Enumerating Longest Increasing Subsequences and Patience Sorting

Sergei Bespamyatnikh and Michael Segal[1]

Department of Computer Science, University of British Columbia, Vancouver, B.C. Canada V6T 1Z4

Abstract

In this paper we present three algorithms that solve three combinatorial optimization problems related to each other. One of them is the *patience sorting* game, invented as a practical method of sorting real decks of cards. The second problem is computing the *longest monotone increasing subsequence* of the given sequence of n positive integers in the range $1, \ldots, n$. The third problem is to *enumerate all the longest monotone increasing subsequences* of the given permutation.

Key words: Algorithms, longest increasing subsequence, van Emde Boas tree

1 Introduction

In this paper we consider the following three related problems:

Longest increasing subsequence of permutation. Given an arbitrary permutation π of $\{1, 2, 3, \ldots, n\}$, an *increasing subsequence* $< s_1, s_2, \ldots, s_k >$ of π is a subsequence satisfying

$$s_1 < s_2 < \ldots < s_k; \quad \pi(s_1) < \pi(s_2) < \ldots < \pi(s_k)$$

The goal is to find the longest increasing subsequence of a permutation π.

[1] Work by MichaelSegal has been supported by the Pacific Institute for Mathematical Studies, Canada

Enumerating all increasing subsequences of permutation. Given an arbitrary permutation π of $\{1, 2, 3, \ldots, n\}$, find all longest increasing subsequences of a permutation.

Patience sorting. Take a deck of cards labeled $1, 2, 3, \ldots, n$. The deck is shuffled, cards are turned up one at a time and dealt into piles on the table, according to the rule : A card with a low index may be placed on a card with a higher index, or may be put into a new pile to the right of the existing piles. At each stage we see the top card on each pile. If the turned up card is higher than the cards showing, then it *must* be put into a new pile to the right of the others.
The target of the game is to finish with as few piles as possible.

There are a lot of papers that deal with the longest increasing subsequences and patience sorting problems. The patience sorting problem was discovered by Mallows [7] who actually proposed it as a way for manually sorting cards. In the same paper Mallows show that the number of piles in patience sorting relates to the Young tableaux that was invented by Schensted [8] in order to study the length of the longest increasing subsequence $l(\pi)$. Floyd [4] described the patience sorting in letters between him and Knuth [6] who gave an $O(n \log n)$-time algorithm for computing longest increasing subsequence for an arbitrary sequence of n numbers. In a very recent paper, Aldous and Diaconis [1] proved several interesting results related to this problem. In particular, they proved that the *greedy* strategy (that is, to always place a card on the leftmost possible pile) is optimal and, moreover, the number of piles the greedy strategy ends with is equal to $l(\pi)$. The brute-force approach in [1] requires $O(n^2)$ comparisons. They [1] pointed out that according to the paper by Fredman [3] the algorithm to find $l(\pi)$ (and, thus, patience sorting) must perform $\Omega(n \log n)$ comparisons. Nevertheless, Hunt and Szymanski [5] gave an $O(n \log \log n)$ runtime algorithm for computing the longest increasing subsequence for a given permutation. Their algorithm actually solves the more general problem of computing the longest common subsequence of two sequences. As a result this algorithm applied to the longest increasing subsequence problem is rather complicated and requires redundant extra space (although remains $O(n)$).

We will present a direct, simple algorithm with $O(n \log \log n)$ runtime in order to solve the longest increasing subsequence problem which can be used to report all such subsequences in optimal time. The previous approach [5] does not allow to do this. Moreover, we show how to extend our approach to solve patience sorting problem.

We present our algorithm for computing longest increasing subsequence and enumerating all the subsequences in the next Section. In Section 3 we describe how to change this algorithm in order to solve the patience sorting problem.

We conclude in Section 4.

2 Longest Increasing Subsequence

We recall that the input of our problem is some permutation π of n numbers. For each element $\pi(i)$, $1 \leq i \leq n$ the algorithm computes the length of the longest increasing subsequence that ends on $\pi(i)$. We keep all these values in an array L. In other words, $L[\pi(i)]$ is the length of the longest increasing subsequence that ends on $\pi(i)$. The main idea of the algorithm is to maintain a list T such that j-th element of this list is the smallest element of permutation π that increasing subsequence of length j ends with. To implement T we use the data structure invented by van Emde Boas [9] (see also [2]) that allows to maintain the sorted list of integers in the range $1, \ldots, n$ in $O(\log \log n)$ time per insertion and deletion.

The data structure T allows the following list operations:

- *insert(i)* - insert the number i into S,
- *delete(i)* - delete the number i from S,
- $next(i)$ - get the successor of i in S, if it does not exist return **nil** (takes $O(1)$ time provided i is already inserted into S),
- $prev(i)$ - get the predecessor of i in S, if it does not exist return **nil** (takes $O(1)$ time provided i is already inserted into S).

First Stage

At the first stage we proceed from the left to the right of the permutation π. Consider the moment when the i-th element $\pi(i)$ is processed. We need to determine the length $L[\pi(i)]$ of the longest increasing subsequence that ends on $\pi(i)$. This length is defined by longest increasing subsequence that ends on some element of π that is smaller than $\pi(i)$ and has been considered before. In order to do this we insert the number $\pi(i)$ in the list T. The length $L[\pi(i)]$ is equal to 1 plus the length associated with the predecessor of $\pi(i)$ in the list T, i.e. $L[\pi(i)] = 1 + L[prev(\pi(i))]$. If there is no predecessor we set $L[\pi(i)] = 1$. If the successor of $\pi(i)$ in the list T has the same associated length, then we delete the successor of $\pi(i)$ from T. If there is no successor of $\pi(i)$ we are done and proceed to the next step.

Second Stage

At the second stage we have filled array L. It turns out that L contains enough information to construct the longest increasing subsequence of π in linear time. Indeed, the length $k = l(\pi)$ of this subsequence $\bar{l} = <s_1, s_2, \ldots, s_k>$ is determined by the largest value that is stored in L. This subsequence $\bar{l}$ satisfies the following property:

$$L[\pi(s_1)] = 1, L[\pi(s_2)] = 2, \ldots, L[\pi(s_k)] = l(\pi).$$

The index $\pi(s_k)$ of the cell that stores the largest value in L is equal to the last element of the list T. We can find it either in the list T or by simple scanning the array L.

To find the remaining elements of the subsequence the algorithm goes from the s_k-th element of permutation to the first one. As was mentioned above j-th element of this subsequence $\bar{l}$ is the first index i such that $L[\pi(i)] = j$.

We give below the formal description of the algorithm (the output is the sequence $<s_1, s_2, \ldots, s_{l(\pi)}>$).

```
Longest Increasing Subsequence
    T = ∅;
  First Stage
    for i = 1 to n do
          m = π(i);
          insert(m)
          if prev(m) ≠ nil then
              L[m] = L[prev(m)] + 1;
            else L[m] = 1;
          if next(m) ≠ nil then
              if L[next(m)] == L[m] then
                    delete(next(m));
  Second Stage
    k = L[1]; index = 1;
    for i = 2 to n do
          if L[i] > k then
                k = L[i]; index = i;
    s_k = index; j = k - 1;
    for i = index to 1 do
          if L[π(i)] = j then
                s_j = i; j = j - 1;
```

Theorem 2.1 *The algorithm above correctly find the longest increasing subsequence of a given permutation and has $O(n \log\log n)$ running time.*

Proof. The correctness of the algorithm follows from the discussion above. The steps 15–18 form an output sequence $< s_1, s_2, \ldots, s_{l(\pi)} >$.

It is easy to evaluate the running time. All steps from 3 to 10 take constant time except steps 4 and 10. Steps 4 and 10 are accomplished at most n times spending $O(\log\log n)$ time. Clearly, the second stage takes linear time. So the total running time is $O(n \log\log n)$. ∎

2.1 Reporting all subsequences

The array L obtained by the previous algorithm contains sufficient information to enumerate all longest increasing subsequences of π.

Theorem 2.2 *All longest increasing subsequences of a given permutation can be reported in optimal $O(n + Kl(\pi))$ time and optimal $O(n)$ space, where K is the number of such subsequences.*

Proof. We first describe our algorithm and then prove its correctness and running time. Recall that $L[\pi(j)]$ is the length of the longest subsequence with $\pi(j)$ as the last element.

Observation: Consider the indices $i_1 < i_2 < \ldots < i_m$, such that $L[i_1] = L[i_2] = \ldots = L[i_m]$. Then, the sequence $< \pi(i_1), \pi(i_2), \ldots, \pi(i_m) >$ is decreasing.

For each element j, $1 \leq j \leq n$, we store two additional indices $left1$ and $left2$. They are defined as follows. The value of $left1(j)$ is the largest index i, such that $L[i] = L[j]$, $i < j$. If such i does not exist, we set $left1(j) =$ **nil**. The value of $left2(j)$ is the largest index i, such that $L[i] = L[j] - 1$, $i < j$. If such i does not exist, we set $left2(j) =$ **nil**. We can compute all the values of $left1$ and $left2$ in linear time by scanning the array L.

Our algorithm is based on a recursive procedure *Enumerate* which on input z reports all longest increasing subsequences with z as the last element. It uses an auxiliary array *Out* for storing the subsequence which is currently being constructed. The array *Out* is filled in the reverse order. The length of this array is equal to $l(\pi)$. The initial call of *Enumerate* is done with parameter $z = \pi(s_k)$, where index s_k was computed at the second stage of the previous algorithm.

Reporting all subsequences

```
   Enumerate(s_k);

 Procedure Enumerate(z)
  // Outputs all the subsequences that ends by z.
 1.   if ((L[z] = l(π)) or (z < Out[L[z] + 1]) then
 2.        Out[L[z]] = z; // z is the current element of subsequence.
 3.   else return;
 4.   z1 = left2(z); // z1 is the predecessor of z in subsequence.
 5.   if z1 = nil then
 6.        print(Out); // Out is already filled.
 7.   else Enumerate(z1); // continue to fill Out.
 8.   while left1(z1) ≠ nil do
 9.       Enumerate(left1(z1)); // start a new subsequence.
```

It is easy to see that the above algorithm requires linear space and runs in time $O(n + Kl(\pi))$, where where K is a number of the longest monotone increasing subsequences. To show the correctness we observe that in fact our algorithm simulates the depth first search strategy. During each step of our algorithm we know the tail of the current longest subsequence, namely $Out[L[z]], Out[L[z]+1], \ldots, Out[l(\pi)]$. The algorithm tries to increase the tail of the current subsequence by looking on all possible values for the $(L[z]-1)$-th position. Our observation above provides an efficient way to find these values using pointers $left1$ and $left2$, cutting the search at line 1. ∎

3 Patience Sorting

A permutation π of $\{1, 2, 3, \ldots, n\}$ can be identified with an arrangement of n-card deck, by specifying that $\pi(i)$ is the index of the card at position i. The algorithm of Aldous and Diaconis [1] applied the following greedy strategy: a card is always placed on the leftmost possible pile.

Let P be an array that stores the top cards of piles. To store information about the cards in piles we use an array $cards$ of length n defined as follows: $cards[i]$ is the index of the card that lies below the card with index i. Let S be the data structure of van Emde Boas [9] that represents the list of top cards.

Consider the moment when the i-th card with index $\pi(i)$ is turned up. Note that the top cards in the piles form an increasing sequence of integers. We need to find the leftmost pile with a top card whose index j is greater than

$\pi(i)$. In order to do this we insert the number $\pi(i)$ in the list S of top cards. The number j is the successor of $\pi(i)$ in the list S after the insertion of $\pi(i)$. To reflect the placement of $\pi(i)$ on j, we set $cards[\pi(i)] = j$ and delete j from S. Eventually, we fill an array P using list S. We give below the formal description of the algorithm.

```
Patience sorting
 1.   S = ∅;
 2.   for i = 1 to n do
 3.          P[i] = card[i] = 0;
 4.   for i = 1 to n do
 5.          k = π(i);
 6.          insert(k);
 7.          j = next(k);
 8.          if j ≠ nil then
 9.              card[k] = j;
10.              delete(j);
11.    k = i = 1;
12.    while k ≠ nil do
13.           P[i] = k;   k = next(k);   i = i + 1;
```

Theorem 3.1 *The algorithm of patience sorting is correct and has $O(n \log\log n)$ running time.*

Proof. The algorithm uses the correct greedy approach [1]. The Steps 11–13 form an output array P containing the top cards that are stored in S. Note that the first element of the list S is 1.

Consider the running time of the algorithm. All steps except steps 6 and 10 take linear time. Steps 6 and 10 are performed at most n times spending $O(\log\log n)$ time which leads to the total $O(n \log\log n)$ running time. ∎

4 Conclusions

In this paper we investigated three related problems and we developed efficient algorithms for solving them. The key idea of the algorithms is based on using van Emde Boas [9] data structure for operations on permutations. We expect that the same technique can be used in order to solve the other permutation problems.

References

[1] D. Aldous and P. Diaconis, "Longest Increasing Subsequences: From Patience Sorting to the Baik-Deift-Johansson", *Bull. Amer. Math. Soc.*, 36 (1999) pp. 413–432.

[2] G.H. Gonnet and R. Baeza-Yates, "Handbook of Algorithms and Data Structures in Pascal and C", Addison Wesley, 1991, pp. 216–217.

[3] M. Fredman, "On computing the length of the longest increasing subsequence", *Discrete Math.*, 11 (1975), pp. 29–35.

[4] B. Floyd, unpublished work, 1964.

[5] J. Hunt and T. Szymanski "A fast algorithm for computing longest common subsequences", *Communications of ACM*, 20 (1977), pp. 350–353.

[6] D. E. Knuth, "Sorting and Searching", *The Art of Computer Programming*, 3 (1973), Addison-Wesley.

[7] C. Mallows, "Patience sorting", *Bull. Inst. Math. Appl.*, 9 (1973), pp. 216–224.

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

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