Algorithms for Testing Length Four Permutations

Yijie Han[1] and Sanjeev Saxena[2]

[1] School of Computing and Engineering, University of Missouri at Kansas City, Kansas City, MO 64110, USA
hanyij@umkc.edu

[2] Dept. of Compute Science and Engineering, Indian Institute of Technology, Kanpur, INDIA-208 016
ssax@cse.iitk.ac.in

Abstract. We present new $\theta(n)$ time algorithms for testing pattern involvement for all length 4 permutations. For most of these permutations the previous best algorithms require $O(n \log n)$ time.

Keywords: Separable permutations, pattern matching, optimal algorithms.

1 Introduction

Pattern containment (also called pattern involvement) is a well studied problem in both Computer Science and Combinatorics [1] (see [7] for a survey of results).

Two permutations $P = p_1p_2...p_k$ and $P' = p'_1p'_2...p'_k$ are said to be order isomorphic if their letters are in the same relative order, i.e., $p_i < p_j$, if and only if, $p'_i < p'_j$. For example permutation 1, 3, 2, 4 is order isomorphic to 7, 19, 15, 23.

A permutation $P = p_1p_2...p_k$ is said to be present in (or is involved in) another permutation $P' = p'_1p'_2...p'_n$ if P' has a subsequence which is order isomorphic to P.

The general problem of testing presence of one permutation in another permutation is NP-complete [4]. However polynomial time algorithms are known when

1. $P = 1, 2, ..., k$. This becomes largest increasing subsequence problem [5].
2. P' is separable [4]. A permutation is separable if it contains pattern 2, 4, 1, 3 or its reverse 3, 1, 4, 2.
3. k is a constant. Brute force algorithm will take $O(n^k)$ time.

In the case when $k = 3$, linear time algorithms are possible [1]. And for the case $k = 4$, [1] have shown that $O(n \log n)$ time is possible. Further, a linear time algorithm exists to test whether a pattern 2, 4, 1, 3 (or its reverse 3, 1, 4,

2) is present; this is basically a test to check whether a pattern is separable [4]. Linear time algorithms are also known for monotone patterns 1, 2, 3, 4 and (its reverse) 4, 3, 2, 1 [5].

Albert et.al. [1] studied the general problem of permutation involvement and proposed $O(n \log n)$ time algorithms for patterns of length 4. Their algorithms use orthogonal range queries of the kind:

Find the smallest number larger than a query item x between positions p and q.

As general orthogonal range queries take $O(\log n)$ query time after $O(n \log n)$ preprocessing time [8] (Theorem 2.12), $O(n \log n)$ time algorithms appear to be the best possible using this approach.

Our improvement comes firstly from use of the "usual" range maxima (minima) queries (instead of orthogonal range queries) of the kind:

Find the largest (smallest) number between positions p and q.

And for the case 1,3,2,4 we use a particular kind of analysis to achieve $\theta(n)$ time.

In this paper, we describe new linear time algorithms for all patterns of length 4.

Some useful "tools" are described in Section 2. Most cases of length 4 patterns are covered in Section 3. In Section 4 we describe the case 1,3,2,4.

2 Preliminaries

We give the definition for range minima and nearest largers problems. We will use routines for range minima and for nearest largers as black boxes.

For the range minima problem, we are given an array $A[1 : n]$, which we preprocess to answer queries of the form:

Given two integers i, j with $1 \leq i \leq j \leq n$ the smallest item in sub-array $A[i : j]$.

Range minima queries take $O(1)$ time after $O(n)$ preprocessing cost [2, **?**]. Range maxima can also be solved in this way.

In the right nearest largers problem [3], for each item i of array $A[1 : n]$, we are to find $j > i$, closest to i, such that $A[j] > A[i]$ (thus items, $A[i+1], A[i+2], ..., A[j-2], A[j-1]$ are all smaller than $A[i]$). Or, $j = \min\{k | A[k] > A[i] \text{ and } k > i\}$.

The right nearest larger also take $O(1)$ time after $O(n)$ preprocessing cost [2, 3]. All nearest largers can be found in $O(n)$ time [2, 3]. The left nearest larger, the left nearest smaller, the right nearest smaller can also be solved in this way.

Let us assume that the permutation is given in array $P[1:n]$. Thus, if the ith item of the pattern is k, then $P[i]=k$. As P is a permutation, all items of P are distinct. Hence, if $P[i]=k$ then we can define the inverse mapping $Position[k]=i$. Thus, item $Y=P(y)$ will be to the right of item $X=P(x)$ in array P, if and only if, $y>x$ or equivalently $Position(Y)>Position(X)$. And item $Z=P(z)$ will be between X and Z if $Position(Z)$ is between (in value) $Position(X)$ and $Position(Y)$, i.e., $Position(X)<Position(Z)<Position(Y)$, or $Position(Y)<Position(Z)<Position(X)$ Let r be the nearest right larger of k in $Position$ array.

P-value	k	$k+1$	$k+2$	$\ldots$	r	$r+1$
$Position$					*	

Then, as items with P-value $k, k+1, , r-1$ are smaller, r is the first (smallest) P-value item larger than k and to its right. Moreover, the nearest smaller of k in the $Position$ array to the right is the first (smallest) P-value item larger than k and to its left.

Similarly, s the nearest larger of k in the $Position$ array on the left, is the first (largest) P-value item smaller than k and to its right. And nearest smaller of k in $Position$ array to its left, is the first (largest) P-value item smaller than k and to its left.

Thus,

Theorem 1: If we know $P[i]=k$, then we can find items closest in values (both larger and smaller than k) on either side of position i using nearest smallers or largers on the $Position$ array. $\square$

We have:

Corollary 1: After preprocessing, if $P[i]=k$, then we can find items closest in values (both larger and smaller than k) on either side of position i in $O(1)$ time. The preprocessing time is $O(n)$ time. $\square$

Let us assume that $P[i]=k$ and $P[j]=l$ with $k<l$.

P-value	k	$k+1$	$k+2$	$\ldots$	l	$l+1$
$Position$	i				j	

If x is the $RangeMinimaPosition(k,l)$ on $Position$-array, then x is the smallest (or the left most) $Position$-value between $Position(k)$ and $Position(l)$ of items with P-value between k and l.

Similarly, if y is the $RangeMaximaPosition(k,l)$ on $Position$-array, then y is the largest (or the right most) $Position$-value between $Position(k)$ and $Position(l)$ of items with P-value between k and l.

Thus,

Theorem 2: Given any $P[i]=k$ and $P[j]=l$, we can find the left most and the right most items with P-values between $P[i]$ and $P[j]$ using range maxima or range minima queries. $\square$

We have the following result:

Corollary 2: If $P[i] = k$, and $P[j] = l$, then we can find the left most and the right most items with P-values between k and l in $O(1)$ time after preprocessing. The preprocessing time is $O(n)$. $\square$

3 Length Four Permutations Except 1324

For length 4 sequences, there will be 24 permutations, but 12 of these will be reverse (i.e., $P_0[i] = P[n - i + 1]$) of some other. Further, 4 out of these 12 will be "complement" (i.e., $P[i] = n - P[i] + 1$), thus in all 8 permutations will be left [1]:

1234, 2134, 2341, 2314, 2143, 1342, 1324, 3142

2413 and 3142 are the cases for separable permutation and linear time algorithm for them are known [5]. In the next section we will give a linear time algorithm for the case 1324 and its reverse 4231. Here we sketch how to deal with other permutations. As techniques for these permutations are similar, the description will be a bit brief.

Depending on the case, we try to see if i can be chosen as a "2" or a "3". We will abbreviate this to just "fix 2" (i.e, $i = i_2$) or "fix 3" (i.e., $i = i_3$). We finally get a witness for tuple (i_1, i_2, i_3, i_4), if $P[i_1] < P[i_2] < P[i_3] < P[i_4]$ and i_1, i_2, i_3, i_4 occur in the same order as $1, 2, 3, 4$. Again, a flag can be set if we have a witness and reset otherwise. Finally a logical "or" will give the answer.

In some of the cases, we have to search for an item (usually 3) which has a still larger item (which can be then chosen as 4). For this to be done efficiently, we define a new array $R[1 : n]$. The element

$$R[i] = \begin{cases} P[i] \text{ if } i \text{ has a larger item to its right} \\ 0 \text{ otherwise} \end{cases}$$

By using right nearest largers, we can easily identify items which have a larger item to their right. Note that in R each nonzero element has a larger element to its right in P. Let us preprocess array R for range maxima queries.

The technique for various patterns is:

1234 Fix 2. 1 the smallest item to its left is $i_1 = RangeMinimaP(1, i)$. Item 3 can be obtained by range maxima on array R, $i_3 = RangeMaximaR(i, n)$. Finally, $i_4 = RangeMaximaP(i_3, n)$.

2134 Fix 2. Index i_1 of 1, the first item less than 2, can be found from right nearest smaller of i_2. And again 3 can be obtained by range maxima on array R, $i_3 = RangeMaximaR(i_1, n)$. Finally, $i_4 = RangeMaximaP(i_3, n)$.

2341 Fix 3. Index i_4 of 4, the first item more than 3, can be found from right nearest larger of i_3. $i_1 = RangeMinimaP(i_4, n)$ will choose 1 as the smallest item on the right of 4. And we use Corollary 1, to find i_2, the index of 2 as the item on left of i, just smaller than $P[i_3]$.

2314 Fix 3. Again we use Corollary 1, to find i_2, the index of 2 as the item on left of i_3, just smaller than $P[i_3]$. We use Corollary 2, to find i_4, the index of 4 as the rightmost item larger than $P[i_3]$. Finally, $i_1 = RangeMinimaP(i_3, i_4)$.

3412 Fix 4. 3 can be found as the largest element smaller than 4 on the left side of 4 using Corollary 1. We create an array of right near larger. For each element e in P that does not have another element f pointing to it (f's right near larger be e) we will change the value of e to max and we will call this array R_1. We then use $RangeMinimaR_1(i_4, n)$ to find 2. 1 would be the closest element on the left of 2 that use 2 as its right near larger.
REMARK Pattern 2, 1, 4, 3 is the reverse of pattern 3, 4, 1, 2

2413 This is reverse of pattern 3142. This is the test of separability of the text. Linear time algorithm is known for this case [5].

1342 Fix 3. 4 is right nearest larger. 1 can be found by $RangeMinimaP(1, i_3)$. Now in the $Position$ array use $RangeMaximaPosition(P[i_1], P[i_3])$ to find the position, i.e. i_2 (i.e. use Corollary 2).

Theorem 3: Given any length 4 permutation (other than 1324 and its reverse 4231), we can test whether it is present (involved) in another permutation of length n in $\theta(n)$ time. $\square$

4 Permutation 1324

In array P we use right nearest larger to build a forest. Within each tree of the forest we define the chain of the tree consisting of the root r of the tree, the largest child c_1 of r, the largest child c_2 of c_1, ..., the largest child c_{i+1} of c_i, ..., etc.

Our intension is to let roots of trees serve as 4 and nodes on the chains to serve as 3. 1 will be found using range minima and 2 will be served by non-chain nodes.

We traverse a tree this way: when we are at node d, we visit the subtrees of d in the order from the subtree rooted at the smallest child of d to the subtree rooted at the largest child of d. After that we visit d. We start at root of the tree. This traversal will label the nodes of the tree.

We start from the rightmost tree and visiting nodes in this tree in the order of the above traversal. Let d be a non-chain node we are visiting and let $d, d_1, d_2, ..., d_t$ be the path in the tree from d to the nearest ancestor d_t where d_t has another child c larger than d. In this case c will be larger than $d, d_1, d_2, ..., d_{t-1}$ and c will be on the left side of $d, d_1, ..., d_{t-1}, d_t$ in array P. Thus we can let d_{t-1} serve as 2, c serve as 3 and d_t serve as 4. 1 will be found using $RangeMinimaP(1, i_3)$. If $RangeMinimaP(1, i_3)$ is larger than 2, then we label $d, d_1, ..., d_{t-1}$ as they cannot serve as 2. They cannot serve as 1 or 3 because of our traversal order (i.e. they may have tried to serve as 1 or 3 before in the traversal).

Thus after we examined all non-chain nodes they cannot serve as 1, nor 2, nor 3. They need not serve as 4 because the root of the tree can serve as 4. Thus these non-chain nodes can be removed.

The chain nodes may later serve as 2 but they cannot serve as 1 or 3 because there are no qualifying 2 for them. All of them except the root need not serve as 4 because the root can serve 4 for them.

Next we examine the second rightmost tree. After we did the same examination for the tree as we did for the rightmost tree only the chain nodes remains to be tested as 2's. However, we have to test the chain nodes in the first tree as 2's using nodes in the second tree as 3's.

In order to do this we have the two lists of nodes each sorted in ascending order. One is the list L_1 of the chain nodes of the first tree and the second list L_2 is the one containing all the tree nodes of the second tree. Let r_1 be the root of the first tree and r_2 be the root of the second tree. We visit these two lists from smallest nodes.

Let a be the current smallest node in L_1 and b be the current smallest node in L_2. If $b < a$ then we remove b from further comparison and get next smallest node from L_2. In this case b is less than all remaining nodes in L_1 and therefore cannot serve as 3 for the remaining nodes in L_1 to serve as 2's. If $r_1 > b > a$ then we let r_1 serve as 4, b serve as 3 and a serve as 2 and use $RangeMinimaP(1, i_3)$ to find 1. If $RangeMinimaP(1, i_3) > 2$ then if a's chain parent p is less than b then p can replace a to serve as 2 and a can be deleted. If $p > b$ then b can be removed. In either case we remove one node. If $r_1 < b$ then we can stop because the remaining nodes cannot serve as 3 because there is no 4 for them.

Thus we visit nodes in the second tree at most twice. In the remaining we have the chain for the second tree left and some nodes on the chain for the first tree left. We can merge these nodes from two chains together and form an ascending list. The merging is not done by examining all nodes of the two chains as doing this way will be too costly. We maintain the above two ascending lists in linked lists. Thus once we find $r_1 < b$ for b as a chain node then we insert the remaining nodes in L_1 between b and b's chain child (here insert into the chain of the second tree and not inserting into L_2). Thus the merge takes constant time. Note now all b's chain descendants are smaller than the remaining nodes in L_1.

Then we view the merged chain node as one chain and we continue working on the third tree from right.

What we have described is the linear time algorithm for pattern 1324.

Theorem 4: Pattern 1324 can be detected in a permutation in $\theta(n)$ time. □

5 Concluding Remarks

We have described $\theta(n)$ time algorithms for testing involvement of all length 4 patterns. Previously most of these patterns have only $O(n \log n)$ time algorithms.

References

1. Albert M.H., Aldred R.E.L., Atkinson M.D., and Holton, D.A. Algorithms for Pattern Involvement in Permutations. Proc. ISAAC 2001, Springer Lecture Notes Computer Sc. vol. 2223 (2001), 355-367.

2. Berkman, O., Matias, Y., and Ragde, P. Triply-logarithmic parallel upper and lower bounds for minimum and range minima over small domains. Journal of Algorithms 28 (August 1998), 197-215.
3. Berkman, O., Schieber, B., and Vishkin, U. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. Journal of Algorithms 14 (May 1993), 344-370.
4. Bose, P., Buss, J. F., and Lubiw, A. Pattern matching for permutations. Information Processing Letters 65 (1998), 277-283.
5. Fredman, M.L. On Computing the length of longest increasing subsequences. Discrete Mathematics 11 (1975), 29-35.
6. Ibarra, L. Finding pattern matchings for permutations. Information Processing Letters 61 (1997), 293-295.
7. Kitaev, S. and Mansour, T. A survey on certain pattern problems. Available at http://www.ru.is/kennarar/sergey/index_files/Papers/survey.ps and http://ajuarna.staff.gunadarma.ac.id/Downloads/files/1662/survey.pdf
8. Preparata, F.P. and Shamos, M.I. Computational Geometry, An Introduction. Springer-Verlag, 1985
9. Saxena, S. Dominance made simple. Information Processing Letters 109 (2009), 419-421.
10. Tamassia, R. and Vitter, J.S. Optimal cooperative search in fractional cascaded data structures. Algorithmica 15 (1996), 154-171.
11. Yugandhar, V., and Saxena, S. Parallel algorithms for separable permutations. Discrete Applied Mathematics 146 (2005), 343-364.