Information & Computation 208 (2010) 1054–1059

Contents lists available at ScienceDirect

Information & Computation

journal homepage: www.elsevier.com/locate/ic

Fast computation of a longest increasing subsequence and application ☆

Maxime Crochemore [a,b,*], Ely Porat [c]

[a] King's College London, Strand, London WC2R 2LS, UK
[b] Université Paris-Est, France
[c] Bar-Ilan University, Ramat-Gan 52900, Israel

ARTICLE INFO

Article history:
Received 7 December 2008
Revised 30 December 2009
Available online 15 May 2010

Keywords:
Design and analysis of algorithms
Longest increasing subsequence
Longest common subsequence
Data structures
Priority queue
Dynamic programming

ABSTRACT

We consider the complexity of computing a longest increasing subsequence (LIS) 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 $O(n \log k)$. The algorithm also improves on the previous $O(n \log\log n)$ bound. The optimality of the new bound is an open question.

Reducing the computation of a longest common subsequence (LCS) between two strings to an LIS computation leads to a simple $O(r \log\log k)$-time algorithm for two sequences having r pairs of matching symbols and an LCS of length k.

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 [1, Chapter 5] for a presentation of Schensted's algorithm [2] 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 three 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 $O(n^2 / \log n)$ [3,4]). 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. [5] and in its subsequent variants. A comparison is based on maximal exact segment 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. Third, the computation of an LCS between two strings reduces to an LIS computation (see [6, Chapter 12]). Although this reduction does not lead to LCS algorithms faster than those derived from dynamic programming techniques, the method is simple and saves a significant amount of memory space in some standard instances of the problem. The method then compares with the Hunt–Szymanski strategy [7] (see also [8]) but with a faster running time.

☆ Work partially supported by the Royal Society, UK. A preliminary version of the article has been presented at the conference "Visions of Computer Science", London, UK, 22–24 September 2008.

* Corresponding author at: King's College London, Strand, London WC2R 2LS, UK.
E-mail addresses: Maxime.Crochemore@kcl.ac.uk (M. Crochemore), PoratEly@cs.biu.ac.il (E. Porat).

0890-5401/$ - see front matter Crown copyright © 2010 Published by Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2010.04.003

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 [9], or the survey by Baik et al. [10] who discuss these questions, and references therein.

Liben-Nowell et al. [11] 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 [12]).

A direct solution to computing a longest increasing subsequence, running in $O(n \log n)$ time, was proposed by Fredman [13]. 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 [14] and references therein). It relies on the priority search trees of van Emde Boas [15,16], 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 split the sequence into blocks of size k in order to feed the priority queue used in the standard algorithm with elements drawn from a restricted universe. 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.

The first step of our algorithm processes the whole sequence with the mere bucket sorting technique to sort each of the blocks. Doing so, our method produces a $O(n \log \log k)$-time algorithm for the LIS computation on an unbounded universe. This step can be avoided if subsequently blocks can be sorted within the bounded time. For example, Han's integer sorting algorithm [17], running in $O(k \log \log k)$ independently of the size of the universe, can be used to do that. It also yields a whole computation that becomes online with respect to blocks because they are treated in sequence. But the techniques implied by Han's algorithm preclude any practical implementation of the computation. Instead, our solution is much more simple and does not have this drawback; it only looses the online feature, which not actually required.

When our result is applied to compute a longest common subsequence between two strings of length n, we get an algorithm running in $O(r \log \log k)$-time with $O(r)$ space, where r is the number of symbol matches between the strings and k is the maximal length of common subsequences. The reduction is used for the same purpose by Rahman and Iliopoulos [18] who achieve the computation through Range Minima Queries, a more expensive technique leading to a $O(r \log \log n)$ time algorithm only.

Section 2 recalls the core algorithm for computing a longest increasing subsequence and Section 3 describes our improved solution. Section 4 shows the consequence on LCS computation, which reduces to LIS computation.

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 $S = (\pi(1), \pi(2), \ldots, \pi(n))$. Its length is denoted by LIS(S).

Elements are processed in the order $\pi(1), \pi(2), \ldots, \pi(n)$. Conceptually we compute, for each length $\ell = 1, 2, \ldots$, 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 [19]). 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 itself 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 ⟵ (); k ⟵ 0
2.  for i ⟵ 1 to n
3.      x ⟵ π(i)
4.      Insert(B, x)
5.      if succ(B, x) exists
6.          Delete(B, succ(B, x))
7.      else k ⟵ k + 1
8.  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 [13]. The running time is indeed $O(n \log k)$ if $k = \mathrm{LIS}(S)$.

Using a more sophisticated priority list implementation in the form of van Emde Boas trees [15,16], each step can be performed in $O(\log \log n)$ amortised time yielding an overall $O(n \log \log n)$ running time algorithm [7].

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 first that a good approximation m of k, $m \geqslant k$, 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 like it is done 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 assigned a key $y = \mathit{key}(x)$ in a one-to-one correspondence. The inverse function is called *item*, then $x = \mathit{item}(y)$. 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. New keys are then defined as ranks of elements in the obtained sorted list. Since we assume $m \geqslant k$, the number of keys in B is no more than m and 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 $(\mathit{item}(y) | y \in B)$ 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 updated 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 $B =$ (a, b, 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 updated to (a, b, f). After processing keys g, d, e, c of elements of C_3, the contents of queue B are successively: (a, b, f, g), (a, b, d, g), (a, b, d, e), and (a, b, c, 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 increasing subsequence of length 4 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, $m \geqslant k$, 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 $m \geqslant k$, 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, possibly no more than k, for m (for instance, $m_0 = 4$), we consider the sequence $(m_i | i \geqslant 0)$ 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 $0 \leqslant j < i$ 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, $m \geqslant k$, 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

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

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. Computing a longest common subsequence

In this section, we recall how the previous algorithm to compute an LIS of a sequence of integer can be used to compute the maximal length of subsequences common (LCS) to two strings (see [6,18]). As a consequence we get a $O(n^2 \log\log k)$-time algorithm to compute the value k of the LCS of two strings of length n. (The running time is $O(mn \log\log k)$ for strings of respective lengths m and n.) This running time reduces to $O(n \log\log k)$ for some usual instances of the question.

Let x and y be two strings of length n over an integer alphabet. With each letter a occurring in x we associate the sequence $p(a)$ of positions of a on y in decreasing order. Let $S(x, y)$ be the sequence of positions on y obtained by concatenating the sequences $p(x[i])$:

$$S(x, y) = p(x[1])p(x[2]) \cdots p(x[n]).$$

The length of $S(x, y)$ is the number of matches between x and y, that is, the size of set $\{(i, j) | x[i] = y[j]\}$, which is $O(n^2)$. Then it is clear that we have the equality of the two lengths:

$$\mathrm{LCS}(x, y) = \mathrm{LIS}(S(x, y)).$$

Note that the sequence $S(x, y)$ is likely to contain several occurrences of the same positions and therefore is not a permutation of integers as stated in Section 2. But Theorem 2 is valid as well for sequences of integers with repetitions, although the problem seems to be more general. A solution is to modify in a fairly simple way the algorithms of previous sections. Here we rather consider the solution that comes down to that of a permutation as follows: each element q occurring at position i on $S(x, y)$ is renamed as the rank of the pair (q, b_q) in the lexicographically sorted list of all these pairs. The value b_q is the number of occurrences of q from position i in $S(x, y)$. The process is yet similar to the renaming by ranking used above. This leads to a new sequence that is now a permutation of an initial segment of the natural numbers (without 0) and to which the previous LIS algorithm applies.

Example. Let us consider the two strings $x = \texttt{ababa}$ and $y = \texttt{aabba}$. Then, $p(\texttt{a}) = (5, 2, 1)$, $p(\texttt{b}) = (4, 3)$, and

$$S(\texttt{ababa}, \texttt{aabba}) = (5, 2, 1, 4, 3, 5, 2, 1, 4, 3, 5, 2, 1).$$

We associate with it the sequence of pairs

$$((5, 3), (2, 3), (1, 3), (4, 2), (3, 2), (5, 2), (2, 2), (1, 2), (4, 1), (3, 1), (5, 1), (2, 1), (1, 1)).$$

When the pairs are replaced with their ranks, we get the new sequence

$$(13, 6, 3, 10, 8, 12, 5, 2, 9, 7, 11, 4, 1)$$

which LIS is 4.

The longest increasing subsequence $(3, 5, 7, 11)$ corresponds to the list $((1, 1), (3, 2), (4, 3), (5, 5))$ of pairs of positions on x and y, respectively, which gives the longest common subsequence $\texttt{aaba}$ between x and y.

The conclusion of the section is stated in the next corollary.

Corollary 3. *Let k be the maximal length of longest common subsequences between two strings of length n. Let r be their number of matches. Then, computing k and extracting a longest common subsequence of the two strings can be done in time $O(r \log\log k)$, which is also $O(n^2 \log\log k)$, within $O(r)$ memory space.*

The LCS notion is extensively used in sequence comparison and approximate pattern matching, and is usually extended to the notion of an alignment between sequences (see [6, Chapter 15] and [20, Chapter 8]). In bioinformatics FastA is one of the fastest tools used to search for an approximate match with a pattern in a database of genomic or protein sequences. The algorithm underlying the software is based on a heuristics that locates positions where the pattern is likely to occur. Then a dynamic programming technique is used to tune up the search close to these positions: the algorithm runs a standard alignment algorithm inside a band or strip around a diagonal of the dynamic programming table. The width of the band is a parameter of the software, say w, given by the user.

When the strip alignment is transposed with our above notation, computing this alignment amounts to compute the LIS of the sequence $S(x, y) = p(x[1])p(x[1]) \cdots p(x[n])$ where each $p(x[i])$ is restricted to the decreasing sequence of positions of $x[i]$ in the segment $y[j - w \,..\, j + w]$ ($j - i$ has a fixed value, it is the index of the selected diagonal). The length of $S(x, y)$, smaller than wn, is then $O(n)$ and the running time of the LIS computation becomes $O(n \log\log n)$.

5. Conclusion

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 [17]). But some of them are affiliated with 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.

References

[1] M. Lothaire, Algebraic Combinatorics on Words, No. 90 in Encyclopedia of Mathematics and its Applications, Cambridge University Press, Cambridge, UK, 2002.
[2] C. Schensted, Largest increasing and decreasing subsequences, Can. J. Math. 13 (1961) 179–191.
[3] W. Masek, M. Paterson, A faster algorithm for computing string edit distances, J. Comput. Syst. Sci. 20 (1980) 18–31.
[4] M. Crochemore, G.M. Landau, M. Ziv-Ukelson, A sub-quadratic sequence alignment algorithm for unrestricted cost matrices, SIAM J. Comput. 32 (6) (2003) 1654–1673.
[5] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, S.L. Salzberg, Alignment of whole genomes, Nucleic Acid Res. 27 (11) (1999) 2369–2376.

[6] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, Cambridge, 1997.
[7] J.W. Hunt, T.G. Szymanski, A fast algorithm for computing longest subsequences, Commun. ACM 20 (5) (1977) 350–353.
[8] A. Apostolico, Improving the worst-case performance of the Hunt–Szymanski strategy for the longest common subsequence of two strings, Inform. Process. Lett. 23 (2) (1986) 63–69.
[9] J.M. Steele, Variations on the monotone subsequence theme of Erdös and Szekeres, in: D. Aldous, P. Diaconis, J. Spencer, J.M. Steele (Eds.), Discrete Probability and Algorithms, The IMA Volumes in Mathematics and its Applications, vol. 72, Springer-Verlag, Berlin, 1995, pp. 111–131.
[10] J. Baik, P. Deift, K. Johansson, On the distribution of the length of the longest increasing subsequence of random permutations, J. ACM 12 (4) (1999) 1119–1178.
[11] D. Liben-Nowell, E. Vee, A. Zhu, Finding longest increasing and common subsequences in streaming data, J. Comb. Optim. 11 (2) (2006) 155–175.
[12] S. Muthukrishnan, Data streams: algorithms and applications, Foundations and Trends in Theoretical Computer Science, vol. 1, Now Publishers Inc., Hanover, MA, 2005
[13] M.L. Fredman, On computing the length of longest increasing subsequences, Discrete Math. 11 (1975) 29–35.
[14] I.-H. Yang, C.-P. Huang, K.-M. Chao, A fast algorithm for computing a longest increasing subsequence, Inform. Process. Lett. 93 (5) (2005) 249–253.
[15] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, Inform. Process. Lett. 6 (1977) 80–82.
[16] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Math. Syst. Theory 10 (1977) 99–127.
[17] Y. Han, Deterministic sorting in $O(n \log\log n)$ time and linear space, J. Algorithms 50 (2004) 96–105.
[18] M.S. Rahman, C.S. Iliopoulos, A new efficient algorithm for computing the longest common subsequence, in: M.-Y. Kao, X.-Y. Li (Eds.), Algorithmic Aspects in Information and Management (AAIM 2007), Lecture Notes in Computer Science, vol. 4508, Springer, 2007, pp. 82–90.
[19] S. Bespamyathnikh, M. Segal, Enumerating longest increasing subsequences and patience sorting, Inform. Process. Lett. 76 (1–2) (2000) 7–11.
[20] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on Strings, Cambridge University Press, Cambridge, 2007.