Bit-Vector Approaches for Solving the Increasing Subsequence Problems with Sliding Windows

Zhi-Cheng Shen, Chang-Biau Yang and Kuo-Si Huang

***Abstract*—In this paper, we define the *most increasing interval subsequence problem with sliding windows* (MIISW), a variant of the most increasing interval subsequence problem. By leveraging bit-vector data structures, we propose efficient algorithms for solving both the *longest increasing subsequence with sliding windows* (LISW) problem and the MIISW problem. Given a numeric sequence A with a fixed window size w, the objective of the LISW problem is to find the largest LIS in every window. Similarly, given an interval sequence X with a fixed window size w, the MIISW problem seeks to find the largest MIIS in every window. When $w \leq \beta$, the time complexities of our algorithms for both problems are O(nw), where n is the length of the input sequence, and β represents the size of a word. When $w > \beta$, the time complexities for both algorithms are O($nw \times \lceil w/\beta \rceil$).**

***Index Terms*—longest increasing subsequence, most increasing interval subsequence, sliding window, bitwise operation, interval**

I. Introduction

The the *longest increasing subsequence* (LIS) problem [1, 3, 4, 7, 11, 13, 14, 17–19] aims to find the longest strictly increasing subsequence within a given numeric sequence. For instance, consider the sequence $A = \langle 7, 6, 8, 4, 10, 5 \rangle$. Its LIS length is 3, and two possible LIS sequences answers are $\langle 7, 8, 10 \rangle$ and $\langle 6, 8, 10 \rangle$.

Over the past decades, researchers have extensively studied the LIS problem due to its wide applicability in various fields. In 1961, Schensted [18] introduced the concept of LIS and proposed an algorithm with a time complexity of O($n \log n$), where n represents the sequence length. Fredman [11] later proved that the comparison operation in LIS requires at least $n \log n - n \log \log n$ + O(n) units of time. Hunt and Szymanski [13] designed another algorithm in 1977 with the same time complexity of O($n \log n$). When the given sequence is a permutation of integers ranging from 1 through n, the time complexity can be reduced to O($n \log \log n$) [13] by using the van Emde Boas tree data structure.

The related researches on the *longest increasing subsequence* (LIS) are summarized in Table I.

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

Zhi-Cheng Shen is with Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan.

Chang-Biau Yang is with Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan. E-mail: cbyang@cse.nsysu.edu.tw. (Corresponding author)

Kuo-Si Huang is with Department of Business Computing, National Kaohsiung University of Science and Technology, Kaohsiung, Taiwan. E-mail: huangks@nkust.edu.tw.

The *longest increasing subsequence with sliding windows* (LISW) problem [3] is a variant of the LIS problem. First defined by Albert *et al.* [3] in 2004, the LISW problem involves finding the longest increasing subsequence within each sliding window of a given sequence. Some algorithms have been proposed to solve this problem. The row tower method proposed by Albert *et al.* [3] achieves a time complexity of O($n \log \log n + OUTPUT$) and a space complexity of O(n). Chen *et al.* [5] proposed an O(nL)-time algorithm based on the canonical antichain partition, where L is the output length. In 2012, Deorowicz [9] employed the cover merge method, achieving a time complexity of O($n \log \log n + \min(nL, n\lceil L^3/w \rceil) \log \lceil w/L^2 + 1 \rceil$). In 2017, Li *et al.* [15] presented the quadruple neighbor list data structure, which solves the problem in O($n \log n$) time. This structure also supports some constrained conditions, including limitations on the sum of elements, and the index difference between the first and last elements. In 2024, Ho and Yang [12] utilized the row tower data structure to solve the *longest almost increasing subsequence with sliding windows* (LaISW) problem, achieving a time complexity of O(nL).

Another related problem is the *longest increasing cyclic subsequence* (LICS) [2, 9], aiming to find the LIS in all rotations of a given numeric sequence A. A rotation means removing some prefix elements and appending them at the end, resulting in a circular sequence. Notably, if the window size is set to $n = |A|$ and the input sequence A is repeated twice, the LISW algorithm can be applied to solve the LICS problem. Therefore, LISW can be regarded as a more general form of LICS. Albert *et al.* [2] first defined LICS in 2007, and solved the problem in O($n^{3/2} logn$) time. Deorowicz [8] later proposed a cover-merge algorithm with a time complexity of O($\min(nL, nlogn + L^3 \log n)$).

Previous research has primarily focused on numeric sequences, but real-world sequences are often more complex. For instance, daily temperatures fluctuate within a certain range, yet they are commonly represented by average values. Such simplification may overlook situations with significant day-night temperature differences. Additionally, obtaining the same result in different regions may be misleading. Therefore, it is crucial to preserve the integrity of the original data.

In this paper, we explore the incorporation of the concept of intervals into the LIS problem. The *most increasing interval subsequence* (MIIS) problem [6] aims to find a subsequence containing the maximum number of intervals. Here, each interval in the sequence is treated as an individual element. This problem was first introduced by Chen and Yang [6] in

TABLE I: The time complexities of the previous LIS, LICS, LISW, and MIISW algorithms. m and n: length of the input sequences; w: window size; L: length of the answer; w': the maximal antichain size; D_n: the depth of the measure at position n; β: size of a word.

Year	Author(s)	Time complexity	Note
The longest increasing subsequence (LIS) problem [12]			
1961	Schensted [18]	O($n \log n$)	Young tableau, binary search
1977	Hunt and Szymanski [13]	O($n \log \log n$)	match pair, van Emde Boas tree
2000	Bespamyatnikh and Segal [4]	O($n \log \log n$)	all answers, patience sorting
2009	Tseng *et al.* [19]	O($n \log \log n$)	minimum height
2010	Crochemore and Porat [7]	O($n \log \log L$)	split blocks
2010	Elmasry [10]	O($n \log L$)	dynamic programming almost increasing
2013	Alam and Rahman [1]	O($n \log n$)	divide-and-conquer
2017	Kloks *et al.* [14]	O($w' \log \min(\frac{n}{w'}, D_n)$)	partially ordered sets
2018	Rani and Rajpoot [17]	O($n \log n$)	divide-and-conquer
2018	Zhu *et al.* [20]	O(nm)	common increasing
The longest increasing cyclic subsequence (LICS) problem			
2007	Albert *et al.* [2]	O($n^{3/2} \log n$)	Monte Carlo
2009	Deorowicz [8]	O($min(nL,$ $n \log n + L^3 \log n)$)	cover merge
The longest increasing subsequence in sliding window (LISW)			
2004	Albert *et al.* [3]	O($n \log \log n + nL$)	row tower
2007	Chen *et al.* [5]	O(nL)	canonical antichain
2012	Deorowicz [9]	O($n \log \log n +$ $min(nL, n\lceil L^3/w \rceil)$ $\log\lceil w/L^2+1 \rceil$)	cover merge
2017	Li *et al.*[15]	O(nw)	quadruple neighbor list
2024	Ho and Yang [12]	O(nL)	row tower, almost increasing
2025	This paper	O($nw \times \lceil w/\beta \rceil$)	window, bit string
The most increasing interval subsequence (MIIS) problem			
2025	Chen and Yang [6]	O($n \log^2 n$)	binary search
2025	This paper	O($nw \times \lceil w/\beta \rceil$)	window, bit-vector

2025, who solved the problem in O($n \log^2 n$) time.

The *most increasing interval subsequence with sliding windows* (MIISW) problem identifies the longest increasing interval subsequence within a fixed-size window, making it useful for discovering increasing price sequences over specific time periods in stock market analysis. For instance, given a time series of stock prices over n trading days, MIISW can be applied to each window of length w to detect upward trends. This approach not only reduces computational complexity but also enables investors to efficiently focus on potential market opportunities.

In this paper, we propose an algorithm utilizing a bit-vector data structure to leverage bitwise operations for accelerating the execution of LISW and MIISW. When $w \leq \beta$, the time complexity of the proposed LISW and MIISW algorithms is O(nw), where w denotes the window size and β is the word size. When $w > \beta$, the time complexity becomes O($nw \times \lceil w/\beta \rceil$).

The remainder of this paper is structured as follows: Section II introduces several variants of the *longest increasing subsequence* (LIS) problem and formally defines the *most increasing interval subsequence in sliding window* (MIISW) problem. In Section III, we present bit-vector algorithms for solving the MIISW and LISW problems, with a time complexity of O(nw), where n represents the length of the input sequence and w is the window size. Finally, our conclusions are summarized in Section V.

II. Preliminaries

A. Longest Increasing Subsequence with Sliding Windows

Definition 1. (longest increasing subsequence with sliding windows) [3] *Given a numeric sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$, and an integer w representing the window size, the LIS length of each window $W_k = A_{k..k+w-1}$, $1 \leq k \leq n-w+1$ is represented by LIS(W_k). The* longest increasing subsequence with sliding windows (*LISW*) *is the maximum value among all LIS(W_k), denoted as LISW(A)=* $\max\{LIS(W_k) \mid 1 \leq k \leq n-w+1\}$.

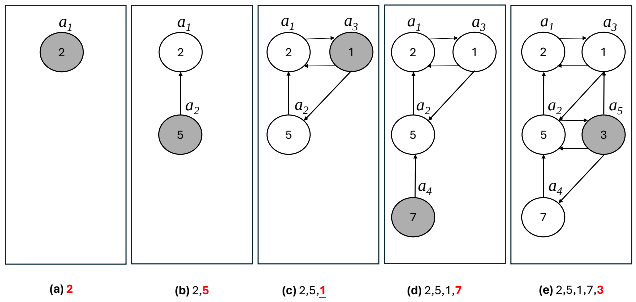

Fig. 1: Insertions of the QN-list with a sequence $A = \langle 2,\ 5,\ 1,\ 7,\ 3\rangle$.

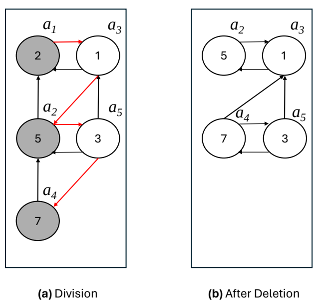

Fig. 2: The deletion of the first element $a_1 = 2$ in the QN-list with a sequence $A = \langle 2,\ 5,\ 1,\ 7,\ 3\rangle$.

For example, if $A = \langle 2,\ 7,\ 1,\ 3,\ 5\rangle$ and $w = 3$, then the LISW sequence would be $W_3 = \langle 1,\ 3,\ 5\rangle$, with a length of 3.

Alber *et al.*[3] introduced this problem in 2004 and proposed an algorithm with a time complexity of O($n \log\log n + OUTPUT$) by using the van Emde Boas data structure. Later, Li *et al.* [15] proposed an algorithm in 2017 with a time complexity of O($n \log n$). Li *et al.* used the QN-list method to address the sliding window problem.

Figure 1 illustrates the insertion of the QN-list with a given sequence $A = \langle 2,\ 5,\ 1,\ 7,\ 3\rangle$. The deletion process can be divided into two parts: horizontal update and vertical update, as shown in Figure 2.

B. Most Increasing Interval Subsequence

Definition 2. (interval) [6] *An* interval*, denoted as* $x = [x_s,\ x_e]$*, represents a consecutive range of integers spanning from a starting point* x_s *to an ending point* x_e*, where* $x_s \leq x_e$*. This interval is inclusive of both* x_s *and* x_e*, and its length, denoted as* $|x|$*, is calculated as* $x_e - x_s + 1$*.*

Definition 3. (interval comparison) [6] *Given two intervals* $x = [x_s,\ x_e]$ *and* $y = [y_s,\ y_e]$*, if* $x_s < y_s$ *and* $x_e < y_e$*, then we say that* $x < y$*.*

For example, consider $x = [3, 7]$ and $y = [5, 9]$. We observe that both the smallest value $x_s = 3$ and the largest value $x_e = 7$ in x are smaller than the corresponding values $y_s = 5$ and $y_e = 9$, respectively. Therefore, we say that $x < y$. However, when comparing interval $z = [5, 11]$ with y, their relationship cannot be definitively determined. In such cases, the relationship is considered undefined.

Definition 4. (MIIS problem) [6] *Given an interval sequence* $X = \langle x_1,\ x_2,\ x_3,\ \ldots,\ x_n\rangle$*, the* most increasing interval subsequence *(MIIS) problem aims to find find a strictly increasing subsequence of intervals that contains the maximum possible number of intervals.*

Suppose that $X = \langle [1, 5],\ [2, 4],\ [2, 6],\ [3, 9],\ [4, 10],\ [5, 8]\rangle$. Then, the answer of MIIS is $\langle [1, 5],\ [2, 6],\ [3, 9],\ [4, 10]\rangle$, containing 4 intervals.

C. Most Increasing Interval Subsequence with Sliding Windows

In this paper, we define the following problem.

Definition 5. (most increasing interval subsequence with sliding windows) *Given an interval sequence* $X = \langle x_1,\ x_2,\ \ldots,\ x_n\rangle$*, where each element* x_i *is an interval, along with a predefined window size* w*, the problem of the* most increasing interval subsequence with sliding windows *(MIISW) aims to find the MIIS in each window* $W_k = \langle x_k,\ x_{k+1},\ \ldots,\ x_{k+w-1}\rangle$*,* $1 \leq k \leq n - w + 1$*. Furthermore, the goal is to decide the value MIISW*$(X) = \max\{$*MIIS*$(W_k)|\ 1 \leq k \leq n - w + 1\}$*, where MIIS(*$W_k$*) represents the length of the MIIS answer in window* W_k*.*

For example, suppose $X = \langle [1, 5],\ [2, 4],\ [2, 10],\ [3, 9],\ [4, 10],\ [5, 8]\rangle$ and $w = 4$. In this case, the MIISW is found in the window $W_2 = \langle [2, 4], [2, 10], [3, 9], [4, 10]\rangle$, with the resulting subsequence $\langle [2, 4], [3, 9], [4, 10]\rangle$, consisting of 3 intervals.

III. The Algorithms with Bit-Vector Operations

In this section, we present a novel sequence algorithm that differs from previous approaches. This algorithm utilizes a machine word of β bits to perform bitwise operations, enabling the simultaneous processing of up to β elements. Most importantly, with slight modifications, this algorithm can be applied to both the LISW problem and the MIISW problem.

A. The Bit-Vectors

Definition 6. (ending list E) *Let* $T = \langle t_1,\ t_2,\ \ldots,\ t_n\rangle$ *be a sequence, where each* t_i *is either a single numeric number, or an interval represented by a starting value and an ending value, and* w *be the window size for sliding the windows.* $E = \langle e_1,\ e_2,\ \ldots,\ e_n\rangle$ *denotes the ending list, where each* e_i *records the LIS or MIIS length ending at* t_i *within window* k*,*

TABLE II: An example for E in each window with $A = \langle 14, 6, 8, 5, 10, 17\rangle$ and $w = 4$.

Window	e_1	e_2	e_3	e_4	e_5	e_6	Comment
$k = 1$: $\langle 14, 6, 8, 5\rangle$	**1**	**1**	**2**	**1**	1	1	LIS length ending at $a_3 = 8$ is 2.
$k = 2$: $\langle 6, 8, 5, 10\rangle$	1	**1**	**2**	**1**	**3**	1	LIS length ending at $a_5 = 10$ is 3.
$k = 3$: $\langle 8, 5, 10, 17\rangle$	1	1	**1**	**1**	**2**	**3**	LIS length ending at $a_6 = 17$ is 3.

for $1 \le i \le n$, $0 \le j \le w-1$, *and* $1 \le k \le n-w+1$. e_i *is given by Equation 1.*

$$e_i = \begin{cases} 1 & \text{if } i < k \text{ or } i > k+w-1 \\ & (outside\ window\ k), \\ 1 + \max\{e_{k+j}\} & \text{if } k \le k+j < i \le k+w-1 \\ & \text{and } t_{k+j} < t_i, \\ 1 & otherwise. \end{cases} \tag{1}$$

For example, consider a numeric sequence $A = \langle 14, 6, 8, 5, 10, 17\rangle$, with a sequence length of $n = 6$ and a window size of $w = 4$. The ending list E is shown in Table II. In the window $k = 2$, we have $e_5 = 1 + \max\{e_{k+j}\} = 3$, with $k + j = 3$.

Definition 7. (length list L) *Let* $T = \langle t_1, t_2, \ldots, t_n\rangle$ *be a sequence, where each* t_i *is either a single numeric number, or an interval,* w *be the window size for sliding the windows, and* β *be the word size, where* $w \le \beta$. *Each* L_i, $1 \le i \le w$, *corresponding length i within window k,* $1 \le k \le n-w+1$, *is a one-dimensional Boolean array with a size of* β *bits, defined as follows.*

$$L_i[j] = \begin{cases} 1 & \text{if } t_{k+j} \text{ is the ending element at the LIS of} \\ & \text{length } i \text{ within window } k,\ 0 \le j \le \beta - 1, \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

The index of window k starts from k, while the index of L_i starts from 0. Therefore, we need to access a_{k+j} in window k to compute $L_i[j]$. For example, consider a numeric sequence $A = \langle 14, 6, 8, 5, 10, 17\rangle$, window size $w = 4$, and $\beta = 5$. The values of each L_i in different windows are shown in Table III. In the window $k = 2$, the length of the LIS ending at $a_5 = 10$ is 3, so $L_3 = 01000$. Here, each L_4 is 0, because there is no LIS of length 4 in any window.

Note that the rightmost (least significant) bit of L_i (i.e. $L_i[0]$) corresponds to e_k (first element in window k). In the bit-vector representation, the rightmost is bit 0 and the leftmost is bit $\beta-1$. This is the opposite of the conventional representation of sequences, where smaller indices are shown on the left. Also note that the index of a sequence starts from 1, while the index of a bit vector starts from 0.

TABLE III: An example of L in each window with $A = \langle 14, 6, 8, 5, 10, 17\rangle$, $w = 4$, and $\beta = 5$.

Window	L_1	L_2	L_3	L_4
$k = 1$: $\langle 14, 6, 8, 5\rangle$	01011	00100	00000	00000
$k = 2$: $\langle 6, 8, 5, 10\rangle$	00101	00010	01000	00000
$k = 3$: $\langle 8, 5, 10, 17\rangle$	00011	00100	01000	00000

TABLE IV: An example of F for each window with $A = \langle 14, 6, 8, 5, 10, 17\rangle$, $w = 4$, and $\beta = 5$.

F	Bit vector	Comment
Window $k = 1$: [14, 6, 8, 5]		
F_1	00000	a_1 is the first element.
F_2	00000	
F_3	00010	$a_3 = 8 > a_2 = 6$.
F_4	00000	
F_5	00000	out of window.
F_6	00000	out of window.
Window $k = 2$: [6, 8, 5, 10]		
F_1	00000	out of window.
F_2	00000	a_2 is the first element.
F_3	00001	$a_3 = 8 > a_2 = 6$.
F_4	00000	$a_4 = 5 < a_2 = 6$, $a_4 = 5 < a_3 = 8$.
F_5	00111	$a_5 = 10 > a_4 = 5$, $a_5 = 10 > a_3 = 8$, $a_5 = 10 > a_2 = 6$.
F_6	00000	out of window.
Window $k = 3$: [8, 5, 10, 17]		
F_1	00000	out of window.
F_2	00000	out of window.
F_3	00000	a_3 is the first element.
F_4	00000	$a_4 = 5 < a_3 = 8$.
F_5	00011	$a_5 = 10 > a_4 = 5$, $a_5 = 10 > a_3 = 8$.
F_6	00111	$a_6 = 17 > a_5 = 10$, $a_6 = 17 > a_4 = 5$, $a_6 = 17 > a_3 = 8$.

B. The Longest Increasing Subsequence Problem

In this section, it is assumed that the window size is w, and the word size is β bits, with $w \le \beta$.

Definition 8. (former list F of LISW) *Let* $A = \langle a_1, a_2, \ldots, a_n\rangle$ *be a sequence, where each* a_i *is a single numeric value,* w *be the window size, and* β *be the word size, where* $w \le \beta$. *Each* F_i $(1 \le i \le n)$ *represents the positions of elements smaller than* a_i *on the left side within window* k $(1 \le k \le n - w + 1)$. *Each* F_i *is a one-dimensional Boolean array of* β *bits, defined as follows:*

$$F_i[j] = \begin{cases} 1 & \text{if } k \le k+j < i \le k+w-1,\ a_{k+j} < a_i, \\ & \text{and } 0 \le j \le \beta - 1, \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

Table IV illustrates an example of F_i.

For initialization, the LISW former list F of the window $k = 1$ is constructed in Algorithm 1. The LISW algorithm is presented in Algorithm 2.

Algorithm 1 Construction of the LISW former list in the window $k = 1$.

Input: A numeric sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$, a window size w and a word size β.
Output: The LISW former list F.

$F_i \leftarrow 0,\ 1 \leq i \leq n$ ▷ initialization, the size of each F_i is β bits.
for $i = 1$ to w **do** ▷ window $k = 1$.
  for $j = 1$ to $i - 1$ **do**
    if $a_j < a_i$ **then** ▷ less than a_i in the left side.
      $F_i \leftarrow F_i \mid (1 << (j-1))$ ▷ bitwise OR operation, set bit $j - 1$ to 1.
return F

Since Algorithm 1 only needs to determine the formers within the window range, its time complexity is O(w^2). Thus, the time complexity of Algorithm 2 is O($nw + w^2$) = O(nw).

Tables V and VI demonstrate the calculations for the windows $k = 1$ and $k = 2$, respectively. The final LISW of A has a length of 3, and the corresponding sequence is $\langle 6, 8, 10 \rangle$. The following explains the steps of Algorithm 2.

Step 1: Initialize window $k = 1$.

1) Compute F for the window $k = 1$ using Algorithm 1.
2) Initialize each element e_i to 1, which indicates that the LIS length ending at each a_i is 1.
3) Initialize L, where L_1 is set to 1 because the LIS ending at a_1 has a length of 1, and the other L_i are initially set to 0 for subsequent updates.

Step 2: Process window $k = 1$. For each element a_i in the window $k = 1$ (starting from a_2), perform the following:

1) Attempt to extend the LIS length ending at a_i, $2 \leq i \leq w$.
2) If F_i & $L_j = 0$, it means that none of the former elements of a_i are in L_j, so the LIS length ending at a_i cannot be extended to $j + 1$.
3) Set e_i to j, indicating that the LIS length ending at a_i is updated to j.

Step 3: Process Windows $k = 2$ to $k = n - w + 1$.

1) **Right-shift F:**
   Perform a logical right shift on F by one bit, filling the leftmost bit with 0 and discarding the rightmost bit, effectively removing the first element from the previous window.
2) **Reinitialize L and e_k:**
   Reinitialize L and set $e_k = 1$, since the LIS lengths in the new window (starting from $k = j$) must be recalculated. The LIS length ending at the first element a_k is set to 1 since it only includes itself.
3) **Update F for the new element a_{k+w-1}:**
   Update F_{k+w-1} by checking each a_j in the new window ($k \leq j \leq k + w - 2$). If $a_j < a_{k+w-1}$, set the corresponding bit in F_{k+w-1} to 1.

Algorithm 2 Calculating the length of the LISW answer.

Input: A numeric sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$, a window size w, and a word size β.
Output: The LISW length len.

Compute F of window $k = 1$ with Algorithm 1
$e_i \leftarrow 1,\ 1 \leq i \leq n$ ▷ initialization.
$L_1 \leftarrow 1$; $L_i \leftarrow 0,\ 2 \leq i \leq w$ ▷ initialization, the size of each L_i is β bits.
for $i = 2$ **to** w **do** ▷ a_i of window $k = 1$.
  for $j = 1$ **to** w **do**
    if $(F_i \ \& \ L_j) = 0$ **then** ▷ no extension, update e_i and L_j.
      $L_j \leftarrow L_j \mid (1 << (i-1))$ ▷ bitwise OR, set bit $i - 1$ to 1.
      $e_i \leftarrow j$; **break** ▷ the length ending at a_i is j.
for $k = 2$ **to** $n - w + 1$ **do** ▷ k is the starting index of the sliding window.
  $F_i >> 1,\ k \leq i \leq k + w - 1$ ▷ logical right shift for F.
  $L_1 \leftarrow 1$; $L_i \leftarrow 0,\ 2 \leq i \leq w$ ▷ initialization, the size of each L_i is β bits.
  $e_k \leftarrow 1$ ▷ a_k is the first element in the window.
  for $j = k$ **to** $k + w - 2$ **do** ▷ update F for new element in the window.
    if $(a_j < a_{k+w-1})$ **then** ▷ a_{k+w-1} is the new (last) element in the window.
      $F_{k+w-1} \leftarrow F_{k+w-1} \mid (1 << (j-k))$ ▷ bitwise OR, set bit $j - k$ to 1.
  for $i = k + 1$ **to** $k + w - 2$ **do** ▷ execute $w - 2$ times.
    if $e_i \geq 2$ **then** ▷ the length ending at a_i might decrease by at most 1.
      if $(F_i \ \& \ L_{e_i - 1}) = 0$ **then** ▷ the length ending at a_i decrease by 1.
        $e_i \leftarrow e_i - 1$ ▷ the length ending at a_i is $e_i - 1$.
    $L_{e_i} \leftarrow L_{e_i} \mid (1 << (i-k))$ ▷ bitwise OR, set bit $i - k$ to 1.
  for $j = 1$ **to** w **do** ▷ process the new element a_{k+w-1} in the window.
    if $(F_{k+w-1} \ \& \ L_j) = 0$ **then** ▷ no extension, update e_i and L_j.
      $L_j \leftarrow L_j \mid (1 << (w-1))$ ▷ bitwise OR, set bit $w - 1$ to 1.
      $e_{k+w-1} \leftarrow j$; **break** ▷ the length ending at a_{k+w-1} is j.
  $len \leftarrow \max(len, \max(E))$ ▷ update the LISW length.

TABLE V: An example for the calculation of the window $k = 1$ with $A = \langle 14, 6, 8, 5, 10, 17 \rangle$, $w = 4$, and $\beta = 5$.

	E	F	L
Step 1: Initial: $\langle 14, 6, 8, 5 \rangle$	$e_1 = 1$ $e_2 = 1$ $e_3 = 1$ $e_4 = 1$ $e_5 = 1$ $e_6 = 1$	$F_1 = 00000$ $F_2 = 00000$ $F_3 = 000\mathbf{1}0$ $F_4 = 00000$	$L_1 = 00001$ $L_2 = 00000$ $L_3 = 00000$ $L_4 = 00000$
Step 2: F_2 & $L_1 = 0$	$e_1 = 1$ $e_2 = \mathbf{1}$ $e_3 = 1$ $e_4 = 1$ $e_5 = 1$ $e_6 = 1$	$F_1 = 00000$ $F_2 = 00000$ $F_3 = 00010$ $F_4 = 00000$	$L_1 = 000\mathbf{1}1$ $L_2 = 00000$ $L_3 = 00000$ $L_4 = 00000$
Step 3: F_3 & $L_1 \neq 0$ F_3 & $L_2 = 0$	$e_1 = 1$ $e_2 = 1$ $e_3 = \mathbf{2}$ $e_4 = 1$ $e_5 = 1$ $e_6 = 1$	$F_1 = 00000$ $F_2 = 00000$ $F_3 = 00010$ $F_4 = 00000$	$L_1 = 00011$ $L_2 = 00\mathbf{1}00$ $L_3 = 00000$ $L_4 = 00000$
Step 4: F_4 & $L_1 = 0$	$e_1 = 1$ $e_2 = 1$ $e_3 = 2$ $e_4 = \mathbf{1}$ $e_5 = 1$ $e_6 = 1$	$F_1 = 00000$ $F_2 = 00000$ $F_3 = 00010$ $F_4 = 00000$	$L_1 = 0\mathbf{1}011$ $L_2 = 00100$ $L_3 = 00000$ $L_4 = 00000$

TABLE VI: An example for the calculation of the window $k = 2$ with $A = \langle 14, 6, 8, 5, 10, 17 \rangle$, $w = 4$, and $\beta = 5$.

	E	F	L
Step 1: Initial: $\langle 6, 8, 5, 10 \rangle$	$e_1 = 1$ $e_2 = \mathbf{1}$ $e_3 = 2$ $e_4 = 1$ $e_5 = 1$ $e_6 = 1$	$F_2 = 00000$ $F_3 = 00001$ $F_4 = 00000$ $F_5 = 00\mathbf{111}$	$L_1 = 00001$ $L_2 = 00000$ $L_3 = 00000$ $L_4 = 00000$
Step 2: F_3 & $L_1 \neq 0$	$e_1 = 1$ $e_2 = 1$ $e_3 = \mathbf{2}$ $e_4 = 1$ $e_5 = 1$ $e_6 = 1$	$F_2 = 00000$ $F_3 = 00001$ $F_4 = 00000$ $F_5 = 00111$	$L_1 = 00001$ $L_2 = 000\mathbf{1}0$ $L_3 = 00000$ $L_4 = 00000$
Step 3: $e_4 = 1$, no need to do & operation.	$e_1 = 1$ $e_2 = 1$ $e_3 = 2$ $e_4 = \mathbf{1}$ $e_5 = 1$ $e_6 = 1$	$F_2 = 00000$ $F_3 = 00001$ $F_4 = 00000$ $F_5 = 00111$	$L_1 = 00\mathbf{1}01$ $L_2 = 00010$ $L_3 = 00000$ $L_4 = 00000$
Step 4: F_5 & $L_1 \neq 0$ F_5 & $L_2 \neq 0$ F_5 & $L_3 = 0$	$e_1 = 1$ $e_2 = 1$ $e_3 = 2$ $e_4 = 1$ $e_5 = \mathbf{3}$ $e_6 = 1$	$F_2 = 00000$ $F_3 = 00001$ $F_4 = 00000$ $F_5 = 00111$	$L_1 = 00101$ $L_2 = 00010$ $L_3 = 0\mathbf{1}000$ $L_4 = 00000$

4) **Process each element in the new window**:
For each a_i ($k+1 \leq i \leq k+w-2$), its LIS length e_i may decrease by at most one, since only the first element of the previous window is removed.
   - If $e_i \geq 2$, check if F_i & $L_{e_i - 1} = 0$. If so, decrease e_i by 1.
   - Update L_{e_i} by setting bit $(i-k)$ to 1 to mark the existence of this length of LIS ending at a_i.

5) **Process the new element** a_{k+w-1}:
Find the smallest j ($1 \leq j \leq w$) with F_{k+w-1} & $L_j = 0$. Set bit $(w-1)$ in L_j to 1 and $e_{k+w-1} = j$, indicating the LIS length ending at a_{k+w-1} is j.

6) **Update the length of LISW**:
Update len as the maximum of the current len and the largest value in E.

C. *The Most Increasing Interval Subsequence Problem*

The QN-list method, proposed by Li *et al.* [15], effectively solves the LIS with sliding windows. However, it cannot be directly applied to the MIISW problem. For example, consider the sequence $X = \langle [6,8], [4,10], [5,11], [7,9] \rangle$ with a window size $w = 4$. As shown in Figure 3, overlapping intervals (e.g., $x_4 = [7,9]$ and $x_3 = [5,11]$) increase the time complexity to $O(n^3)$, making the approach impractical. Therefore, we adopt the bit-vector method to solve MIISW.

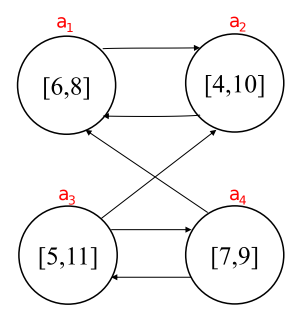

Fig. 3: The QN-list for the most increasing interval subsequence with sliding windows.

To tackle the MIISW problem using bit-vector operations, we also define a former list to record the smaller elements on the left side, as follows.

Definition 9. (former list M of MIISW) *Let $X = \langle x_1, x_2, \ldots, x_n \rangle$ be an interval sequence, where each x_i is an interval represented by a starting value and an ending value, w be the window size for sliding windows, and β be the word size, where $w \leq \beta$. Each M_i ($1 \leq i \leq n$) corresponds to the smaller elements of x_i on the left side within window k ($1 \leq k \leq n - w + 1$). It is a one-dimensional Boolean array with*

TABLE VII: The values of M for each window with $X = \langle[2,7], [3,9], [8,10], [6,9], [2,12], [3,13]\rangle$, $w = 4$, and $\beta = 5$.

M	Bit vector	Comment
Window $k = 1$**:** $\langle[2,7]$**,** $[3,9]$**,** $[8,10]$**,** $[6,9]\rangle$		
M_1	00000	x_1 is the first element.
M_2	00001	$x_2 = [3,9] > x_1 = [2,7]$.
M_3	00011	$x_3 = [8,10] > x_1 = [2,7]$, $x_3 = [8,10] > x_2 = [3,9]$.
M_4	00001	$x_4 = [6,9] > x_1 = [2,7]$.
M_5	00000	out of window.
M_6	00000	out of window.
Window $k = 2$**:** $\langle[3,9]$**,** $[8,10]$**,** $[6,9]$**,** $[2,12]\rangle$		
M_1	00000	out of window.
M_2	00000	x_2 is the first element.
M_3	00001	$x_3 = [8,10] > x_2 = [3,9]$.
M_4	00000	
M_5	00000	
M_6	00000	out of window.
Window $k = 3$**:** $\langle[8,10]$**,** $[6,9]$**,** $[2,12]$**,** $[3,13]\rangle$		
M_1	00000	out of window.
M_2	00000	out of window.
M_3	00000	x_3 is the first element.
M_4	00000	
M_5	00000	
M_6	00100	$x_6 = [3,13] > x_5 = [2,12]$.

a size of β bits, defined as follows.

$$M_i[j] = \begin{cases} 1 & \text{if } k \le k+j < i \le k+w-1,\ a_{k+j} < a_i, \\ & \text{and } 0 \le j \le \beta - 1, \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

Examples for the values of M in each window are shown in Table VII.

The MIISW former list for window $k = 1$ is constructed in Algorithm 3, and the complete MIISW algorithm is presented in Algorithm 4. The time complexity analysis for Algorithm 4 is the same as that of Algorithm 2, both being $O(nw)$.

Tables VIII and IX illustrate the calculations of the windows $k = 1$ and $k = 2$, respectively.

After sliding through all the windows, the final MIISW(X) is 3. The MIISW sequence is $\langle[2,7], [3,9], [8,10]\rangle$.

By examining the methods for calculating LISW and MIISW, we can see that, except for the different algorithms used for F and M, the remainder of the algorithms are identical. The reason is that we have transformed the data structure of this problem into bit vectors, allowing us to solve LISW and MIISW using simple bitwise operations.

IV. Window Size Exceeding Word Size

In this section, we discuss the case where the window size w is greater than the word size β, that is, $w > \beta$. To handle this case, we use a long bit vector $S = \langle s_{m-1}, s_{m-2}, \ldots, s_0\rangle$ to store the entire bit sequence, where each s_i is a bit vector of β bits. Note that the bits in s_0 correspond to the least significant bits.

In cases where $w > \beta$, the details of bitwise operations are implemented in Algorithm 5, Algorithm 6, and Algorithm 7. Each of these algorithms requires O($\lceil w/\beta \rceil$) time. Therefore, the overall time complexity of the algorithm for solving these problem with $w > \beta$ is O($nw \times \lceil w/\beta \rceil$).

See Table X for an example. Consider calculating the length

Algorithm 3 Initialization of the MIISW former list.

Input: An interval sequence $X = \langle x_1, x_2, x_3, \ldots, x_n\rangle$ and a window size w.

Output: The MIISW former list M.

$M_i \leftarrow 0,\ 1 \le j \le n$ ▷ initialize candidate, the size of each M_i is β bits.

for $i = 1$ to w **do** ▷ window $k = 1$.

  for $j = 1$ to $i - 1$ **do**

    if $(x_j < x_i)$ **then** ▷ by Definition 3, less then a_i in the left side.

      $M_i \leftarrow\mid (M_i \leftarrow 1 << (j-1))$ ▷ bitwise OR operation, set bit $j-1$ to 1.

return M

TABLE VIII: The calculation of the window $k = 1$ for MIISW with $X = \langle[2,7], [3,9], [8,10], [6,9], [2,12], [3,13]\rangle$, $w = 4$, and $\beta = 5$.

	E	M	L
Step 1: Initial: $\langle[2,7], [3,9],$ $[8,10], [6,9]\rangle$	$e_1 = 1$ $e_2 = 1$ $e_3 = 1$ $e_4 = 1$ $e_5 = 1$ $e_6 = 1$	$M_1 = 00000$ $M_2 = 0000\mathbf{1}$ $M_3 = 000\mathbf{11}$ $M_4 = 0000\mathbf{1}$	$L_1 = 00001$ $L_2 = 00000$ $L_3 = 00000$ $L_4 = 00000$
Step 2: M_2 & $L_1 \neq 0$ M_2 & $L_2 = 0$	$e_1 = 1$ $e_2 = \mathbf{2}$ $e_3 = 1$ $e_4 = 1$ $e_5 = 1$ $e_6 = 1$	$M_1 = 00000$ $M_2 = 00001$ $M_3 = 00011$ $M_4 = 00001$	$L_1 = 00001$ $L_2 = 000\mathbf{1}0$ $L_3 = 00000$ $L_4 = 00000$
Step 3: M_3 & $L_1 \neq 0$ M_3 & $L_2 \neq 0$ M_3 & $L_3 = 0$	$e_1 = 1$ $e_2 = 2$ $e_3 = \mathbf{3}$ $e_4 = 1$ $e_5 = 1$ $e_6 = 1$	$M_1 = 00000$ $M_2 = 00001$ $M_3 = 00011$ $M_4 = 00001$	$L_1 = 00001$ $L_2 = 00010$ $L_3 = 00\mathbf{1}00$ $L_4 = 00000$
Step 4: M_4 & $L_1 \neq 0$ M_4 & $L_2 = 0$	$e_1 = 1$ $e_2 = 2$ $e_3 = 3$ $e_4 = \mathbf{2}$ $e_5 = 1$ $e_6 = 1$	$M_1 = 00000$ $M_2 = 00001$ $M_3 = 00011$ $M_4 = 00001$	$L_1 = 00001$ $L_2 = 0\mathbf{1}010$ $L_3 = 00100$ $L_4 = 00000$

Algorithm 4 Computing the MIISW

Input: An interval sequence $X = \langle x_1, x_2, x_3, \ldots, x_n \rangle$, and a window size w.
Output: The MIISW length len.

Compute M of window $k = 1$ with Algorithm 3
$e_i \leftarrow 1,\ 1 \le i \le n$ ▷ initialization.
$L_1 \leftarrow 1$; $L_i \leftarrow 0,\ 2 \le i \le w$ ▷ initialization, the size of each L_i is β bits.
for $i = 2$ **to** w **do** ▷ x_i of window $k = 1$.
  for $j = 1$ **to** w **do**
    if $(M_i \ \&\ L_j) = 0$ **then** ▷ no extension, update e_i and L_j.
      $L_j \mid (1 << (i-1))$ ▷ bitwise OR, set bit $i-1$ to 1.
      $e_i \leftarrow j$; **break** ▷ the length ending at x_i is j.
for $k = 2$ **to** $n - w + 1$ **do** ▷ k is the starting index of the sliding window.
  $M_i >> 1,\ k \le i \le k + w - 1$ ▷ logical right shift for M.
  $L_1 \leftarrow 1$; $L_i \leftarrow 0,\ 2 \le i \le w$ ▷ initialization, the size of each L_i is β bits.
  $e_k \leftarrow 1$ ▷ x_k is the first element in the window.
  for $j = k$ **to** $k + w - 2$ **do** ▷ update M for new element in the window.
    if $(x_j < x_{k+w-1})$ **then** ▷ x_{k+w-1} is the new (last) element in the window.
      $M_{k+w-1} \leftarrow M_{k+w-1} \mid (1 << (j-k))$ ▷ bitwise OR, set bit $j-k$ to 1.
  for $i = k+1$ **to** $k + w - 2$ **do** ▷ execute $w-2$ times.
    if $e_i \ge 2$ **then** ▷ the length ending at a_i might decrease by at most 1.
      if $(M_i \ \&\ L_{e_i - 1}) = 0$ **then** ▷ the length ending at x_i decrease by 1.
        $e_i \leftarrow e_i - 1$ ▷ the length ending at x_i is $e_i - 1$.
        $L_{e_i} \leftarrow L_{e_i} \mid (1 << (i-k))$ ▷ bitwise OR, set bit $i-k$ to 1.
  for $j = 1$ **to** w **do** ▷ process the new element x_{k+w-1} in the window.
    if $(M_{k+w-1} \ \&\ L_j) = 0$ **then** ▷ no extension, update e_i and L_j.
      $L_j \leftarrow L_j \mid (1 << (w-1))$ ▷ bitwise OR, set bit $w-1$ to 1.
      $e_{k+w-1} \leftarrow j$; **break** ▷ the length ending at x_{k+w-1} is j.
  $len \leftarrow \max(len, \max(E))$ ▷ update the MIISW length.

TABLE IX: The calculation of the window $k = 2$ for MIISW with $X = \langle [2,7], [3,9], [8,10], [6,9], [2,12], [3,13] \rangle$, $w = 4$, and $\beta = 5$.

	E	M	L
Step 1: Initial: $\langle [3,9], [8,10], [6,9], [2,12] \rangle$	$e_1 = 1$ $e_2 = \mathbf{1}$ $e_3 = 3$ $e_4 = 2$ $e_5 = 1$ $e_6 = 1$	$M_2 = 00000$ $M_3 = 00001$ $M_4 = 00000$ $M_5 = 00000$	$L_1 = 00001$ $L_2 = 00000$ $L_3 = 00000$ $L_4 = 00000$
Step 2: $M_3 \ \&\ L_2 = 0$	$e_1 = 1$ $e_2 = 1$ $e_3 = \mathbf{2}$ $e_4 = 2$ $e_5 = 1$ $e_6 = 1$	$M_2 = 00000$ $M_3 = 00001$ $M_4 = 00000$ $M_5 = 00000$	$L_1 = 00001$ $L_2 = 000\mathbf{1}0$ $L_3 = 00000$ $L_4 = 00000$
Step 3: $M_4 \ \&\ L_1 = 0$	$e_1 = 1$ $e_2 = 2$ $e_3 = 2$ $e_4 = \mathbf{1}$ $e_5 = 1$ $e_6 = 1$	$M_2 = 00000$ $M_3 = 00001$ $M_4 = 00000$ $M_5 = 00000$	$L_1 = 00\mathbf{1}01$ $L_2 = 00010$ $L_3 = 00000$ $L_4 = 00000$
Step 4: $M_5 \ \&\ L_1 = 0$	$e_1 = 1$ $e_2 = 1$ $e_3 = 2$ $e_4 = 1$ $e_5 = \mathbf{1}$ $e_6 = 1$	$M_2 = 00000$ $M_3 = 00001$ $M_4 = 00000$ $M_5 = 00000$	$L_1 = 0\mathbf{1}101$ $L_2 = 00010$ $L_3 = 00000$ $L_4 = 00000$

Algorithm 5 Logical right shift of the former list by 1 bit.

Input: A long bit vector $S = \langle s_{m-1}, s_{m-2}, \ldots, s_0 \rangle$, where s_i has a word size of β bits.
Output: $S >> 1$, logical right shift of S by 1 bit.

$c_i \leftarrow s_i \ \&\ 1,\ n-1 \ge i \ge 1$ ▷ extract the LSB of s_i to c_i.
$s_{n-1} >> 1$
for $i = n-2$ to 0 **do**
  $s_i >> 1$
  $s_i \leftarrow s_i \mid (c_{i+1} << (\beta - 1))$ ▷ update the MSB of s_i, set LSB of s_{i+1} to bit $\beta - 1$ of s_i.
return S

of the LIS ending at $a_{12} = 15$. We first represent F_{12} using a long bit vector:

$$S = [s_2, s_1, s_0],$$

where each s_i is a β-bit word. Similarly, L_1 to L_6 are represented by P_1 to P_6. Using Algorithm 6, we perform segmented bitwise AND operations between S and each P_i. We find that:

$$S \ \&\ P_6 \ne 0.$$

Thus, we update P_6 using Algorithm 7. We logically left-shift the value 1 by 11 bits and combine it with P_6 through a bitwise

Algorithm 6 Checking whether the bitwise AND operation result is zero.

Input: Two long bit vectors $S = \langle s_{m-1}, s_{m-2}, \ldots, s_0 \rangle$ and $P = \langle p_{m-1}, p_{m-2}, \ldots, p_0 \rangle$, where each s_i and each p_i has a word size of β bits.
Output: The Boolean value B of $(S \ \& \ P = 0)$.

B =*False*
for $i = m - 1$ to 0 **do**
  if $s_i \ \& \ p_i = 0$ **then** $\triangleright$ segmenting bitwise AND.
    B =*True*
    break
return B

Algorithm 7 Left shift bitwise OR.

Input: A long bit vector $P = \langle p_{m-1}, p_{m-2}, \ldots, p_0 \rangle$, where each p_i has a word size of β bits, the window starting index k, and the position j of element a_j in the sequence.
Output: $P \leftarrow P \mid (1 << (j - k))$, set bit $(j - k)$ of P to 1.

$i = (j - k)/\beta$ $\triangleright$ position j within window k is in word p_i.
$r = (j - k) \bmod \beta$ $\triangleright$ position to be updated within word p_i
$p_i \leftarrow p_i \mid (1 << r)$
return P

TABLE X: An example for calculating the length of the LIS ending at a_{12}, for the window $k = 1$, with $A = \langle 7, 15, 2, 15, 15, 6, 8, 11, 17, 15, 14, 15, 16 \rangle$, $w = 12$, and $\beta = 5$.

S	s_2	s_1	s_0
[00001, 00111, 00101]	00001	00111	00101
	E	P_1 **to** P_3	P_4 **to** P_6
$S \ \& \ P_1 \neq 0$ $S \ \& \ P_2 \neq 0$ $S \ \& \ P_3 \neq 0$ $S \ \& \ P_4 \neq 0$ $S \ \& \ P_5 \neq 0$ $S \ \& \ P_6 = 0$	$e_1 = 1$ $e_2 = 2$ $e_3 = 1$ $e_4 = 2$ $e_5 = 2$ $e_6 = 2$ $e_7 = 3$ $e_8 = 4$ $e_9 = 5$ $e_{10} = 5$ $e_{11} = 5$ $e_{12} = \underline{\mathbf{6}}$ $e_{13} = 1$	$P_1 =$ [00000, 00000, 00101] $P_2 =$ [00000, 00001, 11010] $P_3 =$ [00000, 00010, 00000]	$P_4 =$ [00000, 00100, 00000] $P_5 =$ [00001, 11000, 00000] $P_6 =$ [000**1**0, 00000, 00000]

OR operation. Since:

$$\lfloor (12 - 1)/\beta \rfloor = 2 \quad \text{and} \quad (12 - 1) \bmod \beta = 1,$$

the bit $p_2[1]$ of P_6 is set to 1.

As shown in Table XI, the logical right shift of F_{12} is explained. We use the method of recording carry bits c_i, $n - 1 \geq i \geq 1$ to determine whether the MSB of each s_i should be shifted in as 1 or 0.

TABLE XI: Right shift of F_{12} =[00001,00111,00101] by one bit.

	s_2	s_1	s_0	**Comment**
Step 1: Calculate c_i, $2 \geq i \geq 1$	00001	00111	00101	$s_2 \ \& \ 1 = 1$, set $c_2 = 1$; $s_1 \ \& \ 1 = 1$, set $c_1 = 1$.
Step 2: Right shift s_2	00000	00111	00101	right shift s_2.
Step 3: Right shift s_1	00000	10011	00101	right shift s_1, and (MSB of s_1)$\mid c_2$.
Step 4: Right shift s_0	00000	10011	10010	right shift s_0, and (MSB of s_0)$\mid c_1$.

When the window size exceeds the word size, we can compensate for this limitation through additional steps. However, the time complexity increases as the gap between the window size and the word size widens. The larger the discrepancy, the greater the increase in time complexity.

V. Conclusion

In this paper, we employ the bit-vector method to solve the *longest increasing subsequence in sliding window* (LISW) and *most increasing interval subsequence in sliding window* (MIISW) problems. When $w \leq \beta$, the time complexity of both algorithms is O(nw), where n is the sequence length, w is the window size, and β is the word size. When $w > \beta$, the time complexity increases to O($nw \times \lceil w/\beta \rceil$). The use of bitwise operations not only resolves the challenge of crossing predecessors in MIISW but also greatly enhances computational efficiency.

Furthermore, because the *longest almost increasing subsequence in sliding window* (LaISW) requires comparison with all preceding elements, and currently there is no efficient method in our algorithm for quickly comparing all previous elements during computation, this challenge may be addressed in the future. Additionally, it may be possible to solve the *longest almost wave subsequence* (LaWS) problem [16] using the bit-vector method.

References

[1] M. R. Alam and M. S. Rahman, "A divide and conquer approach and a work-optimal parallel algorithm for the LIS problem," *Information Processing Letters*, Vol. 113, No. 13, pp. 470–476, 2013.

[2] M. H. Albert, M. D. Atkinson, D. Nussbaum, J.-R. Sack, and N. Santoro, "On the longest increasing subsequence of a circular list," *Information Processing Letters*, Vol. 101, No. 2, pp. 55–59, 2007.

[3] M. H. Albert, A. Golynski, A. M. Hamel, A. López-Ortiz, S. Rao, and M. A. Safari, "Longest increasing subsequences in sliding windows," *Theoretical Computer Science*, Vol. 321, No. 2-3, pp. 405–414, 2004.

[4] S. Bespamyatnikh and M. Segal, "Enumerating longest increasing subsequences and patience sorting," *Information Processing Letters*, Vol. 76, No. 1-2, pp. 7–11, 2000.

[5] E. Chen, L. Yang, and H. Yuan, "Longest increasing subsequences in windows based on canonical antichain partition," *Theoretical Computer Science*, Vol. 378, No. 3, pp. 223–236, 2007.

[6] G.-T. Chen and C.-B. Yang, "Efficient algorithms for the increasing interval subsequence problems," *Journal of Information Science and Engineering*, Vol. 41, No. 3, pp. 525–541, 2025.

[7] M. Crochemore and E. Porat, "Fast computation of a longest increasing subsequence and application," *Information and Computation*, Vol. 208, No. 9, pp. 1054–1059, 2010.

[8] S. Deorowicz, "An algorithm for solving the longest increasing circular subsequence problem," *Information Processing Letters*, Vol. 109, No. 12, pp. 630–634, 2009.

[9] S. Deorowicz, "A cover-merging-based algorithm for the longest increasing subsequence in a sliding window problem," *Computing and Informatics*, Vol. 31, No. 6, pp. 1217–1233, 2012.

[10] A. Elmasry, "The longest almost-increasing subsequence," *Information Processing Letters*, Vol. 110, No. 16, pp. 655–658, 2010.

[11] M. L. Fredman, "On computing the length of longest increasing subsequences," *Discrete Mathematics*, Vol. 11, No. 1, pp. 29–35, 1975.

[12] C.-H. Ho and C.-B. Yang, "The longest almost increasing subsequence problem with sliding windows," *Theoretical Computer Science*, Vol. 1005, p. 114639, 2024.

[13] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, Vol. 20, pp. 350–353, 1977.

[14] T. Kloks, R. B. Tan, and J. van Leeuwen, "Tracking maximum ascending subsequences in sequences of partially ordered data," *Technical Report UU-CS-2017-010*, Utrecht, Netherlands, 2017.

[15] Y. Li, L. Zou, H. Zhang, and D. Zhao, "Longest increasing subsequence computation over streaming sequences," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 30, No. 6, pp. 1036–1049, 2017.

[16] S.-C. Lin, K.-S. Huang, and C.-B. Yang, "The longest almost wave subsequence problem," *Proceedings of the 40th Workshop on Combinatorial Mathematics and Computation Theory*, Taoyuan, Taiwan, pp. 1–8, 2023.

[17] S. Rani and D. S. Rajpoot, "Improvised divide and conquer approach for the lis problem," *Journal of Discrete Algorithms*, Vol. 48, pp. 17–26, 2018.

[18] C. Schensted, "Longest increasing and decreasing subsequences," *Canadian Journal of Mathematics*, Vol. 13, pp. 179–191, 1961.

[19] C.-T. Tseng, C.-B. Yang, and H.-Y. Ann, "Minimum height and sequence constrained longest increasing subsequence," *Journal of Internet Technology*, Vol. 10, No. 2, pp. 173–178, 2009.

[20] D. Zhu, L. Wang, T. Wang, and X. Wang, "A simple linear space algorithm for computing a longest common increasing subsequence," *IAENG International Journal of Computer Science*, Vol. 45, No. 3, pp. 472–477, 2018.