ON THE LONGEST UPSEQUENCE PROBLEM FOR PERMUTATIONS

ERKKI MÄKINEN

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1999-7

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1999-7, APRIL 1999

ON THE LONGEST UPSEQUENCE PROBLEM FOR PERMUTATIONS

ERKKI MÄKINEN

University of Tampere
Department of Computer Science
P.O.Box 607
FIN-33101 Tampere, Finland

ISBN 951-44-4587-2
ISSN 0783-6910

On the longest upsequence problem for permutations

Erkki Mäkinen [1]

Department of Computer Science, University of Tampere, P.O. Box 607, FIN-33101 Tampere, Finland

Abstract

Given a permutation of n numbers, its longest upsequence can be found in time $O(n \log\log n)$. Finding the longest upsequence (resp. longest downsequence) of a permutation solves the maximum independent set problem (resp. the clique problem) for the corresponding permutation graph. Moreover, we discuss the problem of effeciently constructing the Young tableau for a given permutation.

Keywords: Algorithms; Permutation; Upsequence; Stratified tree; Young tableau.

1 Introduction

Consider a sequence of values $(v_1, \ldots, v_n)$. If one deletes i (not necessarily adjacent) values from the sequence, one has a subsequence of length $n - i$. This subsequence is called an *upsequence* (resp. *downsequence*) if its values are in nondecreasing (resp. nonincreasing) order. Gries [6, p. 262] gives a simple algorithm for finding the length of the longest upsequence in a given sequence with n values in time $O(n \log n)$. This algorithm scans the sequence from left to right and maintains the minimum values $m_1, \ldots, m_k$ which end the upsequences on length $1, \ldots, k$, respectively, so far found. For a value v_i in the sequence the algorithm operates as follows: It first compares v_i with the smallest value m_1 and the greatest value m_k. If $v_i < m_1$, then v_i is set to be the new value of m_1. On the other hand, if $v_i \geq m_k$, then an upsequence of length $k+1$ is found, and v_i is stored as m_{k+1}. Otherwise ($m_1 \leq v_i < m_k$), the algorithm finds an index j such that $m_{j-1} \leq v_i < m_j$, and sets v_i to new value of m_j. Finding the correct index j takes time $O(\log k)$ resulting the overall time bound $O(n \log n)$.

In its original form [6], the Gries algorithm finds only the minimal values which end the upsequences of lengths from 1 to k. The longest upsequence is easily

[1] E-mail: em@cs.uta.fi. Work supported by the Academy of Finland (Project 35025).

found by recording the positions to which the elements are inserted. After scanning the input sequence, one simply selects the greatest element from the set of values inserted to the position m_k, a smaller element from the set of values inserted to the position m_{k-1}, and so on, until the whole upsequence is found. The result is not necessarily unique, i.e., a permutation may have several equally long upsequences. Finding the upsequence by recording the insertion positions is always possible in linear time. In what follows, we do not further consider this step of the algorithm.

Finding the correct index j can be performed by using binary search. Scanning the sequence consists of "easy cases" ($v_i < m_1$ or $v_i \geq m_k$) and "difficult cases" ($m_1 \leq v_i < m_k$). The former cases take only a constant time, while the latter ones need time $O(\log k)$. In what follows, the above algorithm solving the longest upsequence problem and using normal binary search for finding the index j is referred to as the *Gries algorithm.*

The subject of the present paper is the problem of finding the longest upsequence (or the longest downsequence) of a given permutation. Obviously, when an arbitrary sequence is replaced by a permutation, the time bound $O(n \log n)$ should be improved. It turns out that data structures for restricted universes can be successfully used since the "domain" of the permutation is supposed to be known. The problem of finding the correct index j in the algorithm can be considered as the problem of finding the successor in a given finite set. Knowing the universe (i.e., values $1, \ldots, n$) makes it possible to find the successor in time $O(n \log \log n)$ [13].

Monotonic subsequences of permutations are of interested in various contexts. It is well-known that the upsequences (resp. downsequences) of a permutation are in one-to-one correspondence with the independent sets (resp. the cliques) of the corresponding permutation graph [5]. Hence, finding the longest upsequence (resp. the longest downsequence) solves the maximum independent set problem (resp. the clique problem) of the corresponding permutation graph. Chang and Wang [3] have reported an $O(n \log \log n)$ time algorithm for this problem. We are not able to improve they time bound, but our discussion will make the problem setting clearer and the algorithm itself conceptually simpler.

The longest downsequence of a permutation gives the number of queues needed in sorting the permutation [5, Cor. 7.4.]. Finding the upsequence of a permutation is of interest also in a more general setting of pattern matching for permutations, see [2,7]. Upsequences (and downsequences) are instances of patterns that can be searched in permutations. Finding upsequences is also closely related to Young tableaux [9,12], as will be discussed in Chapter 3.

In the sequel, we usually speak about upsequences only. Analogously results naturally hold for downsequences. Notice that an upsequence of $(p_1, \ldots, p_n)$ is a downsequence of $(p_n, \ldots, p_1)$. Similarly, a problem in a permutation graph G has usually a meaningful countrpart in the complement graph $\overline{G}$, e.g. an independent set of G is a clique in $\overline{G}$.

2 Finding the longest upsequence

Consider a permutation $P = (p_1, \ldots, p_n)$ of numbers from 1 to n. The following algorithm finds the longest upsequence of P. For the time being, we have left open the way of determining index j.

Algorithm 1 (Longest Upsequence)
Input: *a permutation* $P = (p_1, \ldots, p_n)$, $p > 1$, *of* $\{1, \ldots, n\}$
  { $m[1..n]$ *is an array of integers;* $m[i]$ *contains the smallest*
    number ending an upsequence of length i *so far found;*
    k *is the length of the longest upsequence so far found* }
begin
  $m[1] := p_1$;
  $k := 1$;
  for $i = 2, \ldots, n$
  do
    if $p_i > m[k]$
    then begin
        $k := k + 1$;
        $m[k] := p_i$ ***end***
      else
        if $p_i < m[1]$ ***then*** $m[1] := p_i$
        else $m[j] := p_i$, *where* $m[j] < p_i < m[j + 1]$;
  Find the upsequence by using the lists of elements inserted
      to the different positions;
end $\{$*Algorithm*$\}$

The line $m[j] := p_i$, where $m[j] < p_i < m[j + 1]$; of Algorithm 1 can be performed by stratified trees (also called van Emde Boas priority queues) in $O(\log \log n)$ time and in $O(n)$ space [10,13]. This follows essentially because the universe of keys to be stores in the structure is $\{1, \ldots, n\}$.

Suppose that we can use the following standard stratified tree operations:

- $Max(T)$: return the largest key in T
- $Succ(x, T)$: return the successor of x in T
- $Insert(x, T)$: insert x to T
- $Delete(x, T)$: delete x from T.

Our algorithm can now be written as follows.

Algorithm 2 (Longest Upsequence using Stratified Tree)
Input: *a permutation* $P = (p_1, \ldots, p_n)$, $p > 1$, *of* $\{1, \ldots, n\}$
  { *T is a stratified tree* }
begin
  $Insert(p_1, T)$;
  for $i = 2, \ldots, n$
  do
    if $p_i > Max(T)$
      then $Insert(p_i, T)$
      else begin
        $Delete(Succ(p_i, T), T)$;
        $Insert(p_i, T)$
      end
      Find the upsequence by using the lists of elements
        inserted to the different positions;
end {*Algorithm*}

Stratified trees are later streamlined so that their space complexity depends on the number of elements in the structure and not on the size of the universe [11]. In these *bounded ordered dictionaries* insertions and deletions have $O(\log\log n)$ time bound in amortized and randomized sense. However, in our application this improvement in space complexity does not make any difference in the worst case since the structure may well contain (almost) all elements of the universe.

Contrary to the Gries algorithm, Algorithm 2 does not have any "easy cases". Indeed, each item scanned takes time $O(\log\log n)$, since even when $v_i > m[k]$, the new item must be inserted to the stratified tree by using an operation taking $O(\log\log n)$ time.

The average length of the longest upsequence of a random permutation of $\{1, \ldots, n\}$ is $2\sqrt{n}$ [9, p. 68]. Hence, the Gries algorithm finds the longest upsequence of a given permutation in $O(n \log \sqrt{n})$ time on average. (Since $\log \sqrt{n} = \frac{1}{2} \log n$, the notation $O(n \log \sqrt{n})$ only emphasizes the small constant factor involved.)

A typical example of unfavorable permutations for the Gries algorithm is $(1, \frac{n}{2}+1, \frac{n}{2}+2, \ldots, n, 2, 3, \ldots, \frac{n}{2})$. As an input of the Gries algorithm, this permutation causes first $\frac{n}{2}$ easy cases and then $\frac{n}{2}$ difficult cases. The time needed is $\Theta(n \log n)$.

The results of this chapter can be summarized as follows.

Proposition 1 *The longest upsequence of a given permutation can be found in $O(n \log \log n)$ time and in $O(n)$ space.*

Proposition 2 *The Gries algorithm finds the longest upsequence of a given permutation in time $O(n \log \sqrt{n})$ on average.*

An interesting result concerning Proposition 2 is the following: If P is a permutation containing more than n^2 elements, then there is either an upsequence or a downsequence of length greater than n [4]. Hence, for each permutation, the average case time bound of Proposition 2 is reached for at least one of the problems of finding the longest upsequence or finding the longest downsequence.

Algorithms 1 and 2 do not use all information avaliable. For example, we know that each element is inserted to the structure exactly once. Moreover, all updates are either replacements (an element x in the structure is replaced by a smaller element y which is greater than the predecessor (if exists) of x) or insertions in which the new element is greater than all elemets so far in the structure. We pose it open whether it is possible to take advantage of this information and to improve the time bounds given.

The time complexity of the Gries algorithm depends on the length of the longest upsequence. If it is at most $O(\log n)$, then the Gries algorithm has the same time complexity than Algorithm 2 with considerably much smaller constant factor both in time and space complexity. This fact rises the question of finding classes of permutations with known short longest upsequences or downsequences. For example, many of the permutation types studied in [1] are in this category.

3 Young tableaux

A Young tableau of shape $(n_1, n_2, \ldots, n_m)$, where $n_1 \geq n_2 \geq \ldots \geq n_m > 0$, is an arrangement of $n_1 + n_2 + \ldots + n_m$ distinct intergers in an array with m rows such that in row i there are n_i elements, each row is in increasing order from left to right, and each column is in increasing order from top to bottom. The link between permutations and Young tableaux is that the number of involutions (i.e., permutations that are their own inverses) of $\{1, \ldots, n\}$ is the same as the number of tableaux that can be formed from the elements

$\{1, \ldots, n\}$. (For more information concerning Young tableaux, see [9, Section 5.1.4].) In this chapter we consider the complexity of constructing the Young tableau of a given permutation.

Given a permutation $P = (p_1, \ldots, p_n)$ its Young tableau is constructed by inserting the elements $p_1, \ldots, p_n$ one by one to the originally empty tableau. Inserting p_i is performed as follows. First, find the correct place for p_i in row 1 by Algorithm 2. If p_i is greater than all elements in row 1, insert it to be the last element of row 1, and halt. Otherwise, continue by inserting the element, say r, replaced by p_i in row 2. Again, halt if r is the greatest in the row. Otherwise, continue with the next replaced element and the next row, until a row is found where all elements are smaller than the new element to be inserted or a new row is started.

Consider, for example, constructing the Young tableau for $(3, 5, 4, 9, 8, 2, 7, 6, 1)$. After inserting 3, 5, 4, 9, 8, 2, and 7, the tableau has the form shown in Figure 1.

2	4	7
3	8	
5	9	

Figure 1. The tableau after inserting 3, 5, 4, 9, 8, 2, and 7 (in that order).

Inserting 6 causes first the replacement of 7 by 6 in row 1. Then, 8 is replaced by 7 in row 2, and 9 is replaced by 8 in row 3. Finally, a new row is started with 9 as the only element. The resulting tableau is shown in Figure 2.

2	4	6
3	7	
5	8	
9		

Figure 2. The tableau of Figure 1 after inserting 6.

Inserting 1 to the tableau obtained would cause changes only in the first column: 2 replaced by 1, 3 replaced by 2, 5 relaced by 3, 9 relaced by 5, and a new row with 9 as the only element is started.

Given a permutation, the first row of its Young tableau can be produced by Algorithm 2. The number of columns equals the length of the longest upsequence. Similarly, the number of rows equals the length of the longest downsequence [12].

The rows of the Young tableau for $(p_1, \ldots, p_n)$ are the columns of the tableau for $(p_n, \ldots, p_1)$. That is, reversing the permutation tranposes the tableau [9]. Hence, the Young tableau of (1,6,7,2,8,9,4,5,3), i.e. the reverse of the permutation considered in Figures 1 and 2, is the tableau shown in Figure 3.

1	2	3	5	9
4	7	8		
6				

Figure 3. The Young tableau of the permutation (1,6,7,2,8,9,4,5,3).

In what follows we consider the problem of efficiently constructing the Young tableau for a given permutation. The first row of the tableau is obtained by Algorithm 2, as well the first column is obtained when applying the algorithm to the reversed permutation. In the worst case, it is sufficient to construct $\sqrt{n}$ times the next row and column. We count the number of elements inserted to the tableau in order to be able to halt as soon as all elements are inserted. Namely, the number of both rows and columns can be greater than $\sqrt{n}$ although more than $\sqrt{n}$ rows and columns are never needed to fill the tableau.

In order to be able to continue the construction of the tableau after the first row (or column), we have to augment Algorithm 3 such that it also outputs the new permutation to be used when determining the second row. Namely, the process of constructing the first row changes the order of the elements. All the elements of the input permutation are first inserted to row 1. The new order of the elements essentially depends on how long each element stays there.

As an example, consider again the permutation (3,5,4,9,8,2,7,6,1) discussed earlier in the connection with Figures 1 and 2. The first row contains elements 1, 4, and 6. Deleting these from the permutation would give the input (3,5,9,8,2,7). However, the correct input is (5,9,3,8,7,2). It can be produced simply by recording the order in which the elements are replaced by other elements while Algorithm 3 is executed.

Algorithm 3 (Young tableau)
Input: *a permutation* $P = (p_1, \ldots, p_n)$, $p > 1$, *of* $\{1, \ldots, n\}$
begin
  rowpermutation:= P;
  columnpermutation:= $(p_n, \ldots, p_1)$;
  i:= 1;
  counter:= 0;
  while *rows left* ***and*** *columns left* ***and*** *counter* $< n$ ***do***
  begin
    construct the i^{th} *row by Algorithm 3 from rowpermutation;*
    rowpermutation:= the elements not in the i^{th} *row in the order*
      they are replaced while constructing the i^{th} *row;*
    construct the i^{th} *column by Algorithm 3 from columnpermutation;*
    columnpermutation:= the elements not in the i^{th} *column in the order*
      they are replaced while constructing the i^{th} *row;*
    counter:= counter + the number of elements inserted in the i^{th} *round*
    i:= i + 1;
  end
end $\{$*Algorithm*$\}$

For notational simplicity, suppose that $n = k^2$, for some k. By Proposition 1, the first round of the while-loop takes time $O(n \log\log n)$. Since our algorithm constructs both rows and columns, the worst case is the one, where the resulting Young tableau is a $k \times k$ square. In this case, the second round takes time $O((n-k)\log\log(n-k))$, and the $(i+1)^{st}$ round takes time $O((n-ik)\log\log(n-ik))$.

Hence, the total time needed is

$$\sum_{i=1}^{k} ik \log\log ik.$$

Since $\log\log ik < \log\log n$, we have

$$\sum_{k=1}^{k} ik \log\log ik < k \log\log \sum_{i=1}^{k} i = k\frac{k(k+1)}{2}\log\log n.$$

By replacing $k = \sqrt{n}$, we have obtained the time bound $(n^{1.5}\log\log n)$.

Theorem 3 *There exists an* $O(n^{1.5}\log\log n)$ *time algorithm for constructing the Young tableau for a given permutation.*

A marginally better algorithm would result, if we could avoid considering all but the diagonal elements twice (or more times). This would give the time bound $\sum_{i=1}^{k} i^2 \log\log i^2$. However, we do not further elaborate this idea, since it would imply no improvement in the order of time complexity.

Notice that a trivial algorithm runs in $O(n^2)$ time by simply traversing the tableau row by row as long as all operations caused by inserting an element to the first row are done.

Knuth [9, p. 55] mentions that the minimum running time of "tableau sorting" is propotional to $n^{1.5}$. Tableau sorting consists of constructing the Young tableau and then deleting the elements in increasing order.

References

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

[2] Brosejint Bose, Jonathan F. Buss, and Anna Lubiw, Pattern matching for permutations. *Inform. Process. Lett.* **65** (1998), 227–283.

[3] Maw-Shang Chang and Fu-Hsing 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] P. Erdös and G. Szekeres, A combinatorial problem in geometry. *Compositio Math.* **2** (1935), 463–470.

[5] Martin Charles Golumbic, *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, 1980.

[6] David Gries, *The Science of Programming.* Springer, 1981.

[7] Louis Ibarra, Finding pattern matchings for permutations. *Inform. Process. Lett.* **61** (1997), 293–295.

[8] Rolf G. Karlsson, Algorithms in a restricted universe. Dept. of Computer Science. Univesity of Waterloo, Research Report CS-84-50, November 1984.

[9] Donald E. Knuth, *The Art of Computer Programming. Vol. 3, Sorting and Searching. Second Edition.* Addison-Wesley, 1998.

[10] Kurt Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching.* Springer, 1984.

[11] Kurt Mehlhorn and Stefan Näher, Bounded ordered dictionaries in $O(\log \log n)$ time and $O(n)$ space. *Inform. Process. Lett.* **35** (1990), 183–189.

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

[13] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space. *Inform. Process. Lett.* **6** (1977), 80–82.