J Comb Optim
DOI 10.1007/s10878-017-0170-9

The longest commonly positioned increasing subsequences problem

Xiaozhou He[1] · **Yinfeng Xu**[1,2]

Abstract Based on the well-known longest increasing subsequence problem and longest common increasing subsequence (LCIS) problem, we propose the longest commonly positioned increasing subsequences (LCPIS) problem. Let $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_n \rangle$ be two input sequences. Let $Asub = \langle a_{i_1}, a_{i_2}, \ldots, a_{i_l} \rangle$ be a subsequence of A and $Bsub = \langle b_{j_1}, b_{j_2}, \ldots, b_{j_l} \rangle$ be a subsequence of B such that $a_{i_k} \leq a_{i_{k+1}}, b_{j_k} \leq b_{j_{k+1}} (1 \leq k < l)$, and a_{i_k} and b_{j_k} $(1 \leq k \leq l)$ are commonly positioned (have the same index $i_k = j_k$) in A and B respectively but these two elements do not need to be equal. The LCPIS problem aims at finding a pair of subsequences $Asub$ and $Bsub$ as long as possible. When all the elements of the two input sequences are positive integers, this paper presents an algorithm with $O(n \log n \log \log M)$ time to compute the LCPIS, where $M = min\{max_{1 \leq i \leq n} a_i, max_{1 \leq j \leq n} b_j\}$. And we also show a dual relationship between the LCPIS problem and the LCIS problem.

Keywords Longest increasing subsequence · Common positions · Algorithms · Dual relationship · Longest common increasing subsequence

1 Introduction

The two classic computer science problems—longest increasing subsequence (LIS) problem and the longest common subsequence (LCS) problem—have been studied

✉ Xiaozhou He
xiaozhouhe126@qq.com

Yinfeng Xu
yinfengxu126@163.com

1 Business School, Sichuan University, Chengdu, China

2 State Key Lab for Manufacturing Systems Engineering, Xi'an, China

Published online: 09 September 2017

for decades. The LCS problem, which is to find the length of the longest subsequence common to both two given sequences, has been studied by many such as Masek and Paterson (1980) and Bergroth et al. (2000). The LIS problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are in ascending order (Crochemore and Porat 2010; Fredman 1975).

Based on the two classical problems above, there is the longest common increasing subsequence (LCIS) problem which is to find a longest common subsequence that is monotone increasing of two given sequences. Let m and n ($m \leq n$) be the lengths of two input sequences respectively, and let l be the length of the output LCIS. Yang et al. (2005) defined this problem and solved it using an algorithm within $O(mn)$ time and space. Then, when $m = n$ and r, the number of ordered pairs of positions at which the two sequences match, is relatively small, Chan et al. (2007) gave an improved algorithm which runs in $O(min(r \log l, nl + r) \log \log n + n \log n)$ time. Later, if l is small, Kutz et al. (2011) solved the problem with an algorithm running in $O((m + nl) \log \log \sigma + m \log m)$ time and $O(m)$ space, where σ is the size of the alphabet.

In this paper we propose the longest commonly positioned increasing subsequences (LCPIS) problem. This problem aims at finding a pair of increasing subsequences as long as possible from two input sequences such that the subsequences are constituted by certain pairs of elements which are commonly positioned (i.e., have the same index) in the two input sequences. This problem may appear in the model abstracted from the analysis of securities market when we study the relationship between the prices of two securities. A typical scenario is analyzing the long-time relationship between a security and a relative index such as a mining company's stock price and the industrial production index. If we want to buy this stock and hold it for a long time, it would be better that the stock price has a rising trend in a long period, during which time we do not require that price continuously increases without breaks because some short-term price fluctuation does not have impact on the ultimate profit. Thus this long-term rising trend can be described by a long enough increasing subsequence of the price sequence. As the mining company performance is positively correlated with the industrial production index, we assume that the stock and the index have the same trend. Therefore, we abstract their similar rising trend in a long period by using a pair of commonly positioned increasing subsequences of the stock price sequence and the index sequence. This kind of scenario motivates this LCPIS problem.

By using the subsequence to reflect the similarity of two sequences, LCS and LCIS describe the similarity on the element values of the two sequences, while LCPIS shows the similarity on the position of the elements of the two sequences.

Let $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_n \rangle$ be two input sequences. Based on the characteristic of the LCPIS, we can find it by processing A and B synchronously in terms of the indexes. Then we can compute a LCPIS by applying the well-known LIS-algorithm which maintains a list such that the jth element of this list is the smallest ending number of j-length increasing subsequence (Fredman 1975). Obviously, we must do some modification to the LIS-algorithm to fit this LCPIS algorithm, since each element of the LCPIS consists of two numbers. From this, we notice that the efficiency of solving a LCPIS problem may be improved if there is any improvement on the solution of the LIS problem.

The paper is organized as follows. In the next section, we state problem definitions. Based on the well-known LIS algorithm, Sect. 3 presents the algorithm for this LCPIS problem. In Sect. 4, we give some supplementary proof of the correctness of the algorithm and prove the time complexity. In Sect. 5, we present the dual relationship between the LCPIS problem and the LCIS problem.

2 Definitions

Let $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_n \rangle$ be two input sequences. And let $Asub = \langle a_{i_1}, a_{i_2}, \ldots, a_{i_l} \rangle$ and $Bsub = \langle b_{j_1}, b_{j_2}, \ldots, b_{j_l} \rangle$ be respectively a subsequence A and B. We give the following definitions.

LCPIS We give the definition of the LCPIS compared with the previous LCS and LCIS which also describe sequence similarity.

(i) *Asub* and *Bsub* are a pair of LCPIS of A and B if *Asub* and *Bsub* are sequences as long as possible such that $i_k = j_k$, $1 \le k \le l$ and $a_{i_k} \le a_{i_{k+1}}$, $b_{i_k} \le b_{i_{k+1}}$, $1 \le k < l$.

(ii) *Asub* and *Bsub* are a LCS of A and B if $Asub = Bsub$, i.e., $a_{i_k} = b_{j_k}$, $1 \le k \le l$, as long as possible.

(iii) *Asub* and *Bsub* are a LCIS of A and B if $Asub = Bsub$ as long as possible such that $a_{i_k} \le a_{i_{k+1}}$, $1 \le k < l$.

Example 1 Let $A = \langle 1, 3, 6, 4, 5, 2, 5, 9, 7, 8 \rangle$ and $B = \langle 2, 4, 3, 5, 3, 7, 2, 1, 6, 8 \rangle$ be two sequences both consisting of ten positive integers. $Asub = \langle 1, 3, 4, 7, 8 \rangle$ and $Bsub = \langle 2, 4, 5, 6, 8 \rangle$ are a pair of LCPIS of A and B, where the elements of *Asub* and *Bsub* are the first, second, fourth, ninth and tenth numbers of A and B respectively. By contrast, $\langle 3, 5, 7, 8 \rangle$ is a LCIS, and $\langle 4, 5, 2, 8 \rangle$ is a LCS. □

Couple For $1 \le i \le n$, we say that (a_i, b_i), where a_i and b_i are commonly positioned (have the same index) in A and B respectively, is a *couple*. We also say that a_i and b_i are the *first number* and *the second number* of this couple, respectively.

Partial order relaitons For any two couples (a_p, b_p) and (a_q, b_q), we say that (a_p, b_p) is *weaker than* (a_q, b_q) or (a_q, b_q) is *stronger than* (a_p, b_p) if and only if $a_p \le a_q$ and $b_p \le b_q$.

We process A and B synchronously in terms of the index. Hence we can draw on the method used to solve the LIS problem, and thus, we make some modification to the LIS-problem-algorithm used by Fredman (1975) to fit this LCPIS problem. Fredman maintains a table $T(j)$ $(1 \le j \le n)$ to store the smallest ending number of all the j-length increasing subsequences and compute $T(j)$ $(1 \le j \le n)$ by setting them as the first row of the Young tableau. Similarly, we give the following definition.

Smallest ending couple First we call the couple made up of the two ending numbers of a pair of CPIS the *ending couple*. Then we say that an *ending couple* is a *smallest ending couple* of all pairs of j-length CPIS if there is not any other *ending couple* of a pair of j-length CPIS weaker than it.

There are likely more than one *smallest ending couples* of all pairs of j-length CPIS since not every two couples are comparable, which is the main difference between LCPIS and LIS when designing an algorithm.

Smallest ending sets We define a series of *smallest ending sets* $U_j, 1 \leq j \leq n$, where U_j $(1 \leq j \leq n)$ is the set of all the *smallest ending couples* of all pairs of j-length CPIS.

Remark Assuming that the output LCPIS is of L-length, U_j $(L < j \leq n)$ is an empty set.

For the sake of finding a pair of LCPIS, we use an array $Link[i]$ $(1 \leq i \leq n)$ to record the index of the couple immediately preceding (a_i, b_i) in a pair of CPIS in the step that (a_i, b_i) proceeds.

3 Algorithm

Lemma 1 *For arbitrary two couples in any set, their first numbers (or their second numbers) are not equal.*

Proof It is easy to prove this lemma by contradiction. Assume that there is two couples such as (a_p, b_p) and (a_q, b_q) $(p < q)$ in a set such that their first number is equal: $a_p = a_q$. Thus, if $b_p \leq b_q$, then (a_p, b_p) is weaker than (a_q, b_q) which means (a_q, b_q) can rank after (a_p, b_p) in one CPIS. Else if $b_p > b_q$, then (a_p, b_p) is stronger than (a_q, b_q). In this case, although (a_p, b_p) and (a_q, b_q) can be ending couples of CPIS of the same length, (a_p, b_p) is not a *smallest ending couple*. So the assumption does not hold and similar result goes to the 'second number'. □

Lemma 2 *If we sort the couples in any set with their first numbers, then the second numbers of these couples are in the descending order.*

Proof Let the *smallest ending couples* of all pairs of k-length CPIS in U_k be $\{(a_{i_1}, b_{i_1}), (a_{i_2}, b_{i_2}), \ldots, (a_{i_t}, b_{i_t})\}$. So we can sort the couples in U_k as $\langle (a_{j_1}, b_{j_1}), (a_{j_2}, b_{j_2}), \ldots, (a_{j_t}, b_{j_t})\rangle$ such that $a_{j_1} < a_{j_2} < \cdots < a_{j_t}$ and our aim is to prove $b_{j_1} > b_{j_2} > \cdots > b_{j_t}$. For any two couples (a_{j_p}, b_{j_p}) and (a_{j_q}, b_{j_q}) $(1 \leq p < q \leq t)$ in U_k, there must be $b_{j_p} > b_{j_q}$ since $a_{j_p} < a_{j_q}$. Otherwise, (a_{j_q}, b_{j_q}) is not a *smallest ending couple*. Thereby, $b_{j_1} > b_{j_2} > \cdots > b_{j_t}$ is proved. □

To put $(a_i, b_i), 1 \leq i \leq n$ one by one into the proper set, by Lemma 2, we can construct n vEB trees $T_1, T_2, \ldots, T_n$ to store the *smallest ending couples* belonging to $U_1, U_2, \ldots, U_n$ respectively and insert $(a_i, b_i), 1 \leq i \leq n$ in the corresponding tree. Since we use vEB tree, this algorithm can only be implemented when the elements of the input sequences are all integers. Without loss of generality, we assume that the largest number of A is smaller than that of B. Thus, in each tree we index the *smallest ending couples* by their first numbers. So the universe size of each tree is $M = max_{1 \leq i \leq n} a_i$. A vEB tree is the data structure proposed by van Emde Boas (1977) that can support operations of an ordered integral array W. Four operations that vEB tree can implement are in the following: (let the largest number in W be V)

- *insert (x)* which adds x to W in $O(\log \log V)$ time;
- *delete (x)* which removes x from W in $O(\log \log V)$ time;

- *predecessor (x)* which returns the largest number of w less than x in $O(\log\log V)$ time;
- *successor (x)* which returns the smallest number of W greater than x in $O(\log\log V)$ time.

In the first implement of the algorithm, we store (a_1, b_1) into T_1, and set $T_2, \ldots, T_n$ be empty vEB trees. Then as i proceeds from 2 until n, perform a binary search on the trees that are non-empty at the beginning of that loop, which are assumed to be $T_1, \ldots, T_k$ $(1 \le k \le i)$, to find the largest index t such that T_t $(1 \le t \le k)$ has a couple weaker than (a_i, b_i), then insert (a_i, b_i) into T_{t+1} and remove couples that are no longer *smallest ending couples* from T_{t+1} in the loop.

往後移

Before giving the algorithm, we prove some lemmas for the feasibility of the insertion:

Claim *In each loop, for any (a_i, b_i) in the non-empty tree T_p, there is at least one couple in the tree T_{p-1} weaker than (a_i, b_i).*

Proof In the process of inserting (a_i, b_i) into tree T_p in previous loop, we must first find a couple in T_{p-1} weaker than (a_i, b_i). We assume this weaker couple to be (a_j, b_j). If (a_j, b_j) is still in T_{p-1} in the loop the *Claim* discuss, then (a_j, b_j) is the couple we need. Else if (a_j, b_j) has been removed, which means (a_j, b_j) is no longer a *smallest ending couple* in T_{p-1}, then there must be another couple (a_k, b_k) inserted into T_{p-1} and weaker than (a_j, b_j). Thus, (a_k, b_k) is also weaker than (a_i, b_i). □

Lemma 3 *In each loop, for any couple (a_i, b_i) and any two non-empty trees T_p and T_q $(1 \le p < q \le n)$:*

(i) *if there is a couple in T_p weaker than (a_i, b_i), then there is a couple in each tree among $T_1, \ldots, T_p$ weaker than (a_i, b_i);*
(ii) *if there is no couple in T_q weaker than (a_i, b_i), then there is no couple in $T_q, \ldots, T_n$ weaker than (a_i, b_i).*

Proof It is easy to prove (i) by using Claim 1 repeatedly. We can prove (ii) by contradiction. If there is a couple in T_r $(r > q)$ weaker than (a_i, b_i), then by using (i) $q - p$ times, there is a couple in T_q weaker than (a_i, b_i), which contradicts the assumption. □

Corollary 1 *We assume that $T_1, \ldots, T_k (k \le i-1)$ are non-empty trees and the rest are empty after $(a_1, b_1), \ldots, (a_{i-1}, b_{i-1})$ have been processed. There are three cases when inserting (a_i, b_i):*

(i) *If there is no couple in T_1 weaker than (a_i, b_i), then there is no couple in all the trees weaker than (a_i, b_i). So (a_i, b_i) cannot be a smallest ending couple of a pair of CPIS longer than 1 and (a_i, b_i) should be inserted into T_1.*
(ii) *Else if there is a couple in T_k weaker than (a_i, b_i), then (a_i, b_i) is the smallest ending couple of a pair of $k+1$-length CPIS the other k numbers of this pair come from $T_1, \ldots, T_t$ respectively. Thus, (a_i, b_i) should be inserted into T_{k+1} as its first element.*

(iii) *Otherwise, there must be exactly one tree* T_t $(1 \leq t < k)$ *such that there is a couple in each tree among* $T_1, \ldots, T_t$ *weaker than* (a_i, b_i) *and there is no couple in* $T_{t+1}, \ldots, T_k$ *weaker than* (a_i, b_i). *In this case,* (a_i, b_i) *is the smallest ending couple of a pair of* $t+1$*-length CPIS and* (a_i, b_i) *should be inserted into* T_{t+1}.

□

In order to find the tree T_{t+1} that (a_i, b_i) is inserted into, by Lemma 3, we can apply binary search on $T_1, \ldots, T_k$. Firstly, we determine if there is a couple in tree $T_{\lfloor k/2 \rfloor}$ weaker than (a_i, b_i). Since $T_{\lfloor k/2 \rfloor}$ is a vEB tree and the *smallest ending couples* in it is sorted with their *first numbers*, we can use operation *predecessor* (a_i) on these couples to find the couple with the largest *first number* which is not greater than a_i, and we assume this couple to be (a_x, b_x). And by Lemma 2, b_x is the smallest second number among those couples that have the *first numbers* less than a_i. Thus, we compare b_x with b_i. If $b_x \leq b_i$, then $T_{\lfloor k/2 \rfloor}$ contains (a_x, b_x) weaker than (a_i, b_i) and we continue this search on $T_{\lfloor k/2 \rfloor + 1}, \ldots, T_k$. Otherwise, there is no couple in $T_{\lfloor k/2 \rfloor}$ weaker than (a_i, b_i) and we continue this search on $T_1, \ldots, T_{\lfloor k/2 \rfloor - 1}$.

Example 2 *In the loop* i, *for a tree* T_p, *we have to determine if there is a couple in* T_p *weaker than* (a_i, b_i). Assume that T_p consists of $\{(a_{p_1}, b_{p_1}), \ldots, (a_{p_m}, b_{p_m})\}$ such that $a_{p_1} < \cdots < a_{p_m}$ and $b_{p_1} > \cdots > b_{p_m}$. First, find the largest *first number* which is not greater than a_i (assumed to be a_{p_x}), which means $a_{p_1} < \cdots < a_{p_x} \leq a_i < a_{p_{x+1}} < \cdots < a_{p_m}$. Then, compare b_{p_x} with b_i: if b_{p_x} is not greater than b_i, then at least (a_{p_x}, b_{p_x}) weaker than (a_i, b_i); otherwise, there is no couple in T_p weaker than (a_i, b_i) for $b_{p_1} > \cdots > b_{p_x}$. □

Remark Example 2 gives a detailed illustration to determine whether there is a couple in any tree weaker than (a_i, b_i) in the loop i.

After completing this search and getting T_t, we assign $Link[i]$ the index of the couple that is weaker than (a_i, b_i) in T_t, and use the operation *insert* (a_i) to insert (a_i, b_i) into tree T_{t+1}. Then, by Lemma 2, we use operation *predecessor* (x) $(x = a_i$ initially) to find the next couple with a greater *first number* one by one and use operation *delete*(x) to remove them if they have a second number greater than b_i until the first couple that has a second number less than b_i. remove (>ai, >bi)

Example 3 In the loop i, assume that T_{t+1} consists of $\{(a_{t_1}, b_{t_1}), \ldots, (a_{t_q}, b_{t_q})\}$ before inserting (a_i, b_i) such that $a_{t_1} < \cdots < a_{t_q}$ and $b_{t_1} > \cdots > b_{t_q}$. Also assume that $a_{t_1} < \cdots < a_{t_y} \leq a_i < a_{t_{y+1}} < \cdots < a_{t_q}$ and $b_{t_1} > \cdots > b_{t_z} > b_i \geq b_{t_{z+1}} > \cdots > b_{t_q}$ such that $y \leq z$. Thus, after inserting (a_i, b_i) into T_{t+1}, $\{(a_{t_{y+1}}, b_{t_{y+1}}), \ldots, (a_{t_z}, b_{t_q})\}$ are not *smallest ending couples* anymore for they are weaker than (a_i, b_i). So we should remove them from T_{t+1}. □

Remark Example 3 illustrates why we may remove some couples from the tree after we insert (a_i, b_i) into it in the loop i.

After all the couples from A and B are processed, L (i.e., the length of a pair of LCPIS) turns out to be the number of non-empty trees. Also, We can arbitrarily choose

a couple from the last non-empty tree and find the couple ranking right in front of it in a pair of LCPIS by $Link[i]$ $(1 \leq i \leq n)$, and then repeat this process $L-1$ times to form a pair of LCPIS.

See Algorithm 1, the brief process of the algorithm.

Algorithm 1 the longest pair of CPIS of two sequences A, B

Input: two positive integral sequences: $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_n \rangle$
Output: the length of a pair of LCPIS L; a pair of LCPIS: A_{sub} and B_{sub}
1: compute $max_{1 \leq i \leq n} a_i$ and $max_{1 \leq i \leq n} b_i$, and get M
2: construct n vEB trees $T_1, T_2, \ldots, T_n$ to store the *smallest ending couples* of increasing subsequences of length $1, 2, \ldots, n$ respectively
3: store (a_1, b_1) in T_1 and it is indexed by a_i
4: **for** $i = 2$ to n **do**
5: find the largest index t such that T_t has a couple weaker than (a_i, b_i) by using binary search on non-empty trees;
6: assign the index of the couple in T_t that is weaker than (a_i, b_i) to $Link[i]$;
7: insert (a_i, b_i) to T_{t+1} and then remove those couples that are no longer *smallest ending couples* in T_{t+1}
8: **end for**
9: $L :=$ the number of non-empty trees
10: choose a couple from the last non-empty tree arbitrarily and form a pair of LCPIS, A_{sub} and B_{sub}, by $Link[i]$

4 The correctness and time complexity of the algorithm

By Lemma 3, we can get a pair of CPIS by choosing one couple from each non-empty tree. Now we prove that there cannot be a pair of CPIS with a length longer than L (i.e., the number of non-empty trees) by contradiction. If there exists such a longer pair of CPIS, by the pigeon hole principle, two couples of this pair must be members (including those are removed later) of a same tree, since every couple is or has been inserted into a tree though some of them are removed later. In any tree, only those 'removed' couples can be stronger than the 'later-inserted' couples, but the first number (resp. second number) of every 'later-inserted' couple ranks after that of 'removed couples' in the input sequence A (resp. B). So there cannot be more than one couple in a tree appear in a pair of CPIS, which contradicts that a pair of CPIS can be longer than L.

Theorem 1 *The LCPIS of two sequences of n positive integers can be computed in* $O(n \log n \log \log M)$ *time, where* M *is the less one of the maximum of* A *and the maximum of* B.

Proof The whole implementation can be divided into four parts as follows:

① At the beginning of the algorithm, we find the maximum of A and the maximum of B in $O(n)$ time.
② Next, we spend $O(\log n)$ time applying binary search on n trees to determine which tree (a_i, b_i) should be inserted to, and for each tree we spend $O(\log \log M)$

time finding the proper couple to be compared with (a_i, b_i). In the loop that i proceeds from 2 to n, this process takes $O(n \log n \log\log M)$ time.

③ Then, after finding the tree that (a_i, b_i) should be inserted to, we insert (a_i, b_i) in this tree and remove those couples that are not a *smallest ending couple* anymore. Even though we do not know how many couples we should remove in one step, there are at most n couples being removed in all steps since every couple can be inserted and removed at most one time. Since inserting one couple, finding a couple needed to be removed, and removing a couple each needs $O(\log\log M)$ time, it takes $O(n \log\log M)$ time to complete this process in the loop.

④ At last, it takes $O(n)$ time to find the length of a pair of longest subsequences and output a pair of LCPIS.

All in all, the algorithm takes $O(n \log n \log\log M)$ time. □

5 The dual relationship between the LCPIS and LCIS

Notice that LCPIS is to find a pair of subsequences, while LCIS is to find a common subsequence. Nevertheless, these two problems are closely related to each other. In this section, we establish a certain dual relationship between the common increasing subsequence (CIS) and the commonly positioned increasing subsequence (CPIS) by the indexes. To describe concisely, the definitions of *index sequence* and *dual sequence* are used.

Index sequence For any sequence $S = \langle s_1, \ldots, s_n \rangle$ and any S's subsequence $Ssub = \langle s_{i_1}, s_{i_2}, \ldots, s_{i_l} \rangle$, we name $\langle i_1, i_2, \ldots, i_l \rangle$ the *index sequence* of $Ssub$. Similarly, after sorting S to be in ascending order to be $S^* = \langle s_{j_1}, s_{j_2}, \ldots, s_{j_n} \rangle$, we name $\langle j_1, j_2, \ldots, j_n \rangle$ the *index sequence* of S^*.

Remark When we sort S, if there is two equal elements, then it is necessary to make sure that the order of these two elements does not change in S^*. Formally speaking, if there exists $s_p = s_q$, $p < q$ in S, then in S^*, $s_p = s_{j_k}$, $s_q = s_{j_{k+1}}$ for some k.

Dual sequence Using the notation in the last definition, we say that the *index sequence* of S^*, i.e., $\langle j_1, j_2, \ldots, j_n \rangle$, is the *dual sequence* of S. This dual sequence is denoted as D_S.

Let $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_n \rangle$ be two given input sequences and $Asub = \langle a_{x_1}, a_{x_2}, \ldots, a_{x_l} \rangle$ and $Bsub = \langle b_{y_1}, b_{y_2}, \ldots, b_{y_l} \rangle$ be an increasing subsequence of them, respectively. The *index sequences* of *Asub* and *Bsub* are $I_{Asub} = \langle x_1, x_2, \ldots, x_l \rangle$ and $I_{Bsub} = \langle y_1, y_2, \ldots, y_l \rangle$, respectively.

Sort A and B in ascending order, respectively, to be $A^* = \langle a_{i_1}, a_{i_2}, \ldots, a_{i_n} \rangle$ and $B^* = \langle b_{j_1}, b_{j_2}, \ldots, b_{j_n} \rangle$. And thus, the *dual sequences* of A and B are $D_A = \langle i_1, i_2, \ldots, i_n \rangle$ and $D_B = \langle j_1, j_2, \ldots, j_n \rangle$, respectively.

Firstly, we describe the relationship between the CIS of A and B with the CPIS of their *dual sequences*.

Lemma 4 *I_{Asub} (resp. I_{Bsub}) is an increasing subsequence of D_A (resp. D_B).*

Proof *Asub* is a subsequence of A^*, since *Asub* is an increasing subsequence of A and A^* is the ascending order of A. Hence, I_{Asub} is a subsequence of D_A. In addition,

I_{Asub} is naturally in ascending order. Therefore, I_{Asub} is an increasing subsequence of D_A. Similarly, we can get the corresponding consequence on I_{Bsub}. □

Lemma 5 *If (i) A and B are two permutations of* $\{1, 2, \ldots, n\}$*, and (ii) Asub* $=$ *Bsub is a CIS of A and B, then* I_{Asub} *and* I_{Bsub} *make up a pair of CPIS of the dual sequences* D_A *and* D_B*.*

Proof The position that a_{x_k} ranks in A^* is the same with that b_{y_k} ranks in B^*, because $Asub = Bsub$, i.e., $a_{x_k} = b_{y_k}, k = 1, \ldots, l$, and $A^* = B^* = \langle 1, 2, \ldots, n\rangle$. Hence, x_k and $y_k, k = 1, \ldots, l$ rank the identical position in D_A and D_B, respectively. And with Lemma 4, we prove this lemma. □

Secondly, we describe the relationship between the CPIS of A and B with the CIS of their *dual sequences*.

Lemma 6 *If Asub and Bsub make up a pair of CPIS of A and B, then* I_{Asub} *and* I_{Bsub} *is a CIS of the dual sequences* D_A *and* D_B*.*

Proof According the condition, $x_k = y_k, k = 1, \ldots, l$, i.e., $I_{Asub} = I_{Bsub}$. And with Lemma 4, we prove this lemma. □

On the basis of Lemmas 5 and 6, it is not hard to get the following conclusion.

Theorem 2 *For two given sequences A and B, we have:*

(i) *The LCPIS of A and B have the same length with the LCIS of their dual sequences;*
(ii) *If A and B are two permutations of* $\{1, 2, \ldots, n\}$*, the LCIS of A and B have the same length with the LCPIS of their dual sequences.*

Through Theorem 2, we know that any LCPIS problem can be converted to a LCIS problem which has been solved by some algorithms (see Sect. 1), and vise versa. Thus, any improvement on the complexity of the algorithm solving one of these two problems can improve the efficiency or space usage of solving the other problem.

Acknowledgements This research was supported by the National Natural Science Foundation of China under Grant 71371129.

References

Bergroth L, Hakonen H, Raita T (2000) A survey of longest common subsequence algorithms. In: String processing and information retrieval, SPIRE 2000. Proceedings. Seventh international symposium on, IEEE. pp 39–48

Chan WT, Zhang Y, Fung SP, Ye D, Zhu H (2007) Efficient algorithms for finding a longest common increasing subsequence. J Comb Optim 13(3):277–288

Crochemore M, Porat E (2010) Fast computation of a longest increasing subsequence and application. Inf Comput 208(9):1054–1059

Fredman ML (1975) On computing the length of longest increasing subsequences. Discrete Math 11(1):29–35

Kutz M, Brodal GS, Kaligosi K, Katriel I (2011) Faster algorithms for computing longest common increasing subsequences. J Discrete Algorithms 9(4):314–325

Masek WJ, Paterson MS (1980) A faster algorithm computing string edit distances. J Comput Syst Sci 20(1):18–31

van Emde Boas P (1977) Preserving order in a forest in less than logarithmic time and linear space. Inf Process Lett 6(3):80–82

Yang IH, Huang CP, Chao KM (2005) A fast algorithm for computing a longest common increasing subsequence. Inf Process Lett 93(5):249–253