Longest Increasing Subsequences in Sliding Windows

Michael H. Albert[1], Alexander Golynski[2], Angèle M. Hamel[3], Alejandro López-Ortiz[2], S. Srinivasa Rao[2], Mohammad Ali Safari[2]

[1] Department of Computer Science, University of Otago, Dunedin, New Zealand, `malbert@cs.otago.ac.nz`

[2] School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada, `{agolynski,alopez-o,ssrao,masafari}@uwaterloo.ca`

[3] Department of Physics and Computer Science, Wilfrid Laurier University, Waterloo, ON, N2L 3C5, Canada, `ahamel@wlu.ca`

Abstract. We consider the problem of finding the longest increasing subsequence in a sliding window over a given sequence (LISW). We propose an output-sensitive data structure that solves this problem in time $O(n \log\log n + \text{OUTPUT})$ for a sequence of n elements. This data structure substantially improves over the naive generalization of the longest increasing subsequence algorithm and in fact produces an output-sensitive optimal solution.

1 Introduction

Given a sequence $a_1 a_2 \cdots a_n$ of distinct values from a linearly ordered set its *longest increasing subsequence* (LIS) is a subsequence of maximum length, whose values increase as the indices increase. The underlying set of the given sequence can be, and usually is, taken to be $\{1, 2, \cdots, n\}$, so that the sequence can be viewed as a permutation $\pi = \pi(1)\pi(2)\cdots\pi(n)$. In this setting the LIS consists of a sequence of indices $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ such that $\pi(i_1) < \pi(i_2) < \cdots < \pi(i_k)$ where k is the largest number for which such a sequence exists.

The *longest increasing subsequence problem* refers either to identifying the longest increasing subsequence(s) or, alternatively, to determining the length k of the LIS. In either of these forms, this problem has been the subject of intense study by mathematicians and computer scientists alike (see Subsection 2.1 for a more detailed discussion of previous work). This problem has interesting properties both from a purely combinatorial perspective (see e.g. [13]) as well as actual applications in fields such as DNA sequence matching [4].

In this paper, we consider the problem of finding the length of a longest increasing subsequence in every window of a sequence (LISW) of given width w; that is, the LIS's of substrings of π of the form $\pi(i+1)\pi(i+2)\dots\pi(i+w)$. This work is inspired by the study of the relationship and theoretical underpinnings of a problem and its windowed version [5, 6]. We propose an output sensitive data structure which solves LISW in optimal time; that is, linear on the size of the output.

2 Problem Definition

Given a sequence $\pi = \pi(1)\pi(2)\dots\pi(n)$ and a window size $w \leq n$, the windows W_i of α of width w are the subsequences $\pi(i+1)\pi(i+2)\dots\pi(i+w)$ for $0 \leq i \leq n-w$. The general problem that we consider is that of determining a LIS in each of the windows W_i. Within this framework, several related questions can be posed regarding this problem, each with potentially different time complexity:

Local Max Value For each window report the length k of the longest increasing subsequence in that window.

Local Max Sequence Explicitly list a longest increasing sequence for each window.

Global Max Sequence Find the window with the longest increasing sequence among all windows.

We will deal with the Local Max Sequence form of the LISW. The algorithm we present runs in linear time on the size of the output for this problem and hence is optimal both in worst case and adaptive sense. The same algorithm solves the other two versions of the problem described above, although its optimality in these cases is an open question.

2.1 Previous Work

Algorithms for finding the length of the LIS date back to Schensted [14] and Robinson [12] with a generalization due to Knuth [10]. These algorithms have time complexity $O(n \log n)$ which is optimal in the comparison model. Hunt and Szmanski [9] give an algorithm with time complexity $O(n \log \log n)$ using the van Emde Boas data structure. Chang and Wang [3] also give an $O(n \log \log n)$ algorithm based on a permutation graph interpretation. Bespamyatnikh and Segal [2] present an $O(n \log \log n)$ algorithm that determines *all* longest increasing subsequences. The algorithm we present here is $O(n \log \log n + \text{OUTPUT})$. Probabilistic results related to this problem have been discussed in Aldous and Diaconis [1] and Groenboom [7]. The question also has application in bioinformatics in the MUMmer system for finding matches between DNA sequences [4].

2.2 Background of our results

For the problem we consider there is an obvious naive algorithm which simply computes the LIS in each window separately. Using the methods of the preceding paragraph, this gives an algorithm whose complexity is $O(nw \log\log n)$. In the case where the average length of the LIS in each window is $\Theta(w)$ then, our algorithm offers no asymptotic improvement over this method.

However, it is well known in the permutation case that the average length of the LIS of a permutation of length n is asymptotically $2\sqrt{n}$ (see [1] for this result, and references). Suppose that a permutation π of length n is chosen uniformly at random. Consider any fixed window of π. The relative ordering of values observed in that window will also be uniformly chosen from among the patterns of permutations of length w. Thus the expected length of an LIS in any given window is asymptotically $2\sqrt{w}$ and by linearity of expectation, the expected total length of all LIS's is $(n-w+1)2\sqrt{w} = O(n\sqrt{w})$. So, in the random case, or in any situation where the average length of the LIS in each window is $o(w)$, our algorithm offers a significant improvement on the naive one.

This improvement is obtained largely through the judicious use of a particular data structure. This data structure implicitly represents infomration pertinent to determining the LIS's of the current window and to determining the LIS's of all suffixes of the current window. This information can then be used to update the structure each time we drop an element off the beginning of the window and add one to the end. As with most algorithms concerned with aspects of the LIS problem, our starting point will be the original constructions of Robinson and Schensted, so it will be helpful to review these next.

2.3 Tableaux and the Robinson–Schensted–Knuth Algorithm

The Robinson–Schensted–Knuth algorithm (see [15] and references therein; for background see also Knuth [11] or Sagan [13]) is based on the concept of a tableau which can be used to determine increasing subsequences of a permutation. More formally,

Definition 1. *A tableau of shape $\lambda = \lambda_1, \lambda_2, \ldots, \lambda_m$ where $\lambda_1 + \lambda_2 + \ldots \lambda_m = n$ is a collection of n elements arranged in left-justified rows such that row i has λ_i elements, and the elements increase weakly across rows and increase strictly down columns.*

See Figure 1 for an example of a tableau.

Although we concern ourselves mostly with permutations, we will discuss the Robinson–Schensted–Knuth algorithm in its full generality as applied to sequences of possibly repeated elements that come from a linearly ordered set. The algorithm we introduce in this paper uses a generalization of the Robinson–Schensted–Knuth algorithm and, in particular, uses the same "bumping" rules as Robinson–Schensted–Knuth.

Given a sequence $\alpha = \alpha_1, \alpha_2, \ldots, \alpha_n$, the Robinson–Schensted–Knuth algorithm constructs a pair of tableaux P and Q both of shape λ for λ some partition of n. We describe here just the construction of P as that includes the bumping technique we use. Given α, elements $\alpha_1, \alpha_2, \ldots, \alpha_n$ are inserted one at a time in that order to form P. At step 1 place a single element, α_1, as the first element of the first row of P. At step i, place α_i using the following algorithm: Scan the first row of P from left to right to locate the smallest element t that is greater than α_i. If no such element t exists, place α_i at the end of the first row of P. If t does exist, remove t from the first row of P and put α_i in its place. We say α_i *bumps* t. Then scan the second row of P from left to right to locate the smallest element in the second row of P that is greater than t. If no such element exists, place t at the end of the second row of P. If t does bump an element, insert that element into the third row of P and continue bumping elements until the currently bumped element comes to rest at the end of a row in P. Continue until all elements of α are exhausted.

3	35	25	257	247	2478	1478	1468	
		3	3	35	35	25	257	= P
						3	3	

Fig. 1. Tableau P created by Robinson–Schensted–Knuth Algorithm for $\pi = 35274816$.

The length of the first row of tableau P is equal to the length of the LIS for π. This sequence can be determined via Schensted's basic subsequences. Schensted [14] defined the ith basic subsequence to be the sequence of elements that had occupied the ith position in the first row of P. It is easy to see that the basic subsequences are decreasing and that each element belongs to exactly one basic subsequence. Any longest increasing subsequence includes exactly one element from each basic subsequence and an increasing subsequence can be determined by associating each element a with the element b to its left when it entered the first row of P. This result shows the significance of the first row of the Robinson–

Schensted–Knuth construction and indeed in our algorithm we make use of the first row only and discard the rest.

3 Algorithm

In order to deal with the problem of determining the longest increasing subsequences in the windows of a permutation we first consider a data structure which addresses a slightly more general question. In this structure we maintain information about the LIS of a sequence in such a way that we can

- Remove the first element of the sequence,
- Add an element to the end of the sequence,
- Query the data structure for the length of the current LIS.

For a given sequence $\alpha = \alpha_1\alpha_2\cdots\alpha_n$ be the initial sequence let $\alpha_i^j = \alpha_i\alpha_{i+1}\cdots\alpha_j$ denote the subsequence from the i-th to the j-th element. We apply Robinson–Schensted–Knuth to α but keep track of only the first row in the tableau. We call this row the *principal row* of α and denote it by $P(\alpha)$. Our data structure will maintain principal rows for all the suffixes of the current sequence α; that is, all the rows $P(\alpha_1^n), P(\alpha_2^n), \ldots, P(\alpha_n^n)$. It will be helpful to think of these rows as lying one above the other in a *row tower*. See Figure 2.

Now consider this data structure applied to the LISW problem, beginning with the subseqnece $\pi(1)\pi(2)\cdots\pi(w)$ of a permutation π of length n.

The removal operation is easy: to remove the first element we need only delete the first row of the row tower. Adding a new element corresponds to inserting it using a Robinson–Schensted–Knuth approach in each of the rows stored so far and creating a new row consisting of this element only. The length of the LIS of the current window is the length of the first principal row we store.

A naive implementation of this data structure using a van Emde Boas priority queue for each row takes $O(1)$ time for expiring, $O(w \log\log n)$ time for adding each element and $O(1)$ time for outputting the length of each subsequence. Total time complexity would be $O(nw \log\log n)$.

However, observe that each row other than the first in the row tower is either the same as the one above, or can be obtained from it by deleting a single element. This claim is easily verified by induction. In the trivial case this claim holds when the first element is added, since there is only a single row in this case. Now consider two consecutive rows before insertion of a new element b. If they are the same then they will remain the same after inserting b. Alternatively if they differ in a single element r, then if b does not bump r from the first row, they will still differ in the same way. If b does bump r, then either it bumps the next element of the second row, or is added to the end of that row. In the first case the two rows still differ by one deletion (the next element after r), while in the second case they are now the same. Thus we have proven:

Lemma 1. *Let sequence S be a suffix of sequence T. Then $P(S)$ is a subsequence of $P(T)$ and $|P(T)| - |P(S)| \leq |T| - |S|$.*

Since we now know that the row tower forms an inclusion chain we can remove duplicate rows and record the original multiplicity of remaining rows in a sequence m. From now on, when we refer to the row tower, we will assume that the rows have been made distinct in this way. After this modification the data structure still supports all the operations as described above, but the time complexity for adding is $O(\ell \log\log n)$ and space is $O(n\ell)$ at this time, where ℓ denotes the length of the current LIS.

Suppose that the first row of the row tower contains ℓ symbols. Then, to each position in this row, we associate the number of the last row in which this symbol occurs. Since each row differs from the preceding one by the removal of exactly one symbol, this gives a permutation σ on the elements $1, 2, \ldots \ell$. We call σ the *drop out permutation* of the row tower. We can also define a *drop out sequence*, d, of drop out times, by replacing each element of the drop out permutation by the actual index of the last row in which the corresponding element of the principal row occurs.

Row Towers:

3 →	35 →	25 →	257 →	247 →	2478 →	1478 →	1468 = R_1
	5	2	27	24	248	148	146 = R_2
		2	27	24	248	148	146 = R_3
			7	4	48	18	16 = R_4
				4	48	18	16 = R_5
					8	1	16 = R_6
						1	16 = R_7
							6 = R_8

Fig. 2. Row towers for $\pi = 35274816$.

For the example in Figure 2, we have $m = (1, 2, 4, 1)$, $d = (7, 3, 8, 1)$ and $\sigma = (3, 2, 4, 1)$. Figure 3 illustrates the transformation of the row tower as the window slides.

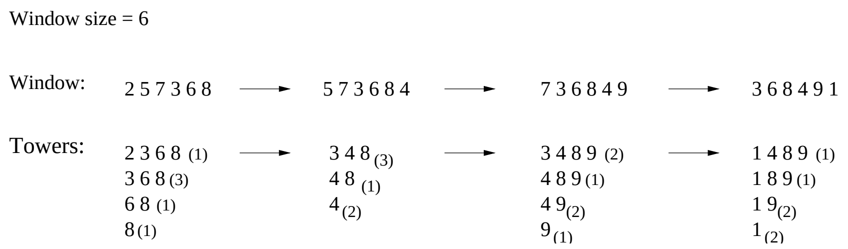

Fig. 3. Construction for $\pi = 257368491$.

We use the values of the principal row, d, and σ, as an implicit representation of all but the first row in the row tower. That is, this data structure has three components, the principal row $R = R_1$, the drop out sequence d, and the drop out permutation σ. Although it is clear that the first two of these suffice to describe the complete row tower, we will make use of the third when we wish to produce actual LIS's from each window, rather than simply the length of the LIS in each window. Next we describe how to update these parts under expire and add operations.

The expire operation simply subtracts 1 from each element of d and deletes the element with expiry time 0 (if there is one) from R. If no deletion occurs then σ is unchanged. Otherwise, the element 1 is deleted from σ and the remaining values are decreased by 1.

The add operation for an element b requires that b be used to bump an element out of each row of the row tower (unless it is appended to all of them). Since, as we have observed, the rows form an inclusion chain, if b bumps a certain element s out of a row, then it bumps the element s out of all further rows to which s belongs. In other words, the drop out time for s changes to the index of the first row from which it is bumped by b. Now consider the next row of the row tower (if one exists) after s has dropped out. In this row there may or may not be elements larger than s. If there are such elements then b bumps the smallest of them. If not, then b is appended to the end of this and all subsequent rows.

So there is a sequence of indices $i_1 < i_2 < \cdots < i_k$ for the sequence d defined as follows: i_1 is the least index of an element of the principal row which is larger than b (if no such index exists, then the sequence is empty), and i_{t+1} is the least index larger than i_t for which $d(i_{t+1}) > d(i_t)$. These indices represent the elements of the principal row which are bumped by b. Since b is placed in position i_1 in the first row and does not drop out until the very end, the sequence d is updated according to:

$$\begin{aligned} d(i_{t+1}) &= d(i_t) \text{ for } t = 1, 2, \ldots, k-1 \\ d(i_1) &= w + 1. \end{aligned}$$

Similarly, the update of σ is:

$$\begin{aligned} \sigma(i_{t+1}) &= \sigma(i_t) \text{ for } t = 1, 2, \ldots, k-1 \\ \sigma(i_1) &= \ell. \end{aligned}$$

At this point, we have a data structure with expire/add time $O(\ell)$ per element, query time (for the length of the LIS in the current window) $O(1)$ and space $O(n)$.

In order to support the operation of outputting an LIS we maintain a tree for each row R_i. In the tree associated to R_1 the paths from vertices to the root will constitute reversals of (some) increasing sequences in the current window. In particular, the path from the last element of R_1 to the root will be an LIS.

The reason for including multiple trees is to allow for the expiry operation. At the point where a principal row expires, it will be necessary to have access to the tree for the new principal row. The basic idea is simply that whenever an element is added to a row it is also added to the tree corresponding to that row and its parent in the tree is the element of the row immediately to its left. The property claimed of paths from vertices to the root then follows immediately.

However, the difficulty with this approach is that all but the first row have implicit representations, and hence looking up the predecessor of an element in each row as required above is a non-trivial operation. We overcome this difficulty by noting that each parent operation takes us one column to the left in some row tower. This row tower is not necessarily the current one, since the element whose parent we seek may already have been bumped from the current row tower, as happens for instance when we look for the LIS in 1342, the element 3 which is 4's parent, no longer occurs in the row tower after 2 arrives. Suppose that we have a value v and a column c that v occupies in some row tower. When v is first added to the row tower, there is a unique row in which it occupies column c. We set the parent of v in column c to be the predecessor of v in that row. In other words, at the time that v is added we establish an array whose entry in position c is the parent of v in column $c-1$. This can easily be accomplished from the explicit information available.

Namely, when v is first added, it is added in say column C. Its predecessor in that column is its immediate predecessor, say p_1 in the principal row. This remains its predecessor in columns $C-1$ through $C-\sigma(p_1)+1$. In column $C-\sigma(p_1)$ its parent will be the right most element p_2 of the principal row which satisfies $\sigma(p_2) > \sigma(p_1)$, and this will remain its parent through column $C-\sigma(p_2)+1$. Thus, by scanning leftwards along the principal row we can create references to all the parents of v in each column. When we exhaust the elements to the left of v (that is, thinking in terms of the row tower, when we reach the final block of rows of which v is the initial element) the parent of v is simply set to the root element of the tree.

Now, we can construct the reversal of the LIS in a given window in constant time per element. Namely, we begin with the rightmost element of the principal row (column ℓ). Using the array associated with this element we determine its parent in column $\ell-1$, the second (last) element of the LIS. In turn using the array associated with that element we find its parent in column $\ell-2$, and so on.

Hence the data structure proposed computes longest subsequences on a sliding window, with a cost for the ith window of $O(\ell_i)$ where ℓ_i is the length of the longest increasing sequence in window i. Thus, the total time is given by $\sum_{i=0}^{n-w} \ell_i = \textsc{output}$, plus the cost of initializing the structure, namely $O(n \log\log n)$.

Theorem 1. *The algorithm described above computes the $n-w+1$ longest increasing sequences, one for each window in total time $O(n \log \log n$ $+\textsc{output})$.*

As an interesting side benefit, the algorithm obtained computes the LISW in an on-line fashion.

4 Conclusions and Open Problems

We proposed a data structure for finding the longest increasing subsequence in a sliding window over a given sequence (LISW). The data structure uses an implicit representation of principal rows for each of the subsequences on a window, and results in an output-sensitive algorithm. This data structure substantially improves over the naive generalization of the longest increasing subsequence algorithm. An on-line, output-sensitive optimal algorithm is derived from this data structure. The time complexity is $O(n \log\log n + \text{OUTPUT})$.

Other variations of the problem remain open, in particular the exact time complexity of the global max sequence problem remains an open question. Another interesting case is the off-line case, in which a pre-processing step in $o(n \log\log n)$ time is allowed. Then a query is issued for the longest subsequence within a given window which must be answered in time $o(w \log\log n)$.

Acknowledgments We wish to thank the participants of the Algorithmic Problem Session at the University of Waterloo for many helpful discussions.

References

1. 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.
2. S. Bespamyatnikh and M. Segal, Enumerating longest increasing subsequences and patience sorting, *Info. Proc. Lett.* 76 (2000), 7–11.
3. M.-S. Chang and F.-H. Wang, Efficient algorithms for the maximum weight clique and maximum weight independent set problems on permtuation graphs, *Info. Proc. Lett.* 43 (1992), 293–295.
4. A.L. Delcher, S. Kasif, R.D. Feischmann, J. Peterson, O. White, S.L. Salzberg, Alignment of whole genomes, *Nucleic Acids Res.* 27 (1999), 2369-2376.
5. P.B. Gibbons and S. Tirthapura, Distributed Streams Algorithms for Sliding Windows, *In Proc. 14th ACM Symp. on Parallel Algs. and Architectures (SPAA)*, 2002.
6. M. Datar, A. Gionis, P. Indyk and R. Motwani, Maintaining Stream Statistics over Sliding Windows, *In ACM-SIAM Symp. on Discrete Algorithms(SODA)*, 2002.
7. P. Groeneboom, Hydrodynamical methods for analyzing longest increasing subsequences, *J. of Comp. and Appl. Math.* 142 (2002), 83–105.
8. M. Hamermesh, *Group Theory and its Application to Physical Problems*, New York: Dover, 1962.
9. J. Hunt and T. Szymanski, A fast algorithm for computing longest common subsequences, *Comm. ACM* 20 (1977), 350-353.
10. D.E. Knuth, Permutations, matrices, and generalized Young tableaux, *Pacific J. Math.* 34 (1970), 709–727.
11. D.E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Reading, Mass.: Addison–Wesley, 1973.
12. G. de B. Robinson, On representations of the symmetric group, *Amer. J. Math.* 60 (1938),745–760.
13. B. Sagan, *The Symmetric Group*, Pacific Grove, Calf.: Wadsworth and Brooks/Cole, 1991.
14. C. Schensted, Longest increasing and decreasing subsequences, *Can. J. Math.* 13 (1961), 179–191.
15. M. Van Leeuwen, The Robinson–Schensted and Schüzenberger algorithms, an elementary approach, *Elect. J. Comb.* Foata Festschrift, Vol. 3, no. 2 (1996) R15.