International Journal of Foundations of Computer Science

The Longest Wave Subsequence Problem: Generalizations of the Longest Increasing Subsequence Problem *

Guan-Zhi Chen

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

Chang-Biau Yang

Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan
cbyang@cse.nsysu.edu.tw†

Yu-Cheng Chang

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

Received (23 Nov. 2023)
Revised (8 April 2024)
Accepted (Day Month Year)
Communicated by (xxxxxxxxxx)

The *longest increasing subsequence* (LIS) problem aims to find the subsequence exhibiting an increasing trend in a numeric sequence with the maximum length. In this paper, we generalize the LIS problem to the *longest wave subsequence* (LWS) problem, which encompasses two versions: LWSt and LWSr. Given a numeric sequence A of distinct values and a target trend sequence T, the LWSt problem aims to identify the longest subsequence of A that preserves the trend of the prefix of T. And, the LWSr problem aims to find the longest subsequence of A within r segments, alternating increasing and decreasing subsequences. We propose two efficient algorithms for solving the two versions of the LWS problem. For the LWSt problem, the time complexity of our algorithm is $O(n \log n)$, where n represents the length of the given numeric sequence A. Additionally, we propose an $O(rn \log n)$-time algorithm for solving the LWSr problem. In both algorithms, we utilize the priority queues for the insertion, deletion, and successor operations.

Keywords: longest increasing subsequence; longest wave subsequence; trend-preserving; increasing/decreasing segment; priority queue.

*A preliminary version of this paper was presented at the 37th Workshop on Combinatorial Mathematics and Computation Theory, July, 2020, Kaohsiung, Taiwan.
†Corresponding author.

1. Introduction

The pattern matching problem [12, 20] has been studied extensively in stringology. Given a text A and a pattern B, the pattern matching problem is to find all occurrences of B in A. This problem finds applications in diverse fields such as image processing [28, 29], database searching [7, 27], and genetic sequence analysis [13, 15, 25]. Sometimes, the focus shifts from finding a specific pattern to identifying trends in a sequence. For example, in the stock market, analysts may be interested in identifying changing patterns of stock prices over a period rather than individual price points. This has led to the *order-preserving pattern matching* (OPPM) problem [5, 6, 14, 16, 18, 21], which has attracted considerable attention in recent years.

Given two numeric strings $A = \langle a_1, a_2, \cdots, a_n \rangle$ and $B = \langle b_1, b_2, \cdots, b_n \rangle$, with the same length, A and B are said to be *order-preserving* if the rank of each a_i in A is identical to the rank of b_i. Here, the rank of a_i in A means the number of elements in A which are less than or equal to a_i. For example, suppose that $A = \langle 24,\ 31,\ 42,\ 40,\ 26 \rangle$ and $B = \langle 10,\ 18,\ 25,\ 21,\ 16 \rangle$. Then, we have ranks $\langle 1, 3, 5, 4, 2 \rangle$ for both A and B. Thus, A and B are order-preserving.

In this paper, we introduce the concept of *trend-preserving*, especially relevant in time series prediction, such as forecasting the future price of a stock. A and B are said to be *trend-preserving* if $\text{sign}(a_i - a_{i-1}) = \text{sign}(b_i - b_{i-1})$, for $2 \leq i \leq n = |A| = |B|$. For example, $A' = \langle 28, 31, 42, 40, 26 \rangle$ and $B = \langle 10, 18, 25, 21, 16 \rangle$ are trend-preserving. But A and B are not order-preserving, because the ranks for A' are $\langle 2, 3, 5, 4, 1 \rangle$ and the ranks for B are $\langle 1, 3, 5, 4, 2 \rangle$.

While order-preserving focuses on the relative ranks of elements, trend-preserving considers the overall trend or pattern of the sequences. For example, in stock price prediction, it may be more important to preserve the trend of price movements over time rather than the exact order of prices. Therefore, the trend-preserving concept provides a more flexible criterion for certain applications compared to the strict order-preserving criteria.

The *longest increasing subsequence* (LIS) problem has gained significant attention from researchers in the past [1, 2, 3, 8, 9, 11, 17, 23, 24, 26, 31]. Given a numeric sequence A of length n, the LIS is the increasing subsequence of A with the maximum length. In 1961, Schensted [26] first defined the LIS problem, and he also presented an $O(n \log n)$-time algorithm based on the *Young tableau*. In 1977, if A is a permutation of $\{1, 2, \ldots, n\}$, Hunt and Szmanski [17] gave an $O(n \log \log n)$-time algorithm by using the van Emde Boas tree [4]. In 2000, Bespamyatnikh and Segal [3] contributed an $O(n \log \log n)$-time algorithm, capable of reporting all LIS answers. In 2010, if A is a permutation of $\{1, 2, \ldots, n\}$ and the maximum length of the LIS is parameterized to a value L', Crochemore and Porat [8] showed that the problem can be solved in $O(n \log \log L')$ time on the RAM model. In 2013, Alam and Rahman [1] applied the divide-and-conquer approach to finding the LIS in $O(n \log n)$ time.

Table 1. The time complexities of the algorithms for the LIS-related problems. n: length of the input sequence A; L': maximum length of the answer; r: maximum number of segments in LWSr; L: answer length; C: constrained sequence; w: window size; w': size of the maximum antichain; LaIS: longest almost increasing subsequence; LISW: longest increasing subsequence with sliding windows.

Year	Author(s)	Time complexity	Note
1961	Schensted [26]	$O(n \log n)$	LIS, binary search, Young tableau
1977	Hunt and Szymanski [17]	$O(n \log \log n)$	LIS, permutation, van Emde Boas tree
2000	Bespamyatnikh and Segal [3]	$O(n \log \log n)$	all LIS answers
2009	Tseng *et al.* [30]	MHLIS: $O(n \log n)$ SCLIS: $O(n \log(n + \|C\|))$	minimum height LIS, sequence constrained LIS
2010	Crochemore and Porat [8]	$O(n \log \log L')$	parameterized LIS, permutation, van Emde Boas tree
2013	Alam and Rahman [1]	$O(n \log n)$	LIS, divide-and-conquer
2017	Kloks *et al.* [19]	$O(w'n \log \min(\frac{n}{w'}, L))$	partially ordered set
2010	Amr Elmasry [10]	$O(n \log L)$	LaIS, dynamic programming
2018	Li *et al.* [22]	$O(nw)$	LISW, quadruple neighbor list
2024	This paper	$O(n \log n)$	the LWSt problem
2024	This paper	$O(rn \log n)$	the LWSr problem

The time complexities of the previously published algorithms for the LIS-related problems are listed in Table 1.

In this paper, we introduce the *longest wave subsequence* (LWS) problem, a generalization of the LIS problem. The LWS problem here includes two versions of problems. The first one is referred to as the LWSt problem, which finds the *LWS with a given trend*. Given a numeric sequence A composed of n distinct values and a target trend sequence T of length n, the LWSt problem is to find the longest subsequence in A that is *trend-preserving* with respect to the prefix of T. When the trend sequence T comprises only one type of sequence, such as strictly increasing, the LWSt problem degenerates into the traditional LIS problem. For the LWSt problem, we propose an algorithm with $O(n \log n)$ time, drawing inspiration from algorithms designed for solving the LIS problem [24, 31].

The second variant is the *LWS problem within r segments*, referred to as the LWSr problem. In various scenarios, such as analyzing stock market trends, a series of data may exhibit distinct trend segments, alternating increasing and decreasing. The LWSr problem aims to identify the longest subsequence within r segments in a given sequence A, where each subsequence forming a segment is either increasing or decreasing. It is worth noting that when $r = 1$, the LWSr problem also degenerates into the traditional LIS problem. We present an $O(rn \log n)$-time algorithm for efficiently solving the LWSr problem.

The rest of this paper is organized as follows. Section 2 introduces the background knowledge of the LIS problem, and presents the formal definitions of the LWSt and LWSr problems. In Section 3, we propose an algorithm for solving the LWSt problem. In Section 4, we present the LWSr algorithm. Finally, the conclusion and discussion are given in Section 5.

Table 2. An example of the folklore algorithm [24, 31] for finding the LIS of $A = \langle 4, 6, 1, 7, 3, 5, 8, 2\rangle$, where $S[x]$ stores the best (smallest) ending element for the current IS of length x. The LIS content is $\langle 1, 3, 5, 8\rangle$, with length 4.

A \ S(length)	1	2	3	4
$a_1 = 4$	**4**			
$a_2 = 6$	4	**6**		
$a_3 = 1$	**1**	6		
$a_4 = 7$	1	6	**7**	
$a_5 = 3$	1	**3**	7	
$a_6 = 5$	1	3	**5**	
$a_7 = 8$	1	3	5	**8**
$a_8 = 2$	1	**2**	5	8

2. Preliminaries

2.1. *Notations and the Longest Increasing Subsequence Problem*

In this paper, we denote a sequence using an upper-case letter, such as A or T. Given a sequence $A = \langle a_1, a_2, \ldots, a_n\rangle$ with $|A| = n$, a_i represents the ith element of A; the notation $i..j$ is the index range from index i to index j; $A_{i..j}$ represents the substring or consecutive elements of A with index range $i..j$. We set $A_{i..j} = \emptyset$ if $i > j$.

Given a numeric sequence $A = \langle a_1, a_2, \ldots, a_n\rangle$, an *increasing subsequence* (IS) of A is a subsequence $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_k}\rangle$, obtained from A, where $a_{i_x} < a_{i_y}$ and $1 \le i_x < i_y \le n$ for $1 \le x < y \le k$. The *longest increasing subsequence* (LIS) [17, 24, 26, 31] is the IS of A with the maximum length. Note that the LIS may not be unique. For example, suppose that $A = \langle 4, 6, 1, 7, 3, 5, 8, 2\rangle$. Then its LIS length is 4, with answer sequences $\langle 4, 6, 7, 8\rangle$, or $\langle 1, 3, 5, 8\rangle$. A *decreasing subsequence* (DS) and the *longest decreasing subsequence* (LDS) are defined similarly.

A folklore algorithm [24, 31] can be used to solve the LIS problem with $O(n \log n)$ time. An example of the folklore algorithm is illustrated in Table 2.

The folklore algorithm maintains an array $S[\cdot]$, where $S[x]$ denotes the best (smallest) ending element of the current IS with length x. In Table 2, each row represents an iteration when it deals with an input element from A, in which the underlined elements indicate the updates. Whenever it linearly scans an element a_i from A, it updates $S[\cdot]$ by either replacement or extension with the binary search scheme. a_i replaces the smallest element that is greater than a_i in $S[\cdot]$ (i.e. the successor of a_i); otherwise, a_i is appended to $S[\cdot]$ as the last element and the LIS length is increased. The content of the LIS in this example is $\langle 1, 3, 5, 8\rangle$, which can be easily obtained by the backtracking technique.

2.2. *The Longest Wave Subsequence*

In this subsection, we define the *longest wave subsequence* (LWS) problem.

Definition 1. (trend of a numeric sequence) *Given a numeric sequence of distinct values* $W = \langle w_1, w_2, \ldots, w_n \rangle$*, the* trend sequence $T = \langle t_1, t_2, \ldots, t_n \rangle$ *of* W *is defined as*

$$t_i = \begin{cases} 0, & \text{if } w_{i-1} > w_i \text{ and } 2 \leq i \leq n; \\ 1, & \text{if } w_{i-1} < w_i \text{ and } 2 \leq i \leq n; \\ \text{complement of } t_2, & \text{if } i = 1. \end{cases} \tag{1}$$

For example, suppose that $W = \langle 2, 6, 9, 8, 5 \rangle$. Then, its trend sequence is $T = \langle 0, 1, 1, 0, 0 \rangle$. Note that t_1 is set as the complement of t_2 and t_1 is used only for initialization. Here, the trend of $W_{1..3}$ is increasing and $W_{3..5}$ is decreasing.

Definition 2. (increasing segment, decreasing segment) *Given a numeric sequence* $W = \langle w_1, w_2, \ldots, w_n \rangle$ *of distinct values with trend sequence* $T = \langle t_1, t_2, \ldots, t_n \rangle$*, an* increasing segment $W^+_{i..j}$ *is a substring of* W *such that* $t_{i+1} = t_{i+2} = \ldots = t_j = 1$ *and* $t_i = t_{j+1} = 0$*, for* $1 \leq i < j \leq n$*. Similarly, a* decreasing segment $W^-_{i..j}$ *is a substring of* W *such that* $t_{i+1} = t_{i+2} = \ldots = t_j = 0$ *and* $t_i = t_{j+1} = 1$*, for* $1 \leq i < j \leq n$*. Here,* t_{n+1}*, used for boundary conditions, is set as the complement of* t_n*.*

Definition 2 specifies a segment as the consecutive elements of the same trend with the maximal length in a sequence. This implies that a sequence can be decomposed into alternately increasing and decreasing segments. For example, consider a sequence $W = \langle 1, 2, 4, 9, 7, 3, 5, 6, 8 \rangle$, whose trend sequence $T = \langle 0, 1, 1, 1, 0, 0, 1, 1, 1 \rangle$. W can be decomposed into $W^+_{1..4}$, $W^-_{4..6}$ and $W^+_{6..9}$. Position 4, where $w_4 = 9$, is an overlapping element of $W^+_{1..4}$ and $W^-_{4..6}$. We refer to position 4 as a *turning point*, serving as the ending element of the former segment and the starting element of the latter segment. Similarly, position 6 is the turning point of $W^-_{4..6}$ and $W^+_{6..9}$.

Definition 3. (turning point) *Given a sequence* W *with trend sequence* T*,* p *is a* turning point *if* $t_p \neq t_{p+1}$*,* $1 \leq p \leq |W| - 1$*. In other words, at the turning point* p*,* w_p *is the overlapping element of two neighboring segments in* W*.*

Note that we set $p = 1$ as the dummy turning point of the first segment. Thus, there are exactly r turning points if there are r segments in W. In the following, we will discuss two versions of the LWS problem, the LWSt and the LWSr problems, each addressing different aspects of LWS.

2.2.1. *The Longest Wave Subsequence Problem with Trend*

Suppose we have two numeric strings (or substrings) $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_n \rangle$. Let $Rank(A, i)$ denote the number of elements in A

that are less than or equal to a_i. Similarly, $Rank(B, i)$ represents the rank of b_i in B. A and B are said to be *order-preserving* [5, 6, 14, 16, 18, 21] if $Rank(A, i) = Rank(B, i)$, for all $1 \leq i \leq n$. For example, suppose that $A = \langle 24,\ 31,\ 42,\ 40,\ 26\rangle$ and $B = \langle 10,\ 18,\ 25,\ 21,\ 16\rangle$. Then, we have $Rank(A, i)_{i=1,2,\ldots,n} = \langle 1, 3, 5, 4, 2\rangle$ and $Rank(B, i)_{i=1,2,\ldots,n} = \langle 1, 3, 5, 4, 2\rangle$. Thus, A and B are order-preserving.

The following definition describes the *trend-preserving* relationship between two sequences.

Definition 4. (trend-preserving) *Given two numeric sequences A and B with the same length, where the values in each sequence are distinct, A and B are said to be* trend-preserving *if A and B have the same trend.*

For example, $A = \langle 28,\ 31,\ 42,\ 40,\ 26\rangle$ and $B = \langle 10,\ 18,\ 25,\ 21,\ 16\rangle$ have the same trend $T = \langle 0,\ 1,\ 1,\ 0,\ 0\rangle$, so they are trend-preserving. However, A and B are not order-preserving since the elements' ranks are not all identical in corresponding positions in A and B. It is clear that any two order-preserving sequences are always trend-preserving. On the contrary, two trend-preserving sequences may or may not be order-preserving.

The first variant of the LWS problem, the *longest wave subsequence problem with trend*, denoted as LWSt, is defined as follows.

Definition 5. (LWSt problem) *Given a numeric sequence $A = \langle a_1,\ a_2,\ \ldots,\ a_n\rangle$ of distinct values and a predefined target trend sequence $T = \langle t_1,\ t_2,\ \ldots,\ t_n\rangle$, with the same length n, where $t_i \in \{0, 1\}$ for $1 \leq i \leq n$, and $t_1 \neq t_2$, the* longest wave subsequence with trend (LWSt) *is a longest subsequence $W = \langle w_1,\ w_2,\ \ldots,\ w_{n'}\rangle$ obtained from A such that $w_{i-1} < w_i$ if $t_i = 1$ and $w_{i-1} > w_i$ if $t_i = 0$, for $2 \leq i \leq n' \leq n$.*

Note that we stipulate $t_1 \neq t_2$, as T is a predefined trend sequence and not calculated from A. As another view of Definition 5, the LWSt problem is to find W of maximum length such that its trend aligns with the prefix of T. For example, suppose that $A = \langle 4,\ 6,\ 1,\ 7,\ 3,\ 5,\ 8,\ 2\rangle$, and $T = \langle 0,\ 1,\ 1,\ 0,\ 1,\ 1,\ 1,\ 1\rangle$. The LWSt answer is $\langle 4,\ 6,\ 7,\ 3,\ 5,\ 8\rangle$ with length 6, whose trend $\langle 0,\ 1,\ 1,\ 0,\ 1,\ 1\rangle$ perfectly matches a certain prefix of T. In other words, W and the prefix of T are trend-preserving. In addition, when there is only one turning point in the trend sequence, such as $\langle 0,\ 1,\ 1,\ 1,\ \ldots\rangle$, the LWSt problem simplifies to the traditional LIS problem.

2.2.2. *The Longest Wave Subsequence Problem within r Segments*

The longest increasing (or decreasing) subsequence problem considers a subsequence of only a fixed trend. In some applications, it may allow several interleaving up and down trends, or alternating peaks and valleys, in the subsequence, creating a wave-like pattern. The second variant of the LWS problem is defined as follows.

Definition 6. (LWSr problem) *Given a numeric sequence $A = \langle a_1,\ a_2,\ \ldots,\ a_n\rangle$ of distinct values and the constraint number r of segments, the problem of the* longest

Table 3. The LWSr answers of $A = \langle 4, 6, 1, 7, 3, 5, 8, 2\rangle$ for $r = 1$, 2, or 3.

r	LWSr answers	length
1	$\langle 4, 6, 7, 8\rangle$, $\langle 1, 3, 5, 8\rangle$	4
2	$\langle 4, 6, 7, 3, 2\rangle$, $\langle 4, 6, 7, 5, 2\rangle$, $\langle 4, 6, 7, 8, 2\rangle$, $\langle 1, 3, 5, 8, 2\rangle$	5
3	$\langle 4, 6, 1, 3, 5, 8\rangle$, $\langle 4, 6, 7, 3, 5, 8\rangle$	6

wave subsequence within r segments (LWSr) *is to find a longest subsequence* $W = \langle w_1, w_2, \ldots, w_{n'}\rangle$ *obtained from* A *such that the number of turning points in* W *is at most* r.

In Definition 6, i is a turning point, $i \geq 2$, if and only if ($w_{i-1} > w_i$ and $w_i < w_{i+1}$) or ($w_{i-1} < w_i$ and $w_i > w_{i+1}$). Note that $i = 1$ is always considered as the first turning point.

Without loss of generality, it is assumed that the first segment is increasing, that is $w_1 < w_2$. Table 3 shows the LWSr examples with $A = \langle 4, 6, 1, 7, 3, 5, 8, 2\rangle$. When $r = 1$, it is identical to the traditional LIS problem, and the LWSr is $\langle 4, 6, 7, 8\rangle$ or $\langle 1, 3, 5, 8\rangle$ with length 4. When $r = 2$, it allows increasing first and then decreasing. When $r = 3$, it allows increasing first, then decreasing, and again increasing.

As an important note, for a given sequence A, suppose that the LWSr answers W and W' are required to contain exactly r and r' segments, respectively. Then it cannot be guaranteed that $|W| \leq |W'|$ if $r < r'$. This situation is exemplified in $A = \langle 3, 4, 1, 6, 7\rangle$. When the LWSr is required to have exactly $r = 1$ segment, then the LWSr answer is $\langle 3, 4, 6, 7\rangle$ with length 4. When the LWSr is required to have exactly $r = 2$ segments, then the LWSr answer is $\langle 3, 4, 1\rangle$ with length 3.

3. The Algorithm for the Longest Wave Subsequence Problem with Trend

To solve the LWSt problem, we need some notations as follows.

Definition 7. (best ending element) [24, 31] *Given a sequence* A*, among all increasing* (*decreasing*) *subsequences of* A *with a certain length, the* best ending element *is the last element which is the minimum* (*maximum*).

Definition 8. (solution list) *Given a sequence* A*, for solving the LWSt problem, each element* $S[x]$ *in a* solution list $S[\cdot]$ *stores the best element at length* x.

Definition 9. (successor) *Given an element* a_i*, the* successor *in an increasing sequence is the smallest element that is greater than* a_i*, and the* successor *in a decreasing sequence is the largest element that is less than* a_i.

We first demonstrate our idea for solving the LWSt problem with the example in Table 4. According to the turning point list, the first range for updating $S[\cdot]$

Table 4. An example for the LWSt algorithm with an input sequence $A = \langle 4, 6, 1, 7, 3, 5, 8, 2\rangle$ and a trend sequence $T = \langle 0, 1, 1, 0, 1, 1, 1, 1\rangle$, where the turning point list is $P = \langle 1, 3, 4\rangle$. The LWSt answer is $\langle 4, 6, 7, 3, 5, 8\rangle$ with length 6.

T	0	1	1	0	1	1	1	1
A \ S(length)	1	2	3	4	5	6	7	8
$a_1 = 4$	**4**							
$a_2 = 6$	4	**6**						
$a_3 = 1$	**1**	6						
$a_4 = 7$	1	6	**7**					
$a_5 = 3$	1	6	7	**3**				
$a_6 = 5$	1	6	7	3	**5**			
$a_7 = 8$	1	6	7	3	5	**8**		
$a_8 = 2$	1	6	7	**2**	5	8		

is $S[1..3]$, which is increasing. When sequentially dealing with a_1, a_2, a_3 and a_4, the action is exactly the same as the traditional LIS algorithm. After obtaining $S[3] = 7 = a_4$, we encounter the second turning point $p_2 = 3$. Hence, the next range for updating $S[\cdot]$ is shifted to $S[3..4]$, which is decreasing. And $S[3] = 7$ is currently the starting element of the new range.

The process of $a_5 = 3$ is the same as the traditional LDS algorithm performed in range $S[3..4]$. We again encounter the next turning point $p_3 = 4$. Thus, the next update range becomes $S[4..8]$, which is increasing, and $S[4] = 3$ is currently the starting element of the new range. Then, for a_6, a_7 and a_8, we perform the the traditional LIS algorithm on the new range. Finally, the LWSt answer is obtained by a simple backtracking technique.

For each new element a_i, we perform the following two possible actions:

(1) Replace its successor to maintain the same length, thereby obtaining a better ending element at that length.

(2) Extend the length if its successor is ∞ for an increasing segment, or $-\infty$ for a decreasing segment.

Thus, the insertion position of a_i in $S[\cdot]$ can be determined by finding the *successor* of a_i in the current range of $S[\cdot]$ with the binary search scheme. It is sufficient to obtain the length x of LWSt by conducting an update within the current range. We need not update a_i into any previous range. This property will be proved in Theorem 1.

Our algorithm for solving the LWSt problem is presented in Algorithm 1. The state flag $c = 1$ ($c = 0$) indicates that the current range is increasing (decreasing). In addition, the turning point list P is used to shift the updating range to the next one.

Theorem 1. *Suppose we are given an input sequence A and a trend sequence T*

Algorithm 1. Computing the LWSt length

Input: A numeric sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$ of distinct values and a trend sequence $T = \langle t_1, t_2, \ldots, t_n \rangle$, where $t_i \in \{0, 1\}$, $1 \leq i \leq n$, and $t_1 \neq t_2$

Output: the LWSt length L

$S[i] \leftarrow \infty$, if $t_i = 1$ and $1 \leq i \leq n$
$S[i] \leftarrow -\infty$, if $t_i = 0$ and $1 \leq i \leq n$
$j = 1$
for $i = 1$ to $n - 1$ **do** ▷ build turning point list $P = \langle p_1, p_2, \ldots, p_r \rangle$
  if $t_i \neq t_{i+1}$ **then** ▷ turning point
    $p_j \leftarrow i$, $j \leftarrow j + 1$
$left \leftarrow p_1$, $right \leftarrow p_2$ ▷ range of current segment, $p_1 = 1$
$c \leftarrow t_2$ ▷ state flag, $c = 1$ for increasing; $c = 0$ for decreasing
$S[1] \leftarrow a_1$
for $i = 2$ to n **do**
  if $c = 1$ **then** ▷ increasing segment
    find minimal x such that $left \leq x \leq right$ and $a_i < S[x]$ ▷ successor
  else ▷ decreasing segment
    find minimal x such that $left \leq x \leq right$ and $a_i > S[x]$ ▷ successor
  $S[x] \leftarrow a_i$ ▷ best ending element with LWSt length x
  if $x = right$ **then** ▷ a turning point
    $c \leftarrow 1 - c$ ▷ state changed between 0 and 1 alternately
    $left \leftarrow right$
    $right \leftarrow$ next turning point in P, or n if no next turning point
$L \leftarrow$ maximal x for $S[x] \neq -\infty$ and $S[x] \neq \infty$ ▷ LWSt length
return L

with r turning points $P = \langle p_1, p_2, \ldots, p_r \rangle$ and $p_{r+1} = \infty$. After processing $A_{1..i-1}$, let the LWSt solution list be $\langle S_1, S_2, \ldots, S_{r'} \rangle$, where $1 \leq r' \leq r$, and each S_i represents the segment $S[p_i..p_{i+1}]$. Then, the update of $S_{r'}$ ($S[p_{r'}..p_{r'+1}]$) when processing a_i is sufficient to obtain the correct LWSt.

Proof. Without loss of generality, we assume that the first segment is increasing. Let ℓ denote the length of the LWSt solution list $\langle a'_1, a'_2, \ldots, a'_\ell \rangle$ after processing $A_{1..k-1}$. Let $L(a_i)$ denote the maximal length of the LWSt answer ending at a_i.

We first prove the case that the process is only in the first segment, that is $\ell < p_2$. When $k = 2$, it is trivially correct since there is only one element $\langle a_1 \rangle$ in the LWSt solution list after $A_{1..1} = a_1$ is processed.

For the hypothesis, assume that a_{k-1}, $k \geq 2$ has been processed and the LWSt solution list $\langle a'_1, a'_2, \ldots, a'_\ell \rangle$ is correctly obtained, where $\ell < p_2$. Now a_k is to be processed. If $a_k > a'_\ell$, then the LWSt length can be increased by appending a_k to the answer, and the solution list becomes $\langle a'_1, a'_2, \ldots, a'_\ell, a'_{\ell+1} = a_k \rangle$. Otherwise, a_k

will replace a'_j that a'_j is the smallest for $a'_j > a_k$, $1 \le j \le \ell$. Thus, it is correctly processed in the first segment, even when $\ell + 1 = p_2$ (turning point).

Next consider that two or more segments have been produced. For the hypothesis, assume that a_{k-1}, $k \ge 2$, has been processed and the LWSt solution list $\langle a'_1, a'_2, \ldots, a'_\ell \rangle$ is correctly obtained, where $\ell \ge p_2$. Without loss of generality, it is assumed that the last segment is decreasing. Let j be the last turning point before ℓ. That is, $p_{r'} = j$ and a'_j is the starting element in the last segment $S_{r'}$ and $a'_j > a'_{j+1} > \cdots > a'_\ell$. When a_k is processed, there are two possible actions: (1) a_k is added only in the last segment; (2) a_k is added in each segment. We want to prove that after a_{k+1} is processed, the value of $L(a_{k+1})$ for action 1 of a_k is not less than that (denoted as $L'(a_{k+1})$) for action 2. That is, $L(a_{k+1}) \ge L'(a_{k+1})$.

Three cases are considered as follows.

Case 1: $a_k > a'_j$.

For action 1, a_k will substitute a'_j in segment $S_{r'}$ and a_k becomes the new a'_j. That is $a'_j = a_k$ and $L(a_k) = j$. For action 2, a_k will still substitute a'_j, which is the last of the increasing segment $S_{r'-1}$ and the first of the decreasing segment $S_{r'}$. That is $L'(a_k) = j$. Therefore, $L(a_k) = L'(a_k) = j$. Then, the process results of a_{k+1} for both actions are the same. That is, $L(a_{k+1}) = L'(a_{k+1})$.

Case 2: $a'_j > a_k > a'_{j-1}$.

For action 1, we will find a proper position μ for a_k (substitution or appendant) in the last segment $S_{r'}$, so $L(a_k) = \mu > j$. For action 2, a_k will substitute a'_j in $S_{r'-1}$. However, a'_j is also the first of $S_{r'}$, then the substitution will reduce the extendability of $S_{r'}$. Thus, we should not substitute a'_j by a_k. Therefore, the process result of a_{k+1} for action 1 is not worse than action 2. That is, $L(a_{k+1}) \ge L'(a_{k+1})$.

Case 3: $a_k < a'_j$ and $a_k < a'_{j-1}$.

In this case, a'_j will not be replaced by a_k in both $S_{r'-1}$ and $S_{r'}$. Thus, the update of $S_{r'-1}$ is independent to the update of $S_{r'}$. In other words, we can find a proper position for a_k in the decreasing segment $S_{r'-1}$ and another proper position in the increasing segment $S_{r'}$. Then, the process results of a_{k+1} for both actions are same. That is, $L(a_{k+1}) = L'(a_{k+1})$.

The situations for updating a_k into S_1, S_2, ..., $S_{r'-2}$ have the same results. Thus, the theorem holds. □

Theorem 2. *Algorithm 1 solves the LWSt problem in $O(n \log n)$ time and $O(n)$ space.*

Proof. Algorithm 1 only maintains the solution list $S[\cdot]$ with length n to record the best ending element for each LWSt length. While processing each element a_i of A, it updates $S[\cdot]$ by finding the successor of a_i in the current range of $S[\cdot]$. This can be accomplished by the binary search scheme, which takes $\mathrm{O}(\log n)$ time. Therefore, the overall time complexity is $\mathrm{O}(n \log n)$, and the space complexity is $\mathrm{O}(n)$. □

When the length of each segment is a constant, such as $T = \langle 0, 1, 1, 0, 0, 1, 1,$

$\ldots\rangle$, the successor of a_i in the current range can be found in O(1) time. Therefore, for solving the LWSt problem with constant-length segments, our algorithm takes only $O(n)$ time.

4. The Algorithm for the Longest Wave Subsequence Problem within r Segments

Without loss of generality, it is assumed that the first segment in the LWSr problem is increasing. The LWSr problem obviously degenerates into the traditional LIS problem when $r = 1$. When $r = 2$, the answer is divided into two parts by the turning point. The problem can be solved by applying the traditional LIS algorithm forward and backward, respectively, to get two parts of the answer. For the forward part, we scan A from a_1 to a_n and obtain a list $F = \langle f_1, f_2, \cdots, f_n\rangle$ of the LIS lengths, where f_i, $1 \le i \le n$, denotes the LIS length of $A_{1..i}$. For the backward part, we scan A from a_n to a_1 and obtain a list $B = \langle b_1, b_2, \cdots, b_n\rangle$, where b_i, $1 \le i \le n$, denotes the LIS length of the reverse of $A_{i..n}$ (the LDS length of $A_{i..n}$). Then, we consider the turning point at each possible position i, and find the maximal value of $f_i + b_{i+1}$, which represents the sum of the LIS length of $A_{1..i}$ and the LDS length of $A_{i+1..n}$.

In the above algorithm, we apply the LIS algorithm forward and backward, resulting in a time complexity of $O(n \log n)$ for solving the LWSr problem within 2 segments. When $r \ge 3$, the decision of the turning points becomes more complex, making it impractical to generalize the above algorithm for $r \ge 3$.

In our LWSr algorithm, we denote a 2-tuple (e, l) as the solution status, where e denotes the best ending element and l represents its LWSr length. For example, (7, 4) represents an LWSr of length 4 which ends at element 7.

For solving the LWSr problem, we use r priority queues for storing these 2-tuple elements. Each queue Q_j corresponds to segment j, where $1 \le j \le r$. For an increasing segment j (j is odd), Q_j is a min-priority queue. For a decreasing segment j (j is even), Q_j is a max-priority queue. The 2-tuple elements are stored in the priority queue based on their e values. In a min-priority (max-priority) queue, the minimum (maximum) element is stored as the first one.

In a priority queue, there are two common operations, *Predecessor* and *Successor*. $Predecessor(Q, x)$ returns the previous element of x in Q. On the other hand, $Successor(Q, x)$ returns the next element of x in Q.

Definition 10. *For any 2-tuples* (e_1, l_1), $(e_2, l_2) \in Q_j$, $(e_1, l_1) \ne (e_2, l_2)$, *we say that* (e_1, l_1) dominates (e_2, l_2) *in a min-priority* (*max-priority*) *queue* Q_j, *if* $e_1 \le e_2$ *and* $l_1 \ge l_2$ ($e_1 \ge e_2$ *and* $l_1 \ge l_2$). *Any pair of 2-tuples in* Q_j *do not dominate each other.*

Table 5 illustrates an example of our concept for solving the LWSr problem. In the main procedure of our algorithm, when a new element a_i is processed, the following works are performed in each Q_j for $j = 1, 2, \ldots, r$.

Table 5. An example of the LWSr algorithm within $r = 3$ segments for $A = \langle 4, 6, 1, 7, 3, 5, 8, 2\rangle$. Here, each red bold element indicates an update, and each crossed-out element means that it is dominated and then deleted. The LWSr answer is $\langle 4, 6, 7, 3, 5, 8\rangle$ with length 6.

length		1	2	3	4	5	6
$a_1 = 4$	Q_1	**(4, 1)**					
	Q_2	**(4, 1)**					
	Q_3	**(4, 1)**					
$a_2 = 6$	Q_1	(4, 1)	**(6, 2)**				
	Q_2	~~(4, 1)~~	**(6, 2)**				
	Q_3	(4, 1)	**(6, 2)**				
$a_3 = 1$	Q_1	~~(4, 1)~~ **(1, 1)**	(6, 2)				
	Q_2		(6, 2)	**(1, 3)**			
	Q_3	~~(4, 1)~~	~~(6, 2)~~	**(1, 3)**			
$a_4 = 7$	Q_1	(1, 1)	(6, 2)	**(7, 3)**			
	Q_2		~~(6, 2)~~	~~(1, 3)~~ **(7, 3)**			
	Q_3			(1, 3)	**(7, 4)**		
$a_5 = 3$	Q_1	(1, 1)	~~(6, 2)~~ **(3, 2)**	(7, 3)			
	Q_2			(7, 3)	**(3, 4)**		
	Q_3			(1, 3)	~~(7, 4)~~ **(3, 4)**		
$a_6 = 5$	Q_1	(1, 1)	(3, 2)	~~(7, 3)~~ **(5, 3)**			
	Q_2			(7, 3)	~~(3, 4)~~ **(5, 4)**		
	Q_3			(1, 3)	(3, 4)	**(5, 5)**	
$a_7 = 8$	Q_1	(1, 1)	(3, 2)	(5, 3)	**(8, 4)**		
	Q_2			~~(7, 3)~~	~~(5, 4)~~ **(8, 4)**		
	Q_3			(1, 3)	(3, 4)	(5, 5)	**(8, 6)**
$a_8 = 2$	Q_1	(1, 1)	~~(3, 2)~~ **(2, 2)**	(5, 3)	(8, 4)		
	Q_2				(8, 4)	**(2, 5)**	
	Q_3			(1, 3)	~~(3, 4)~~	~~(5, 5)~~ **(2, 5)**	(8, 6)

Step 1: $(e_j, l_j) \leftarrow Successor(Q_j, a_i)$. Note that Q_j may be either a min-priority or max-priority queue.

Step 2.1: If there does not exist (e_j, l_j), it indicates that we can append a_i in segment j to increase its length. We set the length l'_j for a_i to be $L_j + 1$, where L_j denotes the maximum length in Q_j, and L_j is set to 0 for initialization. And then set $L_j = l'_j$.

Step 2.2: If there exists (e_j, l_j), e_j is replaced by a_i with the same length l_j. In addition, we have to consider the length for a_i obtained in Q_{j-1}. Thus, $l'_j \leftarrow \max\{l_j, l'_{j-1}\}$.

Step 3: Insert (a_i, l'_j) into Q_j.

Step 4: Remove all dominated 2-tuples in Q_j by iteratively applying

$Successor(Q_j, a_i)$.

For example, in Table 5, when $a_8 = 2$ is processed, (2, 2) replaces (3, 2) in Q_1; (2, 5) is appended to Q_2; (2, 5) is inserted into Q_3 whose l' is obtained from Q_2. Additionally, (3, 4) and (5, 5) in Q_3 are removed, while (1, 3) is retained as (1, 3) and (2, 5) do not dominate each other. The final answer is the maximal l of the 2-tuple element in Q_3, with length 6.

Our algorithm for solving the LWSr problem is formally presented in Algorithm 2. The initialization is performed in Lines 1 through 5. In Line 4, L_j denotes the LWSr length within j segments, obtained in Q_j. Lines 6 through 17 present the main procedure of our algorithm.

Algorithm 2. Computing the LWSr length

Input: A numeric sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$ of distinct values, and the number r of segments

Output: The LWSr length L_r

1: Build an empty min-priority queue Q_j if $1 \leq j \leq r$ and j is odd.
2: Build an empty max-priority queue Q_j if $1 \leq j \leq r$ and j is even.
3: Insert $(a_1, 1)$ into Q_j, for $1 \leq j \leq r$ ▷ initialization
4: $L_j \leftarrow 1$, for $1 \leq j \leq r$ ▷ LWSr length obtained in Q_j
5: $l'_0 \leftarrow 0$ ▷ boundary condition for latter use
6: **for** $i = 2$ to n **do** ▷ processing for a_i
7: **for** $j = 1$ to r **do**
8: $(e_j, l_j) \leftarrow Successor(Q_j, a_i)$
9: **if** (e_j, l_j) is null **then** ▷ append a_i
10: $l'_j \leftarrow L_j + 1$ ▷ assign length for a_i
11: $L_j \leftarrow l'_j$ ▷ increase length in Q_j
12: **else** ▷ replacement
13: $l'_j \leftarrow \max\{l_j, l'_{j-1}\}$ ▷ max length for a_i from Q_j or Q_{j-1}
14: Insert (a_i, l'_j) into Q_j
15: **while** $(e_j, l_j) \leftarrow Successor(Q_j, a_i) \neq$ null and (a_i, l'_j) dominates (e_j, l_j) **do**
16: Remove (e_j, l_j) from Q_j ▷ remove each dominated 2-tuple
17: **end while**
18: **return** L_r

Theorem 3. *Suppose that we are given an input sequence A and a constant r representing the number of segments. Algorithm 2 gives the LWSr length in the queue Q_r.*

Proof. Algorithm 2 maintains r queues $\langle Q_1, Q_2, \ldots, Q_r \rangle$ and updates each Q_j, for $1 \leq j \leq r$. Without loss of generality, we assume the first segment is increasing and

Q_0 is a virtual queue for boundary use. When $j = 1$, it is similar to the traditional LIS problem.

When $j = 2$, we recognize the continuity of segments. In this case, the second segment must be extended from the first segment. If the currently considered element is smaller than the last element in Q_1, it turns the wave to obtain a longer LWSr with 2 segments ($|Q_1| < |Q_2|$). Otherwise, it extends the first segment and Q_2 keeps the same length as segment 1 ($|Q_1| = |Q_2|$).

When $j = 3$, if $|Q_1| < |Q_2|$ and the currently considered element is larger than the last element in Q_2, it once again alters the wave to achieve a longer LWSr with 3 segments ($|Q_2| < |Q_3|$). Otherwise, it extends the second segment and Q_3 maintains the same length as the first 2 segments ($|Q_2| = |Q_3|$). If $|Q_1| = |Q_2|$, indicating only one increasing segment, then $|Q_1| = |Q_3|$, where Q_3 is also increasing.

When $j = r$, by recurrence, $\langle Q_1, Q_2, \ldots, Q_r\rangle$ is either extended or inherited from $\langle Q_0, Q_1, \ldots, Q_{r-1}\rangle$. Thus, the longest length within j segments is stored in Q_j. And the theorem holds. □

Theorem 4. *Algorithm 2 solves the LWSr problem with $O(rn \log n)$ time and $O(rn)$ space.*

Proof. Algorithm 2 maintains r queues $\langle Q_1, Q_2, \ldots, Q_r\rangle$ and the size of each queue is at most n. While processing each element of the input sequence A, it takes $\mathrm{O}(\log n)$ time to update each Q_j, for $1 \le j \le r$, by finding the successor and performing the insertion. The removal of the dominated element in each Q_j is executed at most n times, and each domination removal requires $\mathrm{O}(\log n)$ time. Therefore, the overall time complexity of the algorithm is $\mathrm{O}(rn \log n)$, and the space complexity is $\mathrm{O}(rn)$. □

5. Conclusion and Discussion

When we present our LWS algorithms, it is assumed that all elements of the input sequence A are distinct. If there exist duplicate elements in A, two situations have to be discussed. Firstly, if two identical elements are not allowed in the LWS answer, the algorithm can still work correctly. Secondly, if two identical elements are permitted in the LWSt answer, they may appear in either an increasing segment or a decreasing segment. In this case, an increasing (decreasing) segment effectively becomes a non-decreasing (non-increasing) segment to accommodate the duplicate elements. The situation can be addressed by considering an element as the successor of another identical element in a non-decreasing segment, as well as the successor in a non-increasing segment.

The time complexity of our LWSr algorithm can be further improved by using the *van Emde Boas* (vEB) tree [4], a highly efficient data structure for implementing the

priority queue. In a vEB tree, the operation of each insertion, deletion and successor can be performed in $O(n \log \log n)$ time when the input sequence is a permutation of $\{1, 2, \ldots, n\}$. To facilitate this, we can first transform the input into a permutation of $\{1, 2, \ldots, n\}$ using a sorting scheme. Subsequently, with the integration of the vEB tree, the time complexity of the proposed LWSr algorithm can be reduced to $O(n \log n + rn \log \log n)$.

The LWSr problem offers valuable insights into price movements in the stock market. Sometimes we may consider the price trend relationship between two or more stocks. Thus, it is worthy to study the problem of the *longest common wave subsequence within r segments* in the future. As another possible research direction, in the LWSr problem, the length of each segment in the answer may be bounded by an upper bound or a lower bound, or both bounds.

Acknowledgments

This research work was partially supported by the National Science and Technology Council of Taiwan under contract MOST 108-2221-E-110-031.

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* **113**(13) (2013) 470–476.

[2] 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* **321**(2-3) (2004) 405–414.

[3] S. Bespamyatnikh and M. Segal, Enumerating longest increasing subsequences and patience sorting, *Information Processing Letters* **76**(1-2) (2000) 7–11.

[4] P. v. E. Boas, R. Kaas and E. Zijlstra, Design and implementation of an efficient priority queue, *Mathematical Systems Theory* **10**(1) (1976) 99–127.

[5] T. Chhabra and J. Tarhio, A filtration method for order-preserving matching, *Information Processing Letters* **116**(2) (2016) 71–74.

[6] S. Cho, J. C. Na, K. Park and J. S. Sim, A fast algorithm for order-preserving pattern matching, *Information Processing Letters* **115**(2) (2015) 397–402.

[7] A. F. W. Coulson, J. F. Collins and A. Lyall, Protein and nucleic acid sequence database searching: A suitable case for parallel processing, *The Computer Journal* **30**(5) (1987) 420–424.

[8] M. Crochemore and E. Porat, Fast computation of a longest increasing subsequence and application, *Information and Computation* **208**(9) (2010) 1054–1059.

[9] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White and S. L. Salzberg, Alignment of whole genomes, *Nucleic Acids Research* **27**(11) (1999) 2369–2376.

[10] A. Elmasry, The longest almost-increasing subsequence, *Information Processing Letters* **110**(16) (2010) 655–658.

[11] M. L. Fredman, On computing the length of longest increasing subsequences, *Discrete Mathematics* **11**(1) (1975) 29–35.

[12] Z. Galil, On improving the worst case running time of the Boyer-Moore string matching algorithm, *Communications of the ACM* **22**(9) (1979) 505–508.

[13] G. Garai and B. B. Chaudhuri, A distributed hierarchical genetic algorithm for efficient optimization and pattern matching, *Pattern Recognition* **40**(1) (2007) 212–228.
[14] P. Gawrychowski and P. Uznański, Order-preserving pattern matching with k mismatches, *Theoretical Computer Science* **638** (2016) 136–144.
[15] G. Grillo, F. Licciulli, S. Liuni, E. Sbisá and G. Pesole, Patsearch: a program for the detection of patterns and structural motifs in nucleotide sequences, *Nucleic Acids Research* **31**(13) (2003) 3608–3612.
[16] M. M. Hasana, A. S. M. S. Islama, M. S. Rahmana and M. S. Rahman, Order preserving pattern matching revisited, *Pattern Recognition Letters* **55** (2015) 15–21.
[17] J. W. Hunt and T. G. Szymanski, A fast algorithm for computing longest common subsequences, *Communications of the ACM* **20** (1977) 350–353.
[18] J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi and T. Tokuyama, Order-preserving matching, *Theoretical Computer Science* **525** (2014) 68–79.
[19] 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).
[20] D. E. Knuth, J. James H. Morris and V. R. Pratt, Fast pattern matching in strings, *SIAM Journal on Computing* **6**(2) (1977) 323–350.
[21] M. Kubica, T. Kulczyński, J. Radoszewski, W. Rytter and T. Waleń, A linear time algorithm for consecutive permutation pattern matching, *Information Processing Letters* **113**(12) (2013) 430–433.
[22] Y. Li, L. Zou, H. Zhang and D. Zhao, Longest increasing subsequence computation over streaming sequences, **30**(6) (2018) 1036–1049.
[23] S.-F. Lo, K.-T. Tseng, C.-B. Yang and K.-S. Huang, A diagonal-based algorithm for the longest common increasing subsequence problem, *Theoretical Computer Science* **815** (2020) 69–78.
[24] U. Manber, *Introduction to Algorithms: A Creative Approach*, 1st edn. (Addison-Wesley, Boston, USA, 1989).
[25] K. Rohde and P. Bork, A fast, sensitive pattern-matching approach for protein sequences, *Bioinformatics* **9**(2) (1993) 183–189.
[26] C. Schensted, Longest increasing and decreasing subsequences, *Canadian Journal of Mathematics* **13** (1961) 179–191.
[27] S. M. Stephens, J. Y. Chen, M. G. Davidson, S. Thomas and B. M.Trute, Oracle database 10g: a platform for BLAST search and regular expression pattern matching in life sciences, *Nucleic Acids Research* **21**(11) (2005) 2596–2603.
[28] J. Thornton, M. Savvides and B. V. Kumar, A Bayesian approach to deformed pattern matching of iris images, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **29**(4) (2007) 596–606.
[29] A. Trouvé, Diffeomorphisms groups and pattern matching in image analysis, *International Journal of Computer Vision* **28**(3) (1998) 213–221.
[30] C.-T. Tseng, C.-B. Yang and H.-Y. Ann, Minimum height and sequence constrained longest increasing subsequence, *Journal of Internet Technology* **10**(2) (2009) 173–178.
[31] I.-H. Yang, C.-P. Huang and K.-M. Chao, A fast algorithm for computing a longest common increasing subsequence, *Information Processing Letters* **93**(5) (2005) 249–253.