Journal of Discrete Algorithms 20 (2013) 12–20

Contents lists available at SciVerse ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Computing a longest common subsequence that is almost increasing on sequences having no repeated elements ☆

Johra Muhammad Moosa, M. Sohel Rahman *,[1], Fatema Tuz Zohora

AℓEDA Group, Department of CSE, BUET, Dhaka-1000, Bangladesh

ARTICLE INFO

Article history:
Available online 27 December 2012

Keywords:
Algorithms
Longest common subsequence
Longest increasing subsequence
Longest almost increasing subsequence

ABSTRACT

Given two permutations A and B of $[1..n]$ and a fixed constant c, we introduce the notion of a longest common almost increasing subsequence (LCAIS) as a longest common subsequence that can be converted to an increasing subsequence by possibly adding a value, that is at most c, to each of the elements. We present an algorithm for computing LCAIS in $O(n^2)$ space, $O(n(n+c^2))$ time.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

In this paper, we study some problems on permutations and sequences. A sequence is an ordered list of n arbitrary numbers (or elements). A permutation (of length n) is a special kind of sequence where the elements are from $[1..n]$. Throughout this paper, we use the following convention. We use $A = (a_1, \ldots, a_n)$ to denote a sequence A, where the n elements/numbers are put inside the first brackets in order. On the other hand, a permutation A of $[1..n]$ is denoted by $A = \langle a_i, \ldots, a_n \rangle$ where angle brackets are used instead of first brackets.

Given a permutation $A = \langle a_1, \ldots, a_n \rangle$ of $[1..n]$, an increasing subsequence (IS) of A is a subsequence $(a_{j_1}, a_{j_2}, \ldots, a_{j_\ell})$ such that $j_1 < j_2 < \cdots < j_\ell$ and $a_{j_1} < a_{j_2} < \cdots < a_{j_\ell}$. A longest increasing subsequence (LIS) of a permutation A is an IS of A having the maximum length. The goal of the LIS problem is to compute an LIS of A. Given two sequences (not necessarily permutations) $A = (a_1, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_m)$, a common subsequence (CS) refers to a subsequence common to both of the sequences. The longest common subsequence (LCS) of A and B is a common subsequence of them having the maximum length. The goal of the LCS problem is to compute an LCS given two sequences A and B.

The Longest Increasing Subsequence (LIS) and Longest Common Subsequence (LCS) problems are both classical problems in computer science. The LIS problem was first tackled by Robinson [11] more than seventy years ago. Schensted [13] and Knuth [9] gave an $O(n \log n)$ time algorithm to solve the problem. Bespamyatnikh and Segal [1] gave improved algorithms to solve the LIS problem that run in $O(n \log \log n)$ time. Very recently, Crochemore and Porat [5] presented an $O(n \log \log k)$ time algorithm for the problem assuming a RAM model, where k is the size of the output.

By using a simple dynamic programming technique, the LCS problem for two sequences of n elements can be solved in $O(n^2)$ time. The only $o(n^2)$ results known so far are those by Masek and Paterson [10] and Crochemore et al. [4], that run in $O(n^2/\log n)$ and $O(hn^2/\log n)$ time respectively. For the latter work, $h \leqslant 1$ refers to the entropy of the input sequence. Hunt and Szymanski [7] proposed an $O(R \log n + n)$ time algorithm for the LCS problem, where R is the number of ordered pairs

☆ Authors' names are in the alphabetic order of their last names. This research work was part of the undergraduate thesis of Moosa and Zohora under the supervision of Rahman.

* Corresponding author.
E-mail address: sohel.kcl@gmail.com (M.S. Rahman).

[1] Partially supported by a Commonwealth Fellowship and an ACU Titular Fellowship.

1570-8667/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.jda.2012.12.005

of positions where the two input sequences match. Recently, Rahman and Iliopoulos presented an improved LCS algorithm which runs in $O(R \log\log n + n)$ time [8]. Although $R = O(n^2)$, there are large number of applications for which $R \sim n$ [7].

A combination of LCS and LIS gives us another interesting problem, namely, the Longest Common Increasing Subsequence (LCIS) problem. In LCIS, the goal is to compute a longest common subsequence that is also an increasing sequence. The LCIS problem has not been studied until recently when Yang et al. [14] proposed an algorithm having time and space complexity $O(n^2)$ to construct an LCIS of two sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$. Subsequently, Sakai [12] proposed a space-efficient algorithm for the LCIS problem, which takes $O(n^2)$ time and $O(n)$ space. Brodal et al. [2] gave a faster algorithm which runs in $O(n\ell \log\log\sigma + Sort)$ time and $O(n)$ space, where σ is the size of the alphabet, and *Sort* is the time to sort each input sequence. Chan et al. [3] proposed an efficient algorithm to find an LCIS of two sequences in $O(\min(R \log \ell, n\ell + R) \log\log n + Sort)$ time.

Recently Elmasry [6] introduced the concept of an *Almost Increasing Subsequence* (*AIS*). Given a constant $c > 0$, a sequence $y_1, y_2, \ldots, y_k$ is said to be *almost increasing* if and only if $\forall_i y_i > \max_{j=1}^{i-1} y_j - c$. Elmasry used this concept to propose the problem of finding a Longest Almost Increasing Subsequence (LAIS) as an interesting and perhaps more practically useful variant of the LIS problem. In LAIS problem, given an extra parameter c as part of the input, instead of a longest increasing subsequence, the goal is to find a longest **almost** increasing subsequence. Elmasry [6] in fact showed how to efficiently construct an LAIS in $O(n \log k)$ time, where k is the length of the output.

In this paper, we introduce the notion of a Longest Common Almost Increasing Subsequence (LCAIS) which can be seen as a natural extension of both LAIS and LCIS problems. Given two permutations $A = \langle a_1, a_2, \ldots, a_n\rangle$ and $B = \langle b_1, b_2, \ldots, b_n\rangle$ of $[1..n]$ and a constant $c > 0$, a Common Almost Increasing Subsequence (CAIS) is a common subsequence $Y = (y_1, y_2, \ldots, y_m)$ of A and B such that $\forall_i y_i > \max_{j=1}^{i-1} y_j - c$. A longest common almost increasing subsequence (LCAIS) of A and B is a CAIS having the maximum length. Interestingly, an LCAIS can be seen as an LCS that can be converted to an increasing sequence by possibly adding a value, that is at most a fixed constant (c), to each of the elements.

Apart from being interesting from a theoretical point of view, the LCAIS problem seems to have practical motivations as well. The motivation of LCAIS follows naturally from that of LAIS. So, we first present the following discussion on the motivation of LAIS, which is borrowed from [6]. Consider 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. Clearly this scenario also motivates the problem of LCAIS when we plan to compare the similarity of two related activities based on their historical snapshots. Additionally, we can always cite the presence of noise as a motivation of LCAIS as a relaxed version of LCIS. In this paper, we present an efficient algorithm for computing an LCAIS. In particular, we first present an algorithm to solve the LCAIS problem that runs in $O(n^3)$ time and space. Then we show how to implement the algorithm to achieve $O(n(n + c^2))$ time and $O(n^2)$ space complexity.

The rest of the paper is organized as follows. In Section 2 we discuss our LCAIS algorithm after presenting some definitions and notions required to describe our algorithm. Section 3 states and proves some important lemmas and theorems to establish the correctness as well as the claimed time and space complexity of our algorithm. Finally we briefly conclude in Section 4.

2. An LCAIS algorithm

In this section, we present our algorithm to solve the LCAIS problem. In what follows the following notions/definitions will be useful.

Definition 2.1 *(Compatible).* Given a fixed integer c, y is said to be compatible with x if and only if $y > x$ or $y > x - c$. In the latter case, i.e., $y > x - c$, we say y is compatible with x *considering* c. Otherwise, we say that y is incompatible with x.

Definition 2.2 *(Compatibility check).* A compatibility check between an element x and a sequence $S = (s_1, s_2, \ldots, s_p)$ is the procedure of checking whether x is compatible with s_i, $1 \leqslant i \leqslant p$.

Definition 2.3 *(length(X)).* Given a sequence X, we use *length*(X) to denote the length of X.

Definition 2.4 *(match(i, j)).* Given two sequences $A = (a_1, a_2, \ldots, a_i, \ldots, a_m)$ and $B = (b_1, b_2, \ldots, b_j, \ldots, b_n)$, we say that $match(i, j) = a_i (= b_j)$ iff $a_i = b_j$ where $1 \leqslant i \leqslant m$, $1 \leqslant j \leqslant n$.

Now, we are ready to describe our algorithm. Suppose that $A = \langle a_1, a_2, \ldots, a_n\rangle$ and $B = \langle b_1, b_2, \ldots, b_n\rangle$ are two permutations of $[1..n]$. We maintain an array $L_j^i[k]$, $1 \leqslant i, j, k \leqslant n$ as follows: we set $L_j^i[k] = a_i$ if there is a CAIS between $(a_1, a_2, \ldots, a_i)$ and $(b_1, b_2, \ldots, b_j)$ having length k ending at a_i such that a_i is the smallest such number and there exists no CAIS between A and B ending at a_i but having length greater than k.

In order to keep the backtracking information, we use two more variables as follows: we use $L\_index^i_j[k]$ $(1 \leqslant i, j, k \leqslant n)$ to record the index pair (i, j) such that $L^i_j[k] = a_i = b_j$. We also make use of a linked-list named *predlist*, associated with each match a_i, to keep track of the previous members of the CAIS having the maximum length k ending at a_i.

Given two permutations A, B and a constant c, our algorithm runs in three main steps, namely, *initialization* (Step 1), *main computation* (Step 2) and *construction* (Step 3). In what follows, we describe these three steps elaborately.

Step 1: Initialization

Initialization is done by assigning infinity to $L^i_j[k]$ for $1 \leqslant i, j, k \leqslant n$ indicating that there is no LCAIS initially.

Step 2: Main computation process

The main computation is done in a row major order. At the beginning of the computation of Row i, $L^i_j[k]$ is initialized to $L^{i-1}_j[k]$. Until a match is found in Row i, no operation is done.

Now suppose we have a match at Column j, i.e., $a_i = b_j$. We have to find out the previous integers in the CAIS ending at the match $a_i = b_j$. Now we find the largest k' such that $k' < i$ and $L^i_j[k'] - c < a_i$. If such a k' does not exist, we set $L^i_j[1] = a_i$, which means that there is a CAIS of length one containing only element $a_i = b_j$. Otherwise, we compute $predlist(a_i)$ according to the following cases. Notably, for the sake of a better understanding, we identify the cases with an example presented later in Section 2.2.

Case 1. $[L^i_j[k'] < a_i.]$ In this case, we first initialize $predlist(a_i)$ to $predlist(L^i_j[k'])$ and then append $L^i_j[k']$ to $predlist(a_i)$. Then, $L^i_j[k'+1]$ is set to a_i. Additionally, for all $j < j' \leqslant n$, if $a_i < L^i_{j'}[k'+1]$, we set $L^i_{j'}[k'+1] = a_i$. Finally, we set $L\_index^i_{j'}[k'+1] = (i, j)$. This case occurs in $match(3, 3)$ in the example of Section 2.2.

Case 2. $[L^i_j[k'] > a_i$ but $L^i_j[k'] - c < a_i.]$ In this case we perform a *Compatibility Check* between a_i and $predlist(L^i_j[k'])$. Depending on the result we have two different sub-cases as described below.

Case 2.1. [a_i is compatible with all the elements of $predlist(L^i_j[k'])$.] In this case, we do the same as Case 1. Please refer to $match(5, 4)$, $match(7, 7)$, $match(8, 6)$, $match(11, 9)$ and $match(12, 10)$ in the example of Section 2.2.

Case 2.2. [a_i is incompatible with some of the elements of $predlist(L^i_j[k'])$.] Suppose there are $N > 0$ such elements. So, we skip N such elements from $predlist(L^i_j[k'])$. Now, we can prove that $N \leqslant c - 1$ (please refer to Lemma 3.1 in the following section). Suppose ℓ' denotes the length of the LCAIS ending at a_i $(= b_j)$ according to our computation in Case 2. Let a' be the ending character of $predlist(a_i)$. Recall that $L^i_j[k']$ keeps track of a CAIS having length k' and $predlist(L^i_j[k'])$ has length $k' - 1$. Then, we have $\ell' = ((k' - 1) - N) + 1 = k' - N$. At this point we need a *Length Verification* step as argued below.

It is possible that while checking a sequence having length, say k'', where $((k' - 1) - N) \leqslant k'' \leqslant k' - 1$, none of the elements of $predlist(L^i_j[k''])$ are required to be skipped during the *Compatibility Check*. So this *Compatibility Check* might give us a longer CAIS. Also even if N' elements are skipped from $predlist(L^i_j[k''])$, the resultant length of CAIS ending at a_i still could be greater than ℓ' if $N' < N$.

So the *Length Verification* step is executed as follows. Here the idea is to check N sequences with length k'' such that $((k' - 1) - N) \leqslant k'' < k' - 1$ (the fact that checking only k''-length sequences is enough is proved in Lemma 3.3 in a later section). During each iteration of this check, if we find $L^i_j[k''] = a''$, where $a'' - c < a_i$, we perform *Compatibility Check* between a_i and $predlist(a'')$. Let S_1 be the sequence returned after the *Compatibility Check*. Say $\ell'' = length(S_1) + 1$. Now we have the following cases. The correctness of the steps executed in the following cases are proved later in Lemmas 3.2 and 3.5.

Case 2.2.1. $[\ell' < \ell''.]$ So S_1 will make a longer sequence, i.e., length of CAIS ending at a_i can be greater than ℓ'. Hence we set $predlist(a_i) = S_1$.

For an example of this case, let there are two sequences $A = (7, 15, 14, 3, 2, 13, 6, 8, 11)$ and $B = (7, 15, 14, 13, 6, 8, 3, 11)$. We align A with the vertical axis and B with the horizontal axis. As we proceed, $match(6, 4)$ gives us a CAIS $(7, 15, 14, 13)$ and $match(8, 6)$ gives a CAIS $(7, 6, 8)$. So, $L_8[3]$ is set to 8 and $L_8[4]$ is set to 13. Hence, for $match(9, 8) = 11$, CAIS $(7, 15, 14, 13)$ is considered as a candidate *predlist* of $match(9, 8)$ because $L_8[4] = 13 - c < 11$. So for $match(9, 8)$ we set $temp\_predlist = (7, 15, 14, 13)$ and then a $c - 1$ loop is run on it to see if any element needs be discarded. Clearly, here, 15 and 14 will be discarded and length of $temp\_predlist = (7, 13)$ will be 2 now. So the algorithm now searches for a better sequence (Line 17 in the subroutine LAIS_Insert). As a result LAIS_Insert returns $(7, 6, 8)$ as *temp_predlist*, which is of length 3. Since the latter is better, finally our algorithm sets $predlist(11) = (7, 6, 8)$ and we get a CAIS $(7, 6, 8, 11)$.

Case 2.2.2. $[\ell' > \ell''.]$ In this case $predlist(a_i)$ will remain as before.

Case 2.2.3. [$\ell' = \ell''$ & $a'' < a'$.] In this case we set $predlist(a_i) = S_1$. $Match(9, 8)$ of the example in Section 2.2 illustrates this case.

Case 2.2.4. [$\ell' = \ell''$ & $a'' > a'$.] In this case $predlist(a_i)$ whose last element is a' will remain unchanged.

Computation of $predlist(a_i)$ is complete now. Let ℓ''' be the length of the finally selected CAIS ending at a_i. So we set $L^i_j[\ell'''] = a_i$. We also set $L\_index^i_j[\ell'''] = (i, j)$. Finally for all $j < j' \leqslant n$, if $a_i < L^i_{j'}[\ell''']$ we set $L^i_{j'}[\ell'''] = a_i$. Additionally we set $L\_index^i_{j'}[\ell'''] = (i, j)$.

Step 3: Constructing an LCAIS

In this step, we construct an LCAIS. Let ℓ be the largest length such that $L^m_n[\ell] \neq \infty$, where n and m represent the final column and final row respectively in the matrix. This means that there exists a sequence having length ℓ which is the highest possible length. Now we just need to recover the ending character of that sequence. Recall that this is stored in $L\_index^m_n[\ell]$. Suppose that $L\_index^m_n[\ell] = (x_1, y_1)$. Now suppose $a_{x_1} = b_{y_1} = a$. We initialize an empty list S to $predlist(a)$. We now simply append a to S. Then we return S as an LCAIS.

2.1. Non-permutation

While describing our algorithm we have assumed that the input sequences are permutations. However it is easy to realize that in our algorithm we only need that the input sequences do not have any repeated elements. So the input sequences need not be permutations and they even can be of different lengths. In fact our algorithm relies only on the matched elements. As long as we have distinct matches, the algorithm would work correctly. More formally, we only need to ensure that if $match(i_1, j_1) = \alpha = match(i_2, j_2)$, then $i_1 = i_2$ and $j_1 = j_2$. In that sense, our algorithm will work for a class of non-permutation sequences as well. And, in fact, the illustrative example in Section 2.2 has been presented on two sequences that are not really permutations.

Notably, it seems that using a balanced tree like structure we may be able to exploit some more properties of permutations and get a faster algorithm for them. But unfortunately such an algorithm still eludes us. The same fate applies for the case when the inputs are general sequences instead of permutations.

2.2. An illustrative example

In this section, we provide an example to illustrate how our algorithm works. Let the inputs be $A = (20, 7, 15, 1, 14, 3, 6, 13, 11, 18, 10, 9)$, $B = (7, 12, 15, 14, 21, 13, 6, 11, 10, 9)$ and $c = 3$. Clearly, $m = 12$ and $n = 10$, where m, n are the lengths of the sequences A and B respectively. Also note that, following the discussions of Section 2.1, in this example we have considered two sequences that are non-permutations. In particular both A and B are arbitrary sequences having no repetitions. For each $match$, values of L, L_index and $predlist$ are shown below:

1. For $match(2, 1) = 7$, our algorithm has $predlist = null$. It sets $L_1[1]$ through $L_{10}[1]$ to 7 and $L\_index_1[1]$ through $L\_index_{10}[1]$ to $(2, 1)$. Constructed CAIS for this match is (7).
2. For $match(3, 3) = 15$, the algorithm gets $L_3[1] = 7 < 15$. So it sets $predlist(15) = (7)$. Then it sets $L_3[2]$ through $L_{10}[2]$ to 15 and $L\_index_3[2]$ through $L\_index_{10}[2]$ to $(3, 3)$. So CAIS for this match is $(7, 15)$.
3. For $match(5, 4) = 14$, our algorithm gets $L_4[2] = 15 - c < 14$. So it runs a $c - 1$ loop on the $predlist(15)$ to see if anyone needs to be discarded. Since none needs be discarded, it sets $predlist(14) = (7, 15)$. It also sets $L_4[3]$ through $L_{10}[3]$ to 14 and $L\_index_4[3]$ through $L\_index_{10}[3]$ to $(5, 4)$. Here constructed CAIS is $(7, 15, 14)$.
4. For $match(7, 7) = 6$, similarly the algorithm sets $predlist(6) = (7)$. Then it sets $L_7[2]$ through $L_{10}[2]$ to 6 and updates $L\_index_7[2]$ through $L\_index_{10}[2]$ to $(7, 7)$. CAIS for this match is $(7, 6)$.
5. For $match(8, 6) = 13$, the algorithm sets $predlist(13) = (7, 15, 14)$. It sets $L_6[4]$ through $L_{10}[4]$ to 13 and updates $L\_index_6[4]$ through $L\_index_{10}[4]$ to $(8, 6)$. Constructed CAIS for this match is $(7, 15, 14, 13)$.
6. For $match(9, 8) = 11$, the algorithm gets $L_8[4] = 13 - c < 11$. So it sets $temp\_predlist = (7, 15, 14, 13)$ and runs a $c - 1$ loop on this list to see if anyone needs to be discarded. It then discards 15 and 14 and length of $temp\_predlist = (7, 13)$ is reduced to 2. So it goes to Line 17 in the subroutine LAIS_Insert and search for a better sequence. As a result it gets $(7, 6)$ as $temp\_predlist$ which is better than $(7, 13)$.
   So it sets $predlist(11) = (7, 6)$. Then it sets $L_8[3]$ to $L_{10}[3] = 11$ and also updates $L\_index_8[3]$ through $L\_index_{10}[3]$ to $(9, 8)$. Hence CAIS for this match is $(7, 6, 11)$. This case is illustrated in Fig. 1.
7. For $match(11, 9) = 10$, our algorithm sets $predlist(10) = (7, 6, 11)$. It sets $L_9[4]$ through $L_{10}[4]$ to 10 and updates $L\_index_9[4]$ through $L\_index_{10}[4]$ to $(11, 9)$. So CAIS for this match is $(7, 6, 11, 10)$.
8. For $match(12, 10) = 9$, we get the longest sequence. The algorithm sets $predlist(9) = (7, 6, 11, 10)$. Then it sets $L_{10}[5] = 9$ and updates $L\_index_{10}[5]$ to $(12, 10)$. So constructed CAIS for this match is $(7, 6, 11, 10, 9)$.

Note that, if we had not run the search for a better sequence at Line 17 of subroutine *LAIS_Insert*, then we would have missed the sequence $(7, 6, 11)$ and get the sequence $(7, 13, 11)$ for $match(9, 8) = 11$. As a result for $match(11, 9) = 10$, we

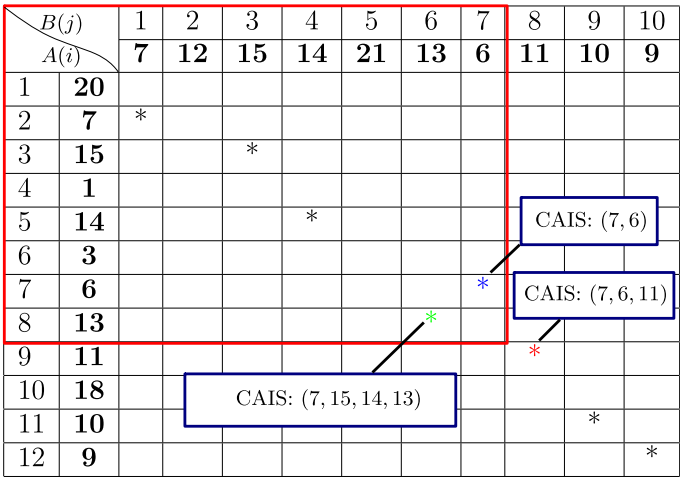

Fig. 1. Illustration for *match*(9, 8).

would get (7, 11, 10) instead of getting the sequence of length 4 namely (7, 6, 11, 10) which would have ultimately caused the omission of the sequence of length 5, namely (7, 6, 11, 10, 9), as the final result.

3. Implementation and analysis

A straightforward implementation of our algorithm described in Section 2 would run in $O(n^3)$ time using $O(n^3)$ space as discussed below. For each row (say, Row i), we search for a match which requires $O(n)$ time. After finding a match a_i, we create CAIS ending at that match. *Compatibility Check* can be done in $i-1$ iterations and, in the *Length Verification* step, $i-1$ sequences are needed to be checked which will require $O(n^2)$ time. So the overall time complexity would be $O(n^3)$. As $L^i_j[k]$ and $L\_index^i_j[k]$ are three-dimensional arrays, the space complexity would also be $O(n^3)$.

In the rest of this section, we discuss how we can achieve $O(n(n+c^2))$ time and $O(n^2)$ space complexity for our algorithm. We present a number of lemmas below to establish the improved time and space complexity.

Lemma 3.1. *It is sufficient to run a $c-1$ loop to perform the Compatibility Check in Case* 2.

Proof. For a match α, our algorithm selects a CAIS of length k ending at α' where $\alpha' - c < \alpha$. Now, it is sufficient to run a $(c-1)$ loop on *predlist*(α') as follows. In the worst case, all the elements in the range $[\alpha'+1..\alpha'+c-1](=[\ell..r])$ exist in *predlist*(α'). Now, since both A and B are permutations (i.e., no elements are repeated) and since for any position $p' < p'' \leqslant p$ in a CAIS $X = (x_1, \ldots, x'_p, \ldots, xp'', \ldots, x_p)$ we must have $x_{p'} - c < x_{p''}$, all the elements in the range $[\ell..r]$ must occur in the rightmost $c-1$ positions of *predlist*(α'). Now α can be incompatible with some of these $(c-1)$ elements. Here if $\alpha' = \alpha + c$ and if all the elements between the range $[\ell..r]$ appear in *predlist*(α'), then all of the last $c-1$ elements of *predlist*(α') will be skipped. So it is enough to check the rightmost $c-1$ elements of *predlist*(α') by running a simple $c-1$ loop. □

Lemma 3.2. *Suppose $S' = (s'_1, s'_2, \ldots, s'_p)$ and $S'' = (s''_1, s''_2, \ldots, s''_p)$ are two sequences such that $|S'| = |S''| = p$ and $s'_p < s''_p$. Suppose for an element a_i both S' and S'' are valid as predlist(a_i). Then we can safely assign predlist$(a_i) = S'$.*

Proof. We can find a match a_{i+q} in Row $i+q$, where $q > 0$, $a_i - c < a_{i+q}$, $L^i_j[k'] - c > a_{i+q}$. If the number of elements skipped after the *Compatibility Check* between a_{i+q} and S' (S'') is N' (N'') then $N' \leqslant N''$. So we can safely consider S'. □

Lemma 3.3. *It is sufficient to check sequences with length k'' in the Length Verification step where $((k'-1)-N) \leqslant k'' < k'-1$.*

Proof. Upper bound is found from the fact that, search for a sequence having length k' will return the same sequence as before (Case 2). So k'' must have length less than $k'-1$. Again $k'' \geqslant ((k'-1)-N)$, because search for a sequence having length less than that will result in a CAIS having smaller length than ℓ', where $\ell' = length(predlist(a_i))$ after *Compatibility Check*, assuming a_i is the current match. Sequence having length $((k'-1)-N)$ is also checked because element $L^i_j[((k'-1)-N)]$ might be smaller than $L^i_j[k']$ which follows from Lemma 3.2. □

Lemma 3.4. *Length Verification step runs in $O(c^2)$ time.*

```
   input : Two sequences, let A = ⟨a_1, a_2, ..., a_i, ..., a_m⟩ and B = ⟨b_1, b_2, ..., b_j, ...b_n⟩ , where m ⩾ n.
   output: LCAIS
 1 Initialization.
 2 for j = 1 to m do
 3     for i = 1 to n do
 4         L_j[i] = ∞;
 5     end
 6 end
 7 Main program.
 8 for i = 1 to m do
 9     z = −1;
10     χ = −1;
11     if i − 1 > n then
12         p = n;
13     end
14     else
15         p = i − 1
16     end
17     for j = 1 to n do
18         if a_i = b_j then                                                    // the match case
19             χ = LAIS_Insert(L_j, L_index, a_i, p, i, j) ;    // returns the insert position that means length of the
               subsequence ending at a_i = b_j
20             z = j ;                                                  // save the index of column
21         end
22         else                                                             // the mismatch case
23             if χ ≠ −1 and L_{j−1}[χ] < L_j[χ] then
24             L_j[χ] = L_{j−1}[χ];
25             L_index_j[χ] = (i, z) ;        // save the (row,column) position of ending character of sequence
               having length χ
26             else χ = −1;
27         end
28     end
29 end
30 recover the LCAIS.
31 x = largest x such that L_n[x] is not ∞, if no such x then return null;
32 (y_1, y_2) = L_index_n[x];
33 let, a be the match(y_1, y_2);
34 then print a and predlist(a);
```

Algorithm 1: The LCAIS algorithm.

In the worst case, $N = c - 1$ (Lemma 3.1) where N is the number of elements skipped in *Compatibility Check*. So we need to check $c - 1$ sequences at most. While checking each sequence we have to check compatibility between the current match and the sequence. It follows that the *Length Verification* step requires $O(c^2)$ time.

Lemma 3.5. *While constructing CAIS ending at α, preserving the CAIS with the maximum length is sufficient.*

Proof. Let at $match(i, j) = \alpha$, a sequence of length k ending at α be formed. Now, for this row (i.e., Row i) and column (i.e., Column j), if there is a sequence S having length $\ell < k$ whose ending character $\alpha' > \alpha$, then we do not need to update that sequence S by replacing the ending element with α. This is because, if in future, for match α'' our algorithm has to work with the sequence ending at α, it will definitely pick the sequence having length k (and not less than k). If $\alpha'' > \alpha$, then the whole sequence will be merged with α'' and will give a sequence of length $k + 1$. If $\alpha'' > \alpha - c$, then the algorithm will also pick the sequence having length k and then will run a $c - 1$ loop to discard incompatible elements. Hence the result follows. □

Lemma 3.6. *Maintaining L_j and L_index_j, $1 \leqslant j \leqslant n$ is sufficient instead of L^i_j & $L\_index^i_j$ where $1 \leqslant i,\ j \leqslant n$.*

Proof. Although it seems that to keep the record of $L^i_j[k]$ we need a three-dimensional array in fact it is not necessary. As we set $L^i_j[k]$ to $L^{i-1}_j[k]$ it is not required to keep records for all values of i. Rather, we run the loop in row major order and keep records for each column. So the notation can safely be changed to $L_j[k]$. So (according to Algorithm 1) for all $1 \leqslant i \leqslant n$, L^i_j and $L\_index^i_j$ share the memory spaces of L_j and L_index_j. □

```
1  LAIS_Insert(array L_j, array L_index, match a, integer p, integer i', integer j ){
2  x = p ;
3  while x ≠ 0 do
4      if L_j[x] < a then
5          ℓ = x + 1; L_index_j[ℓ] = (i', j);
6          set predlist(a) to (L_j[x] + predlist(L_j[x]));
7          return ℓ
8      end
9      else
10         if L_j[x] − c < a then
11             temp_predlist = newlist;
12             check&insert(a, L_j[x], temp_predlist,∞);
13             predlist(a) = temp_predlist;
               // Now check if any integer number has been skipped, if yes search if any better
               sequence exists;
14             current_size = sizeof(temp_predlist);
15             if current_size < x then
16                 skip = x − 1 − current_size;
17                 for i = x − 1 to current_size do
18                     temp_predlist = newlist;
19                     if L_j[i] < a then
20                         set temp_predlist to (L_j[i] + predlist(L_j[i]));
21                         best_l = pick_one(temp_predlist, predlist(a)); predlist(a) = best_l;
22                     end
23                     else if L_j[i] − c < a then
24                         check&insert(a, L_j[x], temp_predlist,skip );
25                         best_l = pick_one(temp_predlist, predlist(a));
26                         predlist(a) = best_l;
27                     end
28                     skip − −; i − −;
29                 end
30                 ℓ = sizeof(predlist(a)) + 1; L_index_j[ℓ] = (i', j);
31                 return ℓ
32             end
33             else
34                 ℓ = x + 1; L_index_j[ℓ] = (i', j);
35                 return ℓ
36             end
37         end
38         else
39             x − −;
40         end
41     end
42 end
43 L_j[1] = a; predlist(a) = null;
44 return 1 ; }
```

Algorithm 2: Subroutine: LAIS_Insert.

Lemma 3.7. *For a match α, predlist(α) needs $O(n)$ space.*

Proof. Variable *predlist* is used to keep the sequence formed by the corresponding match. So for the matches of one CAIS we can keep only one *predlist*, i.e., for the last element of the CAIS. So each *predlist* would require $O(n)$ space. □

Theorem 3.1. *LCAIS can be solved in $O(n(n + c^2))$ time using $O(n^2)$ space.*

Proof. Since we have n columns, for each row (say, Row i) we search for a match which requires $O(n)$ time. Suppose that for Row i, we have a match a_i at Column j. Then a search on $L_j[k]$, where $1 \leqslant k \leqslant i - 1$ is performed until a compatible element is found. If no such element is found then the algorithm will take at most $n - 1$ iterations when $i = n$. Otherwise, if the algorithm finds $L_j[k] < a_i$, it stops there and make the CAIS ending at that match a_i as described in Case 1 of Step 2. But if $L_j[k] > a_i$ and $L_j[k] - c < a_i$, then, at first the *Compatibility Check* is done which runs in $O(c)$ time (Lemma 3.1). Then, the *Length Verification* step is performed which requires $O(c^2)$ time (Lemma 3.4). So, the overall time complexity of constructing

```
1  check&insert(a, candidate, temp_predlist, skip){ // c is the provided constant
2  insert candidate into the temp_predlist;
3  let, k = index of the last member of predlist(candidate);
4  for i = 1 to c − 1 do
5  |  q = kth member of predlist(candidate) ;
6  |  if q = null then
7  |  |  break;
8  |  end
9  |  else if q < a then
10 |  |  insert q into the temp_predlist;
11 |  |  break;
12 |  end
13 |  else if q − c < a then
14 |  |  insert q into the temp_predlist;
15 |  end
16 |  else
17 |  |  skip − −;
18 |  |  if skip ⩽ 0 then    // more than allowed number of integer numbers have been skipped so it can
   |  |  not be a better choice. So return null via temp_predlist
19 |  |  |  temp_predlist = null;
20 |  |  |  return;
21 |  |  end
22 |  end
23 |  k − −;
24 end
25 copy remaining integer numbers of predlist(candidate) to temp_predlist;
26 return }
```

Algorithm 3: Subroutine: check&insert.

```
1  pick_one(predlist, temp_predlist ){
2  if sizeof(predlist) > sizeof(temp_predlist) then
3  |  return predlist;
4  end
5  else if sizeof(predlist) < sizeof(temp_predlist) then
6  |  return temp_predlist;
7  end
8  else                    // both have the same length so break the tie using smallest ending number
9  |  if last_elem(predlist) < last_elem(temp_predlist) then
10 |  return predlist;
11 |  else  return temp_predlist;
12 end
13 }
```

Algorithm 4: Subroutine: pick_one.

LCAIS will be $O(n(n+c^2))$. Finally since we will need n *predlists* in the worst case, it follows from Lemmas 3.6 and 3.7 that the algorithm uses $O(n^2)$ space.

We formally present our algorithm in Algorithms 1–4.

4. Conclusion

In this paper, we have introduced the LCAIS problem and presented an efficient algorithm to solve the problem. Our algorithm for computing an LCAIS runs in $O(n^2)$ space, $O(n(n+c^2))$ time. Note that, although we assumed the two input sequences to be permutations, for our algorithm to work, we only require that the input sequences do not contain repeated elements. However it might be interesting to consider the variant where this restriction is lifted. So, future research endeavor could be directed towards this variant where the input sequences can be any general sequence.

References

[1] S. Bespamyatnikh, M. Segal, Enumerating longest increasing subsequences and patience sorting, Inform. Process. Lett. 76 (1–2) (2000) 7–11.
[2] G. Brodal, K. Kaligosi, I. Katriel, M. Kutz, Faster algorithms for computing longest common increasing subsequences, in: M. Lewenstein, G. Valiente (Eds.), Combinatorial Pattern Matching, in: Lecture Notes in Comput. Sci., vol. 4009, Springer, Berlin/Heidelberg, 2006, pp. 330–341.
[3] W.-T. Chan, Y. Zhang, S. Fung, D. Ye, H. Zhu, Efficient algorithms for finding a longest common increasing subsequence, in: X. Deng, D.-Z. Du (Eds.), Algorithms and Computation, in: Lecture Notes in Comput. Sci., vol. 3827, Springer, Berlin/Heidelberg, 2005, pp. 665–674.
[4] M. Crochemore, G.M. Landau, M. Ziv-Ukelson, A subquadratic sequence alignment algorithm for unrestricted scoring matrices, SIAM J. Comput. 32 (2003).
[5] M. Crochemore, E. Porat, Fast computation of a longest increasing subsequence and application, Inform. and Comput. 208 (9) (2010) 1054–1059.
[6] A. Elmasry, The longest almost-increasing subsequence, Inform. Process. Lett. 110 (16) (2010) 655–658.
[7] J.W. Hunt, T.G. Szymanski, A fast algorithm for computing longest subsequences, Commun. ACM 20 (5) (1977) 350–353.
[8] C.S. Iliopoulos, M.S. Rahman, A new efficient algorithm for computing the longest common subsequence, Theory Comput. Syst. 45 (2) (2009) 355–371.
[9] D.E. Knuth, The Art of Computer Programming, vol. 3: Sorting and Searching, 2nd ed., Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.
[10] W.J. Masek, M.S. Paterson, A faster algorithm computing string edit distances, J. Comput. System Sci. 20 (1) (1980) 18–31.
[11] G.d. Robinson, On the representations of the symmetric group, Amer. J. Math. 60 (1938) 745–760.
[12] Y. Sakai, A linear space algorithm for computing a longest common increasing subsequence, Inform. Process. Lett. 99 (5) (2006) 203–207.
[13] C. Schensted, Longest increasing and decreasing subsequences, Canad. J. Math. 13 (1961) 179–191.
[14] I.-H. Yang, C.-P. Huang, K.-M. Chao, A fast algorithm for computing a longest common increasing subsequence, Inform. Process. Lett. 93 (5) (2005) 249–253.