Computing a Longest Increasing Subsequence of Length k in Time $O(n \log \log k)$

MAXIME CROCHEMORE
King's College London, Strand, London WC2R 2LS, UK
and Université Paris-Est. Maxime.Crochemore@kcl.ac.uk
ELY PORAT
Bar-Ilan University, Ramat-Gan 52900, Israel
PoratEly@cs.biu.ac.il

Abstract

We consider the complexity of computing a longest increasing subsequence parameterised by the length of the output. Namely, we show that the maximal length k of an increasing subsequence of a permutation of the set of integers $\{1, 2, \ldots, n\}$ can be computed in time $O(n \log \log k)$ in the RAM model, improving the previous 30-year bound of $O(n \log \log n)$. The optimality of the new bound is an open question.

Keywords: Design and analysis of algorithms, Longest increasing subsequence, Data structures, Priority queue

1. LONGEST INCREASING SUBSEQUENCE

We consider the problem of extracting a longest increasing subsequence (LIS) from a sequence of integers. The sequence S is assumed to be a permutation of the set $\{1, 2, \ldots, n\}$, but having multiple occurrences of integers between 1 and n in the sequence of length n does not change the result (see Section 4).

The question is related to the representation of permutations, elements of the symmetric group on $\{1, 2, \ldots, n\}$, with Young tableaux. This is certainly why it has attracted a lot of attention. See the chapter by Lascoux, Leclerc, and Thibon in [8, Chapter 5] for a presentation of Schensted's algorithm [12] in this context.

The question is also related to the computation of a longest common subsequence (LCS) of two strings, and to their alignment, in at least two ways. First, the LIS of S is the LCS between S and the sequence $(1, 2, \ldots, n)$. This remark leads to an $O(n^2)$ running time algorithm implementing the standard dynamic programming technique used for finding longest common subsequences (it can indeed be reduced to $O(n^2/\log n)$ [9, 2]). Though simple, this solution cannot compete against direct computations. Second, the LIS question is involved in the solution to whole-genome comparisons proposed by Delcher et al [3] and in its subsequent variants. A comparison is based on maximal exact matches between the two input genome sequences, matches that are additionally constrained to occur only once in each sequence. An LIS is used to extract a long subsequence of matches that are compatible between each other, i.e. they appear in the same order along the two sequences, for producing an alignment of the complete genomes.

There is an extensive literature about the distribution of the maximal length of increasing subsequences in random permutations. As a notable result, it is known that the expected LIS length is $\sim 2\sqrt{n}$ as $n \to \infty$. See [13], or the survey by Odlyzko and Rains [11] who discuss these questions, and references therein.

Liben-Nowell et al. [7] explore the LIS problem in the streaming model which typically aims at reducing to a polylogarithmic amount the memory space required by the computation, in addition to efficient running time (see also [10]).

A direct solution to computing a longest increasing subsequence, running in $O(n \log n)$ time, was proposed by Fredman [4]. The solution is optimal if the elements are drawn from an arbitrary set due to the $\Omega(n \log n)$ lower bound for sorting n elements. Parameterised by the LIS length k, the running time is $O(n \log k)$. On integer alphabets, the fastest known solution runs in $O(n \log \log n)$ time (see [16] and references therein). It relies on the priority search trees of van Emde Boas [14, 15], which provide $O(\log \log n)$ amortised time per operation when keys are drawn from the set $\{1, 2, \ldots, n\}$.

The solution presented in this paper breaks the long-standing $O(n \log \log n)$ upper bound down to $O(n \log \log k)$, where k is the maximal length of increasing subsequences. It extends immediately to the computation of a longest increasing subsequence (not only its length). We assume the RAM model for evaluating the running time and the algorithm can be viewed as a parameterised solution (k is the length of the output). This is certainly a result mostly of theoretical nature but it opens the road to a possible linear-time LIS computation.

To get the $O(n \log \log k)$ bound, we feed the priority queue used in the standard algorithm with elements drawn from a restricted range. This is done through a series of careful renamings of the elements. Downsizing the key universe to size $O(k)$ leads the priority queue to work in amortised time $O(\log \log k)$ and yields the announced result. But the length k of a longest increasing subsequence is not an input of the algorithm: we show that an approximation of it is enough, and how to compute such an approximation.

Section 2 recalls the core algorithm for computing a longest increasing subsequence and Section 3 describes our improved solution.

2. CORE ALGORITHM

We recall the core algorithm for computing a longest increasing subsequence, starting with the computation of its length.

Let π be a permutation of $\{1, 2, \ldots, n\}$. The aim is to extract a longest increasing subsequence from the sequence $S = (\pi(1), \pi(2), \ldots, \pi(n))$.

Elements are processed in the order $\pi(1)$, $\pi(2)$, ..., $\pi(n)$. Conceptually we compute, for each length $\ell = 1, 2, ..$, the smallest last element that can end an increasing subsequence of that length. It is called the best element for that length and denoted by $B[\ell]$.

Note that best elements $B[1]$, $B[2]$, ..., form an increasing sequence. This fact is used for the choice of a data structure to implement the list and is essential for getting an efficient computation.

One step in the algorithm is as follows (see [1]). The currently processed element $\pi(i)$ can extend any increasing subsequence having the last element smaller than it. If $\pi(i)$ is larger than all the best elements computed so far for the sequence $(\pi(1), \pi(2), \ldots, \pi(i-1))$, it produces an increasing subsequence longer than any previous one, for which it is the last element. Otherwise, $\pi(i)$ becomes the best element for an existing length: it replaces the smallest element greater than it in B. This leads to the next algorithm to compute the maximal length of increasing subsequences of S, in which B is a priority queue that stores the best elements.

LIS(π, n)
1. $B \longleftarrow ()$; $k \longleftarrow 0$
2. for $i \longleftarrow 1$ to n
3. $\quad x \longleftarrow \pi(i)$
4. $\quad$ Insert(B, x)
5. $\quad$ if $succ(B, x)$ exists

6. Delete$(B, succ(B, x))$
7. else $k \longleftarrow k + 1$
7. return k

Example Let $S_0 = (12, 8, 9, 1, 11, 6, 7, 2, 10, 4, 5, 3)$. The queue B is initially empty. Its contents after processing sequentially each of the elements are successively: (12), (8), $(8, 9)$, $(1, 9)$, $(1, 9, 11)$, $(1, 6, 11)$, $(1, 6, 7)$, $(1, 2, 7)$, $(1, 2, 7, 10)$, $(1, 2, 4, 10)$, $(1, 2, 4, 5)$, $(1, 2, 3, 5)$. The length of an LIS is then 4 the size of B at the end of the run.

Computing a longest increasing subsequence (not just its length) is a simple extension of the algorithm. Instead of storing best elements only in the queue B, it suffices to store pairs of the form (x, y) where y is a best element predecessor of x. Then, tracing back predecessor information from the last best element in B produces a longest increasing subsequence.

Example (continued) The predecessor of 5 when it was inserted in B was 4, that of 4 was 2, that of 2 was 1, which gives the LIS: $(1, 2, 4, 5)$ of $S_0 = (12, 8, 9, 1, 11, 6, 7, 2, 10, 4, 5, 3)$. Considering value 10, which is also a best element for an increasing subsequence of length 4, we get in the same way another LIS: $(1, 6, 7, 10)$.

The running time of the algorithm relies mainly on the implementation of the queue B of best elements. Using an array and binary search (since elements are naturally sorted) to locate the position of the next element x (i.e. to implement the operations Insert, Delete, $prev$, and $succ$) yields a $O(n \log n)$ running time algorithm [4].

Using a more sophisticated priority list implementation in the form of van Emde Boas trees [14, 15], each step can be performed in $O(\log \log n)$ amortised time yielding an overall $O(n \log \log n)$ running time algorithm [6].

In the next section we keep the same algorithm and the same priority list implementation but process the initial sequence differently to get the announced running time.

3. IMPROVEMENT BY RENAMING

In order to compute a longest increasing subsequence, having length k, from the sequence S of length n in time $O(n \log \log k)$ we want a priority queue that works in $O(\log \log k)$ amortised time per operation. Our strategy to get this result is to downsize the key universe of the queue to size $O(k)$. This is done through a series of careful renamings of the elements of the sequence.

We assume that a good approximation m of k, $m \geq k$, is given. We discuss how to find such an m at the end of the section.

The solution splits the initial sequence S into blocks of size m (except of course the last block that can be smaller), and processes each block separately in the order of the sequence. We discuss these two points.

Splitting S into blocks and sorting them. The sequence S is split into blocks, C_j, $j = 1, \ldots, \lceil n/m \rceil$, of consecutive elements:

$$C_j = (\pi((j-1)m + 1),\ \pi((j-1)m + 2),\ \ldots,\ \pi((j-1)m + m)).$$

We also consider sorted blocks: C_j^s is the sorted list of elements of C_j. Sorted and unsorted blocks are kept in memory.

Sorting all the blocks individually by radix sort would take too much time because the elements in a given block are not in a limited range. To sort them all in linear time, we sort them altogether but identify the block of each element. To do so, we associate with each element $\pi(i)$ the pair $(\lceil i/m \rceil, \pi(i))$ composed of its block number and itself. Pairs are then sorted lexicographically using radix sort. And since the first component of each pair identifies its block, we get all the blocks sorted.

The whole procedure runs in time $O(n)$ because the elements and the block numbers are in the set $\{1, 2, \ldots, n\}$.

Processing a block. In the modified algorithm, instead of processing an element x of S as in Lines 4-7 of Algorithm LIS, we deal with a key associated with it. All the elements of a block are treated online. Before going to the next block some work as to be done to assign keys to elements.

When processing a block, each element x is associated a key $y = key(x)$ in a one-to-one correspondence. The inverse function is called elt, then $x = elt(y)$. Keys are in the set $\{1, 2, \ldots, 2m\}$ and are inserted in the queue B.

To assign keys in the designated range we merge elements whose keys are in the queue B with the current sorted block. Note that elements whose keys are in B are in increasing order as already mentioned in Section 2, which is essential for merging. Keys are then ranks of elements in the obtained sorted list. Since we assume $m \geq k$, the number of keys in B is no more than m, the length of the sorted list is no more than $2m$, which implies that keys are in the set $\{1, 2, \ldots, 2m\}$.

After keys are assigned, we update B with the new keys of elements that are conceptually in the queue.

The last step in the treatment of a block is to process all its elements in the order of the block. The key of each element is dealt with as in Algorithm LIS.

The next scheme summarise the processing of a block.

Processing a block
1. merge $(elt(y) \mid y \in B)$ with the next sorted block
2. assign new keys in the order of the list
3. update keys in B correspondingly
4. insert in B keys of elements of the block in the order of the block

Example (continued) We consider $m = 4$ for the improved algorithm, and go on with the example sequence $S_0 = (12, 8, 9, 1, 11, 6, 7, 2, 10, 4, 5, 3)$.

To avoid confusion in the description between elements of S and their keys, these are denoted by letters `a`, `b`, `c`, ...

The three blocks are $C_1 = (12, 8, 9, 1)$, $C_2 = (11, 6, 7, 2)$, $C_3 = (10, 4, 5, 3)$, their sorted versions are $C_1^s = (1, 8, 9, 12)$, $C_2^s = (2, 6, 7, 11)$, $C_3^s = (3, 4, 5, 10)$.

Processing the first block. Keys of 12, 8, 9, 1 are `d`, `b`, `c`, `a` respectively. After processing the key of each element, the contents of queue B are successively: (`d`), (`b`), (`b`, `c`), and (`a`, `c`).

Processing the second block. Queue $B =$ (`a`, `c`) corresponds to the list of elements $(1, 9)$. It is merged with C_2^s producing the list $(1, 2, 6, 7, 9, 11)$. The content of B is update to (`a`, `e`). After processing keys `f`, `c`, `d`, `b` of elements of C_2, the contents of queue B are successively: (`a`, `e`, `f`), (`a`, `c`, `f`), (`a`, `c`, `d`), and (`a`, `b`, `d`).

Processing the third block. Queue $B = (\mathtt{a}, \mathtt{b}, \mathtt{d})$ corresponds to the list of elements $(1, 2, 7)$. It is merged with C_3^s producing the list $(1, 2, 3, 4, 5, 7, 10)$. The content of B is update to $(\mathtt{a}, \mathtt{b}, \mathtt{f})$. After processing keys g, d, e, c of elements of C_3, the contents of queue B are successively: $(\mathtt{a}, \mathtt{b}, \mathtt{f}, \mathtt{g})$, $(\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{g})$, $(\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{e})$, and $(\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{e})$.

The list of elements whose keys are in B is: $(1, 2, 3, 5)$, which give an LIS of length 4 ending with 5. Computing an LIS can be done as explained above.

In the implementation of Algorithm LIS, the cost of all renamings is $O(n)$ if radix sorting is used. Each operation on the queue (Insert, Delete, Update) takes only $O(\log \log m)$ amortised time because the elements in B belong to the set $\{1, 2, \ldots, 2m\}$. This gives the following statement.

Lemma 1 *The implementation of Algorithm LIS with blocks of size m, $m \geq k$, and renamings runs in time $O(n \log \log m)$ for a sequence of length n.*

Finding the size of blocks. In the above presentation an approximation m of the length k of longest increasing subsequences of S satisfying $m \geq k$, is assumed to be given. We discuss now how to find it.

The idea is to try increasing values of m until we get the approximation leading to the announced running time. Starting with some value m_0, expected to be no more than k, for m (for instance, $m_0 = 4$), we consider the sequence $(m_i \mid i \geq 0)$ defined by $m_i = {m_{i-1}}^{\log m_{i-1}}$ for $i > 0$.

For a given value of m in the sequence, we run Algorithm LIS implemented as described above but with this change: the run stops if the size of the queue B becomes larger than m, and the algorithm signals the fact. Therefore, the first time the algorithm does not stop due to this condition is when the value of m is the smallest value in the list that is larger than k. Let m_i be this value.

Doing so, the running time of the modified Algorithm LIS is $O(n \log \log m_j)$ for $0 \leq j < i$ because during all these runs the queue B contains no more than m_j elements that all belong to $\{1, 2, \ldots, 2m_j\}$. For the value m_i the run finishes normally because the condition of its complete execution, $m \geq k$, is met. The running time for this value of m is $O(n \log \log m_i)$.

Noting that $\log \log(m^{\log m}) = 2 \log \log m$, the total running of the whole execution of Algorithm LIS for $m = m_0, m_1, \ldots, m_i$ is

$$O(n(\Sigma_{j=0,\ldots,i} 1/2^{j-1}) \log \log m_i),$$

which is also $O(n \log \log m_i)$, and eventually $O(n \log \log k)$ because $m < k^{\log k}$ implies $\log \log m < 2 \log \log k$.

The conclusion lies in the next statement.

Theorem 2 *Let S be a permutation of the integers $\{1, 2, \ldots, n\}$ and let k be the maximal length of its increasing subsequences. Computing k and extracting a longest increasing subsequence from S can be done in time $O(n \log \log k)$.*

4. CONCLUSION

The result stated in Theorem 2 is valid for sequences of integers with repetitions. Although the problem seems more general its solution comes down to that of a permutation as follows: each element x occurring at position i in S is renamed as the rank of the pair (x, i) in the lexicographically sorted list of all these pairs. This process is yet similar to the renaming by ranking used in our solution.

As to whether the upper bound in Theorem 2 is optimal, and not linear, raises the question of finding a totally different approach to compute longest increasing subsequences because the implementation of Schensted's algorithm is squeezed as much as possible with the present

solution. But this is unlikely to happen if we consider that many researchers have already worked on the problem.

Another possible way for exploring the complexity of the problem is to use other techniques for sorting integers (see for example [5]). But some of them are affiliated to van Emde Boas' method and are mostly designed to avoid the non-linear space coming from the large range of input integers. This is not the problem we have for computing the LIS of a permutation, though the techniques might simplify the solution or give a direct answer to the question.

5. ACKNOWLEDGEMENTS

We are grateful to the anonymous reviewer for very detailed comments and for pointing reference [5] to us.

REFERENCES

[1] S. Bespamyathnikh and M. Segal. Enumerating longest increasing subsequences and patience sorting. *Information Processing Letters*, 76(1-2):7–11, 2000.

[2] M. Crochemore, G. M. Landau, and M. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. *SIAM Journal of Computing*, 32(6):1654–1673, 2003.

[3] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic Acid Research*, 27(11):2369–2376, 1999.

[4] M. L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11:29–35, 1975.

[5] Y. Han. Deterministic sorting in $o(n \log \log n)$ time and linear space. *Journal of Algorithms*, 50:96–105, 2004.

[6] J. Hunt and T. Szymanski. A fast algorithm for computing longest increasing subsequences. *Communications of ACM*, 20:350–353, 1977.

[7] D. Liben-Nowell, E. Vee, and A. Zhu. Finding longest increasing and common subsequences in streaming data. *Journal of Combinatorial Optimization*, 11(2):155–175, 2006.

[8] M. Lothaire. *Algebraic Combinatorics on Words*. Number 90 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, Cambridge, UK, 2002.

[9] W. Masek and M. Paterson. A faster algorithm for computing string edit distances. *J. Comput. Syst. Sci.*, 20:18–31, 1980.

[10] S. Muthukrishnan. *Data Streams: Algorithms and Applications*, volume 1 of *Foundations and Trends in Theoretical Computer Science*. Now Publishers Inc., Hanover, MA, 2005.

[11] A. M. Odlyzko and E. M. Rains. On longest increasing subsequences in random permutations. 1999.

[12] C. Schensted. Largest increasing and decreasing subsequences. *Canadian Journal of Mathematics*, 13:179–191, 1961.

[13] J. M. Steele. Variations on the monotone subsequence theme of Erdös and Szekeres. In D. Aldous, P. Diaconis, J. Spencer, and J. M. Steele, editors, *Discrete Probability and Algorithms*, volume 72 of *The IMA volumes in mathematics and its applications*, pages 111–131. Springer-Verlag, Berlin, 1995.

[14] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1977.

[15] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[16] I.-H. Yang, C.-P. Huang, and K.-M. Chao. A fast algorithm for computing a longest increasing subsequence. *Information Processing Letters*, 93(5):249–253, 2005.