Theoretical Computer Science 1005 (2024) 114639

Contents lists available at ScienceDirect

Theoretical Computer Science

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

The longest almost increasing subsequence problem with sliding windows [☆,☆☆]

Cheng-Han Ho, Chang-Biau Yang [*]

Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan

A R T I C L E I N F O

Communicated by V. Bonifaci

Keywords:
Longest increasing subsequence
Longest almost increasing subsequence
Sliding window
Row tower
Principal row

A B S T R A C T

In this paper, we first define the *longest almost increasing subsequence with sliding windows* (LaISW), a generalization that combines the *longest increasing subsequence with sliding windows* (LISW) problem and the *longest almost increasing subsequence* (LaIS) problem. For a numeric sequence A, a window size w and a tolerance constant c, the goal of the LaISW problem is to identify the LaIS within all windows of size w in A. In an almost increasing subsequence, slight decreases smaller than c are permitted. We propose an efficient algorithm for solving the LaISW problem. Instead of constructing the entire row tower, our algorithm computes the change in drop out (occurrence) for each element in the row tower. The time complexity of our algorithm is $O(nL)$, and the space complexity is $O(L)$, where n and L represent the lengths of the input sequence and the LaISW answer, respectively.

1. Introduction

Given a numeric sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$, the *longest increasing subsequence* (LIS) problem [15,12,4,17,8,1] is to find the strictly increasing subsequence of A with the maximum length. That is, it seeks to determine the maximum length of a subsequence from the given sequence where each element is strictly greater than the preceding one. Take $A = \langle 2, 7, 5, 6, 1, 9, 8, 11, 10 \rangle$ as an example. The LIS answers could be $\langle 2, 5, 6, 9, 11 \rangle$, $\langle 2, 5, 6, 9, 10 \rangle$, $\langle 2, 5, 6, 8, 11 \rangle$, or $\langle 2, 5, 6, 8, 10 \rangle$, all of which have a length of 5. It is worth noting that an LIS answer does not contain duplicate elements.

The LIS problem was first defined by Schensted [15] in 1961. He proposed an algorithm with a time complexity of $O(n \log n)$ to solve this problem, where n represents the length of the input sequence. In 1977, Hunt and Szmanski [12] proposed an algorithm with a time complexity of $O(n \log\log n)$, employing the van Emde Boas tree (vEB tree) [18] if the input sequence A is a permutation of $\{1, 2, \ldots, n\}$. In 2000, Bespamyatnikh and Segal [4] devised an algorithm with time complexity $O(n \log\log n)$ by employing the vEB tree. Notably, their algorithm is capable of reporting all LIS answers. In 2010, Crochemore and Porat [8] reduced the time complexity to $O(n \log\log L)$ in the RAM model, where L represents the LIS length. In 2013, Alam and Rahman [1] presented a divide-and-conquer method, whose time complexity is $O(n \log n)$.

☆ This article belongs to Section A: Algorithms, automata, complexity and games, Edited by Paul Spirakis.

☆☆ A preliminary version of this paper was presented at the 10th IEEE Asia-Pacific Conference on Computer Science and Data Engineering 2023 (IEEE CSDE 2023), Dec. 2023, Fiji.

* Corresponding author.
E-mail address: cbyang@cse.nsysu.edu.tw (C.-B. Yang).

https://doi.org/10.1016/j.tcs.2024.114639
Received 17 November 2023; Received in revised form 13 May 2024; Accepted 18 May 2024

Available online 22 May 2024

Table 1
The time complexities of the previous LIS, LaIS, LICS and LISW algorithms. n: length of the input sequence A; c: tolerance constant; w: window size; L: length of the answer; r: number of match pairs; Σ: alphabet set.

The longest increasing subsequence (LIS) problem			
Year	Author(s)	Time complexity	Note
1961	Schensted [15]	$O(n \log n)$	Young tableau, binary Search
1977	Hunt and Szymanski [12]	$O(n \log \log n)$	Match pair, van Emde Boas tree
2000	Bespamyatnikh and Segal [4]	$O(n \log \log n)$	All answers
2010	Crochemore and Porat [8]	$O(n \log \log L)$	Split blocks
2010	Elmasry	$O(n \log L)$	Dynamic programming almost increasing
2013	Alam and Rahman [1]	$O(n \log n)$	Divide-and-conquer
The longest increasing circular subsequence (LICS) problem			
Year	Author(s)	Time complexity	Note
2007	Albert et al. [2]	$O(n^{3/2} \log n)$	Monte Carlo
2009	Deorowicz [9]	$O(min(nL, n \log n + L^3 \log n))$	Cover merge
The longest increasing subsequence with sliding window (LISW)			
Year	Author(s)	Time complexity	Note
2004	Albert et al. [3]	$O(n \log \log n + nL)$	Row tower
2007	Chen et al. [6]	$O(nL)$	Canonical antichain
2012	Deorowicz [10]	$O(n \log \log n + min(nL, n\lceil L^3/w\rceil) \log\lceil w/L^2+1\rceil)$	Cover merge
2018	Li et al. [13]	$O(nw)$	Quadruple neighbor list
2024	This paper	$O(nL)$	Row tower, almost increasing

To obtain a longer subsequence, the strict requirement for a strictly increasing subsequence in LIS can be relaxed in real-world applications. For instance, during an upward trend in the stock market, the stock price may not increase every day; instead, there might be slight decreases on some days during the period. Therefore, Elmasry [11] introduced the concept of the *longest almost increasing subsequence* (LaIS) in 2010, presenting it as a generalized variant of the LIS problem. The LaIS problem permits slight decreases (smaller than a predefined tolerance constant c) from the maximum element encountered thus far. For example, consider a sequence $A = \langle 2, 7, 5, 6, 1, 9, 8, 11, 10\rangle$ along with a tolerance constant $c = 2$. The LaIS answer of A is $\langle 2, 5, 6, 9, 8, 11, 10\rangle$, whose length is 7. The time complexity of the proposed algorithm of Elmasry is $O(n \log L)$ [11].

In 2004, Albert et al. [3] initially introduced the *longest increasing subsequence with sliding windows* (LISW), which is a variant of LIS. They presented a *row tower* method to tackle the LISW problem, with time complexity $O(n \log \log n + nL)$ and space complexity $O(n)$, where L represents the answer length. In 2007, Chen et al. [6] presented an $O(nL)$-time algorithm with the utilization of the canonical antichain partition. In 2012, Deorowicz et al. proposed a cover-merge algorithm with time complexity $O(n \log \log n + \min(nL, n\lceil L^3/w\rceil) \log\lceil w/L^2 + 1\rceil)$, where w represents the window size. In 2018, Li et al. [13] introduced a data structure known as the *quadruple neighbor list*, enabling the problem to be solved in $O(nL)$ time.

The *longest increasing circular subsequence* (LICS) problem represents another variant of LIS. Given a numeric sequence A, the LICS problem is to identify the LIS among all rotations of A, where a rotation involves removing some prefix elements and appending them at the end to create a circular sequence. If the window size is n and the input sequence A is repeated twice, then it is apparent that the LISW algorithm can solve the LICS problem. Consequently, LISW is considered a more generalized version compared to LICS. Initially defined by Albert et al. in 2007, the LICS problem was also addressed by them with time complexity $O(n^{3/2} \log n)$ [2]. In 2009, Deorowicz [9] proposed a cover-merge algorithm for solving the LICS problem with time complexity $O(\min(nL, n \log n + L^3 \log n))$.

The previously related studies are summarized in Table 1.

In this paper, we will first introduce the *longest almost increasing subsequence with sliding window* (LaISW) problem, which is a generalized combination of the LISW and LaIS problems. Given a numeric sequence $A = \langle a_1, a_2, \dots, a_n\rangle$, and a sliding window size w along with a tolerance constant c, the LaISW problem aims to identify the almost increasing subsequence with the maximum length among all substrings of A with length w. We then propose an efficient algorithm to solve the LaISW problem. Rather than constructing the entire row tower, our algorithm calculates the change in drop out (occurrence) for each element in the row tower. The time complexity of our algorithm is $O(nL)$, and the space complexity is $O(L)$.

Table 2
An example for the LaIS algorithm [11], where $A = \langle 2, 7, 5, 6, 1, 9, 8, 11, 10\rangle$, and $c = 4$. The LaIS answer is $\langle 2, 7, 5, 6, 9, 8, 11, 10\rangle$, with length 8.

A \ length		1	2	3	4	5	6	7	8	9
a_1	2	**2**								
a_2	7	2	**7**							
a_3	5	2	**5**	7						
a_4	6	2	5	**6**	7					
a_5	1	**1**	2	6	7					
a_6	9	1	2	6	7	**9**				
a_7	8	1	2	6	7	**8**	9			
a_8	11	1	2	6	7	8	9	**11**		
a_9	10	1	2	6	7	8	9	**10**	11	

The subsequent sections of this paper are organized as follows. Section 2 offers a brief overview of the LaIS problem and the LISW problem, followed by a formal definition of the LaISW problem. In Section 3, we explore some properties of the LaISW. We propose an algorithm for solving the LaISW problem with $O(nL)$ time and $O(L)$ space in Section 4. Section 5 provides a comprehensive example. Lastly, Section 6 presents the conclusions.

2. Preliminaries and problem definitions

The *longest almost increasing subsequence* (LaIS) problem, first defined by Elmasry in 2010 [11], represents a generalized variant of the LIS problem. Unlike the LIS problem, where the answer must be strictly increasing, LaIS introduces flexibility by allowing slight decreases within the sequence as long as they do not exceed a predefined tolerance constant. The LaIS problem aims to find the *almost increasing subsequence* with the maximum length.

Definition 1. *(almost increasing sequence) [11]* Given a numeric sequence $A = \langle a_1, a_2, \ldots, a_n\rangle$ and a tolerance constant c, A is an *almost increasing sequence* (aIS) if $a_i > \max\{a_k | 1 \le k \le i-1\} - c$, for $2 \le i \le n$.

For example, consider a sequence $A = \langle 2, 7, 5, 6, 1, 9, 8, 11, 10\rangle$ and a tolerance constant $c = 4$. The LaIS answer is $\langle 2, 7, 5, 6, 9, 8, 11, 10\rangle$, with length 8. If the tolerance constant is changed to $c = 2$, then the LaIS answer is $\langle 2, 5, 6, 9, 8, 11, 10\rangle$, with length 7. In the case of $c = 2$, 5 cannot be appended behind 7, because it is not true for $5 > 7 - c = 5$. In addition, the answer obtained from LaIS may be longer than that from LIS because LaIS permits some slight decreases.

Table 2 illustrates an example of the LaIS algorithm as proposed by Elmasry [11]. In each iteration, a new element is added, aiming to minimize the largest element retained in the subsequence of each length. The LaIS algorithm can be summarized as follows.

Theorem 1. [11] *In the LaIS algorithm, when a new element a_i is added, it is placed before the first element that is greater than a_i. Then, the first element greater than or equal to $a_i + c$ is removed if there exists any one.*

The *longest increasing subsequence with sliding windows* (LISW) problem seeks to identify the LIS among all substrings of the given sequence with the predefined window size. Albert et al. [3] introduced the concept of principal row, row tower and drop out of the LIS to tackle the LISW problem. The time complexity of their algorithm for LISW is $O(n \log\log n + nL)$, where L represents the length of the LISW answer.

In this paper, we combine the LaIS and LISW problems to define the *longest almost increasing subsequence with sliding windows* (LaISW) problem as follows.

Definition 2. (*LaISW problem*) Given a numeric sequence, $A = \langle a_1, a_2, \ldots, a_n\rangle$, along with a window size w and a tolerance constant c, the *longest almost increasing subsequence with sliding window* (*LaISW*) problem aims to find the longest almost increasing subsequence in all windows $A_{i..i+w-1} = \langle a_i, a_{i+1}, \ldots, a_{i+w-1}\rangle$, where $1 \le i \le n - w + 1$.

For example, suppose that $A = \langle 9, 1, 2, 5, 3, 5, 10, 8, 2\rangle$, $w = 6$ and $c = 3$. The LaISW answer are $\langle 1, 2, 5, 3, 5, 10\rangle$, with length 6, obtained from $A_{2..7} = \langle 1, 2, 5, 3, 5, 10\rangle$, or $\langle 2, 5, 3, 5, 10, 8\rangle$ obtained from $A_{3..8}$.

3. Related properties

To solve the LaISW problem, we first build some properties, including the principal row, row tower and drop out of LaIS.

Definition 3. *(principal row of a substring in LaIS)* Given a numeric sequence $A = \langle a_1, a_2, \ldots, a_n\rangle$, along with a tolerance constant c, the *principal row* of A, denoted by $P = \langle p_1, p_2, \ldots, p_{n'}\rangle$, is formed that p_i, $1 \le i \le n'$, is the smallest (best) maximum value of the almost increasing subsequence of A with length i. Furthermore, let R_i^j denote the *principal row* of $A_{i..j} = \langle a_i, a_{i+1}, \ldots, a_j\rangle$, $1 \le i \le j \le n$.

Table 3
Examples for row tower, removal, insertion and update in LaIS with $A = \langle 2, 7, 5, 6, 1, 9, 8, 11, 10, 3\rangle$, $w = 9$ and $c = 4$. Here, R denotes the row tower and D denotes the drop out.

Seq.	$A_{1..9}$	$A_{2..9}$ (2 is deleted)	$A_{2..10}$ (3 is added)
R	R_1^9: 1, 2, 6, 7, 8, 9, 10, 11	R_2^9: 1, 6, $\underline{7}$, 8, 9, 10, 11	R_2^{10}: 1, 3, 6, 8, 9, 10, 11
	R_2^9: 1, 6, 7, 8, 9, 10, 11	R_3^9: 1, 6, $\underline{8}$, 9, 10, 11	R_3^{10}: 1, 3, 6, 9, 10, 11
	R_3^9: 1, 6, 8, 9, 10, 11	R_4^9: 1, $\underline{8}$, 9, 10, 11	R_4^{10}: 1, 3, 9, 10, 11
	R_4^9: 1, 8, 9, 10, 11	R_5^9: 1, $\underline{8}$, 9, 10, 11	R_5^{10}: 1, 3, 9, 10, 11
	R_5^9: 1, 8, 9, 10, 11	R_6^9: $\underline{8}$, 9, 10, 11	R_6^{10}: 3, 9, 10, 11
	R_6^9: 8, 9, 10, 11	R_7^9: $\underline{8}$, 10, 11	R_7^{10}: 3, 10, 11
	R_7^9: 8, 10, 11	R_8^9: $\underline{10}$, 11	R_8^{10}: 3, 11
	R_8^9: 10, 11	R_9^9: $\underline{10}$	R_9^{10}: 3
	R_9^9: 10		R_{10}^{10}: 3
D	5, 1, 3, 2, 7, 6, 9, 8	4, 2, $\underline{1}$, $\underline{6}$, 5, $\underline{8}$, 7	4, 9, 2, 1, 5, 6, 7

The principal row of a sequence is the final list built by Theorem 1. For example, see Table 3. Suppose that $A = \langle 2, 7, 5, 6, 1, 9, 8, 11, 10, 3\rangle$ and $c = 4$. Then, the principal row of $A_{1..9}$ is $P = R_1^9 = \langle 1, 2, 6, 7, 8, 9, 10, 11\rangle$. In addition, $p_4 = 7$ is the smallest (best) maximum value of the almost increasing subsequence with length 4, obtained from $\langle 2, 7, 5, 6\rangle$.

Definition 4. *(row tower in LaIS)* Given a numeric sequence $A = \langle a_1, a_2, \ldots, a_n\rangle$ and a tolerance constant c, the *row tower* R of LaIS consists of principal rows $R_1^n, R_2^n, \ldots, R_n^n$.

Definition 5. *(drop out) [3]* Given a sequence A, the *drop out* D records the number of occurrences of each element in the first row of the row tower R.

The concept of drop out, initially proposed by Albert et al. [3] for solving the LISW problem, can be adapted by modifying the row tower of LIS to accommodate LaIS. For instance, Table 3 illustrates the row towers of $A_{1..9}$, $A_{2..9}$, and $A_{2..10}$ along with their drop outs.

To design an efficient algorithm for solving the LaISW problem, we first develop the following theorem.

Theorem 2. *Given a numeric sequence $A = \langle a_1, a_2, \ldots, a_n\rangle$ and a tolerance constant c, in the LaIS row tower R, each row R_{i+1}^j can be obtained by removing exactly zero or one element from R_i^j, $1 \le i < i+1 \le j \le n$.*

Proof. We will prove it by induction. Obviously, $R_1^1 = \langle a_1\rangle$, since only a_1 is considered. When $\langle a_1, a_2\rangle$ is considered, there are three cases as follows.

Case 1: $a_1 < a_2$.
a_2 can be appended to a_1 to form an LaIS of length 2. In this case, $R_1^2 = \langle a_1, a_2\rangle$.

Case 2: $a_1 - c < a_2 \le a_1$.
a_2 can still be appended behind a_1 to form an LaIS answer of length 2. In the row tower, a_2 and a_1 are the smallest maxima for forming answers of lengths 1 and 2, respectively. Thus, $R_1^2 = \langle a_2, a_1\rangle$.

Case 3: $a_1 - c \ge a_2$.
a_2 cannot be appended behind a_1 to make a longer LaIS. However, we can replace a_1 by a_2 to get a smaller maximum for an answer of length 1. In this case, $R_1^2 = \langle a_2\rangle$.

It is clear that $R_2^2 = \langle a_2\rangle$. Thus, the theorem holds for $j = 2$.

For the hypothesis, it is assumed that R_i^j and R_{i+1}^j hold for the theorem, where $1 < i \le j$ and $2 \le j \le n-1$. When $i = j$, we have that $R_i^j = \{a_i\}$, consisting of one element, and R_{i+1}^j is empty. Now, we aim to prove that R_i^{j+1} and R_{i+1}^{j+1} also hold for the theorem.

To build R_i^{j+1} from R_i^j, the addition of a_{j+1} involves inserting a_{j+1} into the appropriate position of R_i^j, followed by the removal of the first element $a_{k'}$ such that $a_{k'} \ge a_{j+1} + c$ if one exists

By the hypothesis, two cases for the relation of R_i^j and R_{i+1}^j are considered as follows.

Case 1: $R_i^j = R_{i+1}^j$.
Since the two principal rows are the same, the addition of a_{j+1} performs the same work for the two principal rows. Thus, $R_i^{j+1} = R_{i+1}^{j+1}$.

Case 2: $R_i^j = R_{i+1}^j \cup \{a_{h'}\}$. The addition of a_{j+1} to build R_i^{j+1} and R_{i+1}^{j+1} may or may not remove one element from R_i^j and R_{i+1}^j, respectively. The following four subcases are considered:

Subcase 2.1: There is no removal for both R_i^j and R_{i+1}^j. Then, we get $R_i^{j+1} = R_{i+1}^{j+1} \cup \{a_{h'}\}$.

Subcase 2.2: One removal occurs in R_i^j, but no removal occurs R_{i+1}^j. The only possible element to be removed is $a_{h'}$, since all other elements in R_i^j are the same as R_{i+1}^j. Thus, after the addition of a_{j+1}, we have $R_i^{j+1} = R_{i+1}^{j+1}$.

Subcase 2.3: No removal occurs in R_i^j, but one removal occurs R_{i+1}^j. However, this is impossible since all elements in R_i^j are also present in R_{i+1}^j.

Subcase 2.4: One removal occurs in both R_i^j and R_{i+1}^j. If the removed element in both R_i^j and R_{i+1}^j is $a_{k'}$ which is the first element in both R_i^j and R_{i+1}^j that is greater than or equal to $a_{j+1}+c$, then $R_i^{j+1} = R_{i+1}^{j+1} \cup \{a_{h'}\}$. If $a_{h'}$ is removed from R_i^j, but $a_{k'}$ is removed from R_{i+1}^j, then $R_i^{j+1} = R_{i+1}^{j+1} \cup \{a_{k'}\}$.

In summary, R_i^{j+1} and R_{i+1}^{j+1} also hold for the theorem. Thus, the theorem gets proved. □

Based on Theorem 2, the following corollary can be easily obtained. In other words, for each element in the row tower, if it dies, it will never be reborn.

Corollary 1. *In the LaIS row tower R of a sequence A, R_j^k is a subsequence of R_i^k, $1 \le i \le j \le k \le |A|$.*

4. The algorithm

To slide the window with a fixed size w, three operations are performed in order: (1) *removal operation*: delete the first element of the original window; (2) *insertion operation*: adds the next element into the principal row; (3) *update operation*: update the drop out from the old one to the new one.

Definition 6. (*removal operation from the row tower*) When the first element of a given sequence A is deleted, the *removal operation* deletes the first row of the LaIS row tower R, subtracts 1 from each element in the drop out D. Finally, it removes each element in D if it becomes 0 after the subtraction.

For example, see Table 3. Transitioning from $A_{1..9}$ to $A_{2..9}$ by removing the first element $a_1 = 2$ is equivalent to deleting the first row of the row tower R_1^9. Therefore, the drop out D can be obtained by subtracting one from each element. That is, ⟨5, 1, 3, 2, 7, 6, 9, 8⟩ for $A_{1..9}$ becomes ⟨4, 0, 2, 1, 6, 5, 8, 7⟩, then we remove any zeros to form drop out ⟨4, 2, 1, 6, 5, 8, 7⟩ for $A_{2..9}$.

When a new element is added, the *insertion operation* is performed on the old principal row.

Definition 7. (*insertion operation of the principal row*) *[11]* Let b be the newly element added to the end of the new window. b is first inserted into the principal row P in the front of the successor of b. Then, the first element in P greater than or equal to $b+c$ is deleted if there exists any.

Table 3 illustrates an example for the addition of a new element, from the middle window to the right one. The old principal row (first row on the middle window) is ⟨1, 6, 7, 8, 9, 10, 11⟩. In the right window, a new element $b = 3$ is added. $b = 3$ is inserted before its successor 6, and then 7 is deleted since 7 is the first element greater than or equal to $b + c = 3 + 4 = 7$. After this insertion operation, the new principal row (first row on the right) becomes ⟨1, 3, 6, 8, 9, 10, 11⟩.

The row tower of $A_{2..9}$ can be updated to the row tower of $A_{2..10}$ with a brute-force method. By Definition 7, the brute-force method involves inserting $b = 3$ into each row of the row tower and subsequently removing the first element (if any) greater than or equal to $b + c = 3 + 4 = 7$. Once the new row tower is constructed row by row, establishing the new drop out is straightforward, as it counts the occurrences of each element in the row tower.

Next, we define the *shift path* for updating the drop out more efficiently.

Definition 8. (*shift path of drop out*) When a new element b is inserted into an existing principal row $P = \langle p_1, p_2, \cdots, p_m\rangle$, along with the drop out $D = \langle d_1, d_2, \cdots, d_m\rangle$ and the tolerance constant c, the *shift path of drop out* is represented by the index sequence $\langle i_1, i_2, \cdots, i_{m'}\rangle$, where $m' \le m$. Here, p_{i_1} is the first element greater than or equal to $b + c$, and $d_{i_k} < d_{i_{k+1}}$ for $1 \le k \le m' - 1$. In the shift path, $d_{i_{k+1}}$ is the next element greater than d_{i_k} in D.

See the middle window of Table 3. The shift path of drop out is $\langle i_1, i_2, i_3\rangle = \langle 3, 4, 6\rangle$. $p_{i_1} = p_3 = 7$ is the first element greater than or equal to $b + c = 7$. In D, the next element greater than $d_3 = 1$ is $d_4 = 6$, and the next element greater than $d_4 = 6$ is $d_6 = 8$.

Definition 9. (*update operation of the drop out*) When a new element b is inserted into an existing principal row $P = \langle p_1, p_2, \cdots, p_m\rangle$, along with the drop out $D = \langle d_1, d_2, \cdots, d_m\rangle$ and the tolerance constant c, the *update operation* calculates the new drop out $D' = \langle d'_1, d'_2, d'_3, \cdots\rangle$ based on the old drop out D with *Equation (1)*.

$$\begin{cases} d'_i = d_i & \text{if } p_i \le b; \\ d'_i = w & \text{if } i \text{ is the inserted position index of } b \text{ in } P; \\ d'_i = d_{i-1} & \text{if } b+1 \le p_{i-1} \le b+c-1; \\ d'_{i_{k+1}} = d_{i_k} & \text{if } i_k \text{ is in the shift path and } k \ge 1; \\ d'_i = d_i & \text{otherwise.} \end{cases} \tag{1}$$

Equation (1) provides a more efficient method for getting the new drop out. In the example, $b = 3$ is the newly appended element into $A_{2..9}$ (from the middle to the right). For $A_{2..9}$, the principal row $P = \langle p_1, p_2, \cdots, p_m \rangle = \langle 1, 6, 7, 8, 9, 10, 11 \rangle$, where $m = 7$, and the drop out $D = \langle 4, 2, 1, 6, 5, 8, 7 \rangle$. According to Equation (1), five cases of the update operation are considered to build the new drop out D' after appending $b = 3$ as follows.

Case 1: $p_i \leq b$. We have that $p_1 = 1 \leq b = 3$. So we get $d'_1 = d_1 = 4$. In this case, the number of occurrences of p_1 remains unchanged, since $b = 3$ is inserted after p_1 in P.

Case 2: i is the inserted position index of b in P. In this example, $b = 3$ should be inserted before its successor 6. In other words, $i = 2$. Thus, we get $d'_2 = w = 9$, since $b = 3$ is always alive in the row tower.

Case 3: $b + 1 \leq p_{i-1} \leq b + c - 1$. We can get that $b + 1 = 4 \leq p_2 = 6 \leq b + c - 1 = 6$. That is, $p_{i-1} = p_{3-1}$ satisfies the condition. As a result, $d'_3 = d_2 = 2$, meaning that the survival duration of $p_2 = 6$ is not affected by the insertion of $b = 3$, except that p_2 is shifted to the next position $i = 3$ in the new principal row.

Case 4: i_k is in the shift path, $k \geq 1$. As mentioned before, the shift path of this example is $\langle i_1, i_2, i_3 \rangle = \langle 3, 4, 6 \rangle$. We get $d'_{i_2} = d'_4 = d_{i_1} = d_3 = 1$, and $d'_{i_3} = d'_6 = d_{i_2} = d_4 = 6$.

Case 5: The remaining cases are those that are greater than or equal to $b + c$ but are not in the shift path. In this example, $p_5 = 9$ and $p_7 = 11$ do not fall in the above cases. Thus, we get $d'_5 = d_5 = 5$, and $d'_7 = d_7 = 7$.

Theorem 3. *The new drop out can be correctly obtained by Equation (1).*

Proof. When a new element b is inserted into the row tower R, five cases are considered for updating the dropout D as follows.

Case 1: $p_i \leq b$.
When b is inserted behind p_i, p_i becomes a leading element of b. Thus, the occurrence of p_i remains unchanged.

Case 2: i is the inserted position index of b in P.
After a new element b is added into the row tower, b will be alive in all rows. Therefore, the drop out of b is equal to the window size.

Case 3: $b + 1 \leq p_{i-1} \leq b + c - 1$.
Since the elements are not deleted, the drop out remains unchanged. Additionally, b is added before them, thus the new drop out will be equal to the old drop out of the preceding element.

Case 4: i_k is in the shift path, $k \geq 1$.
To prove this case, we first explain the meaning of the shift path.

According to Corollary 1, in the row tower R of a sequence A, each $R^n_{j'}$ is a subsequence of $R^n_{i'}$, $1 \leq i' \leq j' \leq n$. In other words, the drop out d_j represents the occurrence of p_j in the principal row ($P = R^n_1$) appearing consecutively from the first row to row d_j in the row tower.

With Definition 8, each index of the shift path $= \langle i_1, i_2, i_3, \ldots \rangle$ indicates the first element in a certain row that is greater than or equal to $b + c$. So they will be sequentially deleted within specific rows until none remains, then the deletion process shifts to the next element.

In the new row tower, p_{i_2} is alive from the first row through row d_{i_1}, since p_{i_1} will be deleted from the first row through row d_{i_1}. So, the old drop out d_{i_1} is copied to become the new drop out d'_{i_2}.

Then, p_{i_2} is deleted from row $d_{i_1} + 1$ through row d_{i_2}, and p_{i_3} will be alive in these rows. Thus, the old drop out d_{i_2} is copied to become the new drop out d'_{i_3}. The other drop outs in the shift path perform the similar works.

Case 5: otherwise.
The remaining elements are all greater than or equal to $b + c$, but they do not exist in the shift path. Therefore, each of them is not the first element greater than or equal to $b + c$ in its respective row of the old row tower. Consequently, they will not be deleted in each row, as only the first element greater than or equal to $b + c$ is removed. Additionally, the newly added element b is placed in front of them. With one element added and one element deleted, their indices remain unchanged.

The proof is complete. □

The algorithm for calculating the LaISW length is formally presented in Algorithm 1. Then, *removal* and *insertion* operations are formally presented in Algorithm 2 and Algorithm 3, respectively. The shift path can be built by Algorithm 4. The *update* operation can be executed using Equation (1), and it is shown in Algorithm 5.

If the sequence of the LaISW answer is desired to be output, we can accomplish this task with a simple backtracking scheme. Each time a new element b is added, we need to keep track of its predecessor, defined as the rightmost element that is less than $b + c$ and preceding b in the window. After the LaISW algorithm finishes, the answer sequence can be obtained by tracing back along the path consisting of these predecessors.

It is easy to see that both time and space complexities of each *removal*, *insertion*, *update* and *shift* are $O(L)$, where L denotes the length of the LaISW answer.

Theorem 4. Algorithm 1 *can solve the LaISW problem in $O(nL)$ time and $O(L)$ space, where n denotes the length of the input sequence, and L denotes the LaISW length.*

Algorithm 1 Main algorithm for computing the LaISW length.

```
Input: A numeric sequence $A = \langle a_1, a_2, \dots, a_n \rangle$, a tolerance constant $c$, and a window size $w$.
Output: The LaISW length
1: Initialize $P$ and $D$ as empty.
2: for $i = 1$ to $w$ do                                          ▷ first window
3:     $P' \leftarrow insert(P, a_i, c)$                           ▷ continuously add new elements
4:     $D \leftarrow update(P, P', D, a_i, c, i)$                  ▷ update drop out
5:     $P \leftarrow P'$
6: end for
7: $length \leftarrow size(D)$
8: for $i = w + 1$ to $n$ do                                      ▷ remaining windows
9:     $P', D \leftarrow remove(P, D)$                             ▷ removal operation
10:    $P'' \leftarrow insert(P', a_i, c)$                         ▷ insertion operation
11:    $D \leftarrow update(P', P'', D, a_i, c, w)$                ▷ update operation
12:    $length \leftarrow \max(length, size(D))$                   ▷ keep maximal length
13:    $P \leftarrow P''$
14: end for
15: return $length$
```

Algorithm 2 Removal operation $remove(P, D)$.

```
Input: A principal row $P$, and a drop out $D$
Output: The principal row $P'$ and drop out $D'$ after removing the first element
1: $d_i \leftarrow d_i - 1$, for $1 \le i \le size(D)$            ▷ subtract 1 from each $d_i$
2: for $i = 1$ to $size(D)$ do                                    ▷ remove the item with $d_i = 0$
3:     if $d_i \ne 0$ then
4:         $append(P', p_i)$                                      ▷ put $p_i$ to the end of $P'$
5:         $append(D', d_i)$                                      ▷ put $d_i$ to the end of $D'$
6:     end if
7: end for
8: return $P'$, $D'$
```

Algorithm 3 Insertion operation $insert(P, a_i, c)$.

```
Input: The principal row $P$, the inserted element $a_i$, the tolerance constant $c$
Output: The principal row $P'$ after inserting $a_i$
1: $j \leftarrow successor(P, a_i + 1)$                ▷ $p_j$ is the successor of $a_i$, $p_j \ge a_i + 1$ in $P$
2: $k \leftarrow successor(P, a_i + c)$                ▷ first element $p_k \ge a_i + c$ in $P$
3: $p'_i \leftarrow p_i$, for $1 \le i \le j - 1$
4: $p'_j \leftarrow a_i$
5: $p'_i \leftarrow p_{i-1}$, for $j + 1 \le i \le k$                    ▷ shift and remove $p_k$
6: $p'_i \leftarrow p_i$, for $k + 1 \le i \le size(P)$   ▷ only $p_k$ is removed, and others remain unchanged
7: return $P'$
```

Algorithm 4 Shift path $shift(P, D, a_i, c)$.

```
Input: A principal row $P$, a drop out $D$, the inserted element $a_i$, the tolerance constant $c$
Output: The shift path $S$
1: $k \leftarrow successor(P, a_i + c)$                   ▷ first element $p_k \ge a_i + c$ in $P$
2: $append(S, k)$                                     ▷ add $k$ to $S$ as the first element of $S$
3: $last \leftarrow k$
4: for $i = k + 1$ to $size(D)$ do                 ▷ search for the next larger element in $D$
5:     if $d_i > d_{last}$ then
6:         $append(S, i)$
7:         $last \leftarrow i$
8:     end if
9: end for
10: return $S$
```

5. A complete example

We illustrate our LaISW algorithm with a complete example shown in Table 4. In the first window, consider $A_{1..6} = \langle 9, 1, 2, 5, 3, 5 \rangle$. After constructing the whole row tower, we obtain the principle row $P = \langle 1, 2, 3, 5, 5 \rangle$ and drop out $D = \langle 2, 3, 5, 4, 6 \rangle$, with an aISW length 5.

When sliding to the second window $A_{2..7} = \langle 1, 2, 5, 3, 5, 10 \rangle$, three operations are performed as follows.

(1) Removal: Initially, $D = D_1^6 = \langle 2, 3, 5, 4, 6 \rangle$. After decreasing each element by one, D becomes $\langle 1, 2, 4, 3, 5 \rangle$ temporarily.

(2) Insertion: The current principal row is $P = R_1^6 = \langle 1, 2, 3, 5, 5 \rangle$, and 10 is the newly added element. Since 10 is greater than all elements in P, it is inserted at the end without removing any element. We get $P' = R_2^7 = \langle 1, 2, 3, 5, 5, 10 \rangle$.

Algorithm 5 Update operation $update(P', P'', D, a_i, c, w)$.

```
Input: The principal row $P'$ after removal operation, the principal row $P''$ after insertion operation, an old drop out $D$, the inserted element $a_i$, the tolerance constant $c$, and the window size $w$
Output: The new drop out $D'$
 1: $S \leftarrow shift(P', D, a_i, c)$                          ▷ shift path
 2: $j \leftarrow successor(P'', a_i + 1)$                        ▷ $p_j$ is the successor of $a_i$, $p_j \geq a_i + 1$ in $P''$
 3: $k \leftarrow successor(P'', a_i + c)$                        ▷ first element $p_k \geq a_i + c$ in $P''$
 4: $d'_i \leftarrow d_i$, for $1 \leq i \leq j - 1$              ▷ Equation (1)
 5: $d'_j \leftarrow w$
 6: $d'_i \leftarrow d_{i-1}$, for $j + 1 \leq i \leq k$
 7: $d'_{S(i+1)} \leftarrow d'_{S(i)}$, for $1 \leq i \leq size(S)$
 8: for $i = k + 1$ to $size(P'')$ do                             ▷ otherwise case in Equation (1)
 9:     if $d'_i = null$ then                                     ▷ not in shift path
10:         $d'_i \leftarrow d_i$
11:     end if
12: end for
13: return $D'$
```

Table 4
A complete example of the LaISW algorithm with $A = \langle 9, 1, 2, 5, 3, 5, 10, 8, 2 \rangle$, $c = 3$, and $w = 6$. The LaISW length is 6. R: row tower; P: principal row; D: drop out; L: answer length.

window	window 1 ($A_{1..6}$) 9, 1, 2, 5, 3, 5	window 2 ($A_{2..7}$) 1, 2, 5, 3, 5, 10	window 3 ($A_{3..8}$) 2, 5, 3, 5, 10, 8	window 4 ($A_{4..9}$) 5, 3, 5, 10, 8, 2
R	1, 2, 3, 5, 5 1, 2, 3, 5, 5 2, 3, 5, 5 3, 5, 5 3, 5 5	1, 2, 3, 5, 5, 10 2, 3, 5, 5, 10 3, 5, 5, 10 3, 5, 10 5, 10 10	2, 3, 5, 5, 8, 10 3, 5, 5, 8, 10 3, 5, 8, 10 5, 8, 10 8, 10 8	2, 3, 5, 8, 10 2, 3, 8, 10 2, 8, 10 2, 10 2 2
P	1, 2, 3, 5, 5	1, 2, 3, 5, 5, 10	2, 3, 5, 5, 8, 10	2, 3, 5, 8, 10
D	2, 3, 5, 4, 6	1, 2, 4, 3, 5, 6	1, 3, 2, 4, 6, 5	6, 2, 1, 3, 4
L	5	6	6	5

(3) Update: The first five elements of P are all less than 10. Therefore, the first five elements in D remain unchanged, resulting in $D'_{1..5} = D_{1..5} = \langle 1, 2, 4, 3, 5 \rangle$. Then, the drop out of the new element is added at position 6. So we set $d'_6 = 6$, which is the window size. Thus, we get $D' = \langle 1, 2, 4, 3, 5, 6 \rangle$.

The same three operations are performed when sliding to the third window $A_{3..8} = \langle 2, 5, 3, 5, 10, 8 \rangle$.

(1) Removal: Initially, the old principal row $P = R^7_2 = \langle 1, 2, 3, 5, 5, 10 \rangle$ and $D = \langle 1, 2, 4, 3, 5, 6 \rangle$. After decreasing each element by one, D becomes $\langle 0, 1, 3, 2, 4, 5 \rangle$ temporarily. Since d_1 becomes 0, it is removed, resulting in that P becomes $\langle 2, 3, 5, 5, 10 \rangle$, and D becomes $\langle 1, 3, 2, 4, 5 \rangle$ temporarily.

(2) Insertion: The current $P = \langle 2, 3, 5, 5, 10 \rangle$, and the next element to be added is $b = 8$. 8 will be inserted before its successor, which is 10. Since no element in P is greater than or equal to $b + c = 8 + 3 = 11$, no element is removed. We get $P' = R^8_3 = \langle 2, 3, 5, 5, 8, 10 \rangle$.

(3) Update: The first four elements of P are smaller than $b = 8$. Therefore, the first four elements in D remain unchanged, resulting in $D'_{1..4} = D_{1..4} = \langle 1, 3, 2, 4 \rangle$. Then, the new element $b = 8$ is added at the fifth position, so d_5 = window size = 6. Finally, for the sixth position, since $P_6 = 10 \leq b + c - 1 = 8 + 3 - 1 = 10$, we get $d'_6 = d_5 = 5$. Then we get $D' = \langle 1, 3, 2, 4, 6, 5 \rangle$.

Sliding to $A_{4..9} = \langle 5, 3, 5, 10, 8, 2 \rangle$, the same three operations are performed as follows.

(1) Removal: P becomes $\langle 3, 5, 5, 8, 10 \rangle$, and D becomes $\langle 2, 1, 3, 5, 4 \rangle$ temporarily.

(2) Insertion: The current $P = \langle 3, 5, 5, 8, 10 \rangle$. The next element $b = 2$ will be inserted before its successor 3. $p_2 = 5$ is the first element in P greater than or equal to $b + c = 2 + 3 = 5$, so $p_2 = 5$ is removed. We get $P' = R^9_4 = \langle 2, 3, 5, 8, 10 \rangle$.

(3) Update: With $P = \langle 3, 5, 5, 8, 10 \rangle$ and $D = \langle 2, 1, 3, 5, 4 \rangle$, we can identify the shift path $\langle i_1, i_2, i_3 \rangle = \langle 2, 3, 4 \rangle$, corresponding to $\langle p_2, p_3, p_4 \rangle = \langle 5, 5, 8 \rangle$ and $\langle d_2, d_3, d_4 \rangle = \langle 1, 3, 5 \rangle$. Then, the new drop out $D' = \langle 6, 2, 1, 3, 4 \rangle$ can be obtained with Equation (1).

After executing the algorithm for the sliding window, the LaISW length can be obtained by $\max\{5, 6, 6, 5\} = 6$.

6. Conclusion

In this paper, we incorporate the concept of 'almost increasing' into the LISW problem, so that the integrated problem allows for slight decreases in the answer. This incorporation makes the algorithm more flexible. If we desire to find the LICS (longest increasing circular subsequence) in the almost increasing (LaICS) problem, we can directly apply the proposed LaISW algorithm. However, this is still a preliminary approach, and there could be more efficient methods available for solving the LaICS problem.

In the future, there is potential to further enrich the LISW framework by incorporating additional concepts, such as the *longest wave subsequence* (LWS) [7]. This extension would not only enable LaISW to accommodate slight decreases but also add the possibility of alternating between increasing and decreasing subsequences. Such enhancements could significantly enhance the flexibility of the algorithms for practical applications.

Furthermore, the primary goal of the LaISW problem is to analyze a single input sequence. There is potential for extension to encompass two input sequences, with one being notably longer than the other. This extension might involve sliding the shorter sequence to identify the *longest common almost increasing subsequence* (LCaIS) [5,14,16] across all windows of the longer sequence.

CRediT authorship contribution statement

Cheng-Han Ho: Writing – original draft, Methodology, Formal analysis. **Chang-Biau Yang:** Writing – review & editing, Validation, Supervision, Project administration, Funding acquisition, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Chang-Biau Yang reports financial support was provided by National Science and Technology Council of Taiwan. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgements

This research work was partially supported by National Science and Technology Council of Taiwan under contract NSTC 112-2221-E-110-026-MY2.

References

[1] M.R. Alam, M.S. Rahman, A divide and conquer approach and a work-optimal parallel algorithm for the lis problem, Inf. Process. Lett. 113 (13) (2013) 470–476.
[2] M.H. Albert, M. Atkinson, D. Nussbaum, J.-R. Sack, N. Santoro, On the longest increasing subsequence of a circular list, Inf. Process. Lett. 101 (2) (2007) 55–59.
[3] M.H. Albert, A. Golynski, A.M. Hamel, A. López-Ortiz, S. Rao, M.A. Safari, Longest increasing subsequences in sliding windows, Theor. Comput. Sci. 321 (2–3) (2004) 405–414.
[4] S. Bespamyatnikh, M. Segal, Enumerating longest increasing subsequences and patience sorting, Inf. Process. Lett. 76 (1–2) (2000) 7–11.
[5] M.T.H. Bhuiyan, M.R. Alam, M.S. Rahman, Computing the longest common almost-increasing subsequence, Theor. Comput. Sci. 930 (2022) 157–178.
[6] E. Chen, L. Yang, H. Yuan, Longest increasing subsequences in windows based on canonical antichain partition, Theor. Comput. Sci. 378 (3) (2007) 223–236.
[7] G.-Z. Chen, C.-B. Yang, The longest wave subsequence problem: generalizations of the longest increasing subsequence problem, in: The 37th Workshop on Combinatorial Mathematics and Computation Theory, Kaohsiung, Taiwan, 2020, pp. 28–33.
[8] M. Crochemore, E. Porat, Fast computation of a longest increasing subsequence and application, Inf. Comput. 208 (9) (2010) 1054–1059.
[9] S. Deorowicz, An algorithm for solving the longest increasing circular subsequence problem, Inf. Process. Lett. 109 (12) (2009) 630–634.
[10] S. Deorowicz, A cover-merging-based algorithm for the longest increasing subsequence in a sliding window problem, Comput. Inform. 31 (6) (2012) 1217–1233.
[11] A. Elmasry, The longest almost-increasing subsequence, Inf. Process. Lett. 110 (16) (2010) 655–658.
[12] J.W. Hunt, T.G. Szymanski, A fast algorithm for computing longest common subsequences, Commun. ACM 20 (1977) 350–353.
[13] Y. Li, L. Zou, H. Zhang, D. Zhao, Longest increasing subsequence computation over streaming sequences, IEEE Trans. Knowl. Data Eng. 30 (6) (2018) 1036–1049.
[14] J.M. Moosa, M.S. Rahman, F.T. Zohora, Computing a longest common subsequence that is almost increasing on sequences having no repeated elements, J. Discret. Algorithms 20 (2013) 12–20.
[15] C. Schensted, Longest increasing and decreasing subsequences, Can. J. Math. 13 (1961) 179–191.
[16] T.T. Ta, Y.-K. Shieh, C.L. Lu, Computing a longest common almost-increasing subsequence of two sequences, Theor. Comput. Sci. 854 (2021) 44–51.
[17] C.-T. Tseng, C.-B. Yang, H.-Y. Ann, Minimum height and sequence constrained longest increasing subsequence, J. Internet Technol. 10 (2) (2009) 173–178.
[18] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Math. Syst. Theory 10 (1) (1976) 99–127.