Acta Informatica 18, 171 – 179 (1982)

Acta Informatica
© Springer-Verlag 1982

A Longest Common Subsequence Algorithm Suitable for Similar Text Strings

Narao Nakatsu[1], Yahiko Kambayashi[2], and Shuzo Yajima[2]

[1] Center for Educational Technology, Aichi University of Education, Kariya, Aichi 448, Japan
[2] Department of Information Science, Kyoto University, Kyoto 606, Japan

Summary. Efficient algorithms for computing the longest common subsequence (LCS for short) are discussed. $O(pn)$ algorithm and $O(p(m-p)\log n)$ algorithm [Hirschberg 1977] seem to be best among previously known algorithms, where p is the length of an LCS and m and n are the lengths of given two strings ($m \leqq n$). There are many applications where the expected length of an LCS is close to m.

In this paper, $O(n(m-p))$ algorithm is presented. When p is close to m (in other words, two given strings are similar), the algorithm presented here runs much faster than previously known algorithms.

1. Introduction

The longest common subsequence (LCS) problem is to determine one of the longest subsequences that can be obtained by deleting zero or more symbols from each of two given strings. For example, TUSDAY is the longest common subsequence of TUeSDAY and ThUrSDAY. This paper presents an efficient algorithm for the LCS problem, which is especially suitable when two given strings have a long LCS.

Algorithms for this problem can be used to locate differences of two text strings such as a text and a revised text (or a program and a revised program). Algorithms can be also used to compress data when we need to store similar text strings, since the common subsequences are required to be stored once. The problem is also regarded as a generalization of the string matching problem (for example, see [5]) where some errors in strings are permitted. For example, an algorithm for the LCS problem can be used to find all titles of research papers which contain some given words or words similar to them.

Hirschberg has presented $O(pn)$ and $O(p(m-p)\log n)$ LCS algorithms where p is the length of an LCS and m and n are the lengths of two given text strings ($m \leqq n$ is assumed). The former runs in almost linear time when p is small and the latter runs in time of $O(m \log n)$ when p is large [3].

0001-5903/82/0018/0171/$01.80

In the applications discussed above, usually p is very close to m. This paper presents an $O(n(m-p))$ algorithm which runs in linear time when p is close to m. Thus to compare two similar text strings, the algorithm presented here is much more suitable than the previously known algorithms. Moreover, when only the length of an LCS is required, an algorithm requiring $O(m+n)$ storage space can be easily obtained from our algorithm.

There are, however, many cases when the possible p is unpredictable. A comparison of the above three algorithms is also given in this paper.

2. Basic Definitions and Previous Results

In this section, some basic definitions are presented.

Definition 1. Let $\Sigma = \{a_1, a_2, \ldots, a_k\}$ be any finite set of symbols called an alphabet. A string over Σ is any finite sequence of elements from Σ. Σ^* means the set of all strings over Σ, including the empty string Λ. $\Sigma^* - \{\Lambda\}$ is denoted by Σ^+. $|\Sigma|$ denotes the number of elements in Σ and $|\sigma|$ $(\sigma \in \Sigma^*)$ denotes the length of σ.

Definition 2. A string σ $(\sigma \in \Sigma^+)$ is denoted by $\sigma(1)\,\sigma(2) \ldots \sigma(m)$ $(\sigma(i) \in \Sigma, 1 \leqq i \leqq m = |\sigma|)$ and $\sigma' = \sigma(i_1)\,\sigma(i_2) \ldots \sigma(i_k)$ $(1 \leqq i_1 < i_2 < \ldots < i_k \leqq m)$ is called a subsequence of σ. In other words, a subsequence of a string is obtained by deleting zero or more symbols from the string.

Example 1. For the string $\sigma = abcbcab$, $\sigma' = abca$ is a subsequence of σ.

Definition 3. For given two strings σ and τ defined over Σ, string θ is a common subsequence of σ and τ if θ is a subsequence of both σ and τ. String θ is a <u>longest common subsequence</u> (LCS for short) of σ and τ if θ is a common subsequence of σ and τ of maximum length. The problem for finding an LCS of two given strings is called the LCS problem.

Throughout this paper, m and n denote the lengths of strings σ and τ, respectively. $m \leqq n$ is assumed without loss of generality.

Example 2. For the given two strings $\sigma = abcdbb$ and $\tau = cbacbaaba$, $\sigma' = bcbb$ and $\sigma'' = acbb$ are the two LCSs of the strings σ and τ.

Aho et al. have shown that unless the alphabet size is fixed, every solution to the problem by the "equal-unequal" comparison will consume an amount of time that is proportional to the product of the lengths of two strings [1]. $O(m \log n)$ has been proved to be a lower bound to solve the LCS problem by the "less than-equal-greater than" comparison for an unrestricted alphabet [4]. We consider the problem by "equal-unequal" comparison approach.

Though Aho et al. suggested the existence of linear time algorithms for the LCS problem when the alphabet size is fixed, previously published algorithms require only slightly less running time than $O(mn)$ in the worst case [2, 6].

Hunt and Szymanski presented an algorithm which has the running time of $O((r+m) \log n)$, where r is the total number of ordered pairs of positions at which two strings match [2]. Mukhopadhyay has also shown an algorithm with running time of the same time complexity [6]. r will be mn/k on the average, where $k = |\Sigma|$. These algorithms are not so efficient when m and n are large and k is small.

Hirschberg presented two algorithms [3]. The first one requires $O(pn)$ time and it is efficient when p is small, while the other one requires $O(p(m+1-p)\log n)$ time and it is suitable when p is large, where p is the length of an LCS.

In the following section, an algorithm which is preferred for applications where the expected length of an LCS is large is presented. The algorithm runs in time of $O(n(m-p))$ for finding one LCS and thus in $O(n)$ time when p is close to m.

3. An Algorithm for the Longest Common Subsequence Problem

Definition 4. For a given string $\sigma=\sigma(1)\,\sigma(2)\ldots\sigma(m)$, $\sigma(i{:}m)$ denotes the consecutive subsequence $\sigma(i)\,\sigma(i+1)\ldots\sigma(m)$. For given strings $\sigma=\sigma(1)\,\sigma(2)\ldots\sigma(m)$ and $\tau=\tau(1)\,\tau(2)\ldots\tau(n)$ $(m\leqq n)$, $L_i(k)$ denotes the largest h such that $\sigma(i{:}m)$ and $\tau(h{:}n)$ have an LCS of length k.

From Definition 4, next three lemmas hold. Similar lemmas have been already presented by Hirschberg and Hunt et al. [2, 3]. Lemmas 1 and 2 are easily observed by the definition of $L_i(k)$, and the proofs are omitted.

Lemma 1. $\forall\, i\in[1,m]$, $L_i(1)>L_i(2)>\ldots$

Lemma 2. $\forall\, i\in[1,m-1]$, $\forall\, j\in[1,m]$, $L_{i+1}(j)\leqq L_i(j)$ *holds.*
Undefined $L_i(j)$ *are not considered in above two lemmas.*

Lemma 3.

$$L_i(j)=\begin{cases}\mathrm{Max}(h, L_{i+1}(j)), \text{ where } h \text{ is the largest number such that} \\ \sigma(i)=\tau(h) \text{ and } h<range, \text{ where } range=L_{i+1}(j-1) \text{ if } j\geqq 2 \\ \qquad\qquad\qquad\qquad \text{else } range=n+1. \\ L_{i+1}(j), \text{ if no such } h \text{ exists.}\end{cases}$$

$\mathrm{Max}(x, y)$ *denotes the larger one of* x *and* y.

Proof. By the definition of $L_i(j)$, $h\leqq L_i(j)$ holds and $L_{i+1}(j)\leqq L_i(j)$ holds according to Lemma 2. Therefore $L_i(j)\geqq \mathrm{Max}(h, L_{i+1}(j))$ is proved. By the definition of $L_i(j)$, $\sigma(i+1{:}m)$ and $\tau(L_i(j){:}n)$ must have an LCS of length $j-1$ and then $L_{i+1}(j-1)\geqq L_i(j)$ holds. If we assume that $L_i(j)>\mathrm{Max}(h, L_{i+1}(j))$ is true, $\mathrm{Max}(h, L_{i+1}(j))<L_i(j)\leqq L_{i+1}(j-1)$ must hold. By the definitions of h and $L_i(j)$, $\sigma(i{:}m)$ and $\tau(L_i(j){:}n)$ have an LCS of length j and $\tau(L_i(j))\neq\sigma(i)$ must hold. This suggests that $\sigma(i{:}m)$ and $\tau(L_i(j)+1{:}n)$ or $\sigma(i+1{:}m)$ and $\tau(L_i(j){:}n)$ have an LCS of length j. The former case contradicts the definition of $L_i(j)$ and the latter case contradicts the above assumption. Figure 1 will help readers to understand the proof of Lemma 3. Q.E.D.

Example 3.

```
    123456789          123456
σ = cbacbaaba   and  τ = abcdbb.
```

In this case, we have $L_9(1)=1$, $L_8(1)=6$, $L_8(2)=$ undefined and $L_7(2)=1$.

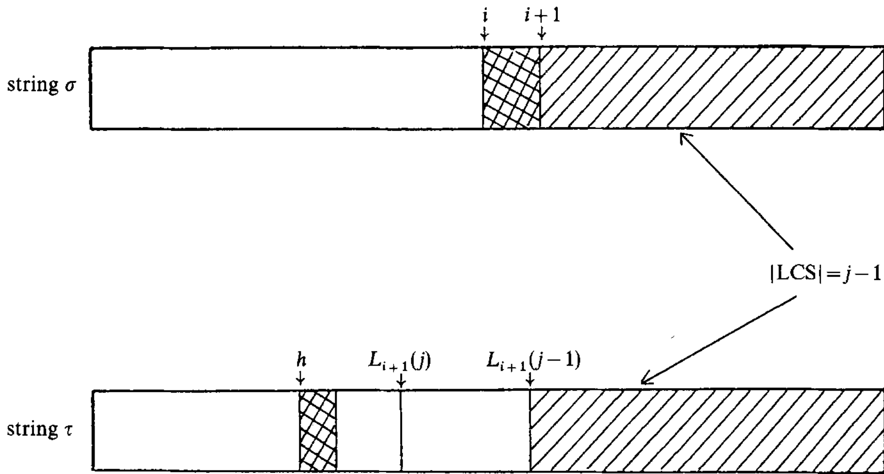

Fig. 1. Lemma 3 holds considering above configuration

We can now present an $O(n(m-p))$ algorithm for finding one LCS. We prepare a triangular matrix M to store each value of $L_i(j)$. An LCS can be obtained by calculating all $L_i(j)$ $(1 \leqq i, j \leqq m)$. Our algorithm is based on the concept that only necessary elements of M which may induce an LCS are calculated using Lemma 3. An example is given before the formal description of the algorithm.

Example 4. Strings σ and τ are given as follows.

$$\begin{matrix} 1234567 & 123456789 \\ \sigma = bcdabab, & \tau = cbacbaaba. \end{matrix}$$

First we prepare a matrix as shown in Fig. 2. We will store $L_i(j)$ in the (j, i) element of the matrix. Since j is the length of an LCS of $\sigma(i:m)$ and $\tau(L_i(j), n)$, $j \leqq m - i + 1$ and $i \leqq m$ hold. Thus we only need the left upper part of the matrix. Undefined $L_i(j)$ are set to be zero for the purpose of the uniform handling. In this example, $L_8(1)$, $L_7(2)$, $L_6(3)$, $L_5(4)$ and $L_4(5)$ are set to be zeros.

Beginning the scan from the last position of τ, we can conclude $L_7(1)$ is 8 since the first "b" encountered in τ is at the 8-th position of τ. $L_6(2)\,(<L_7(1))$ is computed by finding the first "a" ("a" is the 6-th character of σ) which appears in τ before the position $L_7(1)$. Such "a" is found to be in the 7-th position in τ. $L_5(3)$ can be determined by finding the first "b" ("b" is the 5-th character of σ) encountered in τ when τ is scanned backward from $L_6(2)$. Thus $L_5(3) = 5$ is obtained. Continuing this process, $L_4(4) = 3$ is obtained. We find $L_3(5)$ is undefined since "d" is not contained in τ. To compute these all values, i.e., $L_7(1), L_6(2), \ldots, L_3(5)$, we only need to scan out the string τ only once. This fact contributes to the reduction of computation time. After calculating all these values, we again start computation from $L_6(1)$. By Lemma 3, we just need to find the largest h such that $\sigma(i) = \tau(h)$ and $L_{i+1}(j) < h < L_{i+1}(j-1)$. Values $L_{i+1}(j)$ and $L_{i+1}(j-1)$ have been already calculated. To calculate $L_6(1)$, we should notice that $L_6(1) \geqq L_7(1) = 8$.

		1	2	3	4	5	6	7	8
		b	*c*	*d*	*a*	*b*	*a*	*b*	
1	*c*					*9*	9	⑧	0
2	*b*				*8*	8	⑦	0	
3	*a*			*7*	7	⑤	0		
4	*c*		*4*	3	③	0			
5	*b*	*2*	①	0	0				
6	*a*	0	0						
7	*a*								
8	*b*								
9	*a*								

Fig. 2. Matrix M after the application of our algorithm. (Encircled elements represent one LCS obtained by our algorithm. Elements written in italics are needed only when all LCSs are required)

Actually we can find "*a*" at the 9-th position of τ and $L_6(1)$ is found to be 9. Repeating the similar process as above we get $L_5(2)=8$, $L_4(3)=7$, $L_3(4)=3$ and $L_2(5)=1$. For $L_3(4)$, we cannot find the corresponding position and $L_4(4)=3$ is assigned to $L_3(4)$ (Lemma 3). The values $L_7(1)=8$, $L_6(2)=7$, $L_5(3)=5$, $L_4(4)=3$ and $L_2(5)=1$ show the common subsequence $\tau(1)\,\tau(3)\,\tau(5)\,\tau(7)\,\tau(8)$ $=\sigma(2)\,\sigma(4)\,\sigma(5)\,\sigma(6)\,\sigma(7)=cabab$. We should be careful to recover an LCS, because $L_3(4)$ does not represent a real position of an LCS.

We are interested in finding just one such sequence. Since we have already found a common subsequence of length 5, we know further improvement will never be possible by calculating $L_5(1), L_4(2), L_3(3), L_2(4)$ and $L_1(5)$ (note that $L_1(5)$ means an LCS of length 5). Since further calculation of $L_i(j)$ will not improve the result, we stop the computation. If all LCSs are required, $L_5(1), L_4(2), \ldots, L_1(5)$ must be calculated. Then in this example, $\tau(2)\,\tau(3)\,\tau(5)\,\tau(7)\,\tau(8)=babab$, $\tau(2)\,\tau(4)\,\tau(5)\,\tau(7)\,\tau(8)=bcbab$ and $\tau(2)\,\tau(4)\,\tau(7)\,\tau(8)\,\tau(9)=bcaba$ are also LCSs (see Fig. 2).

Our algorithm is formally as follows. For the given two strings StringA and StringB ($|\text{StringA}|=m\leqq|\text{StringB}|=n$), we prepare $(m+1)\times(m+1)$ left upper triangular matrix M, whose (i,j) element ($1\leqq i\leqq m$, $1\leqq j\leqq m$, $j\leqq m+2-i$) denotes $L_j(i)$ (notice that the subscripts are reversed).

Arrays LCSA and LCSB of length m are used to recover one LCS.

Algorithm. $O(n(m-p))$ algorithm for the LCS problem
(initialization)
$s1$: diagonal pos $\leftarrow m$, length MAX $\leftarrow 0$
  clear LCSA and LCSB.
$s2$: **do while** (length MAX < diagonal pos)
    pos A $\leftarrow$ diagonal pos, length LCS $\leftarrow 1$, upper $B \leftarrow n+1$.
(calculate M(i,j) s on one diagonal)
  $s21$: **do while** (pos $A \neq 0$ and upper $B \neq 0$)
      (clear an element of M for uniform handling)
      if diagonal pos $=m$ **or** length LCS > length MAX
          then M(length LCS, pos $A+1$) $\leftarrow 0$.
      lower $B \leftarrow$ Max(1, M(length LCS, pos $A+1$)).
      pos $B \leftarrow$ upper $B-1$.

```
            (calculate one M(i,j) according to Lemma 3)
            do while (pos B ≧ lower B and String A(pos A) ≠ String B(pos B))
               pos B ← pos B − 1. end.
            if pos B ≧ lower B then upper B ← pos B
                                else upper B ← M(length LCS, pos A + 1).
            M(length LCS, pos A) ← upper B.
            if upper B = 0 then length LCS ← length LCS − 1.
            length MAX ← Max(length MAX, length LCS).
            length LCS ← length LCS + 1.
            pos A ← pos A − 1.
         end.
         diagonal pos ← diagonal pos − 1.
      end.
(recover an LCS)
s3: The length of an LCS of String A and String B is length MAX.
if upper B = 0 then pos A ← pos A + 2 else pos A ← pos A + 1.
   pos LCS ← length MAX.
   do while (pos LCS > 0)
      do while (M(pos LCS, pos A) = M(pos LCS, pos A + 1))
                               pos A ← pos A + 1.   end.
      LCS A(pos LCS) ← pos A.  LCS B(pos LCS) ← M(pos LCS, pos A).
      pos A ← pos A + 1, pos LCS ← pos LCS − 1.
   end.
```

An LCS of String$A$ and String$B$ is String$A$ (LCS A(length MAX))

String A(LCS A(length MAX − 1)) ... String A($LCSA$(1))
= String B(LCS B(length MAX))
String B(LCS B(length MAX − 1)) ... String B(LCS B(1)).

end of the algorithm.

Theorem 1. *Assuming that two symbols can be compared in one time unit, above algorithm requires time of* $n(m-p)+(p+1)(m-p+1)$ *units, where p is the length of an* LCS *of* String A *and* String B *(p is denoted by length* MAX *in above algorithm). m and n are equal to* |String A| *and* |String B|*, respectively. Therefore above algorithm requires time of* $O(n(m-p))$ *when* $p<m$. *When* $p=m$, *above algorithm can run in time of* $O(n)$.

Proof. In this algorithm, elements of M are computed in the order shown in Fig. 3. $s1$ requires $O(m)$ computation time. The subloop $s21$ in which elements on one diagonal of M are computed is repeated exactly $(m-p)$ times when $p<m$. If $p=m$ subloop $s21$ is executed only once. In the loop $s21$, all of the symbols in String B are compared sequentially from the last symbol to the first symbol. So total number of comparisons of symbols in loop $s2$ is $(m-p)\,n$. The number of elements of M that are computed in the algorithm is $(p+1)(m-p+1)$ at most including the auxiliary elements. $s3$ requires time of $O(m)$. Then total computation time is $n(m-p)+(p+1)(m-p+1)+O(m)$. Q.E.D.

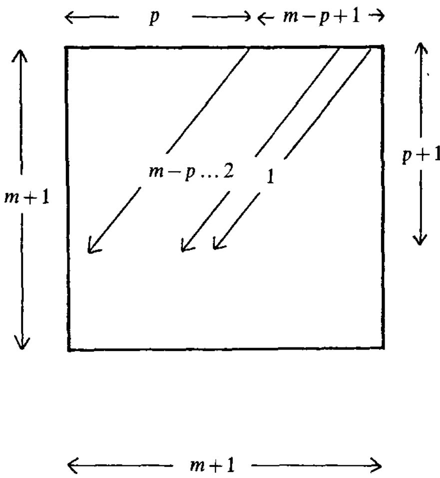

Fig. 3. The order of computing each element of M

For the convenience of understanding, an array of size m^2 is prepared. Some readers may think that the algorithm requires time of $O(m^2)$ if $m^2 \gg (m-p)\,n$, since to clear the array M takes m^2 steps. Only at most $(p+1)(m-p+1)$ elements of M are, however, required in the above algorithm. It is not difficult to map an element of M, $M(i,j)$, to an element of a one-dimensional array, whose size is bounded by $(m+1)(m-p+1)$.

By a little modification of the above algorithm, we can get (1) an algorithm to generate all LCSs, (2) a linear space algorithm to calculate the length of an LCS and (3) another $O(p(m-p)\log n)$ algorithm. We will show an outline of our $O(p(m-p)\log n)$ algorithm.

Basic procedure is the same as the preceding algorithm. The way of calculating each element of M is different. For each symbol a_i in Σ, we prepare a list N_i which is an ordered list, in the descending order, of positions in String B in which symbol a_i occurs. In the preceding algorithm, the largest position h such as String $A(i)$ $=$ String $B(h)$ and $L_{i+1}(j) < h < L_{i+1}(j-1)$ must be determined to calculate $M(j, i)$. If String $A(i) = a_k$ we can determine such h by searching the list N_k in time of $O(\log |N_k|)$, where $|N_k|$ is the length of N_k and it does not exceed $n = |\text{String}\,B|$. To prepare all N_i, $n \log n$ time is required at most. The number of $M(j, i)$ that should be calculated is also $(p+1)(m-p+1)$. So total computation time is $n(\log n) + (p+1)(m-p+1)(\log n) + O(m)$ at most.

Formal description of our $O(p(m-p)\log n)$ algorithm is omitted.

4. Comparison of Algorithms

Hirschberg has shown $O(np)$ and $O(p(m-p)\log n)$ algorithms which seem to be best among previously known algorithms [3]. In this paper, $O(n(m-p))$ algorithm is presented. The efficiency of these algorithms depends on given input strings and we need to select a suitable algorithm for a specific application.

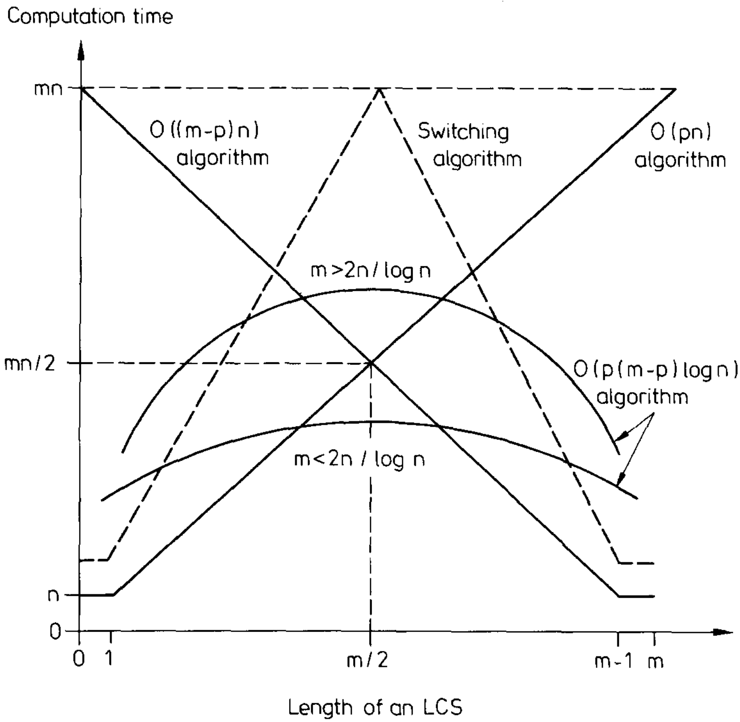

Fig. 4. The computation time of four algorithms

Figure 4 shows the computation time of these three algorithms, where the coefficients of the computation time of these three algorithms are assumed to be the same.

Generally speaking, when the length of an LCS is expected to be short, $O(pn)$ algorithm is better and when the length of an LCS is expected to be close to m, $O((m-p)n)$ algorithm is better.

When the length of an LCS is not estimated beforehand, several approaches are possible. Parallel computation of the above two algorithms would offer shorter running time. Instead of parallel computation, we can execute both algorithms by switching each other step by step. This switching algorithm requires theoretically the running time of $O(\mathrm{Min}(pn, (m-p)n))$, but the coefficient doubles. $O(p(m-p)\log n)$ algorithm may be feasible in this case against the worst case.

Acknowledgment. The authors wish to thank Mr. Jan L. Goodsell (now with the Trilogy Corporation) for his helps in reviewing this paper.

References

1. Aho, A.V., Hirschberg, D.S., Ullman, J.D.: Bounds on the Complexity of the Longest Common Subsequence Problem. J. ACM **23**, 1, 1-12 (1976)

2. Hunt, J.W., Szymanski, T.G.: A Fast Algorithm for Computing Longest Common Subsequences. C. ACM **20**, 5, 350–353 (1977)
3. Hirschberg, D.S.: Algorithms for the Longest Common Subsequence Problem. J. ACM **24**, 4, 664–675 (1977)
4. Hirschberg, D.S.: An Information-Theoretic Lower Bound for the Longest Common Subsequence Problem. Inform. Process. Lett. **7**, 1, 40–41 (1978)
5. Kambayashi, Y., Nakatsu, N., Yajima, S.: Hierarchical String Pattern Matching Using Dynamic Pattern Matching Machines. Proc. IEEE COMPSAC **79**, 813–818 (1979)
6. Mukhopadhyay, A.: A Fast Algorithm for the Longest-Common-Subsequence Problem. Inform. Sci. **20**, 69–82 (1980)

Received May 6, 1980/June 16, 1982