1:n], B[1:n];
integer array THRESH[0:n];
integer i, j, k;
THRESH[0] := 0;
for i := 1 step 1 until n do
  THRESH[i] := n + 1;
for i := 1 step 1 until n do
  for j := n step − 1 until 1 do
    if A[i] = B[j] then
      begin
        find k such that THRESH[k−1] < j ≤ THRESH[k];
        THRESH[k] := j;
      end;
print largest k such that THRESH[k] ≠ n + 1;
```

The correctness of the algorithm follows from consideration of the invariant relation "$THRESH[k] = T_{i-1,k}$ for all k" which holds at the start of each iteration on i, and the invariant relation "$THRESH[k] = T_{i,k}$ for all k" which holds at the end of each iteration on i.

Since the *THRESH* array is monotonically increasing (Lemma 1) we can utilize a binary search to implement the "find" operation in time $O(\log n)$. Thus Algorithm 1 may be implemented to run in $O(n^2 \log n)$ time.

Finally, notice that the direction of the loop on j is crucial. Suppose that for some value of i, $A[i]$ matches several different B elements in a given "threshold" interval, say $B[j_1], \cdots, B[j_m]$ with $THRESH[k-1] = T_{i-1,k-1} < j_1 < \cdots < j_m \leq T_{i-1,k} = THRESH[k]$. From Lemma 3, we see that $T_{i,k} = j_1$ and that $THRESH[k]$ should be updated to this value. Since the inner loop of Algorithm 1 considers values of j in decreasing order, each of the values $j_m, j_{m-1}, \cdots, j_1$ will cause $THRESH[k]$ to take on successively smaller values until it is set equal to the desired value of j_1. If instead the loop on j ran upwards from 1 to n, then not only would $THRESH[k]$ be set to j_1, but $THRESH[k+1]$ would be set to j_2, $THRESH[k+2]$ would be set to j_3 and so forth. Since these latter assignments are unwarranted, we see that the loop on j *must* run downwards.

The Algorithm

A small amount of preprocessing will vastly improve the performance of Algorithm 1. The main source of inefficiency in this algorithm is the inner loop on j in which we repeatedly search for elements of the B sequence which match $A[i]$. Linked list techniques obviate the need for this search.

For each position i we need a list of corresponding j positions such that $A[i] = B[j]$. These lists must be kept in decreasing order in j. All positions of the A sequence which contain the same element may be set up to use the same physical list of matching j's; for the sequences $A = abcbdda$, $B = badbabd$ the desired lists are

$MATCHLIST[1] = \langle 5, 2 \rangle$
$MATCHLIST[2] = \langle 6, 4, 1 \rangle$
$MATCHLIST[3] = \langle\ \rangle$
$MATCHLIST[4] = MATCHLIST[2]$
$MATCHLIST[5] = \langle 7, 3 \rangle$
$MATCHLIST[6] = MATCHLIST[5]$
$MATCHLIST[7] = MATCHLIST[1]$.

We can now display our final algorithm.

Algorithm 2

```
element array A[1:n], B[1:n];
integer array THRESH[0:n];
list array MATCHLIST[1:n];
pointer array LINK[1:n];
pointer PTR;
comment Step 1: build linked lists;
for i := 1 step 1 until n do
  set MATCHLIST[i] := ⟨j1, j2, ···, jp⟩ such that
    j1 > j2 > ··· > jp and A[i] = B[jq] for 1 ≤ q ≤ p;
comment Step 2: initialize the THRESH array;
THRESH[0] := 0;
for i := 1 step 1 until n do
  THRESH[i] := n + 1;
LINK[0] := null;
comment Step 3: compute successive THRESH values;
for i := 1 step 1 until n do
  for j on MATCHLIST[i] do
    begin
      find k such that THRESH[k−1] < j ≤ THRESH[k];
      if j < THRESH[k] then
        begin
          THRESH[k] := j;
          LINK[k] := newnode (i, j, LINK[k−1]);
        end;
    end;
comment Step 4: recover longest common subsequence in reverse
  order;
k := largest k such that THRESH[k] ≠ n + 1;
PTR := LINK[k];
while PTR ≠ null do
  begin
    print (i, j) pair pointed to by PTR;
    advance PTR;
  end;
```

The subroutine *newnode* invoked in step 3 is a subroutine which creates a list node whose fields contain the values of the arguments to *newnode*. These arguments are, respectively, an index of a position in the A sequence, an index of a position in the B sequence, and a pointer to some other list node. The value returned by *newnode* is a pointer to the list node just created.

THEOREM 1. *Algorithm 2 finds and prints a longest common subsequence of the sequences A and B in time* $O((r + n) \log n)$ *and space* $O(r + n)$.

PROOF. Step 1 can be implemented by sorting each sequence while keeping track of each element's original position. We may then merge the sorted sequences creating the *MATCHLIST*s as we go. This step takes a total of $O(n \log n)$ time and $O(n)$ space.

Step 2 clearly takes $O(n)$ time.

The two outer loops of step 3 should be considered as a single loop over all pairs (i, j) such that $A[i] = B[j]$ taken in order of decreasing j within increasing i. In other words, the outer loops of step 3 induce exactly r executions of the innermost statements of step 3. Since these innermost statements involve one binary search plus a few operations which require constant time, we conclude that the time requirement for step 3 is $O(n + r \log n)$.

In this step we also implement a simple backtracking device that will allow us to recover the longest common subsequence. We record each (i, j) pair which causes an element of the *THRESH* array to change value. Thus whenever *THRESH*$[k]$ is defined, *LINK*$[k]$ points to the head of a list of (i, j) pairs describing a common subsequence of length k. Since at most one list node is created per search, Step 3 will require the allocation of at most $O(r)$ list nodes.

In step 4 we recover the actual longest common subsequence. Clearly this takes at most $O(n)$ time. □

We note that certain input sequences such as A = "*aabaabaab* . . ." and B = "*ababab* . . ." cause Algorithm 2 to use $O(r)$ space even if list nodes are reclaimed whenever they become inaccessible. See [4] for an algorithm which never uses more than $O(n)$ space nor less than $O(n^2)$ time.

A Final Note

The key operations in the implementation of our algorithm are the operations of inserting, deleting, and testing membership of elements in a set where all elements are restricted to the first n integers. Peter van Emde Boas has shown that each such operation can be performed in $O(\log \log n)$ time [2]. His data stucture requires $O(n \log \log n)$ time for initialization. Although the necessary algorithms are quite complex, we can use them to present the following theoretical result.

THEOREM 2. (a) *Algorithm 2 can be implemented to have a running time of* $O(r \log \log n + n \log n)$ *over an infinite alphabet.* (b) *Algorithm 2 can be implemented to have a running time of* $O((n + r) \log \log n)$ *over a fixed finite alphabet.* (c) *The longest ascending subsequence of a permutation of the first n integers may be found in* $O(n \log \log n)$ *time.*

PROOF. The problem of part (c) is, of course, equivalent to finding the longest common subsequence of the given permutation and the sequence $1, 2, \cdots, n$. All three parts of the theorem use basically the same algorithm although the implementation of some of the steps varies slightly. We shall present a common analysis.

In all three cases we require $O(n \log \log n)$ time to initialize van Emde Boas's data structures. Step 1 entails a sorting procedure to set up the *MATCHLIST*s. For the infinite alphabet case, this sort can be done in $O(n \log n)$ time. In the other two cases, we can use a distribution sort to create the *MATCHLIST*s in $O(n)$ time. Step 2 takes $O(n)$ time, step 3 takes $O(n + r \log \log n)$ time and step 4 takes $O(n)$ time. Finally, for the permutation case note that each integer appears exactly once in each sequence and thus we have $r = n$. □

Acknowledgments. The authors are indebted to M. Douglas McIlroy who first suggested this problem to us. Harold Stone suggested a variant of the problem (described and solved in [5]) which led to the development of the present algorithm. Alfred V. Aho and Jeffrey D. Ullman provided us with several enlightening conversations including the particular example given following Theorem 1 which shows that our algorithm can require as much as $O(r)$ space. Peter van Emde Boas made several helpful comments on an early draft of this paper.

Received May 1975; revised January 1976

References

1. Chvatal, V., Klarner, D.A., and Knuth, D.E. Selected combinatorial research problems. STAN-CS-72-292, Dep. Comptr. Sci., Stanford U., Stanford, Calif., June 1972.

2. van Emde Boas, P. Preserving order in a forest in less than logarithmic time. 16th Annual Symp. on Foundations Comptr. Sci., Oct. 1975, pp. 75–84.

3. Fredman, M.L. On computing the length of longest increasing subsequences. *Discrete Mathematics 11*, 1 (Jan. 1975), 29–35.

4. Hirschberg, D.S. A linear space algorithm for computing maximal common subsequences. *Comm. ACM 18*, 6 (June 1975), 341–343.

5. Szymanski, T.G. A special case of the maximal common subsequence problem. TR-170, Dep. Electrical Eng., Princeton U., Princeton, N.J., Jan. 1975.

6. Wagner, R.A. and Fischer, M.J. The string-to-string correctior problem. *J. ACM 21*, 1 (Jan. 1975), 168–173.

7. Yao, A.C. and Yao, F.F. On computing the rank function for a set of vectors. UIUCDCS-R-75-699, Dep. Comptr. Sci., U. of Illinois at Urbana-Champaign, Urbana, Ill., Feb. 1975.