An Efficient Algorithm for Enumerating Longest Common Increasing Subsequences

Chun Lin(✉), Chao-Yuan Huang, and Ming-Jer Tsai

National Tsing Hua University, Hsinchu, Taiwan
`s108062571@m108.nthu.edu.tw`

Abstract. The longest common increasing subsequence (LCIS) problem is the combination of two classic problems in algorithms: the longest increasing subsequence (LIS) problem and the longest common subsequence (LCS) problem. In this paper, we propose an algorithm that finds every LCIS of two sequences a, b of length n in $O(n + \sigma + I_a)$ time and space, where σ denotes the size of the alphabet set and I_a the total number of increasing subsequences contained in a (thus, the running time is output-sensitive). Our algorithm employs the trie and some simple data structures, and thus is implementation-wise simple. In addition, it can be proved that our algorithm is optimal in time complexity when $\sigma \leq \log_2 n$.

Keywords: LCIS · Trie · Data structure

1 Introduction

The longest common increasing subsequence (LCIS) problem can be formulated as follows: Given a sequence $a = a_1, a_2, \cdots, a_n$, a sequence $a_{i_1}, a_{i_2}, \cdots, a_{i_k}$ is a subsequence of a if $1 \leq i_j < i_{j+1} \leq n$ for all $1 \leq j < k$. And, given two sequences a, b of length n, the LCIS problem asks for a longest common subsequence of a, b that is strictly increasing.

This problem can be seen as a combination of the longest increasing subsequence (LIS) problem and the longest common subsequence (LCS) problem, and was first introduced by Yang et al. [6] and then applied to the whole genome alignment by Chan et al. [1] in 2005. Yang et al. and Chan et al. proposed algorithms of $O(n^2)$ and $O(min(r \log \sigma, n\sigma + r) \log \log n + Sort_n)$ time, respectively, where $Sort_n$ denotes the time required to sort input sequences a, b, and r the number of ordered pairs (i, j) such that $a_i = b_j$. In 2006, Sakai presented a linear-space and $O(n^2)$-time algorithm using a divide-and-conquer approach [5]. In 2011, Kutz et al. designed an algorithm of $O(n)$ space and $O(nl \log \log \sigma + Sort_n)$ time [3], where l denotes the length of the LCIS of a, b. And, for small alphabet set, algorithms of $O(n)$ and $O(n \log \log n)$ time were proposed for $\sigma = 2$ and $\sigma = 3$, respectively. In 2016, Zhu et al. proposed an

C.-Y. Chen et al. (Eds.): COCOON 2021, LNCS 13025, pp. 25–36, 2021.
https://doi.org/10.1007/978-3-030-89543-3_3

$O(n^2)$-time and linear-space algorithm [7]. Recently, in 2020, Lo et al. proposed an algorithm of $O(n + l(n - l) \log\log \sigma)$ time and $O(n)$ space [4], and Duraj presented the first algorithm of subquadratic time [2].

The rest of this paper is organized as follows: In Sect. 2, the proposed algorithm is presented. In Sect. 3, the correctness and complexity are analyzed. Finally, we conclude this paper in Sect. 4.

2 The Proposed Algorithm

In this section, three assumptions are first introduced. Subsequently, we outline the proposed algorithm, followed by a step-by-step explanation along with the pseudocode. Finally, an example is given.

2.1 Assumptions

Input Format. Given the size of the alphabet set σ, we assume the alphabet set consists of integers $0, 1, ..., \sigma - 1$, i.e., each integer in the input sequences a, b is in $\{0, 1, ..., \sigma - 1\}$.

Fast Computation. We assume that the bitwise shift (or bitwise OR) on one (or two) binary encoded data of no more than σ bits can be done in $O(1)$ time.

Constant Space. We assume that a bitstring of length up to σ takes $O(1)$ space.

Remark that when the desired input format is not satisfied, one can map integers in a, b to the integers in $[0, \sigma - 1]$ without changing the order of integers in $O(n \log \sigma)$ time using a balanced binary search tree.

2.2 Algorithm Overview

The main procedure of the proposed algorithm (Algorithm 1) involves building a trie T containing the information of every increasing subsequence in a. Let IS_u denote the *increasing subsequence* with *binary encoding* u, i.e., the i-th bit in u is 1 if and only if i is contained in IS_u. For example, $IS_{10100100}$ denotes the increasing subsequence $[2, 5, 7]$ as $\sigma = 8$. Then, T has the following properties:

1. A node T_u associated with the length l_u of IS_u exists in T to denote an *increasing subsequence* IS_u if and only if IS_u is found in a. Also, the binary encoding u of IS_u is stored in T_u to help retrieval of sequence information.
2. A directed edge associated with $x \in \{0, 1, ..., \sigma - 1\}$ from T_u to T_v, denoted by the tuple (T_u, T_v, x), exists in T if and only if IS_v is the concatenation of IS_u and x.

After building T, a similar trie-building process is run for sequence b; but instead of building a new trie for b, we walk along the nodes of T that denote the common increasing subsequences of a, b, and meanwhile record all the found longest common increasing subsequences of a, b.

For the complexity of Algorithm 1, the initialization step takes $O(\sigma + n)$ time. Building T takes $O(n + I_a)$ time by using additional data structures that take $O(\sigma + I_a)$ space. And, walking on T takes $O(n + I_a)$ time. To sum up, Algorithm 1 has space and time complexity of $O(n + \sigma + I_a)$.

2.3 Detailed Description

See Algorithm 1 for the pseudocode. Algorithm 1 consists of 5 parts as follows.

Input/Output. Algorithm 1 takes two sequences a, b, the length n of a, b, and the size of the alphabet set σ as the inputs, and outputs a list L containing the binary encoding of every LCIS of a, b.

Initialization for First Loop (Lines 2–19). Firstly, build an array Cnt, where $Cnt[i]$ is the frequency of i in a for all $i \in \{0, 1, ..., \sigma - 1\}$. This can be done in $O(n)$ time by simply scanning a once. Secondly, build a doubly linked list K of nodes to store every integer i with $Cnt[i] > 0$ in an increasing order (from $i = 0$ to $i = \sigma - 1$). Then, a pointer array M of size σ is created. And, for each integer i stored in K, the pointer to the node containing i in K is stored in $M[i]$ so that the node can be removed from K, if necessary, in $O(1)$ time. Thirdly, build the root node T_0 of T, which denotes an empty sequence, and set l_0 to 0. Then, for every i with $Cnt[i] > 0$, create a trie node T_{2^i} containing the binary encoding of the sequence $[i]$, set $l_{2^i} = 1$, and add an edge (T_0, T_{2^i}, i) from T_0 to T_{2^i}. Finally, build σ queues $Next_0, \cdots, Next_{\sigma-1}$, where each queue supports $O(1)$ push and pop (for our purpose, one can also use different data structures such as stacks or dynamic arrays, as long as they support push and pop in $O(1)$ time). Let A_u be the address of T_u. Then, for all i, queue $Next_i$ initially contains A_{2^i} if $Cnt[i] > 0$, and is left empty otherwise.

First Loop (Lines 20–31). Algorithm 1 iterates the following two steps when sequence a is scanned one by one from left to right. Firstly, for the i-th integer a_i in a, we decrease $Cnt[a_i]$ by 1. Secondly, in the inner loop, Algorithm 1 iterates the following two substeps until queue $Next_{a_i}$ is empty. First pop A_u from queue $Next_{a_i}$ and get u from T_u. Then, in the (yet deeper) inner loop, for each integer x with $a_i < x < \sigma$ and $Cnt[x] > 0$ (every such x can be found efficiently using K), first create a new node T_v of T, set l_v to $l_u + 1$, add an edge (T_u, T_v, x) from T_u to T_v, where $v = u + 2^x$, and then push A_v into queue $Next_x$. At last, at the end of the i-th iteration, remove the node containing a_i from K if $Cnt[a_i]$ has become 0.

Algorithm 1: LCIS

```
Input: $(n, \sigma, a, b)$: the length of each sequence, the size of the alphabet set,
       the two sequences
Output: $L$: a list containing the binary code of every LCIS of $(a, b)$
begin
    $Cnt \leftarrow$ new 1D integer array of size $\sigma$;
    $M \leftarrow$ new 1D pointer array of size $\sigma$;
    for $i \leftarrow 0$ to $\sigma - 1$ do
        $Cnt[i] \leftarrow 0$;
    for $i \leftarrow 1$ to $n$ do
        $Cnt[a_i] \leftarrow Cnt[a_i] + 1$;
    $K \leftarrow$ new doubly linked list;
    create trie node $T_0$;
    $l_0 \leftarrow 0$;
    for $i \leftarrow 0$ to $\sigma - 1$ do
        $Next_i \leftarrow$ new queue;
        if $Cnt[i] > 0$ then
            add the node $K_i$ containing $i$ to $K$;
            $M[i] \leftarrow$ address of $K_i$;
            create trie node $T_{2^i}$;
            $l_{2^i} \leftarrow 1$;
            add edge $(T_0, T_{2^i}, i)$;
            push $A_{2^i}$ into $Next_i$;
    for $i \leftarrow 1$ to $n$ do
        $Cnt[a_i] \leftarrow Cnt[a_i] - 1$;
        for $A_u \in Next_{a_i}$ do
            pop $A_u$ from $Next_{a_i}$ and get $u$ from $T_u$;
            for $x \leftarrow a_i + 1$ to $\sigma - 1$ in $K$ do
                $v \leftarrow u + 2^x$;
                create trie node $T_v$;
                $l_v \leftarrow l_u + 1$;
                add edge $(T_u, T_v, x)$;
                push $A_v$ into $Next_x$;
        if $Cnt[a_i] = 0$ then
            remove the node containing $a_i$ from $K$;
    $len \leftarrow 0$;
    $L \leftarrow$ new list;
    insert 0 into $L$;
    for $i \leftarrow 0$ to $\sigma - 1$ do
        if trie node $T_{2^i}$ exists then
            push $A_{2^i}$ into $Next_i$;
    ...(continued in next page)
```

```
37
38      for $i \leftarrow 1$ to $n$ do
39          for $A_u$ in $Next_{b_i}$ do
40              pop $A_u$ from $Next_{b_i}$ and get $u$ and $l_u$ from $T_u$;
41              if $l_u > len$ then
42                  $len \leftarrow l_u$;
43                  empty $L$;
44              if $len = l_u$ then
45                  insert $u$ into $L$;
46              for every edge $(T_u, T_v, x)$ from $T_u$ do
47                  push $A_v$ into $Next_x$;
48      return the list $L$;
```

Initialization for Second Loop (Lines 32–37). Firstly, set *len*, denoting the length of LCIS of a, b currently found, to 0. Secondly, build a list L to store the binary encoding of every common increasing subsequence (CIS) of length *len* of a, b, where L contains only 0 (the binary encoding of the empty sequence) initially. Thirdly, reuse $Next$ queues and for each queue $Next_i$, push A_{2^i} into queue $Next_i$ if T_{2^i} exists in T.

Second Loop (Lines 38–47). Algorithm 1 iterates the following step when sequence b is scanned one by one from left to right. For the i-th integer b_i in b, Algorithm 1 iterates the following substeps until queue $Next_{b_i}$ is empty in the inner loop. First pop one A_u from queue $Next_{b_i}$. Then, since IS_u is a newly found CIS of a, b, we may need to update *len* and L accordingly: 1) if $l_u > len$ (i.e., the length of IS_u is greater than that of any CIS of a, b currently found), empty L and update *len* to l_u, and 2) if $l_u = len$, add u into L. Finally, in the (yet deeper) inner loop, push A_v into queue $Next_x$ for each edge (T_u, T_v, x) from T_u.

2.4 Example

Figures 1a and 1b show the statuses of K, $Next$, and T on the termination of the initialization and iteration 1, respectively, of the first loop of Algorithm 1 for $a = [1, 4, 1, 0, 3]$ and $\sigma = 5$. During the execution of the initialization, $Cnt[0] = Cnt[3] = Cnt[4] = 1$, $Cnt[1] = 2$, and $Cnt[2] = 0$ since the frequencies of integers $0, 1, 2, 3, 4$ are $1, 2, 0, 1, 1$, respectively. And, since $Cnt[i] > 0$ for $i = 0, 1, 3, 4$, the nodes storing integers $0, 1, 3, 4$ are doubly linked in sequence in K, the nodes T_1, T_{10}, T_{1000}, and T_{10000} (which contains the binary encodings of integers $0, 1, 3, 4$, respectively) are created in T, and A_1, A_{10}, A_{1000}, and A_{10000} (which are the addresses of T_1, T_{10}, T_{1000}, and T_{10000}, respectively) are contained in $Next_0$, $Next_1$, $Next_3$, and $Next_4$, respectively. In iteration 1, $a_1 = 1$. Thus, $Cnt[1]$ is

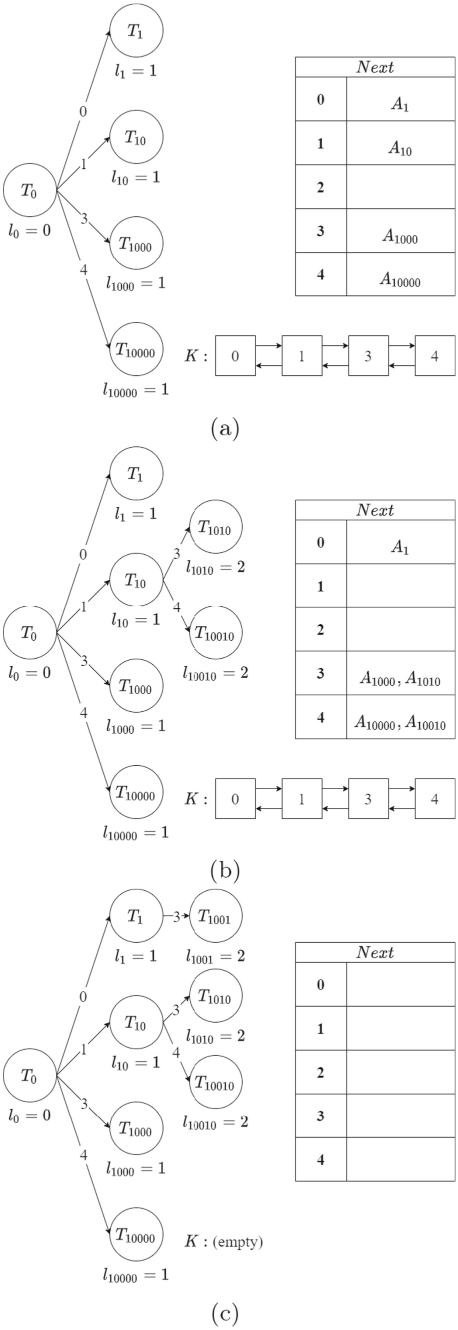

Fig. 1. The statuses of K, $Next$, and T on the termination of (a) the initialization, (b) iteration 1, (c) the last iteration of the first loop of Algorithm 1 as the input sequence a is $[1, 4, 1, 0, 3]$.

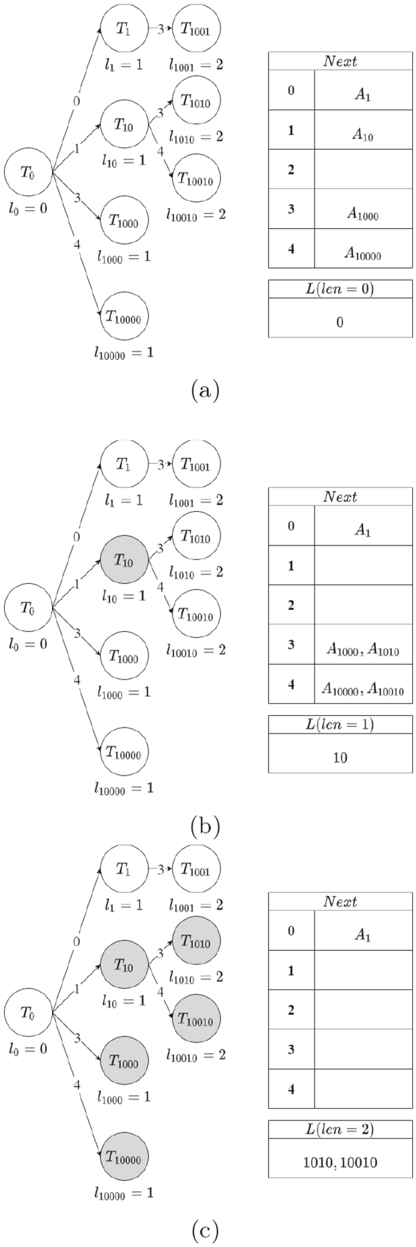

Fig. 2. The statuses of L, $Next$, and the encountered trie nodes in T on the termination (a) the initialization, (b) iteration 1, (c) the last iteration of the second loop of Algorithm 1 as the input sequence b is $[1, 4, 3, 1, 3]$, where the encountered trie nodes in T are shown in grey.

decreased to 1 and A_{10} is popped from $Next_1$. Due to that $a_1 = 1 < x < 5 = \sigma$ and $Cnt[x] > 0$ for $x = 3, 4$, the nodes T_{1010} and T_{10010} (which contains the binary encodings of increasing sequences $[1, 3], [1, 4]$, respectively) are added to T, and A_{1010} and A_{10010} are pushed into $Next_3$ and $Next_4$, respectively. In iteration 2, the node containing integer 4 is removed from K since $a_2 = 4$ and $Cnt[4]$ becomes 0. Similarly, the nodes containing integers 1, 0, and 3 are removed from K in iterations 3, 4, and 5, respectively. In addition, A_{10000} and A_{10010} are popped from $Next_4$ in iteration 2, A_1 is popped from $Next_0$, T_{1001} is added to T, and A_{1001} is pushed into $Next_3$ in iteration 4, and A_{1001} is popped from $Next_3$ in iteration 5. The statuses of K, $Next$, and T on the termination of the first loop is shown in Fig. 1c.

Figures 2a and 2b show the statuses of L, $Next$ and the encountered trie nodes in T on the termination of the initialization and iteration 1, respectively, of the second loop of Algorithm 1 for $b = [1, 4, 3, 1, 3]$. For the second loop, initially, A_1, A_{10}, A_{1000}, and A_{10000} are pushed into $Next_0$, $Next_1$, $Next_3$, and $Next_4$, respectively, since trie nodes T_1, T_{10}, T_{1000}, and T_{10000} exist in T; also, L contains a single element 0, and len is set to 0. In iteration 1, since $b_1 = 1$, A_{10} is popped from $Next_1$, T_{10} is encountered, and L is updated to contain 10 only. Meanwhile, since edge $(T_{10}, T_{1010}, 3)$ exists in T, A_{1010} is pushed into $Next_3$. Similarly, A_{10010} is pushed into $Next_4$. In iteration 2, A_{10000} and A_{10010} is popped from $Next_4$, T_{10000} and T_{10010} are encontered, and L is updated to contain 10010. In iteration 3, A_{1000} and A_{1010} are popped from $Next_3$, T_{1000} and T_{1010} are encontered, and 1010 is inserted to L. In iteration 4 (or 5), the statuses of L and $Next$ remains unchanged since $Next_1$ ($Next_3$) is empty. Figure 2c shows the statuses of L, $Next$ and the encountered trie nodes in T on the termination of the second loop.

3 The Analysis

In this section, we first show the correctness of the proposed algorithm. Subsequently, the time and space complexity of the proposed algorithm is studied.

3.1 Correctness

Lemma 1. *In the first loop, a non-empty increasing subsequence IS_u of a exists if and only if A_u has been popped from some $Next$ queue.*

Proof. It suffices to show for each i $(1 \leq i \leq n)$, on the termination of iteration i of the first loop, a non-empty increasing subsequence IS_u exists in $a_1, a_2, ..., a_i$ (a prefix of a) if and only if A_u has been popped from some $Next$ queue. We show it by induction on the number of iterations executed.

Clearly, on the termination of iteration 1, $[a_1]$ is the only one non-empty increasing subsequence of a. Besides, in the initialization of the first loop, Algorithm 1 pushes A_{2^u} into queue $Next_u$ once for each integer u that exists in a. Since Algorithm 1 pops all items in queue $Next_{a_1}$ in iteration 1, only $A_{2^{a_1}}$ has been popped on the termination of iteration 1. Thus, we have a basis. We then assume the induction hypothesis: on the termination of iteration k, a non-empty increasing subsequence IS_u exists in $a_1, a_2, ..., a_k$ if and only if A_u has been popped from some $Next$ queue. To complete the proof, we only need to show the induction step: on the termination of iteration $k+1$, a non-empty increasing subsequence IS_u exists in $a_1, a_2, ..., a_{k+1}$ if and only if A_u has been popped from some $Next$ queue.

For the *if* part, if A_u is popped from some $Next$ queue before iteration $k+1$, IS_u exists in $a_1, a_2, ..., a_k$ by induction hypothesis, and thus IS_u exists in $a_1, a_2, ..., a_{k+1}$. So, we only need to consider the case where A_u is popped from some $Next$ queue in iteration $k+1$. Note that Algorithm 1 pops all items in queue $Next_{a_{k+1}}$ in iteration $k+1$. Also note that Algorithm 1 only pushes A_u into queue $Next_{a_{k+1}}$ when IS_u ends with a_{k+1}. Let IS_u be the concatenation of IS_v and a_{k+1}. Clearly, if IS_v is an empty sequence, $IS_u = a_{k+1}$ is an increasing subsequence of a. Otherwise, let IS_v end with a_j; then, A_v has been popped from some $Next$ queue on the termination of iteration k and $a_j < a_{k+1}$ because otherwise, A_u is not in $Next$ queues in iteration $k+1$ by Algorithm 1. By induction hypothesis, IS_v is an increasing subsequence in $a_1, a_2, ..., a_k$. This implies IS_u is an increasing subsequence in $a_1, a_2, ..., a_{k+1}$, completing the proof of the *if* part.

For the *only if* part, if IS_u does not end with a_{k+1}, IS_u exists in $a_1, a_2, ..., a_k$, and thus A_u has been popped from some $Next$ queue on the termination of iteration k by the induction hypothesis. So, we only need to consider the case where IS_u ends with a_{k+1}. Since Algorithm 1 pops all items in queue $Next_{a_{k+1}}$ in iteration $k+1$, we only need to show A_u is in queue $Next_{a_{k+1}}$ on the termination of iteration k. Let IS_u be the concatenation of IS_v and a_{k+1}. Then, if IS_v is an empty sequence, A_u is pushed into queue $Next_{a_{k+1}}$ in the initialization of the first loop. Otherwise, IS_v is a non-empty increasing subsequence in $a_1, a_2, ..., a_k$. Let IS_v end with a_j. Since IS_u is an increasing subsequence, we have $a_j < a_{k+1}$. Then, on the termination of iteration k, A_v has been popped from some $Next$ queue by induction hypothesis, and then A_u has been pushed into queue $Next_{a_{k+1}}$ due to $a_j < a_{k+1}$ and $Cnt[a_{k+1}] > 0$, completing the *only if* part.

Theorem 1. *In the first loop, IS_u is a non-empty increasing subsequence of a if and only if a trie node T_u is created in T.*

Proof. Note that Algorithm 1 creates a trie node T_u right before A_u is pushed into some $Next$ queue in the first loop. Thus, a trie node T_u is created in T if and only if A_u has been pushed into some $Next$ queue. Besides, IS_u is a non-empty increasing subsequence of a if and only if A_u has been popped from some $Next$ queue by Lemma 1. Thus, to complete the proof, we only need to show A_u has been pushed into some $Next$ queue if and only if A_u has been popped from some $Next$ queue. Clearly, A_u has been pushed into some $Next$ queue if A_u has been popped from some $Next$ queue. On the other hand, suppose A_u is pushed into some $Next$ queue, say $Next_x$, in iteration j. Then, $Cnt[x] > 0$ in iteration j. This implies x exists in $a_{j+1}, a_{j+2}, \cdots, a_n$. Let a_k be the first integer in $a_{j+1}, a_{j+2}, \cdots, a_n$ such that $x = a_k$. Then, A_u is popped from queue $Next_x$ in iteration k. This completes the proof.

Lemma 2. *In the second loop, a non-empty common increasing subsequence* IS_u *of* a, b *exists if and only if* A_u *has been popped from some* $Next$ *queue.*

Proof. It suffices to show for each i $(1 \leq i \leq n)$, on the termination of iteration i of the second loop, a non-empty common increasing subsequence IS_u of a, b exists in $b_1, b_2, ..., b_i$ (a prefix of b) if and only if A_u has been popped from some $Next$ queue. We show it by induction on the number of iterations executed. In iteration 1, Algorithm 1 pops all items from queue $Next_{b_1}$. Let u be the binary encoding of b_1. Then, b_1 exists in $a_1, a_2, ..., a_n$ if and only if trie node T_u exists in T by Theorem 1, and thus b_1 exists in $a_1, a_2, ..., a_n$ if and only if A_u is pushed into queue $Next_{b_1}$ in the initialization of the second loop. This implies that a non-empty common increasing subsequence IS_u of a, b exists in b_1 if and only if A_u has been popped from queue $Next_{b_1}$ in iteration 1. Thus, we have a basis. We omit the induction hypothesis and the proof of the induction step due to their similarities to that of Lemma 1.

Theorem 2. *By Algorithm 1, the list* L *contains exactly the binary encoding of every LCIS of* a, b.

Proof. By Lemma 2, for every non-empty CIS IS_u of a, b, A_u has been popped from some $Next$ queue in the second loop. In Algorithm 1, when A_u is popped from some $Next$ queue, the binary encoding of IS_u is added to L if the length of IS_u is equal to that of the CIS of a, b in L, and L is updated to contain only the binary encoding of IS_u if the length of IS_u is greater than that of the CIS of a, b in L. This ensures the binary encoding of every LCIS of a, b is contained in L.

3.2 Complexity

Lemma 3. *Every* A_u *for a non-empty increasing subsequence* IS_u *of* a *is pushed into* $Next$ *queues at most once in (a) the first loop and (b) the second loop.*

Proof. The proof of (b) is omitted due to its similarity to that of (a). We show (a) by contradiction. Suppose that A_u is the first one to be pushed into $Next$ queues more than once. Then, $|IS_u|$ must be greater than 1; otherwise, A_u is pushed into $Next$ queues only once in the initialization step.

Let IS_u be the concatenation of IS_v and x (i.e., IS_u ends with x). Note that A_u is pushed into queue $Next_x$ right after A_v is popped from some queue. Thus, A_u is pushed into $Next$ queues the second time right after A_v is popped from some queue the second time. This implies A_v is pushed into some queue twice before A_u is pushed into some queue twice. This constitutes a contradiction.

Theorem 3. *The time and space complexity of Algorithm 1 is* $O(n + \sigma + I_a)$.

Proof. Apart from the input sequences a, b themselves, K and Cnt require $O(\sigma)$ space, and T and $Next$ queues require $O(I_a)$ space by Lemma 3, so the space complexity is $O(n + \sigma + I_a)$. Note that the space complexity can be reduced to $O(n + I_a)$ through removing integers that do not exist in both of a and b from the alphabet set by additional preprocessing before Algorithm 1.

For time complexity, the initialization steps of the first and second loops require $O(n + \sigma)$ time. In the first and second loops, there are $O(I_a)$ queue and trie node operations by Lemma 3. Each queue operation requires $O(1)$ time. And, since the address of T_u is pushed into $Next$ queues, each trie node operation can be achieved in $O(1)$ time. Since Algorithm 1 uses Cnt and K to avoid iterations without queue or trie node operation, the time complexity is $O(n + \sigma + I_a)$.

Remark that due to that $I_a \leq 2^\sigma$, the time complexity of Algorithm 1 is $O(n)$, which is optimal for the LCIS problem, as $\sigma \leq \log_2 n$.

4 Conclusion and Discussion

In this paper, we present an algorithm of $O(n+\sigma+I_a)$ time and space complexity to find every LCIS of sequences a, b of length n. If the proposed algorithm is run on two computers in parallel, the time complexity can be improved to $O(n + \sigma + min(I_a, I_b))$. When the alphabet set is small, an algorithm of $O(n \log\log n)$ time complexity was proposed in the literature for $\sigma = 3$ [3]. By contrast, the proposed algorithm has $O(n)$ time and space complexity as $\sigma \leq \log_2 n$. For the LCIS problem of k $(k > 2)$ sequences, an algorithm of $O(n + \sigma + kI_a)$ time complexity can be obtained through slight modification of the proposed algorithm by just running the second loop for each sequence other than a and keeping track of how many times each node in T is encountered. Whether the proposed algorithm can be modified to better adapt to the cases of more than 2 sequences may be worthy of discussion.

References

1. Chan, W.T., Zhang, Y., Fung, S.P.Y., Ye, D., Zhu, H.: Efficient algorithms for finding a longest common increasing subsequence. J. Comb. Optim. **13**(3), 277–288 (2006). https://doi.org/10.1007/s10878-006-9031-7
2. Duraj, L.: A sub-quadratic algorithm for the longest common increasing subsequence problem. arXiv:1902.06864 [cs], January 2020
3. Kutz, M., Brodal, G.S., Kaligosi, K., Katriel, I.: Faster algorithms for computing longest common increasing subsequences. J. Discret. Algorithms **9**(4), 314–325 (2011). https://doi.org/10.1016/j.jda.2011.03.013
4. Lo, S.F., Tseng, K.T., Yang, C.B., Huang, K.S.: A diagonal-based algorithm for the longest common increasing subsequence problem. Theor. Comput. Sci. **815**, 69–78 (2020). https://doi.org/10.1016/j.tcs.2020.02.024. https://www.sciencedirect.com/science/article/pii/S0304397520301158
5. Sakai, Y.: A linear space algorithm for computing a longest common increasing subsequence. Inf. Process. Lett. **99**(5), 203–207 (2006). https://doi.org/10.1016/j.ipl.2006.05.005

6. Yang, I.H., Huang, C.P., Chao, K.M.: A fast algorithm for computing a longest common increasing subsequence. Inf. Process. Lett. **93**(5), 249–253 (2005). https://doi.org/10.1016/j.ipl.2004.10.014
7. Zhu, D., Wang, L., Wang, T., Wang, X.: A simple linear space algorithm for computing a longest common increasing subsequence. arXiv:1608.07002 [cs], August 2016