AUSTRALASIAN JOURNAL OF COMBINATORICS
Volume **28** (2003), Pages 225–238

# Longest subsequences in permutations

M. H. Albert[1] R. E. L. Aldred[2] M. D. Atkinson[1]

H. P. van Ditmarsch[1] B. D. Handley[2] C. C. Handley[1]

[1]*Department of Computer Science* [2]*Department of Mathematics and Statistics*
*University of Otago*
*New Zealand*

J. Opatrny

*Department of Computer Science*
*University of Concordia*
*Canada*

**Abstract**

For a class of permutations $X$ the *LXS* problem is to identify in a given permutation $\sigma$ of length $n$ its longest subsequence that is isomorphic to a permutation of $X$. In general *LXS* is NP-hard. A general construction that produces polynomial time algorithms for many classes $X$ is given. More efficient algorithms are given when $X$ is defined by avoiding some set of permutations of length 3.

## 1 Introduction

The properties of the Longest Increasing Subsequence (*LIS*) of a sequence of values have been studied for many years. In the case where the sequence of values is a permutation much interest has centred on the probability distribution of the length of an *LIS*; a good survey of these investigations is given in [2]. Computing the *LIS* is a favourite example in many algorithms courses (and books, see [8, 9, 14]) because it neatly illustrates the design paradigm of dynamic programming. The algorithm is due initially to Schensted [11] and has run-time complexity $O(n \log n)$ (proved essentially optimal by Fredman [7]). It has recently become an important part of the MUMmer system [6] for aligning whole genomes.

The length of a *LIS* of a sequence $\sigma$ is a good measure of the "increasing tendency" of $\sigma$. In some data processing situations we may have a sequence of initially sorted data that is subjected to a small number of alterations and so is no longer entirely

increasing. It is useful to be aware of this since some sorting methods have extreme performance (either good or bad) on such data sets (see [3]). In this context it is helpful to define the *defect* of a sequence as the minimal number of elements that have to be deleted to obtain an increasing sequence. Then the smaller the defect of a sequence the more it has a tendency to be increasing. For future reference we note the obvious fact that the defect cannot go up if we pass to a subsequence.

In this paper we consider a problem analogous to that of finding the *LIS*, but in a more general context, that of permutation patterns, which is a rapidly growing area of combinatorics. A permutation $\pi$ is said to occur as a pattern within another permutation $\sigma$ if there is a subsequence of $\sigma$ whose members appear in the same relative order as the elements of $\pi$ (for brevity, we say that such a subsequence is *isomorphic* to $\pi$). For example 3124 occurs as a pattern within 4716352 because of the subsequence 4135. We also write $\pi \preceq \sigma$ and say that $\pi$ is *involved* in $\sigma$ or that $\pi$ is a *subpermutation* of $\sigma$.

With this terminology the *LIS* problem can be stated as follows. Let $I$ be the set of identity permutations of all lengths. An increasing subsequence of a permutation $\sigma$ is just an occurrence within $\sigma$ of a pattern of $I$ so the *LIS* problem is to identify the longest permutation of $I$ that occurs as a pattern within $\sigma$. Formulated like this it seems to be a natural question to replace the set $I$ by other sets of permutations, in other words, to ask the question:

**Longest $X$-subsequence (*LXS*) Problem** *Given a fixed set $X$ of permutations, is there an efficient algorithm that will find, for any given permutation $\sigma$, a longest permutation of $X$ that is involved in $\sigma$?*

Just as for the special case $X = I$ we can define the $X$-defect of a permutation $\sigma$ as the minimal number of symbols from $\sigma$ that must be deleted to obtain a permutation in $X$. However, if we want to use the $X$-defect as a measure of "$X$-ness", we have to overcome the possibility that the $X$-defect could go up when we pass to a subsequence. For example, with $X = \{12, 1243\}$ the $X$-defect of 1243 is 0 whereas the $X$-defect of its subsequence 124 is 1. We circumvent this behaviour by only considering sets $X$ which are *closed* in the sense that whenever $\sigma \in X$ and $\pi \preceq \sigma$ then $\pi \in X$. This is, in fact, a very reasonable restriction to make since closed sets are a natural object of study in the theory of permutation patterns. Obviously, $I$ is a closed set.

One of the important properties of a closed set is that it is determined by "forbidden patterns". More precisely, if we define the basis $B(X)$ of a closed set $X$ to be the set of permutations minimal with respect to not lying in $X$, we have

**Lemma 1.1** *A permutation $\sigma$ lies in a closed set $X$ if and only if $\sigma$ involves no permutation of $B(X)$.*

The *LXS* problem is clearly at least as difficult as the question of recognising whether $\sigma \in X$. The latter question is known to be NP-complete for certain sets $X$ (one example, see [15, 16], is the set of all permutations that can be sorted by 4 stacks

in parallel). On the other hand if $X$ has a polynomial recognition algorithm there is apparently no reason that the *LXS* problem should be tractable. Of course, under this condition, we *can* decide in polynomial time whether the $X$-defect is at most $k$, for any fixed $k$, and in particular we can decide this if $X$ is finitely based. Another connection between the $X$-defect and being finitely based is provided by

**Theorem 1.2** *If $X$ is a finitely based closed set then $X^{+m}$, the closed set of permutations of $X$-defect at most $m$, is also finitely based.*

PROOF: Obviously, $X^{+(m+1)} = (X^{+m})^{+1}$ so the theorem follows by induction provided we can establish the case $m = 1$. Let us suppose that the basis elements of $X$ have length at most $k$. We shall give a bound (best possible, as it happens) on the length of a basis element of $X^{+1}$.

Let $\gamma$ be any permutation of length $n$ and let $S_1, \ldots, S_{t+1}$ be the index subsets of $\{1, \ldots, n\}$ which support subsequences isomorphic to basis elements of $X$; we call these the $X$-basic subsequences of $\gamma$. Let $\gamma - j$ denote the result of deleting the $j^{th}$ element of $\gamma$. Then we have

$$\begin{aligned}
\gamma \notin X^{+1} &\iff \text{for all } j, \gamma - j \notin X \\
&\iff \text{for all } j \text{ there exists } i \text{ with } S_i \subseteq \{1, \ldots, j-1, j+1, \ldots, n\} \\
&\iff \text{for all } j \text{ there exists } i \text{ with } j \notin S_i \\
&\iff \bigcap_i S_i = \emptyset
\end{aligned}$$

If $\gamma$ is a basis element of $X^{+1}$ then its $X$-basic subsequences have trivial intersection. Choose a minimal family $P_1, \ldots, P_{u+1}$ of $X$-basic subsequences of $\gamma$ with trivial intersection. Since $X$-basic subsequences have at most $k$ elements we have $u \leq k$. By the minimal choice there exist indices $x_i$ with $x_i \notin P_i$ but $x_i \in P_j$ for all $j \neq i$. Note that $x_1, \ldots, x_{u+1}$ are distinct.

Also $\cup_i P_i = \{1, \ldots, n\}$ otherwise there is a proper subsequence of $\gamma$ whose $X$-basic subsequences have trivial intersection. So we have

$$\{1, \ldots, n\} = \{x_1, \ldots, x_{u+1}\} \cup \bigcup_{i=1}^{u+1} (P_i \setminus \{x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{u+1}\})$$

It now follows that

$$n \leq u + 1 + (u+1)(k-u) = (u+1)(k-u+1)$$

The maximal value of the RHS occurs at $u = k/2+1$ if $k$ is even and at $u = (k \pm 1)/2$ if $k$ is odd. In any case $n$ is bounded in terms of $k$. ■

**Example 1** $I^{+1}$ is the set of all permutations which, except for at most one symbol, are increasing. By following the method given in the proof above its basis can be computed as $\{321, 2143, 2413, 3142, 3412\}$.

We have seen that we cannot hope for an efficient algorithm to solve the *LXS* problem in general. Nevertheless, for many $X$, the *LXS* problem has a polynomial time algorithm. In section 2 we construct a large number of sets for which there is a polynomial time algorithm (albeit often of rather high degree). Then, in section 3, we generalise the *LIS* problem more directly. Notice that the set $I$ is defined by the basis permutation 21 of length 2. This suggests that the next cases to consider should be those closed sets all of whose basis elements have length 3. There are several such closed sets [12] but for all of them (with one notable exception) we show how to solve the *LXS* problem in time $O(n^2 \log n)$.

Since we appeal frequently to the *LIS* algorithm, and to some of its properties, we review it here and gather together some easy consequences for future reference. Let $\sigma = s_1 s_2 \dots s_n$ be some permutation of length $n$. We scan $\sigma$ from left to right. Having scanned up to symbol $s_i$ we shall have built a list of longest increasing subsequences of each length up to $i$. It turns out that it is sufficient to keep just one subsequence of each length, namely one of those with smallest final element. These smallest final elements are all that need to be maintained if we are interested only in the length of the *LIS*. We update this information when we process the next symbol of $\sigma$ (by doing a binary search) in $O(\log n)$ steps. If we wish to compute the *LIS* itself we need to keep some back pointers to enable it to be reconstructed once the algorithm has completed the scan of $\sigma$. In our exposition (especially in Section 3) we shall give details of algorithms modelled on the *LIS* algorithm that compute only the length of the *LXS*; we rely on the reader to supply the details of the back pointers to compute the actual *LXS* itself.

**Proposition 1.3** *Given a permutation $\sigma = s_1 s_2 \dots s_n$ of length $n$ there are algorithms for the following problems each of time complexity $O(n \log n)$:*

1. *compute a longest increasing subsequence of every initial segment of $\sigma$,*
2. *compute, for every $h$, a longest increasing subsequence of $\sigma$ that has final value no more than $h$,*
3. *compute, for every $i$, a longest increasing subsequence of $\sigma$ that ends with $s_i$.*

*In addition, there is an algorithm of time complexity $O(n^2 \log n)$ to compute, for every $i, j$ with $i \leq j$, a longest increasing subsequence that starts at $s_i$ and ends at $s_j$.*

PROOF: The first problem is solved by applying the standard *LIS* algorithm. The second can be solved either by a simple modification or by applying the *LIS* algorithm to the inverse permutation. The third problem is solved by interpolating an extra step in the main loop of the *LIS* algorithm: when the current $s_i$ is being processed we can determine to which of the subsequences being kept $s_i$ should be appended to get the longest result. For the fourth problem we run the *LIS* algorithm $n$ times. The $i^{th}$ time we run it we maintain only subsequences that begin with $s_i$. ■

Notice that we can replace the word "increasing" with "decreasing". Also we can apply all the algorithms to the reverse of $\sigma$.

## 2 Some polynomial time algorithms

In this section we develop a construction that defines a large number of closed sets for which the *LXS* problem has a polynomial time algorithm. We also give a number of examples to demonstrate the scope of the construction.

Let $\theta = h_1 \dots h_r$ be a permutation of length $r$ and let $X_1, \dots, X_r$ be sets of permutations. We define $\theta[X_1, \dots, X_r]$ as the set of all permutations that can be divided into segments $\phi = \chi_1, \dots, \chi_r$ with the properties

(i) $\chi_i$ is isomorphic to a permutation of $X_i$, and

(ii) $\chi_i < \chi_j$ if and only if $h_i < h_j$.

The second property introduces a piece of notation that we use frequently: if $\alpha$ and $\beta$ are sequences and every component of $\alpha$ is less than every component of $\beta$ then we write $\alpha < \beta$.

**Example 2** Let $\theta = 213$ and suppose that $X_1, X_2, X_3$ are sets containing respectively $21, 132, 312$. Then $\phi = 54|132|867 \in \theta[X_1, X_2, X_3]$ by virtue of the division into segments shown.

Furthermore, if $H$ is any set of permutations of length $r$ then we define

$$H[X_1, \dots, X_r] = \bigcup_{\theta \in H} \theta[X_1, \dots, X_r]$$

We shall use this construction only in the case that the $X_i$ are closed. Since we allow the empty permutation to occur (necessarily a member of every closed set) the segments $\chi_i$ in the above definition are also permitted to be empty. One easily checks

**Lemma 2.1** *If $X_1, \dots, X_r$ are closed then $H[X_1, \dots, X_r]$ is closed.*

As it stands this notation is only a way of making larger closed sets from smaller ones in a simple 'finite' way but, as the next example foreshadows, we can, with the aid of recursion, make considerably more complicated constructions.

**Example 3** Let $S$ be the set of all stack-sortable permutations (those permutations defined by avoiding the permutation 231) and let $T$ be the closed set consisting only of the empty permutation and the permutation 1. Put $\theta = 132$ and consider

$\theta[S, T, S]$. It is not hard to see that this is the class $S$ itself! In other words, $X = S$ is a solution of the equation

$$X = \theta[X, T, X] \tag{1}$$

There are many other solutions of this equation (for example, the set of all permutations). However, given a family of solutions $\{X_i \mid i \in I\}$, one easily checks that $\cap_{i\in I} X_i$ is also a solution (and this is true for non-empty solutions too as all non-empty classes contain the permutation 1). Thus Equation (1) has a minimal non-empty solution $M$ (under set inclusion).

As generally happens in such circumstances the minimal non-empty solution also has an *internal* description. Equation (1) can be used as a rule for generating further members from ones already known. In this case, as we know that the permutation 1 and the empty permutation lie in $M$ we can deduce immediately that $12, 21, 132$ all lie in $M$ and from that we can obtain still larger permutations. It is, in fact, not hard to see that $M = S$.

The above example shows the approach we shall take. We shall consider equations of the form

$$X = H[X_1, \ldots, X_r] \tag{2}$$

where each of $X_1, \ldots, X_r$ is either bound to a definite closed set $D_i$ for which the L$D_i$I problem is polynomially solvable or is the variable $X$ itself. Such an equation always has at least one non-empty solution and so has a least non-empty solution $M$. The fact that each permutation of $M$ is, via equation (2), built up from smaller permutations allows us to devise a dynamic programming algorithm for the longest $M$-subsequence.

Suppose we have a closed set $M$ defined as the minimal non-empty solution of an equation of the form above and we are given a permutation $\sigma$ of length $n$. We shall identify the longest $M$-subsequence of $\sigma$ by solving the problem in larger and larger *windows* on $\sigma$.

Let $I = i, i+1, \ldots, i+p-1$ and $J = j, j+1, \ldots, j+q-1$ be any two integer ranges in $1, 2, \ldots, n$. Two such ranges define an $p \times q$ window on $\sigma$ consisting of the subsequence of $\sigma$ whose positions are in the first range and whose values are in the second. The term "window" is derived from thinking of the graph of $\sigma$, placing an $p \times q$ rectangle within this graph, and observing which points of the graph it captures. We order the windows lexicographically by their dimensions $p$ and $q$. The total number of windows is a quartic polynomial in $n$.

Now, suppose we would like to know the longest $M$-subsequence in a particular $p \times q$ window $W$ defined by intervals $I$ and $J$. So long as we process windows lexicographically we may assume we know the answer for all previous windows. We consider all the ways in which each of $I$ and $J$ may be split into $r$ subintervals; there are at most $O(p^{r-1}q^{r-1}) = O(n^{2r-2})$ such ways. Given any splitting we have a consequent splitting of $W$ into an $r \times r$ array of subwindows. We now examine each $\theta \in H$ in turn. For a typical $\theta$ we identify the $r$ subwindows $W_1, \ldots, W_r$ which match

the form of $\theta$ and we solve the L$X_i$S-problem in $W_i$ (directly if $X_i$ is one of the bound variables, or by looking up a previously computed solution if $X_i = X$) and thereby read off an $M$-subsequence of $W$. The *LMS* of $W$ is the longest $M$-subsequence over all splittings of $W$ and all $\theta \in H$.

This algorithm has to solve a polynomial number of *LXS* problems all of which take polynomial time and so is itself a polynomial time algorithm.

Many of the closed sets considered in the next section can be described by the construction above. To give a better idea of the scope of the construction we give two further examples.

**Example 4** Let P be the class of *separable* permutations that was studied in [5]. Permutations of $P$ are built up from the identity permutation by two types of combination. If $\theta, \phi \in P$ then also $\theta\phi \in P$ if either $\theta < \phi$ or $\theta > \phi$. This definition makes it easy to see that $P$ is the minimal solution of

$$X = H(X, X)$$

where $H = \{12, 21\}$.

**Example 5** Consider the class whose basis is $\{213, 3412\}$. It can be proved that this class is the minimal solution of

$$X = H(T, T, X, T)$$

where $T$ is the class whose only non-empty member is the permutation of length 1 and $H = \{1432\}$.

Finally in this section we note that the methods carry over to systems of equations in several variables. Little expressive power is gained however for the following reason. Suppose the variables (the symbols not bound to known closed sets) are $V_1, V_2, \ldots$ If the equation for a variable $V_i$ occurs on the right hand side of the equation defining a variable $V_j$ then $V_i$ will be a subset of $V_j$. So circular chains of recursive definitions imply that the variables can all be replaced by a single variable. Once that is done the remaining variables are partially ordered by inclusion and we can define each one using previously determined variables. Essentially, this reduces the multivariate case to a collection of single variable problems.

## 3 Basis permutations of length 3

The general algorithms of section 2 can be improved in many cases. This section considers recognition algorithms for longest $X$-subsequence for all closed sets $X$ which can be defined by avoiding permutations of length 3. Such closed sets were first studied in [12] from an enumeration standpoint. Nowadays their internal structure

is well understood and we shall appeal to this piece of combinatorial folk-lore in the descriptions below.

The first thing to note is that we may take advantage of the usual symmetries (since these may be effected in linear time, which will not affect the complexity of our algorithms). That means that the classes $X$ to be considered may be taken to have the following bases that we group into types according to the number of basis permutations. In the following taxonomy we have omitted the finite classes and the classes with 4 or more basis elements.

| Name | Basis | Description |
|---|---|---|
| $A_1$ | 321 | *Merge of two increasing subsequences* |
| $A_2$ | 231 | *Stack sortable permutations* |
| $B_1$ | 321, 312 | *Direct sum of cyclic blocks* |
| $B_2$ | 213, 321 | *Having the profile of* 132 |
| $B_3$ | 231, 312 | *Layered permutations* |
| $B_4$ | 213, 312 | *Increasing segment followed by decreasing segment* |
| $C_1$ | 231, 312, 321 | *Layered with layers of lengths* 1 *and* 2 |
| $C_2$ | 132, 231, 321 | *Initial point then increasing* |
| $C_3$ | 132, 213, 321 | *Cyclic shift of an increasing permutation* |
| $C_4$ | 132, 213, 312 | *Increasing permutation with reversed final segment* |

Each of the finite classes may be taken to have the permutation 321 in its basis. It is known [1] how to recognise (in time $O(n \log n)$) whether a fixed permutation avoiding 321 occurs as a subpermutation, so we can easily solve the *LXS* problem for any finite class in time $O(n \log n)$.

Classes with 4 or more basis elements have a very simple form and we leave the reader to check that their *LXS* problem is virtually trivial.

### 3.1 $A_1$

The algorithm that we present for the $LA_1S$ problem follows the same scheme as the *LIS* algorithm. It scans the given permutation $\sigma$, keeps information relating to $A_1$-subsequences of each length, and updates this information as it processes the current symbol of $\sigma$.

At a typical step, just before we process the next symbol $s$ of $\sigma$, we shall have a record of various $A_1$ subsequences. Rather than storing an entire subsequence we exploit the fact that to avoid 321 is equivalent to having a decomposition as the union of two increasing subsequences, and we store only the final elements $(a, b)$, with $a < b$, of the two subsequences in this decomposition. However, unlike the *LIS* situation, it is necessary to store several $A_1$ subsequences of each length.

The basic idea when processing $x$ is to decide whether $x$ can be appended to any stored 321-avoiding subsequences to make new ones. If we had two $A_1$-subsequences of length $t$ represented by pairs $(a, b)$ and $(c, d)$ with $(a, b) < (c, d)$ then we do not need to store $(c, d)$ as it is *redundant*. This is because any way of extending the

subsequence represented by $(c, d)$ to a maximal $A_1$-subsequence entails a way of also extending the subsequence represented by $(a, b)$. Consequently, if the stored pairs that represent subsequences of length $t$ are $(a_1, b_1), \ldots, (a_r, b_r)$ and we keep them in increasing order of first component $(a_1 < a_2 < \ldots < a_r)$ then we shall have $b_1 > b_2 > \ldots > b_r$. In particular, there will be at most $n$ pairs to keep rather than $n^2$.

We process the current symbol $x$ as follows:

Consider the length $t$ subsequences that we are currently maintaining and suppose they are represented by pairs $(a_1, b_1), \ldots, (a_r, b_r)$ ordered as above. There are three cases:

1. Any pair $(a_j, b_j)$ with $x < a_j$ represents a subsequence that cannot be extended using $x$.

2. If we have a pair with $a_j < x < b_j$ then we can form a new $A_1$-subsequence of length $t+1$ represented by $(x, b_j)$. But, as we are only keeping irredundant pairs we would only store such a new pair if, among those pairs with $a_j < x < b_j$, it was the one with smallest $b_j$. We can identify the unique pair that we need by a binary search.

3. If we have a pair with $a_j < b_j < x$ then we can form two new $A_1$-subsequences of length $t + 1$ represented by $(a_j, x)$ and $(b_j, x)$ but the latter pair is obviously redundant. Thus, among the pairs for which $a_j < b_j < x$ we only have to consider the one with smallest $a_j$ and we can identify it by binary search.

Having identified new candidate pairs to represent $A_1$-subsequences of length $t + 1$ we then need to scan (again by binary search) the existing list of pairs representing subsequences of length $t + 1$. A candidate new pair might have to be discarded because it is redundant; if it is retained then possibly some of the existing pairs might become redundant and be discarded.

On conclusion of the scan we consult the lists that we are keeping and identify the length of the longest $A_1$-subsequence. Each iteration of the main loop has to examine all the lengths that are under consideration and for each one execute some binary searches. Therefore the total time is $O(n^2 \log n)$.

**Note** By an extension of this algorithm we can also solve, in time $O(n^{k-1} \log n)$, the $LM_kS$ problem where $M_k$ is defined by avoiding the permutation $k, \ldots, 1$. This extension has some points in common with the Robinson-Schensted algorithm that associates a pair of standard Young tableaux with a permutation. Indeed, once these tableaux have been constructed one can read off the lengths of the longest increasing $M_k$-subsequences for every value of $k$ (see [13] Theorem 7.23.13).

### 3.2 $A_2$

Our algorithm for the $LA_2S$-subsequence of a permutation $\sigma$ follows the method of Section 2. So the typical step is to construct the longest $A_2$-subsequence in some $p \times q$ window $W$ defined by two ranges, say $I = i, \ldots, i+p-1$ and $J = j, \ldots, j+q-1$. We are looking for a subdivision of $W$ to represent a stack sortable permutation $\alpha m \beta$ with $\alpha < \beta < m$.

Subdivisions with $m < j + q - 1$ correspond to stack permutations in a window defined by $I$ and $j, \ldots, j+q-2$ and we will already have found the longest of these. So we only need to consider subdivisions having $m = j + q - 1$ and those will only arise if $\ell = \sigma^{-1}(m)$ lies in the range $I$. Assuming that this is indeed the case we have to check $O(n)$ subdivisions where $\alpha$ corresponds to a window defined by the intervals $[i, \ell-1]$ and $[j, k-1]$ and $\beta$ corresponds to a window defined by the intervals $[\ell+1, i+p-1]$ and $[k, j+q-2]$ for each $k = j, \ldots, j+q-1$.

Since we can process each window in $O(n)$ steps the entire algorithm runs in time $O(n^5)$.

This algorithm is far worse than the algorithms we have managed to find for the other sets considered in this section. One of the reasons for this is that there seems to be no way to characterise initial segments of a potential stack sortable permutation by a small number of parameters.

### 3.3 $B_1$

Permutations of $B_1$ have the form $\alpha_1\alpha_2\ldots$ where the segments $\alpha_i$ satisfy $\alpha_1 < \alpha_2 < \ldots$ and each has the form $a+1, a+2, \ldots, a+p-1, a$ for some $a$ and $p \geq 1$. We exploit this structure and give an algorithm which is markedly similar to the algorithm for longest $A_1$-subsequence. As in that algorithm we scan the given permutation $\sigma$ from left to right and at each point we maintain a list of $B_1$-subsequences found so far. Rather than storing an entire $B_1$-subsequence we keep only enough that we can tell whether a new element of $\sigma$ can be used to extend it. In this case that means we have to store two values only: the maximal element $m$ say and the greatest element $d$ say that is followed by a smaller one.

Suppose that we process the element $x$ of $\sigma$. Then for every pair $(m, d)$ representing a $B_1$-subsequence of length $t$ we have the potential to create a $B_1$-subsequence of length $t+1$. In fact, if $m < x$ we do indeed obtain such a subsequence and we record it as $(x, d)$ while if $d < x < m$ we have a subsequence represented by the new pair $(m, m)$ (no other case can give a new $B_1$-subsequence of length $t+1$).

As in the analysis for the $A_1$-subsequence algorithm not all pairs need to be stored. For, if $(m, d)$ and $(m', d')$ are two pairs representing $B_1$-subsequences of the same length, and if $(m, d) < (m', d')$, then we may discard $(m', d')$ since any way of extending the sequence it represents allows the sequence represented by $(m, d)$ to be extended also. In particular, only $O(n)$ pairs need be stored for each length. The algorithm and its analysis now follow the same lines as the $LA_1S$ algorithm.

### 3.4 $B_2$

Here we give an algorithm that makes repeated use of the *LIS* algorithm. We are looking for a subsequence of the type $\alpha\beta\gamma$ where each segment is increasing and $\alpha < \gamma < \beta$. We consider each position of $\sigma$ as a possible starting position for a $\gamma$ segment. Let $j$ be a typical such position.

First, by Proposition 1.3, we compute $F(j, h)$ which is the length of the *LIS* that starts at $s_j$ and whose final element has value at most $h$. Next, also by Proposition 1.3, we compute, for each $i < j$, the length $I(i, j)$ of the *LIS* of the first $i$ elements that has final value smaller than $s_j$.

Then we compute, for each value $i = j-1, j-2, \ldots$, the *LIS* that ends before position $j$ and whose first element is $s_i$. This done by the longest *decreasing* subsequence algorithm running on the reversal of the first $j-1$ elements of $\sigma$. This algorithm returns its results, one per $O(\log n)$ steps. If $\ell$ is the length of a typical sequence it finds we can find a $B_2$-subsequence of length $I(i-1, j) + \ell + F(j, s_i - 1)$ and we take the maximal length thereby computed.

Each value of $j$ can be handled in $O(n \log n)$ steps for a total of $O(n^2 \log n)$ steps.

### 3.5 $B_3$

Permutations of $B_3$ have the form $\alpha_1\alpha_2\ldots$ where each $\alpha_i$ is decreasing and $\alpha_1 < \alpha_2 < \ldots$ Following [4] we call them *layered* permutations. We shall describe an algorithm for determining the maximal layered subsequence of a given permutation $\sigma = s_1 s_2 \ldots s_n$ that runs in time $O(n^2 \log n)$, where $n$ is the length of the sequence.

The algorithm requires a preprocessing step that computes, for each pair $i, j$ with $1 \le i \le j \le n$, the longest decreasing subsequence (possibly empty) that starts at $s_i$ and finishes at $s_j$. It is explained how to do this in Proposition 1.3. Let $D(i, j)$ denote the length of such a sequence.

The main part of the algorithm works with a subset of the set of windows of $\sigma$. For convenience we define the following terms:

$M(x, y)$: the length of the maximal layered subsequence in the window defined by the intervals $1, \ldots, x$ and $1 \ldots, y$.

$T(x, y)$: the length of the maximal layered subsequence in the window defined by the intervals $1, \ldots, x$ and $1 \ldots, y$ that has the additional properties that it contains the term $s_x$ (in particular $s_x \le y$) and contains the term $y$ (in particular $s_i = y$ for some $i \le x$). In geometric terms this means that there are points in the layered subsequence on the top and right boundaries of the window. If no such sequence exists, then $T(x, y)$ is defined to be 0.

Like some of the previous algorithms this one also scans $\sigma$ from right to left. Just before processing the symbol $s_i$ it has computed and stored $M(x, y)$ for all $x < i$ and

all $y \leq n$. In processing $s_i$ it first computes $T(i, s_j)$ for all $j \leq n$ according to

$$T(i, s_j) = \begin{cases} 0 \text{ if } D(j, i) = 0 \\ D(j, i) + M(j-1, s_i - 1) \text{ otherwise} \end{cases}$$

Once the $T(i, s_j)$ are calculated we compute the values $M(i, y)$ by

$$M(i, y) = \max(M(i-1, y), M(i, y-1), T(i, y))$$

obtaining $M(i, 1), M(i, 2), \ldots$ in turn.

Each iteration of this loop requires linear time so this part of the algorithm runs in $O(n^2)$. So the dominating factor is finding the maximal decreasing subsequences, at $O(n^2 \log n)$.

### 3.6 $B_4$

Permutations of this class are the juxtaposition of an increasing segment and a decreasing segment. To find the longest $B_4$-subsequence of a given permutation $\sigma$ of length $n$ we run the LIS algorithm on $\sigma$ first. It finds the longest increasing subsequence of *every* initial segment of $\sigma$. Then, by considering the reverse of $\sigma$ we find the longest decreasing segment of every terminal segment. Finally we combine these sets to get the longest $B_4$-subsequence of $\sigma$. The total time is only $O(n \log n)$.

### 3.7 $C_1, C_2, C_3$ and $C_4$

Permutations of the class $C_1$ are layered with layers of sizes 1 and 2 only. We can therefore find the longest $C_1$-subsequence of $s_1 \ldots s_n$ by adapting the algorithm for the class $B_3$. The only modification needed is to define $D(i, j)$ as the longest decreasing sequence of length at most 2 from $s_i$ to $s_j$. Clearly these can be computed in $O(n^2)$ steps and, as noted in the algorithm for $B_3$-subsequences, the remainder of the algorithm runs in $n^2$ steps also.

An algorithm for the longest $C_2$-subsequence is easily derived from the structure of permutations in $C_2$; they are increasing except, possibly, for their initial symbol. So we run the *LIS* algorithm on the segment that begins at the second position and prepend the first symbol to the subsequence it finds.

Permutations of the class $C_3$ are simply cyclic shifts of the identity permutation. Hence we can obtain a $O(n^2 \log n)$ algorithm to find the longest $C_3$-subsequence of $\sigma$ by applying the *LIS* algorithm to every cyclic shift of $\sigma$.

Permutations of the class $C_4$ are obtained from the identity permutation by reversing some final segment. So, to obtain the longest $C_4$-subsequence of a permutation $\sigma$ we apply the *LIS* algorithm to every permutation obtained from $\sigma$ by reversing some final segment. This requires time $O(n^2 \log n)$.

## 4 Concluding remarks

We have given a number of closed sets for which the *LXS* problem can be solved in polynomial time and, by noting that it is at least as hard as the $X$-recognition problem, shown that there are some closed sets for which this is not possible. There remains the question of how closely these two problems are related. For example, does every finitely based closed set $X$ have a polynomial time *LXS* problem?

Another open problem is how efficiently one can solve *LXS* problems for various 'simple' sets $X$. In particular, can the algorithm for finding the longest stack sortable subsequence be improved so that its performance is more comparable with the other sets discussed in Section 3?

The *LIS* problem is often presented as a special case of the Longest Common Subsequence (*LCS*) problem since the *LIS* of $\sigma$ is just the *LCS* of $\sigma$ and $1, 2, \ldots$. The more general *LXS* problem does not seem to be a special case of the *LCS* problem. However, the *LXS* problem does have a more general context which suggests numerous other directions to explore. If $X$ is not the union of two proper closed subsets then [10] there is an infinite permutation $\pi(X)$ whose finite subsequences form (under isomorphism) the set of permutations of $X$. Thus the *LXS* problem could be solved if one had a method for finding the longest permutation involved in two given permutations (admittedly, one of them would be infinite!). The problem of computing, given two permutations, the longest permutation involved in both (even if they are finite) is completely unexplored.

**Acknowledgement** We thank the referee for bringing to our attention the connection between the *LXS* problem (when $X$ is defined by avoiding $k, \ldots, 1$) and the Robinson-Schensted algorithm.

## References

[1] M. H. Albert, R. Aldred, M. D. Atkinson and D. A. Holton, Algorithms for pattern involvement in permutations, in Algorithms and Computation, 12th International Symposium, ISAAC 2001, Proceedings LNCS 2223, P. Eades, T. Takaoka (Eds.) pp.355–366.

[2] D. Aldous and P. Diaconis, Longest increasing subsequences: from patience sorting to the Baik-Deift-Johansson theorem, Bull. Amer. Math. Soc. **36** (1999), 413–432.

[3] R. M. Baer and P. Brock, Natural sorting, J. Soc. Indust. Appl. Math. **10** (1962), 284–304.

[4] M. Bóna, The Solution of a Conjecture of Wilf and Stanley for all layered patterns, J. Combin. Theory, Ser. A **85** (1999), 96–104.

[5] P. Bose, J. F. Buss and A. Lubiw, Pattern matching for permutations, Inform. Process. Lett. **65** (1998), 277–283.

[6] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Paterson, O. White and S. L. Salzberg, Alignment of whole genomes, Nucleic Acids Research **27** (1999), 2369–2376.

[7] M. L. Fredman, On computing the length of longest increasing subsequences, Discrete Math. **11** (1975), 29–35.

[8] D. Gries, The Science of Programming, Springer Verlag (New York) 1981.

[9] U. Manber, Introduction to Algorithms, Addison-Wesley (Reading, Mass.) 1989.

[10] M. M. Murphy, Ph.D. thesis, University of St Andrews, in preparation.

[11] C. Schensted, Longest increasing and decreasing subsequences, Canad. J. Math. **13** (1961), 179–191.

[12] R. Simion and F. W. Schmidt, Restricted permutations, European J. Combin. **6** (1985), 383–406.

[13] R. Stanley, Enumerative Combinatorics, volume 2, Cambridge Studies in Advanced Mathematics 62, Cambridge University Press (Cambridge, UK) 1999.

[14] J. M. Steele, Probability Theory and Optimization, SIAM 1997.

[15] W. Unger, The complexity of colouring circle graphs, Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science, 1992, Springer Lecture Notes in Computer Science **577**, 389–400.

[16] W. Unger, On the $k$-colouring of circle graphs, Proc. 5th Annual Symposium on Theoretical Aspects of Computer Science, 1988, Springer Lecture Notes in Computer Science **294**, 61–72.

(Received 20 June 2002)