Algorithms for pattern involvement in permutations

M. H. Albert
Department of Computer Science

R. E. L. Aldred
Department of Mathematics and Statistics

M. D. Atkinson
Department of Computer Science

D. A. Holton
Department of Mathematics and Statistics
University of Otago

Abstract

We consider the problem of developing algorithms for the recognition of a fixed pattern within a permutation. These methods are based upon using a carefully chosen chain or tree of subpatterns to build up the entire pattern. Generally, large improvements over brute force search can be obtained. Even using on-line versions of these methods provides such improvements, though these are often not as great as for the full method. Furthermore, by using carefully chosen data structures to fine tune the methods, we establish that any pattern of length 4 can be detected in $O(n \log n)$ time. We also improve the complexity bound for detection of a separable pattern from $O(n^6)$ to $O(n^5 \log n)$.

Keywords pattern containment, permutations

1 Introduction

The relation of "pattern containment" or "involvement" on finite permutations has become an active area of research in both computer science and combinatorics. In computer science, pattern containment restrictions are used to describe classes of permutations that are sortable under various conditions [1, 5, 6, 10]. In combinatorics the focus has been more on enumerating permutations under various pattern containment restrictions [7, 8, 9, 11]. In both these areas it is difficult to gather computational data because of the difficulty of testing for pattern containment.

Formally, we say that two sequences are *isomorphic* (or order isomorphic) if the permutations required to sort them are the same. For example, the two sequences $3, 4, 7, 1$ and $5, 7, 9, 2$ are isomorphic. We say that one permutation $\sigma = s_1 \dots s_m$ is *involved* in another permutation $\tau = t_1 \dots t_n$ when $t_1, \dots, t_n$ has a subsequence that is isomorphic to $s_1, \dots, s_m$. We write $\sigma \preceq \tau$ to express this.

It appears to be a difficult problem to decide of two given permutations σ, τ whether $\sigma \preceq \tau$ and in this generality the problem is NP-complete [2]. In this paper we study the case that σ is fixed, of length k say, and τ is the input to the problem. This is the situation that arises when we wish to run many "$\sigma \preceq \tau$" tests with τ varying and σ fixed. In practice most pattern containment investigations are of this type. A brute force approach which simply examines all subsequences of τ of length k would have a worst case execution time of $O(n^k)$, where n is the length of τ. Therefore the problem lies in the complexity class P but of course, for all but small values of k, this execution time will generally be unacceptable.

To the best of our knowledge no previous work has been published on improvements to this upper bound. The best implementation of brute force search that we know of is the program `forbid.c` [4] which uses a tree search approach based on the generating trees defined in [11]; in general however this program does not improve the asymptotics of the worst case.

A few particular cases of the problem have been attacked successfully. There is an $O(n \log \log n)$ algorithm for finding the longest increasing subsequence of a given sequence of length n [3] so this solves the problem for the cases $\sigma = 12 \cdots k$. The permutations $132, 213, 231, 312$ can all be handled in linear time by stack sorting algorithms. Also, an algorithm of time complexity $O(n^6)$ was given in [2] for the case of an arbitrary separable permutation.

In this paper we develop general algorithms whose worst case complexity is considerably smaller than $O(n^k)$ and we look at a number of cases where even further improvement is possible.

In the next section we set up a general apparatus for searching for the pattern σ. As will be seen we are prepared to invest a considerable amount of time (exponential in k) in preprocessing σ in the expectation that we shall then be able to solve instances of the "does τ involve σ" problem much faster than by brute force. We identify an integer $c(\sigma)$ that controls the complexity of our recognition algorithm and report on a statistical study that gives insight into the variation of $c(\sigma)$ as σ varies. This study indicates that the algorithms are never worse than $O(n^{2+k/2} \log n)$ (although that remains unproved) and in some cases are considerably better. We give an example to show that $c(\sigma)$ may be much smaller than the apparent worst case and we briefly discuss how the algorithms solve the associated counting problem. The section ends with a glimpse of an even more general family of algorithms and a minor improvement to the algorithm in [2] for detecting a separable permutation.

Section 3 investigates a special case of our general approach. This special case is particularly suitable for 'on-line' algorithms (where τ can be scanned once only). Moreover it avoids the expensive preprocessing stage and is simple enough that upper bounds can be proved analytically. We give examples of infinite families of permutations that can be detected quickly by an on-line algorithm.

In the final section we examine permutations σ of length 4. We refine our general approach and describe $O(n \log n)$ algorithms to recognise whether $\sigma \preceq \tau$.

2 A General Recognition Framework

In this section we develop a general algorithm for testing whether $\sigma \preceq \tau$ and illustrate its power by a number of case studies. Throughout, σ is a *fixed* permutation of length k and τ a variable permutation of length n.

Before giving the technical notation we sketch the general idea behind our approach. In the first stage of the algorithm we shall identify a suitable sequence

$$\sigma_0 \preceq \sigma_1 \preceq \sigma_2 \preceq \ldots \preceq \sigma_k = \sigma \tag{1}$$

of subsequences of σ (with σ_i of length i). This sequence will be chosen so that the subsequences of τ that are isomorphic to one of the σ_i can be identified and used without being stored in their entirety. This stage of the algorithm may have a cost that is exponential in k but it is independent of the input τ and is not repeated.

The second stage of the algorithm identifies all subsequences of τ that are isomorphic to σ_i, for increasing values of i. Because these subsequences are not stored in their entirety this part of the algorithm is of much lower complexity than naïve search.

In order to handle subsequences of both σ and τ we represent them as sets of pairs. If $\sigma = s_1 \ldots s_k$ is the permutation that maps i to s_i, then σ itself will be represented as the set $\mathcal{S} = \{(i, s_i) \mid 1 \leq i \leq k\}$. Every subset of $\mathcal{S}$ defines a subsequence of σ and vice versa. Subsequences of τ are defined similarly as subsets of the set of pairs $\mathcal{T}$ that defines τ itself.

A sequence such as (1) above is then just a sequence of subsets

$$\emptyset = \mathcal{S}_0 \subseteq \mathcal{S}_1 \subseteq \mathcal{S}_2 \subseteq \ldots \subseteq \mathcal{S}_k = \mathcal{S} \tag{2}$$

in which each subset $\mathcal{S}_i$ is obtained from the previous one by adding a new pair (a_i, b_i). For the moment we shall defer the explanation of how to choose these subsets; once we have described how they are used it will be clear how to choose them optimally. Making this choice is the first step of the algorithm.

Let π_1 and π_2 be the projections that map a pair to its first and second component (respectively). With this view of subsequences, an isomorphism between a subsequence of σ and a subsequence of τ is simply a bijection β between the two corresponding sets of pairs for which the induced maps

$$p = \pi_1(p, v) \mapsto \pi_1(\beta(p, v))$$

$$v = \pi_2(p, v) \mapsto \pi_2(\beta(p, v))$$

are both order preserving. Note that when an isomorphism exists it is unique.

Let Σ_i denote the set of subsets of $\mathcal{T}$ that are isomorphic to the set $\mathcal{S}_i$. The second stage of the recognition algorithm is modelled on the following loop:

Algorithm 1 Basic form of the recognition algorithm

```
for $i := 0$ to $k-1$ do
  for each $(p,v) \in \mathcal{T}$ and each $\theta \in \Sigma_i$ do
    if $\theta \cup \{(p,v)\}$ is isomorphic to $\mathcal{S}_{i+1}$ then
      add it to $\Sigma_{i+1}$
    end if
  end for
end for
```

As it stands, of course, this is simply another version of brute force search crippled by the worst case size of Σ_i. To make significant improvements we need a way of handling many elements in Σ_i simultaneously. We shall introduce a concise form of an element $\theta \in \Sigma_i$, denoted by $R(\theta)$, called its *registration.* The key idea is to process all elements with the same registration simultaneously.

Before giving the technical definition of $R(\theta)$ it will be helpful to consider an example. Let us suppose that $\mathcal{S}_3 = \{(2,4),(5,3),(3,6)\}$ and that $\mathcal{S}_4 = \mathcal{S}_3 \cup \{(4,5)\}$. Suppose also that we have some subsequence $\theta = \{(14,4),(6,9),(10,15)\}$ of τ. Because of the bijection

$$(2,4) \mapsto (6,9), (5,3) \mapsto (14,4), (3,6) \mapsto (10,15)$$

we see that $\mathcal{S}_3 \cong \theta$. The necessary and sufficient condition that this bijection can be extended to an isomorphism between $\mathcal{S}_4$ and $\theta \cup (p,v)$ is $10 < p < 14$ and $9 < v < 15$.

The point of this is that the isomorphism test depends only on 4 items of θ rather than the entire 6 items. In this small case the saving is not very dramatic but it is enough to illustrate the general idea. In general, if $\mathcal{S}_{i+1} = \mathcal{S}_i \cup \{(a_{i+1}, b_{i+1})\}$ we identify in $\mathcal{S}_i$ the two first components which most closely enclose a_{i+1} and the two second components which most closely enclose b_{i+1}. The corresponding items in θ then define the range in which p and v must lie in order that $\theta \cup \{(p,v)\} \cong \mathcal{S}_{i+1}$. Notice that this idea still applies if a_{i+1} (respectively b_{i+1}) is smaller than or greater than any first (respectively second) component; in such a case the range in which p or v must lie is unbounded on one side.

In practice we often need to store considerably more than just these four enclosing components since we must anticipate being able to test isomorphisms with subsets of size greater than $i+1$. To describe precisely what needs to be stored at each stage we define the *registration type* $r(\mathcal{S}_i)$ of each $\mathcal{S}_i$ as

$$r(\mathcal{S}_i) = (P_i, V_i)$$

where

$$P_i = \{j \in \pi_1(\mathcal{S}_i) \mid \text{ either } j+1 \notin \pi_1(\mathcal{S}_i) \text{ or } j-1 \notin \pi_1(\mathcal{S}_i)\}$$

and

$$V_i = \{j \in \pi_2(\mathcal{S}_i) \mid \text{ either } j+1 \notin \pi_2(\mathcal{S}_i) \text{ or } j-1 \notin \pi_2(\mathcal{S}_i)\}$$

Lemma 2.1 *1. If w, x are the two symbols in $\pi_1(\mathcal{S}_i)$ which most closely enclose a_{i+1} (as $w < a_{i+1} < x$) then $w, x \in P_i$. Similarly, if y, z are the two symbols in $\pi_2(\mathcal{S}_i)$ which most closely enclose b_{i+1} then $y, z \in V_i$.*

2. $P_{i+1} \subseteq P_i \cup \{a_{i+1}\}$ and $V_{i+1} \subseteq V_i \cup \{b_{i+1}\}$.

PROOF: For the first part, if $w \notin P_i$ then $w-1$ and $w+1 \in \pi_1(\mathcal{S}_i)$. Therefore $w+1$ and x are a closer enclosing pair for a_{i+1}, a contradiction. The other statements follow similarly.

The second part follows from the definition of P_i and V_i. ■

We shall see later that the sizes of P_i and V_i determine the complexity of a refined version of the algorithm above for recognising whether $\sigma \preceq \tau$. The following example is meant to illustrate that "clever" choices of the sets $\mathcal{S}_i$ can control these values effectively.

Example 2.2 Suppose that $k = 4m$ and consider the permutation:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & \cdots & 2m-1 & 2m & \cdots & 4m \\ 1 & 2m & 4m-1 & 2m-2 & 4m-3 & \cdots & 2m+3 & 2 & \cdots & 2m+2 \end{pmatrix}$$

(in the second row, the odd values decrease by 2 each time, cyclically, beginning from 1, while the even ones decrease by 2 each time beginning from $2m$). If we simply take the set $\mathcal{S}_i$ to consist of the pairs making up the first i columns of this permutation then although $|P_{2m-1}| = 1$, we have $|V_{2m-1}| = 2m-1$. On the other hand, by simply reordering the pairs as follows:

$$\begin{pmatrix} 1 & 2m+1 & 2 & 2m+2 & 3 & 2m+3 & 4 & 2m+4 & \cdots \\ 1 & 2m+1 & 2m & 4m & 4m-1 & 2m-1 & 2m-2 & 4m-2 & \cdots \end{pmatrix}$$

we obtain an arrangement where $|P_i| \le 3$ and $|V_i| \le 4$ for all i.

The registration type $r(\mathcal{S}_i)$ specifies how much of each $\theta \in \mathcal{S}_i$ should be stored by the algorithm. The part that is stored is called the *registration* of θ and is denoted by $R(\theta)$: $R(\theta)$ is the image of $r(\mathcal{S}_i)$ under the natural maps induced by the order isomorphism between θ and $\mathcal{S}_i$. Let $R_i = \{R(\theta) \mid \theta \in \Sigma_i\}$.

The second stage of the recognition algorithm, incorporating the concept of registration, is specified in Algorithm 2.

Proposition 2.3 *When the second stage of algorithm 2 terminates we have $R_i = \mathcal{R}_i$ for $i = 1, 2, \ldots, k$. In particular $\sigma \preceq \tau$ if and only if $|\mathcal{R}_k| > 0$.*

PROOF: Notice that Lemma 2.1 guarantees that $w_\theta, x_\theta, y_\theta, z_\theta$ are present in $R(\theta)$ and that all the symbols needed to compute $R(\phi)$ are available either from $R(\theta)$ itself or from $\{p, v\}$. Note also that the stipulated ranges for p and v are precisely those for which the isomorphism between $\mathcal{S}_i$ and θ can be extended to an isomorphism between $\mathcal{S}_{i+1}$ and ϕ. Therefore each $R(\phi)$ is an element of R_{i+1}. Finally, observe that every $R(\phi) \in R_{i+1}$ is computed by the algorithm; this follows by induction on i. ■

Algorithm 2 Recognition algorithm with registration

```
$\mathcal{R}_i := \emptyset$ {$\mathcal{R}_i$ holds elements of $R_i$; initially none are known}
for $i := 0$ to $k-1$ do
  for each $(p,v) \in \mathcal{T}$ and each $R(\theta) \in \mathcal{R}_i$ do
    Let $w,x,y,z$ be defined as in Lemma 2.1
    Let $w_\theta, x_\theta, y_\theta, z_\theta$ be the elements of $R(\theta)$ that correspond to $w,x,y,z$
    if $w_\theta < p < x_\theta$ and $y_\theta < v < z_\theta$ then {hence $\phi = \theta \cup \{(p,v)\} \in \Sigma_{i+1}$}
      compute $R(\phi)$ and insert it in $\mathcal{R}_{i+1}$
    end if
  end for
end for
```

Next we discuss the run-time of the algorithm. The outer 'for' loop is executed k times but k is independent of n. In a typical iteration of the inner 'for' loop we have to consult each of the n pairs of T and each $R(\theta) \in \mathcal{R}_i$. The computation that is done with a typical (p,v) and $R(\theta)$ incurs a cost that depends on how the sets $\mathcal{R}_i$ are stored; by standard data structuring devices we can contain each of the operations that access and update the set $\mathcal{R}_i$ to a time $O(\log|\mathcal{R}_i|)$.

Taking all this into account the total cost of this stage of the algorithm is $O(n \max_i(|R_i| \log |R_i|))$.

The elements of R_i are sequences of integers in the range $1..n$ and the length of these sequences is $|P_i|+|V_i|$. It will therefore be advantageous to keep $|P_i|+|V_i|$ as small as possible. To this end we define

$$c(\sigma) = \min \max_i(|P_i| + |V_i|)$$

where the minimum is taken over all orderings of the pairs of $\mathcal{S}$. This discussion has proved:

Proposition 2.4 *If the ordering on $\mathcal{S}$ is chosen so as to minimise* $\max_i(|P_i| + |V_i|)$ *the second stage of the algorithm requires time $O(n^{1+c(\sigma)} \log n)$.*

It is now evident what the first stage of the algorithm must do: find the ordering on $\mathcal{S}$ that minimises $\max_i(|P_i| + |V_i|)$. The cost of this first stage of the algorithm will be independent of n so it will not contribute to the asymptotic upper estimate of the time complexity. Nevertheless it is not easy to compute the optimal ordering of $\mathcal{S}$; we have found a method based on a shortest path algorithm to do this in $O(2^k)$ steps.

Statistics

We have generated some thousands of random permutations of degrees up to 17 and computed $c(\sigma)$. In all of them we have found that $c(\sigma) \leq 1 + k/2$. The following table summarises the values of $c(\sigma)$ for samples of random permutations of lengths 8 through 17. In each row, we exhibit the number of permutations observed for each value of $c(\sigma)$ (blanks denoting no observations), and we also provide an example of a particular permutation from the sample which achieved the maximum observed value of $c(\sigma)$. In these examples, two digit numbers 10, 11, and so on, are denoted by the letters A, B, and so on.

	$c(\sigma)$								
k	2	3	4	5	6	7	8	9	Example
8		157	769	74					52814673
9	1	57	682	260					921745863
10		13	469	515	3				72936A5184
11		1	262	676	61				B52A7481639
12			126	662	212				72A419B538C6
13			48	533	412	7			B4159D6C2A738
14			23	365	582	30			3E68BC41927A5D
15			3	55	201	41			2D7CE5A6913F84B
16			1	38	167	83	1		A4F51C9G7D3B82E6
17				25	145	125	5		6$AFDH$842G93EC751B

Counting

Our general approach can easily be modified to count the number of occurrences of σ in τ. The only change we have to make is to keep, with every $\chi \in R_i$, an integer t_χ which records the number of θ for which $R(\theta) = \chi$. Then, whenever we detect that $\phi = \theta \cup \{(p, v)\} \in \Sigma_{i+1}$ we increment t_ω by t_χ, where $\omega = R(\phi)$ and $\chi = R(\theta)$.

A more general algorithm

Our general paradigm can be extended so that (2) is replaced by a 'union' tree. The leaves of the trees are labelled by singleton pairs and the internal nodes are labelled by subsets of $\mathcal{S}$ which are the disjoint unions of the subsets labelling their subtrees. The recognition algorithm processes the nodes of the tree in any order so long as each node is processed after its subtree nodes. To process a node labelled by a subset $\mathcal{U}$ of $\mathcal{S}$ means to find (and store implicitly by registration) all the subsets of $\mathcal{T}$ isomorphic to $\mathcal{U}$. When we do this we shall have available the corresponding information for the subtrees.

In order to get a comparatively efficient algorithm it will be necessary to have short registrations (as we have previously discussed). But the registration information has to be sufficient that we can successfully recognise the subsets of $\mathcal{T}$ isomorphic to $\mathcal{U}$. It is clearly going to be complicated to examine all possible union trees (although independent of n of course) so we have yet to explore the full potential of this idea. Neverthless we offer one example of the power of this approach in the following sketch which improves on the algorithm given in [2].

Suppose that σ is any *separable* permutation. By definition σ may be written as a concatenation $\sigma = \alpha\beta$ where either every entry of α is less than every entry of β or every entry of α is greater than every entry of β; moreover, α, β are isomorphic to separable permutations. Then $\mathcal{S}$ can written as a union $\mathcal{L} \cup \mathcal{M}$ where every member of $\pi_1(\mathcal{L})$ is less than every member of $\pi_1(\mathcal{M})$, and where either every member of $\pi_2(\mathcal{L})$ is less than every member of $\pi_2(\mathcal{M})$ (the positive case) or every member of $\pi_2(\mathcal{L})$ is greater than every member of $\pi_2(\mathcal{M})$ (the negative case). The sets $\mathcal{L}$ and $\mathcal{M}$ are structured in a similar fashion and so we have a natural way of defining a union tree for $\mathcal{S}$. The nodes of this tree are positive or negative according to how they were defined.

The registration type of a node $\mathcal{U}$ is a quadruple that we structure as an ordered pair of ordered pairs $((m_1, m_2), (M_1, M_2))$ where m_1 and m_2 are the minimum values in $\pi_1(\mathcal{U})$ and $\pi_2(\mathcal{U})$, and M_1 and M_2 are the maximum values. Thus the registration of a subset of $\mathcal{T}$ isomorphic to $\mathcal{U}$ is the quadruple which corresponds to the registration type under the isomorphism. It follows that each node is associated with at most n^4 registrations.

The central problem is to compute the set of registrations $R(\mathcal{U})$ at a node $\mathcal{U}$ given the registration sets $R(\mathcal{V}), R(\mathcal{W})$ for the child nodes $\mathcal{V}, \mathcal{W}$. For definiteness assume that $\mathcal{U}$ is a positive node (the negative case is similar). A quadruple $((m_1, m_2), (M_1, M_2))$ is easily seen to belong to $R(\mathcal{U})$ if and only if there exist pairs (a, b) and (c, d) for which

$$((m_1, m_2), (a, b)) \in R(\mathcal{V}) \text{ and } ((c, d), (M_1, M_2)) \in R(\mathcal{W}) \text{ and } (a, b) < (c, d)$$

To compute these quadruples we proceed as follows. First, for every (m_1, m_2) we search $R(\mathcal{V})$ and determine the set of all (w, x) for which $((m_1, m_2), (w, x)) \in R(\mathcal{V})$ and we find the set P_{m_1,m_2} of all minimal pairs in this set. Since the pairs of P_{m_1,m_2} are incomparable we can order them increasingly by first component and have the second components decrease. Next, for every (M_1, M_2) we search $R(\mathcal{W})$ and determine the set of all (y, z) for which $((y, z), (M_1, M_2)) \in R(\mathcal{W})$ and we find the set Q_{M_1,M_2} of all maximal pairs in this set. Again, the pairs of Q_{M_1,M_2} are incomparable so we can order them increasingly by first component and have the second components decrease.

Now, for each $((m_1, m_2), (M_1, M_2))$ we have to test whether there exist pairs $(w, x) \in P_{m_1,m_2}$ and $(y, z) \in Q_{M_1,M_2}$ for which $(w, x) < (y, z)$. Since the components are ordered as explained above this test can be made in time $O(n \log n)$. As there are $O(n^4)$ quadruples each node requires time $O(n^5 \log n)$. There are k nodes in all so the total time is still $O(n^5 \log n)$.

3 On-line algorithms

In this section we study a simpler version of the algorithm presented in the previous section. This simpler version avoids the preprocessing stage (which was exponential in k) while the second stage may be somewhat slower (but still provably better than brute force search). The resulting algorithm only has to scan the input τ once and so is referred to as an 'on-line' algorithm.

The simplification is to take the ordering of $\mathcal{S}$ in which the first components come in the order $1, 2, \ldots, k$. The order in which the second components come is then, by definition, the sequence $s_1, \ldots, s_k$ in the original description of σ and, indeed, the entire algorithm can be presented in the more familiar setting of subsequences of images of σ and τ. Furthermore the first components of registrations need not be kept (indeed it is easily seen that $P_i = \{i\}$) since we shall be processing $\tau = t_1 \ldots t_n$ in left to right order. This form of recognition is described in Algorithm 3.

Algorithm 3 On-line form of the recognition algorithm

```
for $j := 1$ to $n$ do
  for $i := 0$ to $k-1$ do
    for each $R(\theta) \in R_i$ do
      Let $y, z$ be defined as in Lemma 2.1
      Let $y_\theta, z_\theta$ be the elements of $R(\theta)$ that correspond to $y, z$
      if $y_\theta < t_j < z_\theta$ then $\{\phi = \theta t_j \in \Sigma_{i+1}\}$
        add $R(\phi)$ to $R_{i+1}$
      end if
    end for
  end for
end for
```

We define $d(\sigma) = \max |V_i|$. Arguing as in the previous section the execution time of this algorithm is $O(n^{1+d(\sigma)} \log n)$.

Lemma 3.1 $c(\sigma) - 1 \leq d(\sigma) \leq 2k/3$

PROOF: Observe that V_i does not contain 3 consecutive values $j-1, j, j+1$ since, by definition, the condition $j \in V_i$ implies that one of $j-1$ and $j+1$ does not belong to $\pi_2(\mathcal{S}_i)$ and so does not belong to V_i. So, in each triple $3j-2, 3j-1, 3j$, at most two members can belong to V_i and the result follows. ■

Corollary 3.2 *The decision problem $\sigma \preceq \tau$ can be solved in time $O(n^{1+2k/3} \log n)$. More generally, the number of occurrences of σ as a pattern within τ can be computed in this time bound.*

PROOF: The algorithm above stores registrations $R(\theta)$ where θ is now a subsequence of τ that is isomorphic to some σ_i. The registration is a subsequence of θ with enough information that we can determine whether θt_j is isomorphic to σ_{i+1}. As we saw in the previous lemma $|R(\theta)| \leq 2k/3$ and the results follow as in the previous section. ■

In many cases the upper bound given in the corollary can be greatly improved since $d(\sigma)$ is smaller than the upper bound in Lemma 3.1. In addition, we can often exploit special information about σ that is unavailable in general. As an example of such analyses we consider some classes of permutations σ for which very significant improvements can be made. These classes are 'closed sets' in the sense of [1] defined by their avoiding particular permutations. In general, let $A(\omega_1, \omega_2, \ldots)$ denote the set of permutations which do not involve any of $\omega_1, \omega_2, \ldots$.

We begin by considering the set $A(132, 312)$. It is easily seen that a permutation σ belongs to this set if the values of any initial segment of σ form a single interval. Equivalently, any initial segment of σ ends with its minimum or maximum value.

Proposition 3.3 *If $\sigma \in A(132, 312)$ then $d(\sigma) \leq 2$.*

PROOF: The result is almost immediate from the preceding description of $A(132, 312)$. Since the initial segment of length i consists of an interval of values, V_i simply consists of the endpoints of that interval, or only one of the endpoints if 1 or k already occur among the first i positions, and hence has size at most 2. Thus $d(\sigma) = \max_i |V_i| \leq 2$. ■

This proposition establishes that there is an $O(n^3 \log n)$ algorithm for recognising whether $\sigma \preceq \tau$ when $\sigma \in A(132, 312)$. In fact, by a small modification of the registration procedure we can reduce the complexity of this algorithm to $O(n^2 \log n)$.

With notation as in the proposition above, consider the elements of V_i as pairs (a, b) representing the lower and upper endpoints of the corresponding interval. In the naïve version of the algorithm we might well register two such pairs (a, b) and (a', b') where

$$a' < a < b < b'.$$

In this case the pair (a', b') can never be useful in the recognition of σ, since any extensions which they allow will also be allowed by the (a, b) pair.

It follows that the registration information which we need to store for V_i can be thought of as a sequence of pairs $(a_1, b_1), (a_2, b_2), \ldots (a_j, b_j)$ where

$$\begin{aligned} a_1 < a_2 < \cdots < a_j \qquad &\text{and} \\ b_1 < b_2 < \cdots < b_j \end{aligned}$$

In particular there can be at most n such pairs, rather than the $O(n^2)$ which are budgeted for in the standard on-line algorithm. This modification reduces the time complexity as claimed. It transpires that a further reduction to $O(n \log n)$ is possible by the use of the data structures mentioned in the following section.

To within order isomorphism there are only three other infinite sets defined by two length 3 restrictions. By analysing the structure of the permutations σ in these classes we can prove fairly easily

Proposition 3.4 *If σ is a permutation that satisfies at least two length 3 restrictions then $d(\sigma) \leq 3$.*

4 Permutations of length 4

We consider now the problem of finding efficient algorithms for recognising whether $\sigma \leq \tau$ in all cases where $|\sigma| = 4$. At first it seems that there are 24 individual problems of this type to be solved. However, the operations of: reversing a permutation; taking the complement of a permutation; and taking the inverse of a permutation, all respect the ordering $\preceq$, and can be carried out in $O(n \log n)$ time. So, if we can find an efficient algorithm for σ we also have one for its reverse, complement, etc. This reduces the number of cases that we need to consider in the present instance to 7, exemplified by:

$$\sigma = 1234,\ 2134,\ 2341,\ 2314,\ 1324,\ 2143,\ 2413.$$

In the first two cases $d(\sigma) = 1$ and so the on-line algorithms are of complexity $O(n^2 \log n)$. In both cases, and in general when $d(\sigma) = 1$, this is easily reduced to $O(n \log n)$. This is accomplished by storing the registration information R_i as a sorted list in such a way that we can search and insert in $O(\log n)$ time.

In the remaining cases $d(\sigma) = 2$, and so the on-line algorithms are of complexity $O(n^3 \log n)$. As in the case of $A(132, 312)$ though, it is possible to "prune" the registration information when it consists of pairs, to a set of size $O(n)$, and thereby gain an improvement in the running time of the algorithm to $O(n^2 \log n)$. In fact, in each case the running time can be reduced to $O(n \log n)$. To accomplish this, requires the use of a tree-based data structure which permits answering queries of a form similar to:

> What is the smallest $y > x$ which occurred between position q and the present position?

for arbitrary parameters x and q, in $O(\log n)$ time. To take the case of 1324 as an example, such a structure essentially allows registering the "best 3" in a 132 pattern up to the present position (where best in this case means minimum) in $O(\log n)$ time. So the amount of time spent at each element of τ is $O(\log n)$, and the total running time is $O(n \log n)$.

References

[1] M. D. Atkinson: Restricted permutations, Discrete Math. 195 (1999), 27–38.

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

[3] M.-S. Chang, F.-H. Wang: Efficient algorithms for the maximum weight clique and maximum weight independent set problems on permutation graphs, Inform. Process. Lett. 43 (1992), 293–295.

[4] O. Guibert: Personal communication.

[5] D.E. Knuth: *Fundamental Algorithms, The Art of Computer Programming* Vol. 1 (First Edition), Addison-Wesley, Reading, Mass. (1967).

[6] V. R. Pratt: Computing permutations with double-ended queues, parallel stacks and parallel queues, Proc. ACM Symp. Theory of Computing 5 (1973), 268–277.

[7] R. Simion, F. W. Schmidt: Restricted permutations, Europ. J. Combinatorics 6 (1985), 383–406.

[8] Z. E. Stankova: Forbidden subsequences, Discrete Math. 132 (1994), 291–316.

[9] Z. E. Stankova: Classification of forbidden subsequences of length 4, European J. Combin. 17 (1996), 501–517.

[10] R. E. Tarjan: Sorting using networks of queues and stacks, Journal of the ACM 19 (1972), 341–346.

[11] J. West: Generating trees and forbidden sequences, Discrete Math. 157 (1996), 363–374.