1985

The Longest Common Subsequence Problem Revisited

A. Apostolico

C. Guerra

Report Number:
85-543

THE LONGEST COMMON SUBSEQUENCE PROBLEM REVISITED

A. Apostolico
C. Guerra

CSD-TR-543
October 1985
Revised June 1986

ABSTRACT

This paper re-examines, in a unified framework, the problem of finding a longest common subsequence (LCS) of two strings, and proposes simple and generally faster implementations for most known approaches. Let l be the length of an *LCS* between two strings of length m and $n \geq m$, respectively, and let s be the alphabet size. The first revised strategy follows the paradigm of a previous $O(ln)$ time algorithm by Hirschberg. The new version can be implemented in time $O(lm \cdot \min\{\log s, \log m, \log(2n/m)\})$, which is profitable when the input strings differ considerably in size (a looser bound for both versions is $O(mn)$). A natural offspring of this algorithm is also presented which uses only linear space and has the same time bound, except for an additive term $O(m \log m)$. While most existing algorithms use linear space in order to compute only l, the only previously known algorithm computing an LCS in linear space required never less than time $\Theta(nm)$. Another algorithm presented here improves on the Hunt-Szymanski algorithm. This latter takes time $O((r+n)\log n)$, where $r \leq mn$ is the total number of matches between the two input strings. Such a performance is quite good ($O(n \log n)$) when $r \sim n$, but it degrades to $\Theta(mn \log n)$ in the worst case. On the other hand, the variation presented here is never worse than linear-time in the product mn. The exact time bound derived for this variation is $O(m \log n + d \log(2mn/d))$, where $d \leq r$ is the number of *dominant* matches (elsewhere referred to as *minimal candidates*) between the two strings. Finally, a scheme reminiscent in part of that of Hunt-Szymanski is used to set up a natural $O(n(n-k+1))$ time algorithm suitable for similar strings of nearly equal lengths. The bounds $O(n(m-l+1)$ and $O(m(m-l+1)\log n)$ were already obtained elsewhere, though via more involved constructions. It will also be observed that the techniques developed in this paper enable to reduce the second one of such bounds to $O(m(m-l+1) \cdot \min\{\log m, \log s, \log(2n/l)\})$, hence to $O(m(m-l+1))$ for constant alphabet size. All algorithms require an $O(n \log s)$ preprocessing that is nearly standard for the LCS problem, and they make use of simple and handy auxiliary data structures.

Key words and Phrases: Design and analysis of algorithms, Longest common subsequence, Dictionary, Finger-tree, Characteristic tree, Dynamic programming, Efficient merging of linear lists.

1. PRELIMINARIES

We consider *strings* $\alpha,\beta,\gamma,\ldots$ of *symbols* on an *alphabet* $\Sigma=(\sigma_1,\sigma_2,\cdots\sigma_s)$ of size s. A string is identified by writing $\alpha=a_1a_2...a_m$, with $a_i\in\Sigma$ $(i=1,2,...,m)$. The *length* of α is m. A string $\gamma=c_1c_2...c_l$ is a *subsequence* of α if there is a mapping $F:[1,2,...,l]\rightarrow[1,2,...,m]$ such that $F(i)=k$ only if $c_i=a_k$ and F is monotone and strictly increasing. Thus γ can be obtained from α by deleting a certain number of (not necessarily consecutive) symbols.

Let $\alpha=a_1a_2...a_m$ and $\beta=b_1b_2...b_n$ be two strings on Σ with $m\le n$. We say that γ is a *common subsequence* of α and β iff γ is a subsequence of α and also a subsequence of β. The *longest common subsequence* (LCS) *problem* for input strings α and β consists of finding a common subsequence γ of α and β of maximal length. Note that γ is not unique in general.

A dynamic programming strategy to compute the LCS of α and β in $\Theta(mn)$ time and space is readily set up [HC, WF]. Consider the integer matrix $L[0...m,0...n]$, initially filled with zeroes. The following code transforms L in such a way that $L[i,j]$ $(1\le i\le m, 1\le j\le n)$ contains the length of an LCS between $\alpha_i=a_1a_2...a_i$ and $\beta_j=b_1b_2...b_j$.

```
for i=1 to m do
    for j=1 to n do if a_i=b_j then L[i,j] = L[i-1,j-1] + 1
        else L[i,j] = Max{L[i,j-1],L[i-1,j]}.
```

The correctness of this strategy rests on the fact that the final entries of L must observe the following, easy to check, relations:

$$L[i-1,j]\le L[i,j]\le L[i-1,j]+1;$$

$$L[i,j-1]\le L[i,j]\le L[i,j-1]+1;$$

$$L[i-1,j-1]\le L[i,j]\le L[i-1,j-1]+1.$$

It is also easy to show that an LCS can be retrieved, from the L-matrix in final form, in $O(n)$ time. This suggests that the L-matrix may be highly redundant. More efficient algorithms try to limit the computation only to those entries of the matrix which convey essential information. In order to be more precise, we need a few additional definitions.

The ordered pair of *positions* i and j of L, denoted $[i,j]$, is a *match* iff $a_i=b_j=\sigma_t$ for some t, $1\le t\le s$. In the following, r will denote the number of distinct matches between α and β. If $[i,j]$ is a match, and an LCS $\gamma_{i,j}$ of α_i and β_j has length k, then k is the *rank* of $[i,j]$. The match $[i,j]$ is *k-dominant* if it has rank k and for any other pair $[i',j']$ of rank k either $i'>i$ and $j'\le j$ or $i'\le i$ and $j'>j$. The total number of dominant matches will be denoted by d. Let l be the length of an LCS of α and β. It is seen [HI] that, for any $k\le l$, there must be at least one k-dominant match, and that, moreover, there is at least one LCS $\gamma=c_1c_2\cdots c_l$ such that c_k corresponds to a k-dominant match ($k=1,2,...,l$). Thus, computing the k-dominant matches ($k=1,2,...,l$) is all is needed to solve the LCS problem. For a large or a-priori unknown alphabet, and within the decision tree model of computation where comparisons are restricted to give outcomes in $[=,\neq]$, the worst case time lower bound for the LCS problem is $\Theta(mn)$ [AH]. The related preprocessing charges $\Theta(n\log s)$ time and $\Theta(n)$ space.

However, it is easy to see that once all k-dominant matches are available, then $O(m)$ time suffices to retrieve γ. Most known approaches to the LCS problem require $\Theta(n+d)$ space. By contrast, the dynamic programming implementation presented in [HC] takes never more than $\Theta(n)$ space, though never less than $\Theta(mn)$ time.

As an illustration of the concepts introduced thus far, Fig. 1 below displays the nontrivial portion of the final L-matrix for the strings $\alpha=abcdbb$ and $\beta=cbacbaaba\$$, where $\$$ is a 'joker' symbol not in Σ, but matching any symbol of Σ. Entries that correspond to matches are encircled. Emboldened circles circumscribe dominant matches, and boundaries are traced to separate regions with constant L-entry. For our convenience, we will henceforth speak of the L-matrix of α and β referring to the slightly augmented version presented below. Notice that appending $\$$ to β has the effect of transforming each instance of our problem into a corresponding instance with $r\ge m$.

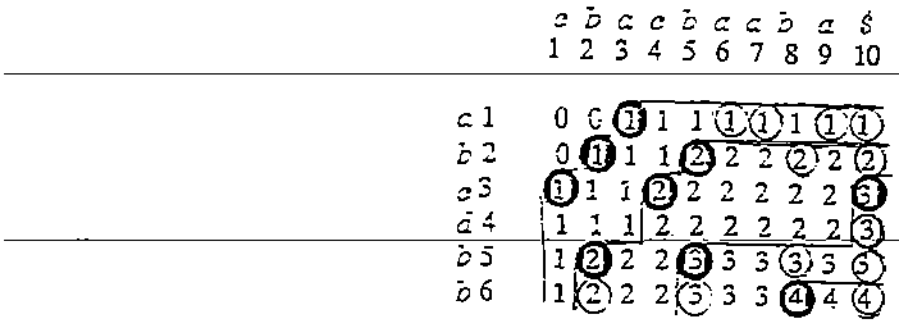

Figure 1
The augmented L-matrix for the strings $\alpha = abcabb$ and $\beta = cbacbaaba$.

The remainder of this paper is organized as a self-contained excursion through several solutions to the LCS problem, and the style of presentation is sometimes deliberately semi-tutorial. In Section 2, we present a strategy which is reminiscent of that in [HI]: only, the direction according to which matching pairs are scanned is inverted, and gain is achieved by exploiting some information on the structure of β (the longer string). The simplest preprocessing of β which makes such information available would require $\Theta(ns)$ time and space, an undesirable cost that is curbed to $\Theta(n)$ in Section 3. Our first new algorithm is also presented in that section, and it is shown there that it can be set up to run in $O(lm \cdot \min\{\log s, \log m, \log(2n/m)\})$ time. We will show in Section 7 that this algorithm can be adapted to run in linear space, at the expense of an extra $m\log m$ term in the above time bound.In Section 4, we revisit the Hunt-Szymanski strategy. We highlight there that an improved version of this strategy is unlikely to be obtainable through the mere rescheduling of the primitive operations contained in Hirschberg's (the expert reader might want to skip this part). We also show, however, that a simple adaptation of the Hunt-Szymanski strategy leads to a natural $O(n(n-l+1))$ time algorithm for the LCS problem. The bounds $O(n(m-l+1))$ and $O(m(m-l+1)\log n)$ were already obtained in [NY], though via more elaborate constructions. It is easily seen, however, that combining some of the techniques of this paper with the construction in [NY] yields a time bound $O(m(m-l+1)\min\{\log s, \log m, \log(2n/l)\}$, hence $O(m(m-l+1))$ if the alphabet size is a constant. Section 5 is devoted to the dissection of the original algorithm in

[HS], which is then reassembled in such a way that the sites of possible speed-ups be more apparent. Section 6 introduces a data structure similar to those recently proposed for the efficient merging of linear lists [ME, BT, BW], but better fit to the special case of our interest. In this section, we give also a construction that uses such a structure to achieve a time bound of $O(m\log n \div d\log(2mn/d))$.

2. HIRSCHBERG's STRATEGY REVISITED

We start by outlining an alternate $\Theta(mn)$ time algorithm for the LCS of α and β. Also this algorithm accepts an $(m+1)(n+1)$ input L-matrix filled with zeroes. The output is again the final L-matrix. The k-dominant matches for each k are identified as follows: the *dummy* pair [0,0] is obviously a 0-dominant match. Suppose now that all the $(k-1)$-dominant matches are known. Then the k-dominant matches can be obtained by scanning the unexplored region of the L-matrix from right to left and top-down, until a stream of matches is found occurring in some row i. The leftmost such match is the k-dominant match $[i,j]$ with smallest i-value. The scan continues at next row and to the left of this match, and this process is repeated at successive rows until all the $(k-1)$-*th* region has been scanned (and identified). Notice that the list of k-dominant matches, in the same order as they are produced, unambiguously encodes the lower border of the k-*th* region. The list (with no more than m entries) produced at some stage suffices to guide the searches involved at the subsequent stage, which highlights that linear space is sufficient if one wishes to compute only the length of γ. (Elaborating on this idea, Hirschberg set up an algorithm [HC], different from that being discussed here, that takes linear space though never less than quadratic time to retrieve γ.)

The approach in [HI] corresponds to an efficient implementation of the schedule of operations which was just described. More precisely, the 0,1,2,...,l-*th* regions of L are produced in succession, on the basis of the following criterion:

1) the topmost and leftmost match in the unexplored region is a dominant match;

2) if $[i,j]$ is a k-dominant match, then any other k-dominant match with $i' > i$ must lie to the left of $[i,j]$,i.e., $j' < j$.

With some preprocessing on α and β, Hirschberg's algorithm performs in time $O(nl+n\log s)$ and space $O(d+n)$ [HI]. Since the product of that preprocessing is necessary to our algorithm as well, we now describe it in detail. For each distinct symbol σ in α, the preprocessing consists of producing the following:

A) A list σ-OCC of all positions of β (in increasing order) which correspond to occurrences of σ†, for each $\sigma \in \Sigma$. In our case, β is always replaced by $\beta\$$, whence the last entry of any σ-OCC list is always $n+1$.

B) The count $N(\sigma)$ of all distinct occurrences of σ in β. In the following, we will retain $N(\sigma)$ exactly as defined here, i.e., without counting $\$$ as an occurrence of σ.

At most s such lists need to be produced, at a cost of $O(n\log s)$. The advantage brought about by the σ-OCC lists is twofold (refer to [HI] for details):

1) the identification of each region can be carried out by traveling (right to left) on such lists rather than on the rows of L, in such a way that each region is identified in $O(n+m)$ steps.

2) if $l=k$, then this circumstance can be detected at once following the identification of the k-*th* border, since in this case none of the σ-OCC lists features non-joker matches in the unexplored region of L.

The alternate strategy by Hunt and Szymanski [HS] also makes use of the σ-OCC lists.

†The σ-OCC lists can be easily allocated in $O(n)$ space in such a way that each one of them is accessible randomly.

We describe now an algorithm similar to that of Hirschberg but characterized with a bound of $O(lm+r+n\log s)$, inclusive of preprocessing. This may be better than $O(ln+n\log s)$ when $m<n$ and r is comparable to m. However, we are only interested in it as a starting point for our discussion, since both its description and evaluation are very simple.

We use an array of integers $PEBBLE[1..m]$, initialized to 1, the role of which shall become apparent later. If $a_i=\sigma_p$, $PEBBLE[i]$ either points to (the location of) an entry $j\leq n$ of σ_p-OCC, and is said to be *active*, or it points to (the location of) $n+1$ and is *inactive*.

Our algorithm consists of l *stages*, stage k being defined as the set of operations involved in identifying all the k-dominant matches. Let a match be k-*internal* ($k=1,2,...,l$) if its rank is larger than k. Then stage k, $1 \leq k \leq l$ begins with all active entries of $PEBBLE$ pointing to $(k-1)$-internal matches and ends with all active entries of $PEBBLE$ pointing to k-internal matches. During stage k the *pebbles*: $PEBBLE[k]$, $PEBBLE[k+1]$, ..., $PEBBLE[m]$, are considered in succession (indeed, no $PEBBLE[i]$ with $i<k$ can be active at stage k). The k-dominant matches detected are appended to the k-*th* list in the array of lists $RANK$ (through the concatenation operation '[ ]'). The entire process terminates as soon as there are no active pebbles left. In practice, one might profitably substitute the *for* loop of *Algorithm* 1 below with a walk through the list of active pebbles, thus gaining a considerable speed-up in some extreme cases. Since only deletions would take place from such a list, its maintenance does not pose special problems. However, the mere introduction of the list of active pebbles does not lead to any improvement in the time bound. Hence we elect to present the less cluttered version below. Although it is not crucial to the general paradigm around which *Algorithm* 1 is built, we prefer to resort from the beginning to an auxiliary table called $SYMB$. This table will be necessary in some of the subsequent versions of this algorithm, and it is defined as follows. $SYMB[j]=k$, if $b_j=\sigma_p$ and j is the k-*th* entry in σ_p-OCC. Thus the table $SYMB$ enables constant time access to the entry in the σ-OCC list that corresponds to the symbol of β occurring at any position. We stipulate that, each time $PEBBLE[i]$ is being handled by the algorithm ($i=1,2,...,m$), then $SYMB[n+1]$ takes the

value $N(a_i)$. The table *SYMB* can be prepared in linear time from the σ-*OCC* lists, quite easily, and we will not spend time on it. Within *Algorithm* 1, *SYMB* is used to speed up the advancement of some *PEBBLE*[i'], if $a_{i'} = a_i$ for some $i < i'$, and T, the *threshold*, was not changed since row i.

Algorithm 1

```
0  for i = 1 to m do PEBBLE[i] = 1; (initialize pebbles)
1  k = 0
2  while there are active pebbles do (start stage k+1)
3  begin T=n+1; k=k+1; RANK[k]=Λ;
4     for i = k to m do (advance pebbles)
      begin
5        t = T;
6        if a_i-OCC[PEBBLE[i]] < T then
         (record a k-dominant match; update threshold)
7           begin RANK[k] = RANK[k] || [i,a_i-OCC[PEBBLE[i]]];
8                 T = a_i-OCC[PEBBLE[i]]
            end;
         (advance pebble, if appropriate)
9        if a_i = b_t
10          then PEBBLE[i] = SYMB[t] + 1
11          else while a_i-OCC[PEBBLE[i]] < t do PEBBLE[i] = PEBBLE[i] + 1
      end;

 end.
```

With each σ-*OCC* list visualized as stretched along each of the corresponding rows of the L-matrix, Fig. 2 depicts the positions occupied by the pebbles at the beginning of each of the stages performed by *Algorithm* 1 on the input strings of Fig. 1. To illustrate the action of the algorithm, we trace its stage 1, which produces the first boundary (consisting of all the 1-dominant matches). The algorithm starts by assigning the value '10' to both T and t, i.e., the variables which will be used to store the current and previous threshold, respectively (lines 3,5). Next, it compares the first occurrence of symbol $a=a_1$ in β (line 6). Such test is passed (3<10), whence the first 1-dominant match is detected and appended to the list *RANK*[1] (line 7).

Moreover, T is updated to the new value '3' (line 8). At this point, the algorithm tries to advance *PEBBLE*[1] onto a 1-internal match. By definition of S, a_i matches S. Moreover, in view of our convention, the current value of *SYMB*[10] is $N(a) = 4$. Thus line 10 is executed following the test of line 9, with the effect of bringing *PEBBLE*[1] to its rightmost position on a-*OCC*, and rendering it inactive. As *Algorithm* 1 proceeds to consider $a_2 = b$, the test of line 6 prompts the detection and recording of a new 1-dominant match on column 2 of the L-matrix. This is followed by the advancement of *PEBBLE*[2] which is thus brought on column 5. *PEBBLE*[3] is subjected to a similar treatment. In our example, the first three pebbles provide all the dominant matches for the first stage. When *PEBBLE*[4] is considered, it does not pass the test of line 6; line 11 has no effect and this pebble is left in its inactive status. The last two pebbles are also left in their original position. We encourage the reader to carry out for himself the remainder of this example, with the aid of Fig. 2.

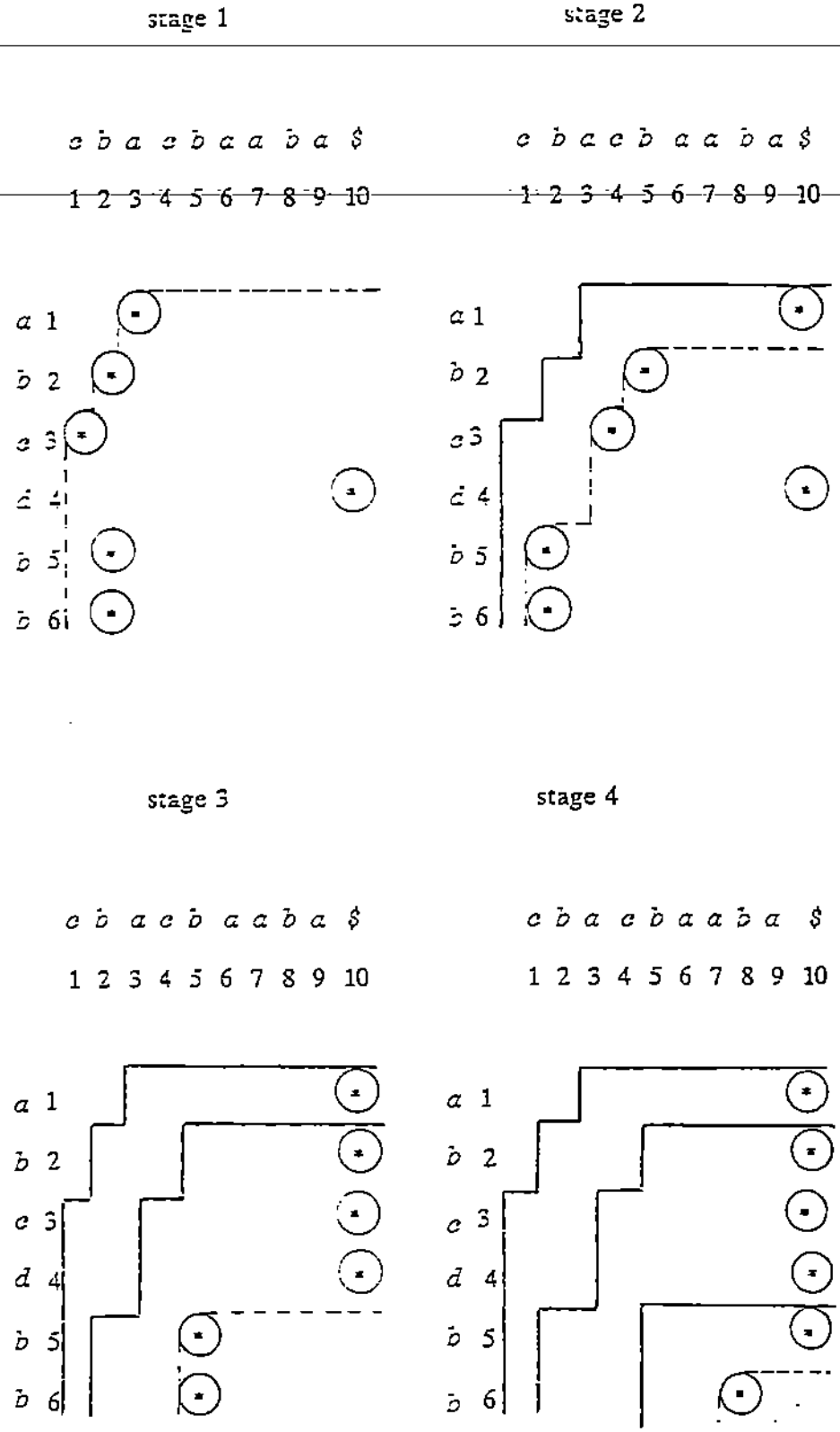

Figure 2

Pebbling the L-matrix of Fig. 1 through the four stages of *Algorithm* 1. The figure relative to stage k ($1 \le k \le 4$) displays the initial positions of the pebbles for that stage. At this point, each active pebble falls either on a k-dominant match or on a k-internal match.

In general, it is easy to check that *Algorithm* 1 maintains the following invariant condition: whenever some pebble is being considered for the k-*th* time, then there can be no match of rank k on the same row and to the left of that pebble. In other words, if such pebble is active, then the match which it points to must be either a k-dominant match or a k-internal match. Unlike the algorithm in [HI], *Algorithm* 1 uses the following heuristic: matches whose ranks have been already determined are not reconsidered at subsequent stages.

Theorem 1. *Algorithm* 1 takes time $O(lm+r)$

Proof.

During stage k, $m-k+1$ pebbles are considered in succession. Each pebble either is advanced some position to the right or it is not moved. The number of advances on one row is bounded by the number of matches on that row, thus the total number of advances is bounded by r. A pebble is considered exactly once during each stage, thus the number of times a pebble can stay put is bounded by l, which yields a total of lm. □

We remark that the above strategy requires $O(lm+r+n \log s)$ inclusive of preprocessing and $O(d)$ ($O(m)$) space to find the LCS (the length of the LCS). If $r < lm$ and m is much smaller than n, this is better than the $O(ln + n \log s)$ in [HI]. When r is large compared to ml, the strongest cause of inefficiency becomes the inner *while* loop of *Algorithm* 1, which generates the $O(r)$ term. We devote the next section to curb this term.

3. IMPROVING EFFICIENCY

The operation of the inner *while* loop of *Algorithm* 1 is basically that of moving the i-*th* pebble to the leftmost position of β which is larger than the value t of the threshold which was updated last. This might involve an uncontrolled number of advancements on the a_i-*OCC* list. The information needed to move the i-*th* pebble is embodied in the structure of β : through the *while* loop we want to move the pebble to the leftmost occurrence of a_i in β which falls past b_t.

For s constant and/or small compared to n (like, for example, in the analysis of molecular sequences [MA, SK]) this information can be made available by a simple preprocessing of the string $\beta\$$. We prepare the $s(n+1)$ table $CLOSEST[\sigma_1...\sigma_s,1...n+1]$ which is filled as follows. $CLOSEST[\sigma_p,n+1]=n+1$, for $p=1,2,...,s$. For $j=1,2....n$, $CLOSEST[\sigma_p,j]=j' \geq j$, where $b_{j'}$ matches σ_p but $b_k \neq \sigma_p$ for $j < k < j'$.

For example, the table *CLOSEST* associated with β=*cbacbaaba* is as follows.

	1	2	3	4	5	6	7	8	9	10
	c	*b*	*a*	*c*	*b*	*a*	*a*	*b*	*a*	$
a	3	3	3	6	6	6	7	9	9	10
b	2	2	5	5	5	8	8	8	10	10
c	1	4	4	4	10	10	10	10	10	10

The preparation of *CLOSEST* in $\Theta(sn)$ time is straightforward. For instance, it can be based on the alternative definition of *CLOSEST* offered by the following, equally straightforward, Lemma.

Lemma 1. Define $CLOSEST[\sigma_p,n+1] = n+1$, $1 \leq p \leq s$. Then, for $j=1,2,...n$, $CLOSEST[\sigma_p,j] = j$ for p such that $\sigma_p=b_j$ and $CLOSEST[\sigma_p,j]=CLOSEST[\sigma_p,j+1]$ for all other values of p.

In addition to the array *CLOSEST*, we may also make use of the auxiliary table *SYMB*, already introduced in connection with *Algorithm* 1.

With these two implements, the inner *while* loop can be removed from *Algorithm* 1, and substituted there with the simple assignment:

$$PEBBLE[i] = SYMB[CLOSEST[a_i, t]]$$

We refer to the algorithm obtained with this modification as *Algorithm* 2. Notice that *Algorithm* 2 does not access *CLOSEST* if $a_i = b_t$.

Theorem 2. *Algorithm* 2 finds an LCS in time $O(lm + sn + n \log s)$ and space $O(d + sn)$.

The proof of Theorem 2 is straightforward and thus is omitted. Notice that, of the three terms in the above upper bound for the time, two are charged by the preprocessing. The processing phase charges a time at most proportional to the product lm. Since all dominant matches are detected during this phase, then lm must be an upper bound for the number of such matches.

The term sn becomes huge as s approaches m. To circumvent this, we replace the table *CLOSEST* with a new table which we call $CLOSE[1...n+1]$, and which is regarded as subdivided into consecutive blocks of size s. Letting $p = j \bmod s$ $(j=1,...,n)$, $CLOSE[j]$ contains the left-most position not smaller than j where σ_p occurs in $\beta\$$. $CLOSE[n+1]$ is set to $n+1$.

The array $CLOSE[j]$ can be obtained in time $\Theta(n)$, for instance by scanning the σ-*OCC* lists one at a time in succession while filling the entries of *CLOSE* relative to each symbol. The manipulations involved in connection with each σ-*OCC* list are similar to the merging of two lists, one of size n/s and the other of size $N(\sigma)$, the number of occurrences of σ in β. Thus each such merge involves less than $n/s + N(\sigma)$ comparisons. There are s such merges totaling less than $s(n/s) + \sum_p N(\sigma_p) = 2n$ comparisons. The total number of assignments is obviously n. We

leave the details of this construction to the reader.

We now assume that the table *CLOSE* has been prepared, and we let $closest[\sigma_p, j]$ be the function whose tabular version is the array *CLOSEST* discussed above.

Lemma 2. For any given p and j $(1 \leq p \leq s, 1 \leq j \leq n)$, $closest[\sigma_p, j]$ can be retrieved from $CLOSE[j]$ and from the σ_p-*OCC* list in time $O(\log s)$.

Proof.

We prove our claim by giving an explicit strategy to identify $closest[\sigma_p, j]$ from p and j. First, compute $j' = (j \ div \ s)s + p$, where *div* stands for the integer division operation. Three cases have to be distinguished according to whether $j'=j$, $j'<j$ or $j'>j$. If $j'=j$ then $CLOSE[j]=closest[\sigma_p, j]$ by definition. Our strategy returns $CLOSE[j]$ as the answer in constant time. Next, suppose $j'<j$. If $CLOSE[j'] = j'' > j$, then clearly $j'' = closest[\sigma_p, j]$. Otherwise $closest[\sigma_p, j]$ is not smaller than $j'' = CLOSE[j']$ but not larger than $j''' = CLOSE[j'+s]$. Now $SYMB[j'']$ and $SYMB[j''']$ point to the corresponding entries in the σ_p-*OCC* list, and there can be no more than s entries in σ_p-*OCC* between these two entries. Thus $closest[\sigma_p, j]$ can be retrieved in $\log s$ steps by performing a binary search in this segment of σ_p-*OCC*. The case $j'>j$ is handled along the same lines as the case just discussed. □

We can now set up still another version of our LCS algorithm, which we call *Algorithm* 3. *Algorithm* 3 does not differ from *Algorithm* 2, except that the assignment:

$$PEBBLE[i] = SYMB[CLOSEST[a_i, t]]$$

is now replaced by:

$$PEBBLE[i] = SYMB[closest[a_i, t]]$$

Theorem 3. *Algorithm* 3 finds an LCS in time $O(lm \log s + n \log s)$ and space $O(d + n)$.

Proof.

Each call to *closest* charges $O(\log s)$ time, in view of Lemma 2. The generic stage can prompt no more than m such calls, and there are precisely l stages. □

If s can be regarded as a small constant, then *Algorithm* 3 takes time $O(\text{Max}\{lm, n\})$. Thus, in particular, the LCS problem between two strings of lengths m and $n=\Omega(m^2)$ has the same time complexity of the pattern matching problem [AU] for the same strings, except that preprocessing is applied here to the 'text' rather than to the 'pattern'. We recall that, under the assumption of constant alphabet size, the algorithm by Masek and Patterson [MP] requires $O(mn/\log n)$ time for all possible values of the ratio n/m.

If n is larger than m^2 and s is larger than m, then limiting the preprocessing to the subset of Σ containing only those symbols which appear in α enables substitution of the $\log s$ in the bound of Theorem 3 with $\min\{\log s, \log m\}$. In intermediate situations, some improvement in the performance of *Algorithm* 3 can be gained from using searching techniques with auxiliary *fingers* [BT,BW,ME]. The unexperienced reader shall become more familiar with such techniques as we proceed with our discussion. For the time being, it will do to mention that finger techniques obtain the result that consecutive search intervals on the same σ-*OCC* list do not overlap during each individual stage.

It is not difficult to see that, with the sole use of fingers, the work at each stage can be bounded by $O(m \log(2n/m))$, thus yielding an overall bound of $O(lm \log(2n/m) + n \log s)$. This latter construction has been proposed very recently in a paper which appeared in the literature during the development of our work [HD]. However, in view of the observation, made above, concerning the cases where s is close to n and $m \ll n$, such a performance conveys some advantage on Hirschberg's only in the rather narrow spectrum of situations where, roughly speaking, $m \log(n/m)$ is small compared to n while n is small compared to m^2. By contrast, the

algorithm which is obtained by combined use of the restriction (to the symbols of α) of the table *CLOSE* and finger techniques, performs in time $O(lm \cdot \min\{\log s, \log m, \log(2n/m)\} + n \log s)$, which is never worse than the time bound of the algorithm in [HI], and can be better than that time bound in a wide variety of instances.

4. A SMOOTH TRANSITION AND A SIMPLE $O(n(n-l+1))$ ALGORITHM

Sometimes the number r of matches can be assumed to be small compared to m^2 (or to the expected value of lm). In these cases, it is much desirable to have an algorithm whose running time is bounded by some slowly growing function of r. This observation is the basis of the LCS algorithm by Hunt and Szymanski [HS], which exhibits a time bound of $O((n + r) \log n)$. The Hunt-Szymanski algorithm (hereafter, *HS* for short) eludes the risk, inherent to Hirschberg's strategy, of wastefully reconsidering the same match many times: this is avoided by establishing, row after row, the ranks of all matches in each row. The main disadvantage of *HS* is that its performance degenerates as r gets close to mn: in these cases this algorithm is outperformed by the algorithm in [HI], which exhibits a bound of $O(ln)$ in all situations.

In this section we examine an LCS algorithm which does not quite coincide with the Hunt-Szymanski strategy. Our discussion will lead to a simple $O(n(n-l+1))$ algorithm, suitable for similar strings. It will also help understand better the developments of the following sections.

The Hunt-Szymanski approach consists of generating all the k-dominant matches, row after row. Using the table *CLOSE* introduced in connection with *Algorithm* 3 above, one might set up a strategy which looks similar to that in [HS], as follows. Let *THRESH*[1...m+1] be an array of *thresholds*, all of whose locations are initialized to the value n+1.

```
Algorithm 4
for i = 1 to m do
 begin PEBBLE[i]=j=1;
    while PEBBLE[i] is active do
     begin
      if a_i-OCC[PEBBLE[i]] < THRESH[j]
      then begin T = THRESH[j]; THRESH[j] = a_i-OCC[PEBBLE[i]];
              record new dominant match; PEBBLE[i] = SYMB[closest[a_i,T]]
           end;
      if a_i-OCC[PEBBLE[i]] = THRESH[j] then PEBBLE[i] = PEBBLE[i]÷1;
      j = j+1;
     end;
 end.
```

We leave it to the reader to recognize *Algorithm* 4 as nothing but a re-scheduling of the operations of *Algorithm* 3, and to evaluate its complexity. He will find it not surprising that both achieve the same time bound. This schedule has some advantages on the previous one in some extreme situations.

However, *Algorithm* 4 does not relate to the Hunt-Szymanski strategy as closely as *Algorithm* 3 relates to Hirschberg's. Indeed, it neglects the basic motivation behind *HS*, namely, the efficient management of those situations where r is expected to be close to n. In fact, there is no strong necessary relation between r and the size of the list *THRESH* which is to be handled at any given row of L. Thus, as long as we allow an unpredictable number of elements of *THRESH* to be considered in connection with any row of L, it is unlikely that an algorithm with the control structure of *Algorithm* 4 could be time-bounded in terms of r. An algorithm with a time bound based on r should be set up as a sequence of more elementary manipulations, each one of which can be charged to a distinct match. We shall see soon that the row-by-row approach of *Algorithm* 4 is compatible with such an objective. Interestingly, the same cannot be said of the approach of the preceding sections [AA].

Before disposing of the scheme of *Algorithm* 4, we show that it subtends a natural $O(n(n-l+1))$ strategy, suitable for similar strings. We shall only outline such an algorithm, the

details of which are tedious but straightforward. Our construction is based on the following simple observations. Let $m=n$ and imagine running *Algorithm* 4 α=*abcdbb* and β=*cbacba*. The L-matrix would consist then of the first six columns of the matrix of Fig. 1, plus a suitable seventh column for \$. The final set of thresholds would be {1,2,5,7}. As can be deduced from Fig. 1, only $k=n-l=2$ positions of α are missing from this final set of thresholds, namely, the positions 3 and 4. We call each such missing position a *gap*, and we use *COTHRESH* to refer to the sorted list of gaps. Clearly, *COTHRESH* can be deduced from *THRESH*, and vice versa. If $k<n$, then *COTHRESH* represents a more compact encoding of the final set of thresholds than *THRESH*. Unfortunately, this is not always true at any stage of the computation, since *THRESH* can be occasionally more sparse and *COTHRESH* correspondingly denser. However, it is straightforward to to see that, in general and for each value of i, the total number of gaps falling within the first i positions of either the i-th row or the i-th column of the L-matrix cannot be larger than k. Indeed, there must be an equal number of gaps, say k', in the i-th row and in the i-th column. If $k'>k$, then the matches contributed to any LCS by the upper-left $i \cdot i$ submatrix of the L-matrix cannot exceed $i-k'$. Since the remaining portion of the L-matrix cannot contribute more than $n-i$ matches, it must be $l \le n-i+i-k'<n-k$.

The above observations suggest that an LCS can be found by extending, one row and one column at a time, the partial solutions relative to all upper-left square submatrices of the L-matrix. At the i-th iteration, we pebble from left to right the $O(k)$ cells of two *COTRESH* lists, one for row i and the other for column i. If the p-th cell of, say, the row-*COTHRESH* contains position $j<i$, and $a_i=b_j$, then $[i,j]$ is a dominant match. The p-th cell of *COTHRESH* is removed from that list. Continuing the scan, the first cell is located with an entry larger than $1+j'$, where j' is the value stored in the immediately preceding cell. Clearly, for some $i'<i$, $[i',j'+1]$ is a dominant match having the same rank of $[i,j]$, whence gap $j'+1$ is inserted in *COTHRESH*. Simple extra bookkeeping enables the retrieval of an LCS at the end of the process. The algorithm performs n iterations, each at a cost of $O(k)$ time.

We start now the presentation of a new implementation of the Hunt-Szymanski strategy, which is not subject to the worst case degenerations of *HS*. Moreover, we will be able to draw for our algorithm a bound which is expressible in terms of d instead of r, namely, $O(m \log n + d \log(2mn/d))$. Intuitively, and with reference to *Algorithm* 4, this can be obtained by dynamically swapping the roles of the two lists which are merged at each row of L, namely, *THRESH* and a_i-*OCC*.

For the reader's convenience, we start our discussion by reproducing *HS* below as *Algorithm* 5. Notice that *HS* preprocesses β to obtain an appropriate number of replicas of the reverse of each σ-*OCC* list: such lists are called *MATCHLIST*s in [HS].

Algorithm 5: 'HS'
element array $\alpha[1{:}m]$,$\beta[1{:}n]$; integer array *THRESH*$[0{:}m]$; list array *MATCHLIST*$[1{:}m]$;
pointer array *LINK*$[1{:}m]$; pointer *PTR*;
begin
(*PHASE* 1: initializations)
  for $i = 1$ to m do
  set MATCHLIST$[i] = \{ j_1, j_2, \ldots, j_p \}$ *such that* $j_1 > j_2 \cdots > j_p$
    and $a_i = b_{j_q}$ *for* $1 \le q \le p$
  set THRESH$[i] = n+1$ *for* $1 \le i \le m$; *THRESH*$[0] = 0$; *LINK*$[0] =$ *null*;
(*PHASE* 2: find k-dominant matches)
  for $i = 1$ to m do
   for j *on MATCHLIST*$[i]$ do
    begin
     find k *such that THRESH*$[k-1] < j \le$ *THRESH*$[k]$;
     if $j <$ *THRESH*$[k]$ **then**
      begin *THRESH*$[k] = j$; *LINK*$[k] =$ *newnode*$(i, j,$ *LINK*$[k-1])$ **end**
    end
(*PHASE* 3: recover LCS γ in reverse order)
  $k =$ *largest* k *such that THRESH*$[k] \ne n+1$; *PTR* = *LINK*$[k]$;
  while *PTR* $\ne$ *null* **do**
   begin *print the match* $[i, j]$ *pointed to by PTR*; *advance PTR* **end**
end.

The principle of operation of *HS* is transparent: by scanning the *MATCHLIST* associated with the i-*th* row, the matches in that row are considered in succession, from right to left; through a binary search in the array *THRESH*, it is assessed whether the match being considered

represents a k-dominant match for some k. In this case the contents of $THRESH[k]$ is suitably updated. We remark that considering the matches in reverse order is crucial to the correct operation of *HS* (the reader is referred to [HS] for details). The total time spent by *HS* is bounded by $O((r+m)\log n + n\log s)$, where the $n\log s$ term is charged by the preprocessing. The space is bounded by $O(d+n)$. As mentioned, this time performance is quite good whenever r is comparable to n: in such instances, the worst case time bound becomes in fact $O(r\log n) \sim O(n\log n)$. However, this performance degenerates to $\Theta(n^2\log n)$ as both r and m get close to n.

5. A MODIFIED PARADIGM FOR HS

The objective of this section is to rearrange *HS* in a harmless way, but so that it be easier for us to distill possible sources of inefficiency. It will turn out that the new scheme can be implemented more efficiently than the original one, as we show in Section 6.

Our main modifications concern the second phase (i.e., finding k-dominant matches) of *HS* as presented above. Slight adjustments of the preprocessing are also required. The first innovation brought about by algorithm *HS* 1 below is that *HS* 1 does not consider all the matches in each row. Rather, *HS* 1 maintains, for each symbol, its associated *active list* of matches, the matches of any such list being characterized by the fact that they are not current thresholds. The second innovation consists of spotting all and only the new dominant matches contributed by any given active list by performing a number of primitive dictionary operations [AU] proportional to the number of these new dominant matches, i.e., independent of the current size of the active list involved.

As regards this second feature, a glance at Fig.1 shows that the bold circles of our example are roughly one half of all circles. While it is obviously always the case that $d \leq r$, there seems to be no general direct proportionality between d and r. For example, consider the following two extreme instances, both offsprings of the assumption $\alpha = \beta$. In the first extreme, we also assume that α and β both represent some permutation of the integers: thus $d = r$, but also $d = n$. In the other extreme, we set instead $\alpha = a^n$, i.e., both strings consist of n replicas of the same symbol a: thus $r = n^2$, but still $d = n$. More generally, *HS* 1 is asymptotically much faster than *HS* whenever $r = \Theta(n^2)$ while $d = O(n)$. As the above brief discussion suggests, these latter conditions are met, in particular, by pairs of nearly identical input strings. Unfortunately, there are still cases where both r and d are quadratic in the size $(m+n)$ of the input.

From now on, we shall find it more convenient for our discussion to 'pebble' the entries, rather than the locations, of the active lists.

Algorithm 6: 'HS1'

THRESH is the list of thresholds initially empty; each 'active' list $AMATCHLIST[\sigma_p]$, $p=1,2,...s$ is initialized to coincide with the corresponding σ_p-*OCC* list. The primitives *INSERT* and *DELETE* have the usual meaning, except they do nothing if the first argument is $n+1$ or the second argument is Λ. *SEARCH*(*key* , *LIST*) returns (a pointer to) the smallest element in *LIST* which is not smaller than *key* ($n+1$, if no such element exists). *SEARCH*($n+1$,*LIST*) returns $n+1$ without performing any action. Notice that all the searches performed within *HS* 1 terminate without success (i.e., *key* is not in *LIST*). The function *char*(*symbol*) returns the element of the alphabet Σ which coincides with *symbol*. By convention, $AMATCHLIST[\$] = \{n+1\} = \Lambda$.

begin
for i = 1 to m do
 begin $\sigma = char(a_i)$; *PEBBLE* = *first*(*AMATCHLIST*[σ]) ; *FLAG* = *true*;
  while *FLAG* do
   begin
    1) T = *SEARCH*(*PEBBLE* , *THRESH*) ; k=*rank*(T);
    2) if $T=n+1$ then *FLAG* = *false*;
    3) *INSERT*(*PEBBLE* , *THRESH*); *DELETE*(T , *THRESH*);
    4) *LINK*[k] = *newnode*(i , *PEBBLE* , *LINK*[$k-1$]);
    5) $\sigma' = char(b_T)$;
    6) *DELETE*(*PEBBLE* , *AMATCHLIST*[σ]);
    7) *PEBBLE* = *SEARCH*(T , *AMATCHLIST*[σ]);
    8) *INSERT*(T , *AMATCHLIST*[σ']);
   end;
 end;
 retrieve γ *as per phase 3 of HS ;*

end.

To illustrate the operation of *HS* 1, we may refer to Fig.1 and interpret it as representing the product of *HS* 1 after it has processed the matches between β = *cbacbaaba* and the symbols in the first six positions of α = *abcdbba*.... At this point, *THRESH* consists of {1,2,5,8}, and $\sigma_1 = a_7 = a$. At this particular stage of our example, it so happens that *AMATCHLIST*[a] = {3,6,7,9} coincides with the a-*OCC* list, i.e,, all occurrences of a in β could become new thresholds. *HS* 1 starts by searching for '3' in *THRESH*, which returns the entry '5'. Since $5 \neq n+1$, then *THRESH* is updated so that it becomes now {1,2,3,8} (line 3). The entry '3' is deleted from *AMATCHLIST*[a] (line 6). The algorithm now searches in *AMATCHLIST*[a] for the old threshold value '5', and this search returns the new *PEBBLE* '6'

(line 7). Finally, '5' is returned to $AMATCHLIST[char(b_5)] = AMATCHLIST[b]$ (line 8). Thus this block terminates with $FLAG = true$. When line 1 is executed next, this provokes the substitution in *THRESH* of the old entry '8' with the new entry '6', which is accompanied by the various list updates. The search of line 7 advances *PEBBLE* to position '9'. As soon as line 1 is executed again, *FLAG* is set to *false*. This will cause the exit from the *while* loop soon after the necessary updates have been performed (notice that some of the updates are dummy in this case, since $T = n+1$, and that the search of line 7 is gratuitous, since *FLAG* was set to *false*). As the final result of the management of a_7, *THRESH* has become {1,2,3,6,9}, while $AMATCHLIST[a]$ shrunk to just {7}. On the other hand, $AMATCHLIST[b]$ was given back the matches '5' and '8'.

In general, the correctness of *HS* 1 can be established as follows. First, we observe that, as long as we stay within the same iteration of the outer loop of *HS* 1, an item j removed from *THRESH* (cf. line 3) will never have to be reinserted in *THRESH*. Notice that this is true irrespective of whether $[i,j]$ is a match (i.e., irrespective of whether $\sigma = \sigma'$). Thus the insertion of line (8) must be executed after the search of line (7). It is easy to check then that the inner loop of *HS* 1 maintains the following invariant condition: if $PEBBLE = j \neq n+1$, then $[i,j]$ is a k-dominant match for some k, and, moreover, the first $k-1$ positions of *THRESH* contain values which are final for row i. As for the outer loop, after *HS* 1 has performed the i-*th* iteration, the following assertions hold.

1) The k-*th* entry of *THRESH* is the smallest position in β such that there is a k-dominant match between α_i and β.

2) $AMATCHLIST[\sigma_t]$ (t=1,2,...,s) contains all and only the occurrences of σ_t in $\beta\$$ which are not currently in *THRESH*.

We are now ready to assess a time bound for *HS* 1. The preprocessing involved in *HS* 1 is quite similar to that in [HS]. The table *char* is thought of as produced during preprocessing, within the bound of $O(n \log s)$ charged by this latter (in fact, *Algorithms* 1-3 also make implicit use of some such table). Thus each subsequent reference to this table can be assumed to take

constant time. *HS* 1 takes at least $\Theta(m)$ time, since it considers each one of the m rows, in succession. Since $n+1$ appears at the end of each *AMATCHLIST* by initialization, then *HS* 1 spends constant time in handling any *trivial* row of L, i.e., any row whose *AMATCHLIST* is found to contain currently only $n+1$.

Theorem 4. In handling all nontrivial rows, *Algorithm HS* 1 performs $\Theta(d)$ searches, insertions and deletions.

Proof.

All the searches, insertions and deletions take place in the *while* loop (lines 1–8) controlled by *FLAG*. There is a fixed number of such primitives within these lines, whence it will do to show that *FLAG* is *true* exactly $\cdot d$ times. With our assumptions, $\sigma = char(a_1)$ and $AMATCHLIST[\sigma] = char(a_1)\text{-}OCC$ is not empty, and the first element on this list (i.e., the leftmost match in the form [1,j]) is a 1-dominant match, as well as the only dominant match in that list. By initialization, *FLAG* is *true* the first time it is tested. Since *THRESH* is empty at this time, lines (3,4) will be executed, whence the first 1-dominant match is recorded. The algorithm also proceeds to the update of the other lists involved, so that at the next step the contents of such lists will be consistent. Moreover, since the *SEARCH* of line (1) returns $n+1$, then *FLAG* is set to the value *false*, which exhausts all manipulations involving matches in the first row. In general, the first match on the *AMATCHLIST* associated with a non trivial row is certainly a k-dominant match for some k. Assume that a certain number of entries of this *AMATCHLIST* have been processed and that: (i) the number of times that *FLAG* was *true* equals the number of dominant matches detected so far, (ii) j identifies the last dominant match detected, and (iii) j is the only such match which has not been recorded yet. It is easy to see that *HS* 1: locates the displacement of this match in *THRESH* (line 1); switches *FLAG* to *false*, if appropriate (line 2); updates the lists and records this new dominant match in *LINK* (lines 3-6, 8), and probes into $AMATCHLIST[\sigma]$ seeking the next position to which the *PEBBLE* should be advanced to mark

the next dominant match (line 7, meaningful only if *FLAG* is *true*). Thus *FLAG* is *true* as long as conditions (i-iii) hold, that is, exactly for d times. □

The actual time bound of *HS* 1 depends on the internal representation which is chosen for the various lists involved. If the lists are represented as priority queues such as 2-3 trees or *AVL* trees [ME], then *HS* 1 runs in $O(d\log n + n\log s)$ time, inclusive of preprocessing, which reduces to $O(d\log\log n + n\log s)$ if one uses a structure better fit to the manipulation of integers [VE]. This already compares favorably with the corresponding bounds in [HS], where r figures in the place of d. One interesting observation, however, is that the sequences of insertions in each list constitute in fact merges of sorted linear sequences. Efficient dynamic structures are available [BW,BT,ME] which support, say, the merging of two lists of sizes k and $f \geq k$ in time $O(k\log(2f/k))$. This leads to speculation that the total time spent by *HS* 1 for the mergings could be bounded by a form such as $O(m\log m + d\log(2mn/d))$. Unfortunately, it does not seem that the $O(k\log(2f/k))$ bound still holds, with such structures, if deletions are intermixed with insertions in an uncontrolled way. Besides, the management of such structures is rather involved, and their storage requirements usually large.

It turns out that the special case which is of interest here is indeed susceptible to efficient implementation on *finger-trees* [AP]. In what follows, however, we provide an alternate construction based on simpler structures, and thus show that the desired performance can be achieved at the expense of almost negligible complications.

6. CHARACTERISTIC TREES

We present a data structure suitable for the efficient implementation of dictionary primitives [AU,ME] on sorted subsequences S of a fixed subsequence U (the *universe*) of the string of integers $1\ 2\cdots n$. We shall assume, to simplify our presentation, that the cardinality m of U is such that $m = 2^c$ for some integer c.

Having chosen U, we associate with it a balanced and complete binary tree T_U with m leaves, labelled in succession with the *keys* in U (i.e., with an integer in $\{1,2...,n\}$). Each interior vertex v of T_U is marked with the ordered pair of keys representing the largest elements of U which appear in the subtrees of T_U rooted at the left and right son of v, respectively (for all our purposes, this information is redundant if $m=n$: however, for uniformity of treatment, we consider it as provided in all cases). Any choice of a subsequence S of U translates in a corresponding instantiation $T_U(S)$ of T_U, as follows (T_U itself can be regarded as $T_U(\phi)$, with ϕ the empty sequence). Each leaf i in $T_U(S)$ is marked '1' if $i \in S$, and '0' otherwise. Thus, the leaves of the tree become a blueprint of the *characteristic function* of the set S with respect to the universe U (see the figure below). In addition, each interior node is marked '0' if neither of its son nodes is marked '1', and '1' otherwise. $T_U(S)$ is called the *U-characteristic tree associated with S* or simply the *C-tree* of S when this raises no confusion about U. The C-tree of S requires only $2m-1$ records (actually, bits when $n=m$), and it can be allocated sequentially, as any *heap*. Thus, for any node in a C-tree, one can travel just as easily upward, downward or horizontally on the same level.

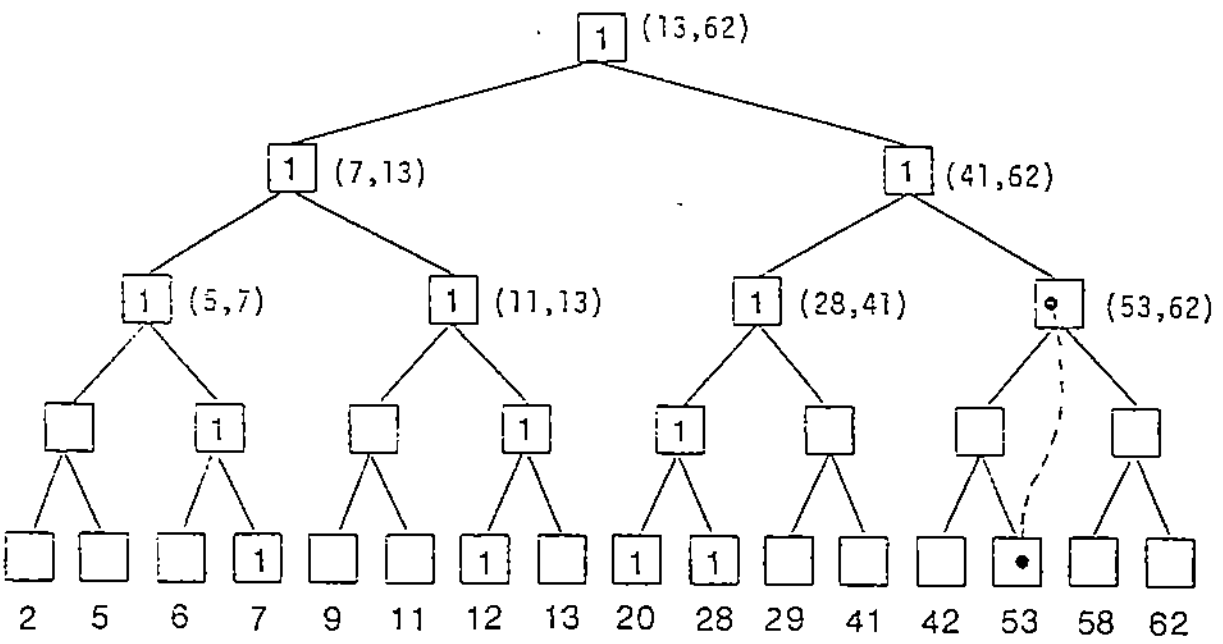

Figure 3

The characteristic tree of the set $S = \{7, 12, 20, 28\}$ relative to the set $U = \{2, 5, \ldots, 62\}$. All '0' marks are omitted, and range information is not reported on the deepest interior nodes. To exemplify just once, leaf 53 is connected to its characteristic node by a broken line.

For any element $i \in U$ which is not (is also) in S, the operation of inserting i in (or deleting it from) $T_U(S)$ is straightforward. The key property of $T_U(S)$ is that if a vertex is marked '1' ('0'), then all its ancestors (descendants) are also marked '1' ('0'). It is convenient to associate, with each leaf i of $T_U(S)$ such that $i \in U-S$, the *characteristic node* $v(i)$ for that leaf, defined as the highest '0'-node on the path from i to the root. Now, to transform $T_U(S)$ into $T_U(S \cup \{i\})$, it suffices to change the marks of all vertices on the path of $T_U(S)$ from i to $v(i)$ (inclusive). Likewise, to delete leaf i, travel from this leaf upward changing all marks to '0', until the first vertex is encountered both sons of which were previously marked '1'. The mark of this node is left unchanged, and the son of this node through which the node itself was reached becomes the new characteristic node of i. The operation of testing for membership in S of a *key* $i \in U$, though immediate, is of not much use. To compensate for this, the operation of searching is not restricted to take arguments from U. More precisely, we define the search of $i \in \{1,2,...n\}$ in $T_U(S)$ as the function which returns the first element j of S not smaller than i (or $n+1$, if no such element exists). We assume that any such search originates at the bottom of T_U and from a *finger* leaf f. The searching technique used requires only slight modifications of otherwise standard searching on a *finger-tree* [ME] (cf. also [BT,BW]). In short, the finger search for i in the C-tree $T_U(S)$ takes the same effort as the finger search for j in T_U, if T_U is regarded this time as the finger tree associated with the set U. For instance, assume that $i > f$. The search starts by climbing from leaf f toward the root, (performing transitions to right neighboring nodes, whenever appropriate) until a node v is found which is marked '1' and which subtends an interval of U the right end of which is larger than i. If j is not in the subtree of T_U rooted at v, the predecessor of j in S certainly is. Which case applies is ascertained by a straightforward downward search which is driven both by the range and boolean information stored in each node. If the element of S returned by this search is the predecessor of j in S, let v' be the deepest 1-marked right neighbor of an ancestor of v. Then j must be the leftmost element of S in the subtree of T_U rooted at v'. The node v' can be easily spotted by resuming the climb from v. Alternatively, this second stage could be avoided

by linking the 1-marked leaves of $T_U(S)$ in a linear list.

In any case, the effort involved in the search is bounded by a constant times the number of nodes that are visited during the climb-up process. Visiting each new node corresponds to doubling the previous guess for the distance separating i from the finger in the key space U, much as it happens in an unbounded search [BY]. This observation supports the following straightforward lemma (cf., for instance, [ME]).

Lemma 3. The search in $T_U(S)$ for an element which falls b positions (i.e., leaves) away from a finger takes $O(\log b)$ steps.

We now consider sequences of consecutive searches in $T_U(S)$, which start with the finger pointing to the leftmost leaf of T_U. By always bringing the finger on the key returned by the search which was performed last, it is easy to maintain inductively that if, for the current query, $f \geq i$, then f is also the result of the search. Thus, each time a climb-up process is performed, this results in moving the finger to the right of its previous position. For k consecutive searches, the total effort is bounded by a constant times the sum

$$\sum_{j=1}^{k} \log b_j$$

where the b_j's represent the widths of the various *intervals*, and these latter are non-overlapping, i.e., $\sum_{j=1}^{k} b_j \leq 2m$. With this constraint, the above sum is maximum when all the b_j's are equal, which yields a bound of $O(k \log(2m/k))$ for the sequence of searches. The problem is more complicated when we consider instead a sequence of consecutive insertions (or a sequence of consecutive deletions), due to the fact that the climb-up is always to be performed there, for the purpose of node re-marking.

It is fortunate that a bound similar to the above can in fact be drawn also for these cases. Again, this is due to the fact that T_U can be regarded, say, as a special 2-3 tree of the kind presented in [BT] (cfr. also [BW,ME]). The specialty consists of our tree having only 2-nodes. To

be more precise, we appeal to the following result in [BT].

Lemma 4. Let T be a 2-3 tree with m leaves numbered 1,2,...,m, and let $i_1,i_2,...,i_k$ be a subsequence of the leaves. Let $i_0 = 0$ and, $b_j = i_j - i_{j-1} + 1$, for $j = 1,2,...,k$. Furthermore, for i and $i' > i$, let $l(i,i')$ be the number of nodes which are on the path from i' to the root but not on the path from i to the root. Finally, let

$$p = \lceil \log m \rceil + 1 + \sum_{j=1}^{k} l(i_{j-1},i_j).$$

Then p obeys the inequality:

$$p \leq 2(\lceil \log m \rceil + \sum_{j=1}^{k} \lceil \log b_j \rceil).$$

For any subsequence Q of U of cardinality k, the expression denoted by p is an upper bound for the process of producing $T_U(Q)$ (T_U) from T_U ($T_U(Q)$) by orderly insertion (deletion) of the elements of Q.

Let now U and $Q = (i_1,i_2,\ldots,i_k)$ be subsequences of $\{1,2,...,n\}$ of cardinalities m and k, respectively, and let S be a subsequence of $\{1,2,...,n\}$. With reference to $T_U(S)$, consider the following three homogeneous series of k operations each. Each series applies a chosen primitive to all the elements of Q, in and orderly fashion. The series which are considered are: (i) the finger-searches of each of the elements of Q (where the finger is suitably initialized to point to i_0=0), (ii) the insertions of each of the elements of Q, (iii) the deletions of each of the elements of Q (the two latter series being defined only when Q is also a subsequence of U).

Lemma 5. p is an upper bound for each of the series (i-iii).

Proof.

Obvious for the searches, for which the bound that was sketched above is actually tighter. Consider then the insertions. Assume that $i_1,i_2,\ldots,i_{j-1}=f$ have been inserted, and let i be the leaf

marked '1' which is the closest such leaf to the left of i_j ($i = 0$ if no such leaf exists). It suffices to show that $i \neq f$ implies $l(i,i_j) \leq l(f,i_j)$. Now $i \neq f$ implies that $f < i$. Since the three individual paths from the root to each of f, i, and i_j all share a common prefix, consider the topmost node whereby this bundle of paths is split. If only one path takes the right branch out of that node, then, since $i < i_j$, this must be the path to i_j, and the assertion holds with equality. On the other hand, if two of the paths depart along the right branch then these must be the paths leading to i and i_j, and $l(f,i_j)$ is at most $l(f,i)$. But since the tree is perfectly balanced, then $l(f,i) = l(f,i_j)$, whence the assertion follows. An analogous argument proves the claim for the series (iii). □

In view of Lemma 4, Lemma 5 can be rephrased by saying that k orderly insertions in an originally non-empty C-tree do not require more effort than in the case where the tree is initially empty. Likewise, the work involved in k orderly deletions cannot exceed the effort of transforming a tree which stores exactly k elements into the empty tree. We leave it as an exercise for the reader to show that the assumption, made at the beginning of this section, that m is a power of 2 can be levied at this point with no substantial penalty on the results presented so far. Lemma 5 does not apply to any hybrid series of dictionary primitives. However, we shall use it to show that it works for the peculiar hybrid series which are involved in $HS\,1$ at each row. Thus, we assume henceforth that all lists in $HS\,1$ are implemented as C-trees, i.e., heap structured complete binary trees, and that each such tree is endowed with a suitable finger. The collective initialization of all trees takes trivially $\Theta(n)$ time.

Theorem 5. $HS\,1$ requires $O(n \log s)$ preprocessing time and $O(m \log n + d \log(2mn/d))$ processing time.

Proof.

It is easy to check that the preprocessing required by $HS\,1$ is basically the same as that required

by HS, whence we can concentrate on the second time-bound. Let d_i denote the number of dominant matches which HS 1 introduces as a result of handling row i. As seen in the discussion of Theorem 4, d_i searches are performed on *THRESH* while considering row i. Observe that the arguments of successive searches constitute a strictly increasing sequence of integers, and that the same can be said of the values returned by those searches. Thus, by Lemma 5, the cost of all searches on this row is bounded, up to a multiplicative constant, by $\log n + \sum_{k=1}^{d_i} \log b_k$, where the intervals b_k are such that $\sum_{k=1}^{d_i} b_k \leq 2n$, since the C-tree of *THRESH* contains n leaves. (Recall that, in the upper bound for the searches, the $\log n$ term is actually unnecessary.) It follows that, up to a multiplicative constant, the total cost on all rows is bounded by $m \log n + \sum_{k=1}^{d} \log b_k$, where now $\sum_{k=1}^{d} b_k \leq 2mn$. With this constraint, the previous sum is maximized by choosing all b_i equal, i.e., $b_i = 2mn/d$. The claimed bound then follows at once.

It is not difficult to show that the same bound holds for the insertions and deletions performed on *THRESH*. We observe the following. First, the two lists of arguments for the insertions and deletions, respectively, represent increasing subsequences of the integers in $[1,n]$. Moreover, the set of items inserted into *THRESH* is disjoint from the set of items deleted from *THRESH*. The second observation enables to deal with each one of the two series separately. In other words, the total work involved in the insertions and deletions affecting *THRESH* at some row is not larger than the work which would be required if one performed all the deletions first, and then performed all the insertions. Thus, through an argument analogous to that used for the searches, the bound follows from Lemma 5, and from the fact that, on each row, *THRESH* is affected by d_i insertions and by a number of deletions which is at least $d_i - 1$ and at most d_i.

We now turn to the primitives collectively performed on all the *AMATCHLIST*s invoked during the management of any single row. The key observation here is that the sum of the cardinalities of all such lists never exceeds n. In fact, there will be exactly n leaves in the forest of C-

trees which implement such lists. If the C-trees corresponding to the various *AMATCHLISTS* s are visualized as aligned one after the other, it is easy to adapt the same argument which was used for *THRESH* to the primitives affecting the collection of these lists. Indeed, the special conditions on the searches, insertions and deletions still hold locally, on each individual list. This leads to our claimed bound, since the d_i insertions in *THRESH* correspond in fact to d_i searches with deletions on *AMATCHLIST*[σ], and an equivalent number of insertions take place in the collection of all lists. □

7. A LINEAR SPACE ALGORITHM

In this section we present an algorithm, *Algorithm* 7, which determines an LCS in linear space and in time equal to that of *Algorithm* 3 up to an additive term $O(m \log m)$ [AG]. As mentioned, the only previous algorithm that computes an LCS in linear space [HC] takes never less than $\Theta(nm)$ time.

Algorithm 7 follows the same divide-and-conquer scheme of [HC]. The algorithm applies the auxiliary procedure *length* recursively to smaller subproblems until it obtains a trivial one. The procedure *length* is a straightforward adaptation of *Algorithm* 3: *length* can work on an arbitrary substrings $a_{i1},...,a_{i2}$, $b_{j1},...,b_{j2}$ of α and β, and that it does not keep track of all dominant matches. Thus *length* computes only the length *lsub* of the LCS for that subproblem. The procedure is called by passing four parameters to it, namely, $i1, i2, j1$ and $j2$. It returns *lsub* and the the array *RANK* which contains the leftmost k-dominant match, for each $k=1,..,l$. At the beginning, the procedure expects to find *PEBBLE*[i] active and pointing to the the entry j of a_i-*OCC*, which corresponds to the leftmost occurrence of a_i in the interval $[j1...j2]$. If the procedure finds that *PEBBLE*[i] falls outside the interval $[j1...j2]$, then it marks this pebble *dead*, if it were not already such. The procedure advances the active pebbles of each row until all of them become *inactive*. A pebble becomes inactive as soon as either the procedure advances it onto an entry of the associated a_i-*OCC* list which is larger than $j2$, or it attempts at advancing the pebble

past the last entry of a_i-*OCC*. When the first case applies, the pebble is retracted by one position on the list: thus by the end of the execution of *length* each non-dead pebble points to the rightmost position that it can occupy in the interval $[j1...j2]$. Following our discussion of Section 3, we implement now *closest* by using both the table *CLOSE* and appropriate fingers on the σ-*OCC* lists.

```
Procedure length (i1, i2, j1, j2, RANK, lsub)
0) RANK [k] = 0, k=1,2,....,(i2−i1); mark dead pebbles outside [j1...j2];
1) k = 0
2) while there are active pebbles do (start stage k÷1)
3) begin T=j2÷1; k=k+1;
    4) for i = i1−1÷k to i2 do (advance pebbles)
     begin
       5) t = T;
       6) if PEBBLE [i] is active and a_i-OCC [PEBBLE [i]] < T then
         (update threshold, update leftmost k-dominant match )
            7) begin  T = a_i-OCC [PEBBLE [i]]; RANK [k]=T end;
         (advance pebble, or make it inactive)
       8) PEBBLE [i]=SYMB [closest [a_i ,t ]];
       9) if PEBBLE [i] is active and a_i-OCC [PEBBLE [i]] > j2 then
          10) begin PEBBLE [i]=PEBBLE [i]−1; make PEBBLE [i] inactive end;
     end;

 end (lsub = k).
```

The procedure *length* detects all k-dominant matches, as is readily checked, although it records only the leftmost k-dominant match incurred for each k. This obtains the linear space bound.

Algorithm 7 is actually based on the four procedures *length*, *lengthrev*, *lcs* and *lcsrev*. The companion procedure of *length*, *lengthrev*, is simply a replica of *length* just made suitable for processing the mirror image of any subproblem on the input strings. Thus, for instance, calling *lengthrev* with parameters: $1,m,1,n$, has the same effect as letting *length* run on the reverse of the input strings. The mirror procedure *lcsrev* is related to the procedure *lcs*, which is still to be described, in the same way. In conclusion, we only need to list *lcs*.

We need to make a few additional assumptions, namely:

– We stipulate that m is a power of 2.

– We remove the previous assumption according to which, upon calling *length* with j-parameters $j1, j2$, the procedure always finds pebbles and fingers pointing to the leftmost positions in the interval $[j1...j2]$. We replace it with the new assumption that either all pebbles and fingers occupy the rightmost positions in the interval $[j1...j2]$, or else they all occupy the leftmost one. Procedure *length* checks at its inception which case applies, and brings all pebbles to their leftmost positions, if necessary. This does not affect the time bound of the procedure.

Algorithm 7: 'Procedure lcs' $(1, m, 1, n, LCS)$
begin
1) if $n=0$ or $m=1$ then *determine LCS in constant time*
else (split the problem into subproblems)
begin
2) *length* $(1, m/2, 1, n, RANK1, lsub_1)$;
3) *lengthrev* $(m/2+1, m, 1, n, RANK2, lsub_2)$;
4) j = *findmax* $(RANK1, RANK2, lsub_1, lsub_2, lsub)$;
(determine the length *lsub* of the LCS for this subproblem)
5) *lcs* $(m/2, m, 1, j, LCS1)$;
6) *lcsrev* $(m/2+1, m, n-j, n, LCS2)$;
7) *combine the two outputs* $LCS1$ and $LCS2$;
end;
end.

The function *findmax* determines the value $j=RANK1[k]$ such that, if $j' = RANK2[k']$ is the smallest entry of $RANK2$ which is larger than j, then $lsub = k+k'$ is a maximum. Thus, the first time *findmax* is executed, it returns $lsub = l$, i.e., the length of any LCS of α and β. Moreover, the match $[i, j]$ with maximum $i \leq m/2$ belongs to an LCS of the two input strings.

The function *findmax* can be straightforwardly implemented in such a way as to require a number of steps proportional to $lsub_1+lsub_2 \leq 2lsub$.

The correctness of *lcs* follows from the arguments in [HC].

Theorem 6. The procedure *lcs* finds an LCS in time $O(m\log m + ml\log(min[s, 2n/m]))$ and space $\Theta(n)$.

Proof. We consider all the executions of *length* and *lengthrev* involved at the $k-th$ level of recursion of our strategy, at once. Such executions are relative to consecutive substrings of α of uniform length $m/(2^k)$, and consecutive substrings of β. Starting from the upper-left corner of the L-matrix, each such substring of β is paired up twice with a substring of α. The upper pairing involves an execution of *length*, the second one an execution of *lengthrev*. We define a *block* at level k as the submatrix which is the domain of two such consecutive subproblems.

All the executions of *findmax* at this level charge $O(l)$ time. Adding up for all values 1,2,...,log$m$ of k yields a bound $O(l\cdot\log m)$ for the total work performed by *findmax*.

The execution of each *length* (*lengthrev*) can be bounded in terms of $m/(2^k)\cdot l_f \log(\min[s, 2n\cdot 2^k/m])$, where l_f denotes the length of the LCS associated with the generic subproblem. There are 2^k calls at level k, yielding a total time:

$$\sum_{f=1}^{2^k} \frac{m}{2^k} l_f \log(min[s, \frac{2n}{m}2^k]),$$

up to a multiplicative constant. Now it is

$$\sum_{f=1}^{2^k} l_f \le 2l.$$

In fact, each l_f cannot be larger than the length of the solution to the corresponding block, and the sum of the 2^{k-1} such lengths cannot exceed l, i.e., the length of the global solution.

Thus we have, in conclusion, that the total work at this level of recursion can be bounded in terms of the quantity:

$$l\cdot\frac{m}{2^k}\cdot\log(min[s, \frac{2n}{m}2^k]) \le l\cdot\frac{m}{2^k}\cdot\log(min[s2^k, \frac{2n}{m}2^k]).$$

The right term can be rewritten as:

$$l\cdot\frac{m}{2^k}\log(2^k\cdot\min[s, \frac{2n}{m}]) = l\cdot k\cdot\frac{2m}{2^k} + l\cdot\frac{m}{2^k}\log(min[s, \frac{2n}{m}]).$$

Adding up through $k = 1,2,...,\log m$ yields:

$$ml \sum_{k=1}^{\log m} \frac{k}{2^k} + ml \log(min[s, \frac{2n}{m}]) \sum_{k=1}^{\log m} \frac{1}{2^k}.$$

Since:

$$\sum_{k=1}^{\log m} \frac{1}{2^k} = 2 - \frac{1}{2^{\log m}} < 2,$$

and:

$$\sum_{k=1}^{\log m} \frac{k}{2^k} = (\log m - 1)\frac{1}{(2m)} + \frac{1}{2},$$

then we obtain the claimed $O(m \log m + ml \log(min[s, 2n/m])$ time bound.

The linear space bound follows from an argument in [HC]. □

8. CONCLUDING REMARKS

Any known approach to the LCS problem computes, in its own way, a *minimal antichain decomposition* (MAD: for this notion and related ones the reader may refer, e.g., to [BO]) for a partial order defined on the set of matches: an LCS is a longest *chain* in the poset of matches, and a set of matches having equal rank is an *antichain*. The MAD problem for posets can be solved in general by flow techniques [BO], although not in time linear in the number of elements of the poset.

The main algorithms discussed in this paper have their natural predecessors in [HI-HC] and [HS]. In terms of MADS, the approach of [HI-HC] consists of computing the antichains one at a time, while that of [HS] extends partial antichains relative to all ranks already discovered, one step at a time. The interested reader shall find that also the approach in [NY], which yields bounds of $O(n(m-l+1))$ or $O(m(m-l+1)\log n)$ falls into this second category. Moreover, it is easy to check that the second bound can be reduced to $O(m(m-l+1)\min\{\log s, \log m, \log 2n/l\})$ by the techniques developed here.

Algorithm 1 and its companion algorithms are inherently off-line, as is the original algorithm in [HI]. Also, the best time bound obtained here with this one-antichain-at-a-time approach turns into $O(ln)$, i.e., the bound already achieved in [HI], when the input strings have nearly equal lengths. Since lm is an upper bound for d, and there are situations where $d=O(n)$ while $lm=\Theta(n^2)$, then *Algorithm* 6 (*HS* 1) appears to be asymptotically faster in general than the other algorithms presented in this paper. Moreover, *HS* 1, like *HS* itself, can be executed on-line. On the other hand, the basic structure of *Algorithm* 3 can be used to set up a linear space LCS algorithm which takes time equal to that of *Algorithm* 3 (when finger-searches are used), up to an additive term $O(m\log n)$. Interestingly, it does not seem that *Algorithm* 6 is amenable to an equivalent (i.e., time-performance-preserving) implementation in linear space. Some of the 'log's in our bounds can be reduced to 'loglog's via the techniques in [VE]. However, the corresponding improvements would not be significant. Finally, we mention that the problem of devising an $O(n\log n)$ time algorithm for the LCS of two strings, or showing than no such algorithm exists, is still open.

Acknowledgements

We are indebted to Z. Galil, K. Mehlhorn, F. P. Preparata and W. Schnyder for enlightening discussions, and to two referees for their very useful comments.

REFERENCES

[AA] Apostolico, A. Remark on Hsu-Du new algorithm for the LCS problem, Tech. rep., Purdue Univ. CS Dept., (1985, submitted for publication).

[AG] Apostolico, A. and C. Guerra. A fast linear space algorithm for computing longest common subsequences, *Proceedings of the 23-rd Allerton Conference*, Monticello, Ill. (1985).

[AP] Apostolico, A. Improving the worst case performance of the Hunt-Szymanski strategy for the longest common subsequence of two strings, Tech. rep., Purdue Univ. CS Dept. (1985), to appear in *Information Processing Letters*.

[AH] Aho, A.D., D.S. Hirschberg and J.D. Ullman. Bounds on the complexity of the maximal common subsequence problem, *JACM* 23, 1, 1-12 (1976).

[AU] Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass, 1976.

[BO] Bogart, K.P., *Introductory Combinatorics*, Pitman, 1983.

[BT] Brown, M.R., and R.E. Tarjan. A representation of linear lists with movable fingers. *Proceedings of the* 10-*th STOC*, San Diego, Ca., 19-29 (1978).

[BW] Brown, M.R., and R.E. Tarjan. A fast merging algorithm, *JACM* 26, 2, 211-226 (1979).

[BY] Bentley, J.L., and A.C-C. Yao, An almost optimal algorithm for unbounded searching, *Inform. Process. Letters* 5, 82-87 (1976).

[HD] Hsu, W.J., and M.W. Du. New algorithms for the LCS problem, *JCSS* 29, 133-152 (1984).

[HI] Hirschberg, D.S. Algorithms for the longest common subsequence problem, *JACM* 24, 4, 664-675 (1977).

[HC] Hirschberg, D.S. A linear space algorithm for computing maximal common subsequences, *CACM* 18, 6, 341-343 (1975)

[HS] Hunt, J.W., and T.G. Szymanski. A fast algorithm for computing longest common subsequences, *CACM* 20, 5, 350-353 (1977).

[MA] Martinez, H.M. (ed.), Mathematical and computational problems in the analysis of molecular sequences, *Bulletin of Mathematical Biology* (Special Issue honoring M. O. Dayhoff) 46, 4, 1984.

[ME] Mehlhorn, K. *Data structures and algorithms 1: sorting and searching*, Springer-Verlag, EATCS Monographs on TCS (1984).

[MP] Masek, W.J., and M.S. Paterson. A faster algorithm for computing string editing distances, *JCSS* 20, 18-31 (1980).

[NY] Nakatsu, N. , Kambayashi, Y. and Yajima, S. A Longest Common Subsequence Algorithm Suitable for Similar Text Strings, *Acta Informatica* 18, 171-179 (1982).

[SK] Sankoff, D. and J. B. Kruskal (eds.), *Time warps, string edits and macromolecules: the theory and practice of sequence comparisons*, Addison-Wesley, Reading (1983).

[VE] van Emde Boas, P. Preserving order in a forest in less than logarithmic time and linear space. *Inf.Proc.Lett.* 6, 3, 80-82 (1977).

[WF] Wagner, R. A., and M.J. Fischer, The string to string correction problem, *Journal of the ACM 21*, 1, 168-173 (1974).