An Optimal Algorithm for Finding the Longest Increasing Subsequence of Every Substring

Chiou-Ting Tseng*, Shyue-Horng Shiau◇ and Chang-Biau Yang*†
*Department of Computer Science and Engineering
National Sun Yat-sen University, Kaohsiung, Taiwan
◇Department of Computer Aided Media Design
Chang Jung Christian University, Tainan, Taiwan
†cbyang@cse.nsysu.edu.tw

Abstract

Given a string $S = \{a_1, a_2, a_3, \cdots, a_n\}$, the longest increasing subsequence (LIS) problem is to find a subsequence of the given string such that the subsequence is increasing and its length is maximal. In a previous result, to find the longest increasing subsequences of each sliding windows with a fixed size w of a given string can be solved in $O(w \log\log n + OUTPUT)$ time, where $O(w \log\log n)$ is taken for preprocessing. In this paper, we solve the problem for finding the longest increasing subsequence of every substring of S. With the straightforward implementation of the previous result, the time required for the preprocessing would be $O(n^2 \log\log n)$. With the modification of the data structure used in the algorithm, our algorithm needs only $O(n^2)$ preprocessing time. Since there are $O(n^2)$ substrings totally in a given string with length n, our algorithm is optimal. The time required for the reporting stage is linear to the size of the output. In other words, our algorithm can find the LIS of every substring in $O(n^2+OUTPUT)$ time, where OUTPUT is the sum of the lengths of the outputs.
Key words: longest increasing subsequence, substring, sliding window, row tower

1 Introduction

Given a string $S = \{a_1, a_2, a_3, \cdots, a_n\}$, the increasing subsequence is a subsequence [13] $IS(S) = \{a_{i_1}, a_{i_2}, a_{i_3}, \cdots, a_{i_k}\}$ that $a_{i_p} < a_{i_q}$ if $i_p < i_q$, for $1 \le p < q \le k$. Three increasing subsequences of S_1=41573 are shown in Table 1.

The *longest increasing subsequence (LIS)* problem is to find the longest among all increasing subsequences. Note that the LIS of a given may not be unique. In Table 1, both 457 and 157 are LIS's of S_1=41573.

Table 1: Three increasing subsequences of S_1 =41573.

$S_1 =$	4	1	5	7	3
$IS(S_1) =$		1			3
$IS(S_1) =$	4		5	7	
$IS(S_1) =$		1	5	7	

The LIS problem has been widely studied in the past decades. There is a straightforward method of finding LIS by finding the longest common subsequence of the input sequence and the sorted input sequence, with time complexity $O(n^2)$. Schensted [11] is the first one that defined the LIS problem and gave an $O(n \log n)$ time algorithm. Hunt [6] improved the algorithm to $O(n \log\log n)$ time. And later, many papers [1, 3–5, 9, 10, 15] studying the LIS problem used the van Emde Boas priority queue [12], which supports insert, delete, find, predecessor, and successor in $O(\log\log|\Sigma|)$ time, where Σ is the alphabet set of the input string. In LIS, the input can be transformed into $|\Sigma| = n$, so it can make the algorithm $O(n \log\log n)$ time. Finding the LIS in streaming data has the limit that the data which has passed can only be retained a limited amount of times, Liben[8] gave an algorithm with $\log(1+\frac{1}{\varepsilon})$ - pass, update time $O(\log l)$ or $O(\log\log\Sigma)$, and space $O(l^{1+\varepsilon}\log\Sigma)$, where l is the length of the longest increasing subsequence. The length distribution of the LIS was analyzed by Aldous and Diaconis [2]. In their result, the average length of the LIS of a sequence with length n is $2\sqrt{n}$.

An extension of the LIS problem is the longest common increasing subsequence (LCIS)

problem, which is given two strings $A = \{a_1, a_2, a_3, \cdots, a_m\}, B = \{b_1, b_2, b_3, \cdots, b_n\}$ and each pair of symbols of A and B are comparable. The common increasing subsequence of A and B is $C = \{c_1, c_2, c_3, \cdots, c_l\}$ where $c_1 = a_{i_1} = b_{j_1}$, $c_2 = a_{i_2} = b_{j_2}$, $\cdots$, $c_l = a_{i_l} = b_{j_l}$ and for all $1 \leq p < q \leq l$, $i_p < i_q$, $j_p < j$, $c_p < c_q$. The LCIS of A and B is the longest among all common increasing subsequences of A and B. Yang, Huang and Chao [14] proposed an algorithm for solving the LCIS problem in $O(n^2)$ time. In 2005, several papers tightened the upper bound, Katriel and Kutz[7] gave an $O(nl \log n + Sort)$ time algorithm, where *Sort* is the time required for sorting sequence B into nondecreasing order. Chan *et al.*[5] gave an $O(\min(r \log l, nl+r) \log \log n + Sort))$ time algorithm, where r is the number of matched pairs between A and B. Brodal et al. [4] gave an $O((m+nl) \log \log |\Sigma| + Sort)$ time algorithm. For small $|\Sigma|$, the algorithm has a tighter bound, $O(m)$ when $|\Sigma| = 2$, $O(m + n \log n)$ when $|\Sigma| = 3$.

The rest of this paper is organized as follows. In Section 2, we shall review the previous result on the LIS in all sliding windows of a fixed size. In Section 3, based on the result of sliding windows, we propose an optimal algorithm for finding the LIS of every substring of a given string. In other words, the window size is not fixed. Our algorithm can find the LIS of every substring in $O(n^2 + OUTPUT)$ time, where OUTPUT is the sum of the lengths of the outputs. Section 4 gives the detailed implementation of our algorithm. Section 5 gives a conclusion and some future works on related topics.

2 A Previous Result on LIS in Sliding Windows

Consider our input string as a deck of cards, a game called patience sorting is as follows. Each time we turn up a card, we put the card into piles on the table obeying that

1. It may be placed on a pile where the new card is smaller than the top card of the pile.
2. The new card is larger than the top cards of all piles, it will be put into a new pile to the right of the existing piles.

The goal of the game is to use as few piles as possible. The greedy strategy to achieve this is to always place the new element on the leftmost possible pile. An example, S_1=41573, is shown in Table 2. For a given string S, we call the top of the multipile as the *representative increasing subsequence* $R(S)$. $R(S)$ can also be viewed as a subset of S that the ith element of $R(S)$ is the smallest ending symbol of increasing subsequences of length i. For S_1, the increasing subsequence of length 1, 2, 3 are $\{4, 1, 5, 7, 3\}$, $\{45, 47, 15, 17, 13, 57\}$ and $\{457, 157\}$ respectively. So $R(S_1)$=137. Note that R may not be a subsequence of S with an example of 137 is not a subsequence of S_1.

Table 3: The row tower of S_2 =4562317829.

			4562317829
S_2=	4562317829	$R(S_2)$ =	126789
$S_2[2,-]$ =	562317829	$R(S_2[2,-])$ =	12789
$S_2[3,-]$ =	62317829	$R(S_2[3,-])$ =	12789
$S_2[4,-]$ =	2317829	$R(S_2[4,-])$ =	12789
$S_2[5,-]$ =	317829	$R(S_2[5,-])$ =	1289
$S_2[6,-]$ =	17829	$R(S_2[6,-])$ =	1289
$S_2[7,-]$ =	7829	$R(S_2[7,-])$ =	289
$S_2[8,-]$ =	829	$R(S_2[8,-])$ =	29
$S_2[9,-]$ =	29	$R(S_2[9,-])$ =	29
$S_2[10,-]$ =	9	$R(S_2[10,-])$ =	9

In the online calculation of LIS, the length of the current LIS is equal to the position of the first element in R which is larger than the new symbol. For example, if we add 6 to the tail of S_1, it becomes S_1'=41573←**6**. In this case, 6 replaces 7 in the 3rd position of $R(S_1)$ and $R(S_1')$ now becomes 136 as shown in the last two column of Table 2. And the LIS's ending at 6 are {136, 156, 456}, all of length 3.

Let $S[i, j]$ denote the subsequence starting from the ith element of $S = \{a_1, a_2, a_3, \cdots, a_n\}$ and ending at the jth element, and let $S[i, -]$ denote $S[i, n]$. A *row tower* T consists of $R(S), R(S[2,-]), R(S[3,-]), \cdots, R(a_n)$, and we call $R(S)$ the *principle row* of T. For example, the row tower of S_2=4562317829 is shown in Table 3. The principle row, $R(S_2)$=126789, is shown at the second row in the table. The row tower of *all prefixes* of S_2 is shown in Table 4. All the 10 prefixes of S_2 is shown at the first row in the table. Each row tower of the 10 prefixes is shown at each colum in the table.

By observation, Albert et al.[1] got that each $R(S[i+1,-])$ is either the same as $R(S[i,-])$ or obtained by removing one element from $R(S[i,-])$, so we can use the principle row and the differences to record the row tower. A data structure is used to represent the row tower, which consists of three

Table 2: Patience Sorting for S_1 =41573.

4	4← **1**	41← **5**	415← **7**	4157← **3**	S_1=41573	41573← **6**	S_1'=415736
	1	**1**	**1**	**1 3**	$R(S_1)$=**137**	**1 3 6**	$R(S_1')$=**136**
4	4	4 **5**	4 **5 7**	4 5 **7**		4 5 7	

Table 4: The row towers of all prefixes of S_2=4562317829.

$prefix(S_2)$=S=	4	45	456	4562	45623	456231	4562317	45623178	456231782	4562317829
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
$R(S)$=	4	45	456	256	236	136	1367	13678	12678	126789
$R(S[2,-])$=		5	56	26	23	13	137	1378	1278	12789
$R(S[3,-])$=			6	2	23	13	137	1378	1278	12789
$R(S[4,-])$=				2	23	13	137	1378	1278	12789
$R(S[5,-])$=					3	1	17	178	128	1289
$R(S[6,-])$=						1	17	178	128	1289
$R(S[7,-])$=							7	78	28	289
$R(S[8,-])$=								8	2	29
$R(S[9,-])$=									2	29
$R(S[10,-])$=										9
pr=	4	45	456	256	236	136	1367	$\underset{\cdot}{1}\underline{3}\underset{\cdot}{6}78$	12678	126789
d=	1	12	123	412	451	641	6417	$\underset{\cdot}{6}\underline{4}\dot{1}78$	69147	691478
σ=	1	12	123	312	231	321	3214	$\hat{3}\bar{2}\dot{1}45$	35124	351246

elements, the principle row pr, the drop out sequence d, of which the ith element d_i is the maximum index of the row that the ith element in the principle row appears, and the drop out permutation σ, which is the order of the elements drop out. For example, consider the eighth row tower in Table 4. The content of the data structure is shown in Table 5. We have pr = 13678, $\underset{\cdot}{1}$ occurs in row 1 through row 6, so d_1 =$\underset{\cdot}{6}$; $\underline{3}$ occurs in row 1 through row 4, so d_2 =$\underline{4}$; $\underset{\cdot}{6}$ occurs in only row 1, so d_3 =$\underset{\cdot}{1}$, and so forth. Finally, we obtain d =$\underset{\cdot}{6}\underline{4}\underset{\cdot}{1}78$. Then the computation of σ is as follows. d_1 =$\underset{\cdot}{6}$ is the 3rd smallest in d, so σ_1 =$\hat{3}$; d_2 =$\underline{4}$ is the 2nd smallest in d, so σ_2 =$\bar{2}$; d_3 =$\underset{\cdot}{1}$ is the smallest in d, so σ_3 =$\dot{1}$, and so forth. Finally, we obtain σ=$\hat{3}\bar{2}\dot{1}45$. The (pr, d, σ) representation of all prefixes of S_2 is given in the lower part of Table 4.

Now, we can handle the problem for finding the LIS of every substring of a fixed window size w. For example, consider S_2 = 4562317829, if w = 8, the LIS's of 45623178, 56231782, 62317829 should be found. Because two neighboring windows have w-1 characters in common, their LIS and R have some duplicates, so we must have a smart way to use the duplicated information. A *window sliding* operation is used to go from one window to the next. The operation consists of two parts: ADD to the right a new symbol and EXPIRE from the left the oldest element. The row towers in the process of window sliding for S_2 with w = 8 is given in Table 6.

Table 5: The (pr,d,σ) representation of the eighth towers of Table 4

	The eighth tower of S	The drop out element $pr = \underset{\cdot}{1}\underline{3}\underset{\cdot}{6}78$	d	σ
row 1	$\underset{\cdot}{1}\underline{3}\underset{\cdot}{6}78$	$\underset{\cdot}{6}$	$\underset{\cdot}{1}$	$\dot{1}$
row 2	$\underset{\cdot}{1}\underline{3}78$			
row 3	$\underset{\cdot}{1}\underline{3}78$			
row 4	$\underset{\cdot}{1}\underline{3}78$	$\underline{3}$	$\underline{4}$	$\bar{2}$
row 5	$\underset{\cdot}{1}78$			
row 6	$\underset{\cdot}{1}78$	$\underset{\cdot}{1}$	$\underset{\cdot}{6}$	$\hat{3}$
row 7	78	$\underline{7}$	$\underset{\cdot}{7}$	4
row 8	8	$\underline{8}$	$\underset{\cdot}{8}$	5

Let us compare the eighth and ninth towers in Table 4. We illustrate them in Table 7. In the first row, $\underline{3}$ is replaced by $\underline{2}$. After $\underline{3}$ disappears, $\underline{2}$ starts to replace $\underline{7}$, so d of $\underline{7}$ is equal to the original d of $\underline{3}$, which is $\underline{4}$. And after $\underline{7}$ disappears, $\underline{2}$ starts to replace $\underline{8}$, so d of $\underline{8}$ is equal to the original d of $\underline{7}$, which is $\underset{\cdot}{7}$. Let us look at $\underline{6}$. Because it disappears earlier than 3, it is not the scapegoat of 3. With the above observation, the replacement is like a chain, the ith element takes the shots for the $(i+1)$th element. So for every element in the

Table 6: The window sliding process for S_2 with window size=8

$S_2 = 4562317829$,window size= 8					
Operation		Add 2	Expire 4	Add 9	Expire 5
S=	45623178	456231782	56231782	562317829	62317829
$R(S)$=	13678	12678	1278	12789	12789
$R(S[2,-])$=	1378	1278	1278	12789	12789
$R(S[3,-])$=	1378	1278	1278	12789	1289
$R(S[4,-])$=	1378	1278	128	1289	1289
$R(S[5,-])$=	178	128	128	1289	289
$R(S[6,-])$=	178	128	28	289	29
$R(S[7,-])$=	78	28	2	29	29
$R(S[8,-])$=	8	2	2	29	9
$R(S[9,-])$=		2		9	
pr=	13678	12678	1278	12789	$1\ddot{2}78\ddot{9}$
d=	64178	69147	5836	58369	47258
σ=	32145	35124	2413	24135	$24\mathring{1}3\dot{5}$

chain, its successor is drop out later than it, and with larger σ value.

The ADD operation consists of two parts: find the place to insert, do the *chain replacement* for d and σ as follows if it is required. For each element a_i in pr, its successor in the chain is the first element right to it in pr with σ larger than it. The successor is the next candidate when a_i is replaced during insertion. For $S_2[1,8]$, $\sigma_1 = 3$, $\sigma_2 = 2 < \sigma_1$, $\sigma_3 = 1 < \sigma_1$, $\sigma_4 = 4 > \sigma_1$, so 7 is the successor of 1. Then we find the successor of 7, $\sigma_5 = 5 > \sigma_4$, so 8 is the successor of 7. Denote the replacement chain p_q as $p_q[1] = q$, $p_q[i+1]$ is the successor of $p_q[i]$ until the end of pr is reached. If the added element replaces position q in pr, the chain replacement is to replace each $d_{p_q[i+1]}$ by $d_{p_q[i]}$. Because the last element in the chain can not replace any other element, its d value disappears. Since σ records the order of the drop out elements in pr, those σ's in front of q with larger σ value have to decrease by one. And since the newly added symbol would find an element to replace in each row in the row tower, so its d is $w+1$ and σ is l. Comparing to the example in the perior paragraph, for $S_2[1,8]$=45623178, $pr = 1\bar{3}678$, $d = 6\mathring{4}1\dot{7}8$, $\sigma = 3\hat{2}1\tilde{4}\acute{5}$, we can get the chain starting from position $\underset{\cdot}{2}$, $p_{\underset{\cdot}{2}} = (\underset{\cdot}{2},\underset{\cdot}{4},\underline{5})$. Let (pr, d, σ) denote the data before the ADD operation, (pr', d', σ') denote the data after ADD. After we add $\bar{2}$ to $S_2[1,8]$, $\bar{2}$ replaces the $\underset{\cdot}{2}$th position of pr, $\bar{3}$, so pr' becomes $1\bar{2}678$, $q = \underset{\cdot}{2}$. The second element in $\mathrm{p}_{\underset{\cdot}{2}}$ is $\underset{\cdot}{4}$ so $d'_{\underset{\cdot}{4}} = d_{\underset{\cdot}{2}} = \mathring{4}$, $\sigma'_{\underset{\cdot}{4}} = \sigma_{\underset{\cdot}{2}} = \hat{2}$, and then $d'_{\underline{5}} = d_{\underset{\cdot}{4}} = \dot{7}$, $\sigma'_{\underline{5}} = \sigma_{\underset{\cdot}{4}} = \tilde{4}$, and then the end of the list is reached. After that, $\sigma_{\underline{5}} = \acute{5}$ disappears, we start from the beginning of the σ and there is no element with larger σ value larger than $\acute{5}$ before the $\underset{\cdot}{2}$th position, and then we set $d'_{\underset{\cdot}{2}} = 9$, $\sigma'_{\underset{\cdot}{2}} = \acute{\bar{5}}$. So $d' = 691\mathring{4}\dot{7}$, $\sigma' = 3\acute{\bar{5}}1\hat{2}\tilde{4}$.

The EXPIRE operation is simply decreases every element of d by one, and if any one element of d becomes zero, delete it and decrease the elements in the σ sequence with larger value than the deleted element by one.

To get the LIS by tracing back, we have to know the predecessor of each element in LIS. In the row tower model, when we slide the window to the next(right) position, the predecessor can be easily obtained. But in (pr, d, σ) representation, the predecessor can not be seen explicitly. Since the predecessor would change when the original predecessor expires, the row range of the new predecessor can be calculated by the σ sequence. Starting from the inserted position to the first element in pr, the candidate predecessor is the one expire later than the current predecessor, in other words, with larger σ value. Let α be the newly added element and $L_\alpha(S)$ denote the LIS ending at α in S. An example of $L_9(S)$ for the last window of S_2 is given in Table 8. The predecessor of 9 in rows 1 through 5 is $\ddot{8}$, rows 6 through 7 is $\acute{\bar{2}}$. Refer to the (pr, d, σ) representation in Table 6. 9 is in the $\underline{5}$th position in pr so its first predecessor is 8 in the $\underline{4}$th position, $\sigma[\underline{4}]$=$\dot{3}$. The first element we find in the left of $\dot{3}$ and larger than $\dot{3}$ is $\mathring{4}$ in the $\underset{\cdot}{2}$nd position. So the predecessor of 9 changes to the $\underset{\cdot}{2}$nd element of pr, $\acute{2}$, when 9 is in the $\tilde{3}$rd position. There is no larger element in the left of $\mathring{4}$ and larger than $\mathring{4}$. So predecessor of 9 is $\ddot{8}$ when 9 is in position $\underline{5}$ down to $\tilde{3}$, and $\acute{\bar{2}}$ when 9 is in position $\underset{\cdot}{2}$.

Table 7: The eighth and ninth towers of Table 3 for $S_2 = 4562317829$. In the drop out sequence, $\dot{7}$ takes the position of $\dot{3}$ and $\dot{8}$ takes the position of $\dot{7}$.

	The eighth tower of S_2	The drop out element	d	σ	The ninth tower of S_2	The drop out element	d	σ
1	1$\underline{3}$$\underline{6}$78	6	1	1	1$\underline{2}$678	6	1	1
2	1$\underline{3}$78				1$\underline{2}$78			
3	1$\underline{3}$78				1$\underline{2}$78			
4	1$\underline{3}$78	$\dot{3}$	4	2	1$\underline{2}$78	$\dot{7}$	4	2
5	1$\underline{7}$8				1$\underline{2}$8			
6	1$\underline{7}$8	1	6	3	1$\underline{2}$8	1	6	3
7	$\underline{7}$8	$\dot{7}$	7	4	$\underline{2}$8	$\dot{8}$	7	4
8	$\underline{8}$	$\dot{8}$	8	5	$\underline{2}$			
9					$\underline{2}$	2	9	5

Table 8: LIS of all suffixes of the last sliding window of S_2

S	$L_9(S)$
62317829	237$\ddot{8}$9
2317829	237$\ddot{8}$9
317829	17$\ddot{8}$9
17829	17$\ddot{8}$9
7829	7$\ddot{8}$9
829	$\ddot{2}$9
29	$\ddot{2}$9
9	9

ADD and EXPIRE in each window requires $O(l)$ time, where l is the LIS length in the window. The first window can be constructed by using the van Emde Boas priority queue with each insertion requiring $O(\log\log|\Sigma|)$ time. So it finds the LIS length or sequence of every sliding window in $O(w\log\log n + \sum_{i=0}^{n-w} l_i)$ time where l_i is the LIS length of window i and $|\Sigma| = O(n)$.

3 Our Algorithm for Finding LIS of Each Substring

Our algorithm is based mainly on the algorithm proposed by Albert *et al.* [1]. When we want to find the LIS of every substring, the straightforward implementation is to apply Albert's algorithm repeatedly with window size varying from 1 to n. It is clear that l_i=$O(w)$. The total time complexity is $\sum_{w=1}^{n}(w\log\log|\Sigma| + nw) = O(n^2\log\log n + n^3)$=$O(n^3)$. In this section, we shall present an algorithm for solving this problem. The time required for preprocessing is $O(n^2)$. Meanwhile, the LIS length of every substring is also obtained. Then, the content of each LIS can be reported in $O(n^3)$ time.

Table 9 show the structure of the row towers. We have the following observations, which can be used to reduce the preprocessing time.

Observation 1. *$R(S[j,k])$ can be calculated by expiring from $R(S[1,k])$ one by one.*

Observation 2. *The R's in the first row can be calculated by inserting the input symbols one by one.*

Proposition 1. *The rank of each d_i's remains unchanged after one EXPIRE operation.*

Proof Because we simply decrease each d_i by one, if $d_{i,a} > d_{i,b}$ then $d'_{i,a} = d_{i,a} - 1 > d_{i,b} - 1 = d'_{i,b}$. Thus the rank (order) remains the same. □

By Observation 2 and Observation 1, there would be n insertions and $\frac{(n-1)^2}{2}$ expirations if we want to find out the LIS's of every substring of S.

Observation 3. *The EXPIRE operation decreases every element of d by one, and if any one element of d becomes zero, it is deleted.*

Suppose the elements in d are ordered as $\{d_{i_1} < d_{i_2} < d_{i_3} < \cdots < d_{i_l}\}$. Our main data structure is based on Observation 3. Let $\delta_{i,1}$=$d_{i,1}$ and δ_{i_p}=$d_{i_p} - d_{i_{p-1}}$. The transformation between d and δ of the last two row tower of Table 6 is given in Table 10. What one EXPIRE operation should do is only to decrease δ_{i_1} by one, and all other δ_{i_p}'s, $2 \le p \le l$, remain unchanged. If δ_{i_1} becomes zero, which means $d_{i_1} = 0$, the first nonzero element of d becomes d_{i_2}. At this point, $d_{i_2} = \delta_{i_2}$. So, we can delete δ_{i_1} and make δ_{i_2} be the leading element of δ sequence. In fact, δ sequence is implemented in a linked list. When we change the leading element,

Table 9: The row tower formation.

$R(S[1,1])$	$R(S[1,2])$	$\cdots$	$R(S[1,n-1])$	$R(S[1,n])$
	$R(S[2,2])$	$\cdots$	$R(S[2,n-1])$	$R(S[2,n])$
		$\ddots$	$\vdots$	$\vdots$
			$R(S[n-1,n-1])$	$R(S[n-1,n])$
				$R(S[n,n])$

Table 10: The δ representation of d sequence.

	$d = 58369$					$d = 47258$				
Sorted d	3	5	6	8	9	2	4	5	7	8
δ	3	2	1	2	1	2	2	1	2	1

we just change the leading pointer to the next element. Because the value of δ_{i_2} is known when δ_{i_1} is decreased to zero, the EXPIRE operation requires only $O(1)$ time.

Our algorithm for finding the LIS of every substring is given in Algorithm 1. A detailed implementation will be given in the next section. As we can see, the chain replacement and predecessor finding operations require $O(l)$ time, so the ADD and EXPIRE operations need $O(l)$ and $O(1)$ time, respectively. Note that we reduce the time complexity of the EXPIRE operation from $O(l)$ to $O(1)$, but time complexity of the ADD operation remains the same. Thus, the total time complexity required for preprocessing is $n * O(l) + \frac{(n-1)^2}{2} * O(1) = O(n^2)$. The total space required for storing pr, σ and δ is $O(n^2)$, which will be explained in the next section. After the preprocessing, for a query on an arbitrary window, we can get the LIS length in $O(1)$ time, and the LIS content in $O(l)$ time. If we want to report the LIS content of all windows with all sizes, then the required time is $n^2 * O(l) = O(n^2 l)$.

Algorithm 1 Find LIS of every substring

```
{Function add}
Find the position to insert, ip.
if the inserted position is not to the last then
    Do chain replacement on δ and σ.
end if
Link the new node to the end of the linked list.
for i=ip-1 down to 1 do
    Find the predecessor of the new element in position i.
end for

{Function expire}
Decrease the δ_{i,1} by one.
if δ_{i,1}=0 then
    Consider δ_{i,2} as the leading element in the future.
end if

{Output}
while The current element is not in the first position do
    The current element is the predecessor of the original element in the original position.
    Decrease the position by one.
end while

{Function Preprocess}
for i = 0 to n do
    Add S[i] to T;
    for j = 0 to i do
        Expire from T;
    end for
end for
```

4 Implementation Details

We define a data structure *snode* to store the values of R and δ. The data structure $RLIS$ consists of two doubly linked lists and σ table. One doubly linked list vl linking the *snode* nodes in the order of R with a pointer to the successor v and a pointer to the predecessor w. We define a notation $VL(A, B)$ that means $A.v = B$, $B.w = A$. And the other doubly linked list el link the nodes according to the order of the σ with a pointer to the successor e and a pointer to the predecessor f, and $EL(A, B)$ means $A.e = B$, $B.f = A$. Let $Ne(Nf)$ denote the *snode* with maximum (mini-

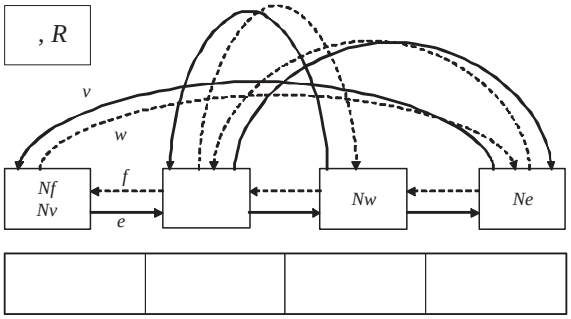

Figure 1: The data structure.

mum) δ value, and $Nv(Nw)$ denote the *snode* with maximum (minimum) R value. The data structure is illustrated in Figure 1, and the representation of the 8th row tower in my data structure is shown in the fisrt part of Figure 2.

Our algorithm consists of two phases. In the first phase, to find the $RLIS$ of all prefixes $S[1, j]$ of S, we ADD the symbols of the input string one by one starting from the first symbol. In the second phase, we continuously apply EXPIRE operations to the RLIS of $S[1, j]$ to get the RLIS of $S[i, j]$.

One ADD operation consists of three parts: update the vl link, update the el link, update the predecessor table t.

An example of the update operation is illustrated in Figure 3 where the array in the bottom represents the order of the value stored in the nodes. Let $d(e)$ and $\delta(R)$ denote the d and δ values of a pr element R, respectively. In this example, b is replaced by h. The first element after b in el with larger σ value is e, so now $\delta(e) = q$. Then the first element after e in el with larger value is f, so now f has $\delta(f) = s$. And after f in el, there is no element larger than f. Suppose originally $d(e) = x$, then $d(f) = x + t$, $d(g) = x + t + u$. After the replacement of b, there is no element R with $d(R) = x + t$, so now $\delta(g) = t + u$. After that, we insert a new node for the new element R, and $\delta(R) = 1$ because the d value of the previously added element is always one less than $d(R)$.

The ADD operation consists of two cases, each has to perform the three parts.

Case 1: The newly added element R is larger than all previous elements.

Because the newly added node would not replace any previous *snode*, so it would not af-

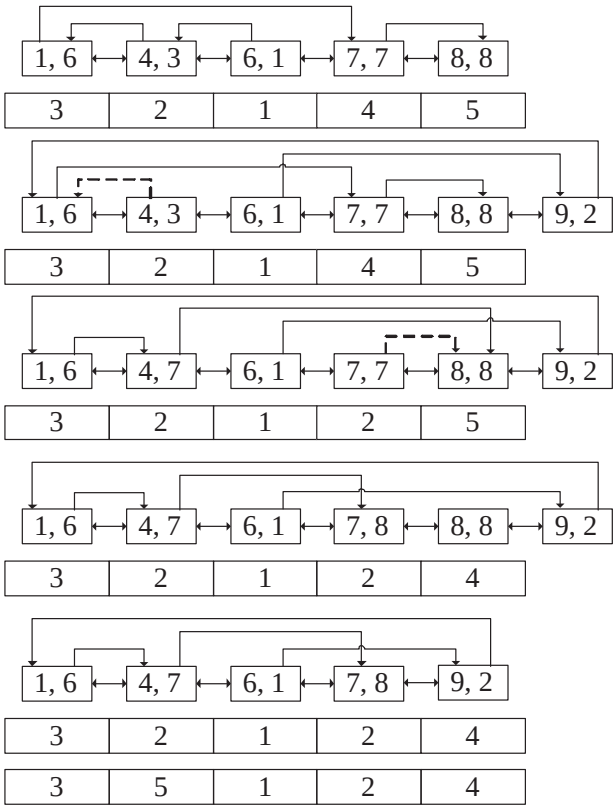

Figure 2: The data structure update for input=4562317829 when the 9th symbol is added. Here, for clear illustration, the d value is shown instead of the δ value.

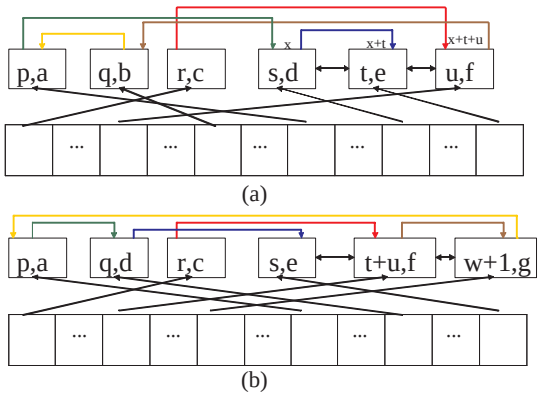

Figure 3: The update of δ sequence.

fect δ of other *snodes*. So we only need to perform $VL(Nv, R)$.

Case 2: The newly added element R is not larger than all previous elements.

Perform $VL(r.w, R)$ and $VL(R, r.v)$. And then perform the chain replacement if it is required.

For both cases, we need to link the new node in the *el* after the current Ne, and set Ne to the new node.

The complete implementation of the ADD operation is given in Algorithm 2, the EXPIRE operation is given in Algorithm 3.

In the first phase, we need $O(n)$ space for each element to record its predecessor, so the total space required to record the predecessor table is $O(n^2)$. For each newly added element, we need to record the $RLIS$ individually, so we require $O(n^2)$ space totally to record the data in the first column in τ.

Although in the second phase, σ might change due to element expiration, we do not need to update it since the σ table is unused in following EXPIRE operations and future query. The predecessor table remains the same during the second phase. The $RLIS$'s in every MD_k can share with the $RLIS$ of $R(S[1, k])$, so no additional space is required for recording the $RLIS$'s. What we need to record for the future query is the LIS length and the ending symbol of R. Thus, we need $O(n^2)$ space for both phases

Figure 2 shows an example for demonstrating the data structure used in the algorithm. For clear demonstration, we use d instead of δ in this example. Figure 2 shows the steps of update when 2 at position 9 is added. Starts at (6, 1), we get $1 < 2$, so it is not a candidate to be replaced. Following the *vl* to (4, 3), we have $3 > 2$, so 3 is the first candidate to be replaced. Then we insert (9, 2) and mark up (4, 3) and start to do the chain replacement. Next we come to (1, 6), because $\sigma(3) = 2 > \sigma(6) = 1$, 6 is not a candidate. Then we come to (7, 7), $\sigma(7) = 4 > 3$, so (7, 7) is the next replaced *snode*. We copy $d = 4$ to (7, 7) and make it become (4, 7), and reconnect *el* with (4, 7), and make $\sigma[4] = 2$. Then (8, 8) is reached, $\sigma(8) = 5 > 4$, so (8, 8) is the next replaced *snode*. We copy $d = 7$ to (8, 8), make it become (7, 8), reconnect *el* with (7, 8) and make $\sigma[5] = 4$. Then we come to the end of the *vl* list, so (8, 8) is the real victim, so we link (7, 8) with (9, 2). Continue to find from the first element of *vl* to the inserted

Algorithm 2 Add

```
1: Find the position to insert, ip.
2: if the inserted position is after Nv then
3:     VL(Nv, R).
4:     Nv = R.
5: else
6:     There is a snode r being replaced.
7:     VL(r.w, R) and VL(R, r.v).
8:     if r = Nw then
9:         Nw = R;
10:    end if
11:    nnow = r;
12:    cp = ip;
13:    rs = σ[ip];
14:    while the end of the list is not reached do
15:        nnow = nnow.v;
16:        cp + +;
17:        if σ[cp] > rs then
18:            temp = nnow;
19:            EL(r → f, nnow);
20:            EL(nnow, r → e);
21:            δ(nnow) = δ(r);
22:            r = temp;
23:            exchange(rs, σ[cp]);
24:        end if
25:    end while
26:    for i = 1 to ip − 1 do
27:        if σ[cp] > rs then
28:            exchange (σ[cp], rs);
29:        end if
30:    end for
31: end if
32: σ[ip] = l;
33: EL(r → f, r → e);
34: δ(r → e)+ = δ(r);
35: EL(Ne, R);
36: Ne = R;
37: δ(R) = 1;
38: if r=Nf then
39:    Nf = Nf → e;
40: end if
41: dtn = σ[ip − 1];
42: nnow = R.w;
43: for i=ip-1 down to 1 do
44:    if σ[i] > dtn then
45:        dtn = σ[i];
46:        an = nnow.R;
47:    end if
48:    t[R][i + 1] = an;
49:    nnow = nnow.w;
50: end for
```

Algorithm 3 Expire

```
1: Decrease δ(Nf) by one.
2: if δ(Nf)=0 then
3:    if Nf = Nv then
4:       Nv = Nv → w
5:    else
6:       VL(Nf.w, Nf.v)
7:    end if
8:    Nf = Nf → e
9: end if
```

Algorithm 4 Output(N, pos)

```
1: while pos!=0 do
2:    output N;
3:    Output(t[N][pos], pos − 1);
4: end while
```

Algorithm 5 Main

```
1: for i = 0 to n do
2:    rt[i] = rt[i − 1].Add(S[i]);
3:    rt[i].Output();
4:    temp = rt[i];
5:    for j = 0 to i do
6:       temp.Expire();
7:       temp.Output(the last element of R in
         rt[i], l);
8:    end for
9: end for
```

position, there is no element with σ larger than 5, so the update of σ also terminates. And then we make $\sigma(2) = 5$. The construction of the predecessor table starts from the previous of 2 in vl, whose value is 1, and there is no element in front of 1 with larger σ than 1. So 1 is the predecessor of 2 at any time.

5 Conclusion and Future Work

In this paper, we proposed an efficient algorithm for solving the problem of finding the LIS of every substring. For given an input string of length n, there are total $O(n^2)$ substrings, whose lengths vary from 1 to n. In the precessing stage, $O(n)$ ADD operations and $O(n^2)$ EXPIRE operation are needed. Originally, in the algorithm proposed by Albert et al.[1], both ADD and EXPIRE operations requires $O(l)$ time, where l represents the LIS length. We improve the EXPIRE operation to reduce the required time to only $O(1)$. And the time required for the ADD operation remains unchanged. Thus, the time complexity of our preprocessing is $O(n^2)$.

After our preprocessing, for each query on a substring of an arbitrary length, we can answer the LIS length in constant time and the LIS content in $O(l)$ time, which is linear time. If one desires to report the LIS contents of all substrings, then $O(OUTPUT) = O(n^3)$ time is required, where $OUTPUT$ represents the sum of the lengths of the output. Our algorithm is optimal in answering either the LIS lengths or contents of all substrings.

In this paper, we solved the LIS window problem for one input string. In the future, the design of efficient algorithm for solving the LCIS window problem for two or more input strings may be worth further study.

References

[1] M. H. Albert, A. Golynski, A. M. Hamel, A. Lopez-Ortiz, S. S. Rao, and M. A. Safari, "Longest increasing subsequences in sliding windows," *Theoretical Computer Science*, pp. 413–432, 2004.

[2] D. Aldous and P. Diaconis, "Longest increasing subsequences: From patience sorting to the baik-deift-johansson theorem," *BAMS: Bulletin of the American Mathematical Society*, Vol. 36, pp. 413–432, 1999.

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

[4] G. S. Brodal, K. Kaligosi, I. Katriel, and M. Kutz, "Faster algorithms for computing longest common increasing subsequences," Tech. Rep. BRICS-RS-05-37, BRICS, Department of Computer Science, University of Aarhus, Dec. 2005.

[5] W. T. Chan, Y. Zhang, S. P. Y. Fung, D. Ye, and H. Zhu, "Efficient algorithms for finding a longest common increasing subsequence.," *The 16th Annual International Symposium on Algorithms and Computation, Hainan, China*, pp. 665–674, 2005.

[6] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, Vol. 20, No. 5, pp. 350–353, 1977.

[7] I. Katriel and M. Kutz, "A faster algorithm for computing a longest common increasing subsequence," Research Report MPI-I-2005-1-007, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, Mar. 2005.

[8] D. Liben-Nowell, E. Vee, and A. Zhu, "Finding longest increasing and common subsequences in streaming data," *11th International Computing and Combinatorics Conference, Kunming, China*, pp. 263–272, 2005.

[9] E. Mäkinen, "On the longest upsequence problem for permutations," Tech. Rep. A-1999-7, Department of Computer Science, University of Tampere, 1999.

[10] F. Malucelli, T. Ottmann, and D. Pretolani, "Efficient labelling algorithms for the maximum noncrossing matching problem," *Discrete Applied Mathematics*, Vol. 47, No. 2, pp. 175–179, 1993.

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

[12] P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and implementation of an efficient priority queue.," *Mathematical Systems Theory*, Vol. 10, pp. 99–127, 1977.

[13] C. B. Yang and R. C. T. Lee, "Systolic algorithms for the longest common subsequence problem.," *Journal of the Chinese Institute of Engineers*, Vol. 10, No. 6, pp. 691–699, 1987.

[14] I. H. Yang, C. P. Huang, and K. M. Chao, "A fast algorithm for computing a longest common increasing subsequence.," *Information Processing Letters*, Vol. 93, No. 5, pp. 249–253, 2005.

[15] M. S. Yu, L. Y. Tseng, and S. J. Chang, "Sequential and parallel algorithms for the maximum-weight idependent set problem on permutation graphs.," *Information Processing Letters*, Vol. 46, No. 1, pp. 7–11, 1993.