Information Processing Letters 110 (2010) 655–658

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

The longest almost-increasing subsequence

Amr Elmasry [1]

Max-Planck Institut für Informatik, Saarbrücken, Germany

ARTICLE INFO

Article history:
Received 11 February 2010
Received in revised form 24 May 2010
Accepted 25 May 2010
Available online 1 June 2010
Communicated by M. Yamashita

Keywords:
Algorithms
Algorithm design and analysis
Data structures

ABSTRACT

Given a sequence of n elements, we introduce the notion of an almost-increasing subsequence as the longest subsequence that can be converted to an increasing subsequence by possibly adding a value, that is at most a fixed constant, to each of the elements. We show how to optimally construct such subsequence in $O(n \log k)$ time, where k is the length of the output subsequence.

1. Introduction

The longest increasing subsequence (LIS) is a subsequence of maximum length where every element is greater than the previous element. The longest increasing subsequence problem refers to either producing the subsequence or just finding its length. The problem was first tackled by Robinson [16] seventy years ago. The classical dynamic-programming algorithm for the problem, which appears in many algorithmic textbooks [9], is due to Schensted [17]. This algorithm runs in $O(n \log k)$, where n is the length of the input sequence, and k is the length of the longest increasing subsequence. Knuth [14] gave generalizations to the problem with relations to Young tableaux. Fredman [12] showed that $O(n \log n)$ comparisons are both necessary and sufficient, to find the length or produce the subsequence, in the comparison-tree model. The same lower bound was also proven for the algebraic decision-tree model [15]. If the input sequence is a permutation of the integers 1 to n, algorithms were introduced to construct the longest increasing subsequence in $O(n \log \log n)$ time [8,13].

The problem is important in practice. Several other problems involve a LIS construction (see, for example, [5]). It has lately gained even more practical importance as it is used in the MUMmer system [10] for aligning genomes.

A related problem is the longest common subsequence (LCS) problem, which considers two sequences and locates a series of entries that appear in the same order in both sequences. Note that we can apply the LCS algorithms to a sequence and its sorted outcome to get a longest increasing subsequence.

Several variants of the LIS problem have been introduced. The longest increasing subsequence of a circular list (LICS) assumes the input sequence to be circular. A randomized algorithm for the LICS that runs in expected $O(n^{3/2} \log n)$ time is given in [3]. The best worst-case bound known is $O(n^2)$, and can be achieved using techniques from [4]. Another variant is to find the longest increasing subsequences that lie in all the sliding windows with a specified width. An algorithm that runs in time proportional to the size of the output subsequences plus an additive bound for constructing one LIS is given in [4]. A generalization of the LIS problem is discussed in [2], where a fixed set of permutations is given and the task is to compute, for a given input sequence, the longest subsequence that is order isomorphic to one of the given permutations. The LIS problem applies when such permutations are the identity permutations.

The combinatorics of the problem are of no less importance. Starting with the work of Erdös and Szekeres, the

E-mail address: elmasry@mpi-inf.mpg.de.

[1] Supported by an Alexander von Humboldt Fellowship.

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2010.05.022

length of a LIS in a random permutation was investigated. The complete limiting distribution of the length of the LIS of a permutation of length n chosen uniformly at random is given in [6]. The expected length of a LIS is shown to be close to $2\sqrt{n}$.

Suppose one is considering the process of monitoring the performance of an activity. We say that the activity is well performing once it is well performing in comparison with a large number of accredited historical snapshots where it was as well performing when deploying the same criteria. Picking the largest number of points when the activity is strictly performing better among such previously selected points is too restricted and unfair. The notion needs to be relaxed to reflect a good progress without necessarily being the best selected so far. On another front, there are applications where the data items have a small amount of noise. In accordance, a relaxed version of the LIS problem is needed.

In this paper, we introduce a variant of the LIS problem that we call the longest almost-increasing subsequence (LaIS) problem. We allow a drop of at most a constant value from the maximum element that appeared so far. Given a sequence $\langle x_1, x_2, \ldots, x_n\rangle$ and a constant $c > 0$, our goal is to construct a longest subsequence $\langle y_1, y_2, \ldots, y_k\rangle$ such that $\forall_i\ y_i > \max_{j=1}^{i-1} y_j - c$. We give an asymptotically optimal algorithm that runs in $O(n \log k)$ time. The main idea is to apply the dynamic programming paradigm to a search-utilizing pointer-based data structure and not to an array.

2. Recursive formulation

Let *LaIS*(h, i), $h \leqslant i$, denote a longest almost-increasing subsequence among the elements of $\langle x_1, \ldots, x_i\rangle$, such that a largest element is x_h and h is minimal. We show that these two parameters fully characterize any *LaIS*, and recursively express *LaIS* in terms of solutions to smaller problems.

The key observation is that any *LaIS*(h, i), $h < i$, can be split into two independent (except for the value of h) subsequences, following the relation:

$$\mathit{LaIS}(h, i) = \mathit{LaIS}(h, h) \cdot T(h, i),$$

where "·" is the concatenation operation, and $T(h, i)$ is the subsequence including every element x_j among $\langle x_{h+1}, \ldots, x_i\rangle$ satisfying $x_h - c < x_j \leqslant x_h$.

The second observation is that *LaIS*(h, h) can be expressed as:

$$\mathit{LaIS}(h, h) = \mathit{LaIS}\big(i', h-1\big) \cdot \langle x_h\rangle,$$

where *length*(*LaIS*$(i', h-1)$) has the maximum value among $i' < h$ satisfying $x_{i'} < x_h$. Note that i' is not necessarily unique.

3. The basic algorithm

The algorithm proceeds in n iterations. After the i-th iteration, the algorithm maintains for each element x_j, among the first i elements, a longest almost-increasing subsequence whose largest element is x_j and j is minimal. For each such subsequence, it is enough to keep track of its length l_j and the minimal index p_j of a largest element among the elements preceding x_j. During the i-th iteration, two tasks are performed: The first task is to find a longest almost-increasing subsequence whose largest element is x_i. This subsequence is constructed by appending x_i to the longest subsequence found so far whose largest element is smaller than x_i. More formally, we look for an index $i' < i$ such that $l_{i'} \geqslant l_j$ among all indexes $j < i$ having $x_j < x_i$. The length l_i is then set to $l_{i'} + 1$, and the index p_i is set to i'. The second task is to append x_i to every subsequence found so far whose largest element is larger or equal to x_i and smaller than $x_i + c$. More formally, for all $j < i$, set l_j to $l_j + 1$ if $x_i \leqslant x_j < x_i + c$. After the n-th iteration, the length of the LaIS is the maximum length l_m among the l_i's stored by the algorithm. To construct a LaIS, we make use of the p_i's to produce the subsequence in reverse order. Using the element x_m corresponding to the maximum length l_m, scan every element x_i, from $i = n$ to $m + 1$, and output the elements satisfying $x_m - c < x_i \leqslant x_m$ followed by x_m. Let x_t be the last element of the subsequence output in the reverse order, scan every element x_i from $i = t - 1$ to $p_t + 1$, and output the elements satisfying $x_{p_t} - c < x_i \leqslant x_{p_t}$ followed by x_{p_t}. The previous step is repeated until we get a value of p_t that indicates the first element of the subsequence (when, for example, $p_t = t$).

A straightforward implementation of the previous algorithm would use two linked lists (or two arrays) each of size n; one for the l_i's and another for the p_i's. This implementation runs in $O(n^2)$ time.

Example

Assume $c = 2$, and consider the following sequence indexed from 1 to 12: $\langle 7, 15, 2, 14, 14, 6, 8, 11, 17, 15, 14, 13\rangle$. We show in Table 1 the two arrays: one holding the lengths l_i's (first row), and another holding the indexes p_i's (second row), after each of the 12 iterations performed by the algorithm.

It follows that the LaIS is of length 6. There are five such subsequences: $\langle 7, 15, 14, 14, 15, 14\rangle$, $\langle 7, 6, 8, 11, 15, 14\rangle$, $\langle 2, 6, 8, 11, 15, 14\rangle$, $\langle 7, 6, 8, 11, 14, 13\rangle$ and $\langle 2, 6, 8, 11, 14, 13\rangle$. Typically, the algorithm stores one p_i and reports one LaIS.

4. The improved algorithm

Instead of storing one l_i corresponding to each x_i, we store one element for every length value. Namely, after the i-th iteration, we store $z_l \leftarrow x_h$ corresponding to length l, where $\mathit{length}(\mathit{LaIS}(h, i)) = l$ and $x_h \leqslant x_{h'}$ for all h' satisfying $\mathit{length}(\mathit{LaIS}(h', i)) = l' \geqslant l$. It follows that $z_j \leqslant z_{j'}$ for $j < j'$. To show that these elements are enough to construct a LaIS, consider two elements $x_h > x_{h'}$ where $\mathit{length}(\mathit{LaIS}(h', i)) \geqslant \mathit{length}(\mathit{LaIS}(h, i))$. Given any almost-increasing subsequence having *LaIS*(h, i) as a prefix subsequence, we can replace *LaIS*(h, i) with *LaIS*(h', i) and get another almost-increasing subsequence with at least the same length.

Table 1
Dry run of the basic algorithm.

Iteration	**1**											
1	**1**											
Iteration	1	**2**										
2	1	**1**										
Iteration	1	2	**1**									
3	1	1	**3**									
Iteration	1	**3**	1	**2**								
4	1	1	3	**1,3**								
Iteration	1	**4**	1	**3**	**2**							
5	1	1	3	1,3	**1,3**							
Iteration	**2**	4	1	3	2	**2**						
6	1	1	3	1,3	1,3	**3**						
Iteration	2	4	1	3	2	2	**3**					
7	1	1	3	1,3	1,3	3	**1,6**					
Iteration	2	4	1	3	2	2	3	**4**				
8	1	1	3	1,3	1,3	3	1,6	**7**				
Iteration	2	4	1	3	2	2	3	4	**5**			
9	1	1	3	1,3	1,3	3	1,6	7	**2,8**			
Iteration	2	**5**	1	3	2	2	3	4	5	**5**		
10	1	1	3	1,3	1,3	3	1,6	7	2,8	**8**		
Iteration	2	**6**	1	**4**	**3**	2	3	4	5	**6**	**5**	
11	1	1	3	1,3	1,3	3	1,6	7	2,8	8	**8**	
Iteration	2	6	1	**5**	**4**	2	3	4	5	6	**6**	**5**
12	1	1	3	1,3	1,3	3	1,6	7	2,8	8	8	**8**

Table 2
Dry run of the improved algorithm.

(a) The array of z_i's

Iteration 1	7					
Iteration 2	7	15				
Iteration 3	2	15				
Iteration 4	2	14	15			
Iteration 5	2	14	14	15		
Iteration 6	2	6	14	15		
Iteration 7	2	6	8	15		
Iteration 8	2	6	8	11		
Iteration 9	2	6	8	11	17	
Iteration 10	2	6	8	11	15	
Iteration 11	2	6	8	11	14	15
Iteration 12	2	6	8	11	13	14

(b) The array of p_i's

Iteration 1	1											
Iteration 2	1	1										
Iteration 3	1	1	3									
Iteration 4	1	1	3	3								
Iteration 5	1	1	3	3	3							
Iteration 6	1	1	3	3	3	3						
Iteration 7	1	1	3	3	3	3	6					
Iteration 8	1	1	3	3	3	3	6	7				
Iteration 9	1	1	3	3	3	3	6	7	8			
Iteration 10	1	1	3	3	3	3	6	7	8	8		
Iteration 11	1	1	3	3	3	3	6	7	8	8	8	
Iteration 12	1	1	3	3	3	3	6	7	8	8	8	8

In the sequel, we call the second task of the i-th iteration of the basic algorithm, which involves incrementing the length of every subsequence whose largest element x_j satisfies $x_i \leqslant x_j < x_i + c$, a *length-shift*. Apart from the length-shift, the improved algorithm is quite similar to that of constructing the longest increasing subsequence [17]. If we use an array to store the sequence $\langle z_1, z_2, \ldots \rangle$, mimicking the algorithm in [17], we will be able to use binary search to locate the predecessor of x_i and that of $x_i + c$ in $O(\log k)$ time. But, a length-shift, which is now performed by sequentially overwriting every z_j in a specified range using $z_{j+1} \leftarrow z_j$, would require $O(k)$ time. This results in an implementation that runs in $O(nk)$ time; still not the claimed bound.

Example

Back to the example of Section 3, where $c = 2$ and the input sequence is $\langle 7, 15, 2, 14, 14, 6, 8, 11, 17, 15, 14, 13 \rangle$. We show in Table 2 the two arrays: one holding the z_i's (part a), and another holding the indexes p_i's (part b), after each of the 12 iterations performed by the algorithm.

The LaIS reported by this version of the algorithm will be $\langle 2, 6, 8, 11, 14, 13 \rangle$.

5. The data structure

In order to achieve the claimed $O(n \log k)$ bound, the inner loop must be executed in $O(\log k)$ time. To efficiently implement a length-shift, we store the z_j's as an ordered linked list, where a node holding z_j points to a node holding its successor element z_{j+1}. During each iteration, a search is performed for the predecessor of x_i. A node that contains x_i is then inserted after this position.

To perform a length-shift, the corresponding range of nodes is determined and the successor of the last node in the range is deleted (if such a node exists). To construct the subsequence, we still maintain an array for the p_i's. This array is modified once per iteration and later used to produce the output.

In summary, at each iteration the algorithm performs: two predecessor searches, one successor finding, an insertion and a possible deletion.

Algorithm 1 The pseudo-code for the LaIS algorithm.

```
for i = 1 to n do
    v ← new_node()
    v.value ← x_i
    v.index ← i
    pred ← predecessor(x_i)
    if (pred ≠ null) then
        p_i ← pred.index
    else
        p_i ← i
    end if
    insert(v)
    s ← successor(predecessor(x_i + c))
    if (s ≠ null) then
        delete(s)
    end if
end for
m ← tail_node().index
for i = n down to m + 1 do
    if (x_m − c < x_i ⩽ x_m) then
        print x_i
    end if
end for
print x_m
t ← m
while (p_t ≠ t) do
    for i = t − 1 down to p_t + 1 do
        if (x_{p_t} − c < x_i ⩽ x_{p_t}) then
            print x_i
        end if
    end for
    print x_{p_t}
    t ← p_t
end while
```

Each of these operations can be executed in $O(\log k)$ time when building a balanced search structure over the nodes of the linked list. We may use, for example, an AVL tree [1] to achieve $O(\log k)$ cost per operation, or a splay tree [18] for an amortized $O(\log k)$ cost per operation. For practical purposes, it would be better to keep a linked list of pointers to find the successor of a given node in constant time, accounting for at least three pointers per node in case using the aforementioned structures. However, from the practical point of view, the best search structure for this application is the jumplist [11]. Using a jumplist: a predecessor search, an insertion and a deletion require $O(\log k)$ amortized time ($O(\log k)$ expected time [7]), finding the successor requires constant time, and we only use and maintain two pointers per node.

Acknowledgements

I would like to thank Rajiv Raman and Saurabh Ray for introducing the problem and for several useful discussions.

References

[1] G. Adelson-Velskii, E. Landis, On an information organization algorithm, Doklady Akademii Nauk SSSR 146 (1962) 263–266.
[2] M. Albert, R. Aldred, M. Atkinson, H. Ditmarsch, B. Handley, C. Handley, J. Opatrny, Longest subsequences in permutations, Australian Journal of Combinatorics 28 (2003) 225–238.
[3] M. Albert, M. Atkinson, D. Nussbaum, J. Sack, N. Santoro, On the longest increasing subsequence of a circular list, Information Processing Letters 101 (2007) 55–59.
[4] M. Albert, A. Golnski, A. Hamel, A. Lopez-Ortiz, S. Rao, M. Safari, Longest increasing subsequences in sliding windows, Theoretical Computer Science 321 (2004) 405–414.
[5] A. Apostolico, M. Atallah, S. Hambrusch, New clique and independent set algorithms for circle graphs, Discrete Applied Mathematics 36 (1992) 1–24.
[6] J. Baik, P. Deift, K. Johannsson, On the distribution of the length of the longest increasing subsequence of random permutations, Journal of the American Mathematical Society 12 (1999) 1119–1178.
[7] H. Brönnimann, F. Cazals, M. Durand, Randomized jumplists: A jump-and-walk dictionary data structure, in: 20th Symposium on Theoretical Aspects of Computer Science, in: LNCS, vol. 2607, 2003, pp. 283–294.
[8] M. Chang, F. Wang, Efficient algorithms for the maximum weight clique and maximum weight independent set problems on permutation graphs, Information Processing Letters 76 (2000) 7–11.
[9] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms, 2nd ed., The MIT Press, Cambridge, 2001.
[10] A. Delcher, S. Kasif, R. Fleischmann, J. Paterson, O. White, S. Salzberg, Alignment of whole genomes, Nucleic Acids Research 27 (1999) 2369–2376.
[11] A. Elmasry, Deterministic jumplists, Nordic Journal of Computing 12 (1) (2005) 27–39.
[12] M. Fredman, On computing the length of longest increasing subsequence, Discrete Mathematics 11 (1975) 29–35.
[13] J. Hunt, T. Szymanski, A fast algorithm for computing longest common subsequences, Communications of the ACM 20 (1977) 350–353.
[14] D. Knuth, Permutations, matrices, and generalized Young tableaux, Pacific Journal of Mathematics 34 (1970) 709–727.
[15] P. Ramanan, Tight $\Omega(n \log n)$ lower bound for finding a longest increasing subsequence, International Journal of Computer Mathematics 65 (3–4) (1997) 161–164.
[16] G. Robinson, On representations of the symmetric group, American Journal of Mathematics 60 (1938) 745–760.
[17] C. Schensted, Longest increasing and decreasing subsequences, Canadian Journal of Mathematics 13 (1961) 179–191.
[18] D. Sleator, R. Tarjan, Self-adjusting binary search trees, Journal of the ACM 32 (3) (1985) 652–686.