An Improved $O(R \log\log n + n)$ Time Algorithm for Computing the Longest Common Subsequence

Daxin Zhu, Lei Wang, and Xiaodong Wang*

***Abstract*—In this paper, we revisit the much studied LCS problem for two given sequences. Based on the algorithm of Iliopoulos and Rahman for solving the LCS problem, we have suggested 3 new improved algorithms. We first reformulate the problem in a very succinct form. The problem LCS is abstracted to an abstract data type DS on an ordered positive integer set with a special operation $Update(S, x)$. For the two input sequences X and Y of equal length n, the first improved algorithm uses a van Emde Boas tree for DS and its time and space complexities are $O(R \log\log n + n)$ and $O(R)$, where R is the number of matched pairs of the two input sequences. The second algorithm uses a balanced binary search tree for DS and its time and space complexities are $O(R \log L + n)$ and $O(R)$, where L is the length of the longest common subsequence of X and Y. The third algorithm uses an ordered vector for DS and its time and space complexities are $O(nL)$ and $O(R)$.**

***Index Terms*—longest common subsequence, NP-hard problems, dynamic programming, time complexity.**

I. Introduction

THE longest common subsequence (LCS) problem is a classic problem in computer science. The problem has several applications in many apparently unrelated fields ranging from file comparison, pattern matching and computational biology [1], [2], [4], [5], [14]–[16].

Given two sequences X and Y, the longest common subsequence (LCS) problem is to find a subsequence of X and Y whose length is the longest among all common subsequences of the two given sequences.

The classic algorithm to LCS problem is the dynamic programming solution of Wagner and Fischer [13], with $O(n^2)$ worst case running time. Masek and Paterson [7] improved this algorithm by using the "Four-Russians" technique to reduce its running time to $O(n^2/\log n)$ in the worst case. Since then, there has been not much improvement on the time complexity in terms of n found in the literature. However, there were several algorithms with time complexities depending on other parameters. For example, Myers in [8] and Nakatsu et al. in [9] presented an $O(nD)$ algorithm, where the parameter D is the Levenshtein distance of the two given sequences. The number of matched pairs of the two input sequences R, is perhaps another interesting and more relevant parameter for LCS problem. Hunt and Szymanski [5] presented an $O((R + n) \log n)$ time algorithm to solve LCS problem. Their paper also cited applications, where $R \sim n$ and thus the algorithm would run in $O(n \log n)$ time for these applications. To the authors' knowledge, the most efficient algorithm sofar for solving the LCS problem is the $O(R \log\log n + n)$ time algorithm presented by Iliopoulos and Rahman [6], [11]. The key point of their algorithm is to solve a restricted dynamic version of the Range Maxima Query problem by using some interesting techniques of [10], and combining them with van Emde Boas structure [12]. Readers are referred to [1] for a more comprehensive comparison of the well-known algorithms for LCS problem and their behavior in various application.

In this paper, we will revisit the classic LCS problem for two given sequences and present new algorithms with some interesting new observations and some novel ideas. We first reformulate the problem in a very succinct form. The problem LCS is abstracted to an abstract data type DS on an ordered positive integer set with a special operation $Update(S, x)$. Three new improved algorithms have been suggested. The first improved algorithm uses a van Emde Boas tree for DS and its time and space complexities are $O(R \log\log n + n)$ and $O(R)$, where R is the number of matched pairs of the two input sequences. The second algorithm uses a balanced binary search tree for DS and its time and space complexities are $O(R \log L + n)$ and $O(R)$, where L is the length of the longest common subsequence of X and Y. The third algorithm uses an ordered vector for DS and its time and space complexities are $O(nL)$ and $O(R)$. The three new algorithms improve the $O(R \log\log n + n)$ time algorithm of Iliopoulos and Rahman. Our novel algorithm has a very simple structure. It is very easy to implement and thus very practical.

The organization of the paper is as follows.

In the following 4 sections, we describe our improved $O(R \log\log n + n)$ time algorithm of Iliopoulos and Rahman for solving LCS problem.

In Section 2, the preliminary knowledge for presenting our algorithm for LCS problem is discussed, and the $O(R \log\log n + n)$ time algorithm of Iliopoulos and Rahman is reviewed briefly. In Section 3, we present our improvements on the algorithm of Iliopoulos and Rahman with time complexity $O(R \log\log n + n)$, where n and R are the lengths of the two given input strings, and the number of matched pairs of the two input sequences, respectively. Some concluding remarks are located in Section 4.

II. An $O(R \log\log n + n)$ time Algorithm

In this section, we briefly review the $O(R \log\log n + n)$ time algorithm of Iliopoulos and Rahman [6] for the sake of completeness.

Manuscript received December 9, 2016; revised February 22, 2017.

This work was supported in part by the Quanzhou Foundation of Science and Technology under Grant No.2013Z38, Fujian Provincial Key Laboratory of Data-Intensive Computing and Fujian University Laboratory of Intelligent Computing and Information Processing.

Daxin Zhu is with Quanzhou Normal University, Quanzhou, China.(email:dex@qztc.edu.cn)

Lei Wang is with Facebook, 1 Hacker Way, Menlo Park, CA 94052, USA.

Xiaodong Wang is with Fujian University of Technology, Fuzhou, China.

*Corresponding author.

A sequence is a string of characters over an alphabet $\sum$. A subsequence of a sequence X is obtained by deleting zero or more characters from X (not necessarily contiguous). A substring of a sequence X is a subsequence of successive characters within X.

For a given sequence $X = x_1x_2\cdots x_n$ of length n, the ith character of X is denoted as $x_i \in \sum$ for any $i = 1, \cdots, n$. A substring of X from position i to j can be denoted as $X[i:j] = x_ix_{i+1}\cdots x_j$. If $i \neq 1$ or $j \neq n$, then the substring $X[i:j] = x_ix_{i+1}\cdots x_j$ is called a proper substring of X. A substring $X[i:j] = x_ix_{i+1}\cdots x_j$ is called a prefix or a suffix of X if $i = 1$ or $j = n$, respectively.

Definiton 1: An appearance of sequence $X = x_1x_2\cdots x_n$ in sequence $Y = y_1y_2\cdots y_n$, for any X and Y, starting at position j is a sequence of strictly increasing indexes $i_1, i_2, \cdots, i_n$ such that $i_1 = j$, and $X = y_{i_1}, y_{i_2}, \cdots, y_{i_n}$.

A match for sequences X and Y is a pair (i, j) such that $x_i = y_j$. The set of all matches, M, is defined as follows:

$$M = \{(i, j)|x_i = y_j, 1 \leq i, j \leq n\}$$

The total number of matches for X and Y is denoted by $R = |M|$.

It is obvious that $R \leq n^2$.

Definiton 2: A common subsequence of the two input sequences $X = x_1x_2\cdots x_n$ and $Y = y_1y_2\cdots y_n$, denoted $cs(X, Y)$, is a subsequence common to both X and Y. The longest common subsequence of X and Y, denoted $LCS(X, Y)$, is a common subsequence whose length is the longest among all common subsequences of the two given sequences. The length of $LCS(X, Y)$ is denoted as $r(X, Y)$.

In this paper, the two given sequences X and Y are assumed to be of equal length. But all the results can be easily extended to the case of two sequences of different length.

Definiton 3: Let $T(i, j)$ denote $r(X[1:i], Y[1:j])$, when $(i, j) \in M$. $T(i, j)$ can be formulated as follows:

In the algorithm of Iliopoulos and Rahman [3], a vector H of length n is used to denote the maximum value so far of column l of T. For the current value of $i \in [1..n]$, $H_i(l) = max_{1\leq k<i,(k,l)\in M}\{T(k, l)\}, 1 \leq l \leq n$. The footnote i is used to indicate that the current row number is i, and the values of $H_i(l), 1 \leq l \leq n$ are not changed for row i in the algorithm. We can omit the footnote i if the current row is clear.

The most important function $RMQ_i(left, right) = max_{left\leq l\leq right}\{H_i(l)\}$ used in the algorithm is a range maxima query on H for the range $[left..right], 1 \leq left \leq right \leq n$. It is clear that

$$T(i, j) = 1 + RMQ_i(1, j - 1), \quad \texttt{if} \quad (i, j) \in M \qquad (2)$$

For the efficient computation of $T(i, j)$, the following facts [10] are utilized in the algorithm of Iliopoulos and Rahman.

Fact 1: Suppose $(i, j) \in M$. Then for all $(i', j) \in M, i' > i$, (resp.$(i, j') \in M, j' > j$), we must have $T(i', j) \geq T(i, j)$ (resp.$T(i, j') \geq T(i, j)$).

Fact 2: The calculation of the entry $T(i, j), (i, j) \in M, 1 \leq i, j \leq n$, is independent of any $T(l, q), (l, q) \in M, l = i, 1 \leq q \leq n$.

The algorithm is proceed in a row by row manner as follows.

Algorithm 1 Iliopoulos-Rahman-LCS

```
for i = 1 to n do
    H ← S {Update H for the next row}
    for all (i, j) ∈ M do
        T(i, j) ← 1 + RMQ(1, j − 1)
        S(j) ← T(i, j)
    end for
end for
```

In the algorithm, another vector S, of length n, is invoked as a temporary vector. After calculating $T(i, j)$, the vector S is restored from T, $S(j) = T(i, j)$, if $(i, j) \in M$. Therefore, at the end of processing of row i, S is actually H_{i+1}. The algorithm continues in this way as long as it is in the same row. As soon as it comes a new row, the vector H is updated with new values from S. In the algorithm above, $RMQ(1, j - 1)$ is actually $RMQ_i(1, j - 1)$. We can omit the footnote i since the processing is in the same row i. The correctness of the above procedure follows from Facts 1 and 2. The problem RMQ can be solved in $O(n)$ preprocessing time and $O(1)$ time per query [2]. Therefore, for the constant time range maxima query, an $O(n)$ preprocessing time has to be paid as soon as H is updated. Due to Fact 2, it is sufficient to perform this preprocessing once per row. So, the computational effort added for this preprocessing is $O(n^2)$ in total.

The most complicated part of the algorithm of Iliopoulos and Rahman is in the computation of $RMQ(1, j - 1)$. To eliminate the n^2 term from the running time of the algorithm, a van Emde Boas tree is used to store the information in H. This data structure can support the operations $Search$, $Insert$, $Delete$, Min, Max, $Succ$, and $Pred$ in $O(\log\log n)$ time. By using these operations the query $RMQ(1, j - 1)$ can then be answered in $O(\log\log n)$ time. The $O(n)$ preprocessing step of the algorithm can then be avoided and hence the n^2 term can be eliminated from the running time. However, as a price to pay, the query time of $RMQ(1, j - 1)$ increases to $O(\log\log n)$.

Finally, the time complexity is improved to $O(R\log\log n + n)$, using $O(n^2)$ space. The vEB structure described in the algorithm of Iliopoulos and Rahman is somewhat involved. For more details of the data structure, the readers are referred to [10].

III. Improvements of the algorithm

In this section, we will present several improvements on the algorithm of Iliopoulos and Rahman.

A. A first improvement

In the algorithm of Iliopoulos and Rahman, the set $M = \{(i, j)|x_i = y_j, 1 \leq i, j \leq n\}$ is constructed explicitly in the lexicographic order of the matches such that all the matches can be processed in the correct order. To this purpose, two separate lists, L_X and L_Y are built in $O(n)$ time. For each symbol $c \in \sum$, $L_X(c)$ (resp. $L_Y(c)$) stores, in sorted order, the positions of c in X (resp. Y), if any. Then, a van Emde Boas structure is used to build the set M in $O(R\log\log n)$ time. This preprocessing step can be removed completely from our improved algorithm, since we do not need to access

$$T(i,j) = \begin{cases} \text{Undefined} & \text{if } (i,j) \notin M \\ 1 & \text{if } (i,j) \in M \text{and } (i',j') \notin M, i' < i, j' < j \\ \max_{\substack{1 \le l_i < i \\ 1 \le l_j < j \\ (l_i,l_j) \in M}} \{T(l_i,l_j)\} & \text{Otherwise} \end{cases} \quad (1)$$

each match (i,j) directly. It will be clear in the description of the algorithm in latter section.

Secondly, to avoid overlap, the information in H are stored separately in row i and $i+1$. We have noticed that, the only query on H used in the algorithm is of the form $RMQ(1,j-1) = \max_{1 \le l < j}\{H(l)\}$. It is obvious that if in each row i, we process all $j \in L_Y(x_i)$ in a decreasing order, i.e. the order of j from large to small, then every overlap for current j will not change the values $RMQ(1,j-1)$ for the succeed j. Therefore, the information in H can be updated in same row immediately. In our improved algorithm, the list L_Y is built in this way such that for each symbol $c \in \sum$, the positions of c in Y are listed from large to small.

B. The key improvements

By the definition of $H_i(l) = max_{1 \le k < i,(k,l) \in M}\{T(k,l)\}, 1 \le l \le n$ for the current value of $i \in [1..n]$, it is not difficult to observe the following fact.

Fact 3: For all $1 \le i' < i \le n$, and $1 \le j \le n$, $H_{i'}(j) \le H_i(j)$.

The information in H we maintained is used for computing $RMQ(1,j-1)$ in the algorithm. If we use a vector Q of length n to store the values of $Q(j) = RMQ(1,j)$ for all $1 \le j \le n$, then $RMQ(1,j-1)$ will be computed more directly by $Q(j-1)$ in $O(1)$ time. For example, if $H = (0,1,2,1,2,0,1)$, then $Q = (0,1,2,2,2,2,2)$. An important albeit easily observable fact about Q is that, its components $Q(j), 1 \le j \le n$ are nondecreasing.

Fact 4: Suppose $Q(j) = RMQ(1,j) = max_{1 \le l \le j}\{H(l)\}$. Then the values of $Q(j), 1 \le j \le n$ form a nondecreasing sequence. This sequence is under a very special form. If $Q(j) < Q(j+1), 1 \le j < n$, then $Q(j+1) = Q(j)+1$. Therefore, the values of this sequence are taken from a consecutive integer set $\{0,1,\cdots,L\}$, where $L = r(X,Y)$ is the length of the longest common subsequence of X and Y.

Proof.

For any $1 \le j' < j \le n$, $Q(j') = max_{1 \le l \le j'}\{H(l)\}$ and $Q(j) = max_{1 \le l \le j}\{H(l)\}$. It follows from $j' < j$ that $\{l | 1 \le l \le j'\} \subset \{l | 1 \le l \le j\}$. It follows then $Q(j') \le Q(j)$. This proves that the values of $Q(j), 1 \le j \le n$ form a nondecreasing sequence.

In the case of $Q(j) < Q(j+1), 1 \le j < n$, since $Q(j+1)$ and $Q(j)$ are both nonnegative integers, we have:

$$Q(j+1) \ge Q(j) + 1 \quad (3)$$

On the other hand, by the definition of Q, if current row number is i, we have:

$$\begin{aligned} Q(j+1) &= \max_{1 \le l \le j+1}\{H_i(l)\} \\ &= \max\{\max_{1 \le l \le j}\{H_i(l)\}, H_i(j+1)\} \\ &= \max\{Q(j), H_i(j+1)\} \\ &= H_i(j+1) \end{aligned} \quad (4)$$

It follows from the definition of $H_i(j+1)$ that

$$\begin{aligned} H_i(j+1) &= \max_{1 \le k < i,(k,j+1) \in M}\{T(k,j+1)\} \\ &= T(k',j+1), 1 \le k' < i, (k',j+1) \in M \\ &= Q_{k'}(j) + 1 \end{aligned} \quad (5)$$

It follows from Fact 3 and $k' < i$ that $H_{k'}(j) \le H_i(j)$, and thus $Q_{k'}(j) \le Q_i(j) = Q(j)$.

It follows from (4) and (5) that

$$\begin{aligned} Q(j+1) &= H_i(j+1) \\ &= Q_{k'}(j) + 1 \\ &\le Q(j) + 1 \end{aligned} \quad (6)$$

Combining (3) and (6) we then have, $Q(j+1) = Q(j)+1$. The proof is completed. □

By the definition of Q we know, if current row number is i, then for $1 \le j \le n$,

$$\begin{aligned} Q(j) &= \max_{1 \le l \le j}\{H_i(l)\} \\ &= \max_{1 \le l \le j} \max_{1 \le k < i,(k,j) \in M}\{T(k,j)\} \\ &= \max_{\substack{1 \le k < i \\ 1 \le l \le j}}\{T(k,l)\} \end{aligned} \quad (7)$$

At the end of the algorithm, all the n rows are treated. At this time, we have, for $1 \le j \le n$,

$$Q(j) = \max_{\substack{1 \le k \le n \\ 1 \le l \le j}}\{T(k,l)\} \quad (8)$$

In other words, $Q(j) = r(X, Y[1:j])$, $1 \le j \le n$. Especially, $Q(n) = L = r(X,Y)$. If we want compute $L = r(X,Y)$, but not $LCS(X,Y)$, then we do not need to store the 2 dimensional array T. At the end of the algorithm, $Q(n)$ will return the answer.

Because of its unusual form, Q can be viewed as a piecewise linear function. As we know, it is sufficient to record the break points of such functions to calculate their values. Therefore, we can use another vector P of length at most L to record the break points of Q. Let $m = \max_{1 \le i \le n}\{Q(i)\}$. For $1 \le j \le n$, the value of $P(j)$ can be defined as follows:

$$P(j) = \begin{cases} \min_{1 \le i \le n}\{i | Q(i) = j\} & \text{if } j \le m \\ n+1 & \text{Otherwise} \end{cases} \quad (9)$$

Let

$$S(j) = P(j), \quad \texttt{if} \quad 1 \le j \le n, 1 \le P(j) \le n \tag{10}$$

It is clear that S forms an increasing sequence of length at most L.

For instance, if $Q = (0, 1, 2, 2, 2, 2, 2)$, then $P = (2, 3, 8, 8, 8, 8, 8)$, and $S = \{2, 3\}$.

Let $\alpha = |S|$, then $S(\alpha)$ is the maximal element of S.

By the definition of S, if $k < \alpha$, then for any $S(k) \le j < S(k+1)$, we have $Q(j) = k$. In the case of $k = \alpha$, for any $S(\alpha) \le j \le n$, we have $Q(j) = \alpha$. Therefore, $Q(j)$ can be easily computed by S as follows.

$$Q(j) = \begin{cases} k & \text{if } S(k) \le j < S(k+1) \\ \alpha & j \ge S(\alpha) \end{cases} \tag{11}$$

Furthermore, we can wind up the following fact.

Fact 5: Suppose $P(j), 1 \le j \le n$ be defined by formula (9). Then $S = \{P(j) | 1 \le j \le n, 1 \le P(j) \le n\}$ forms an increasing sequence of length at most L. This sequence has a very unique property. For each $P(t) \in S$, and any $P(t) \le j < P(t+1)$, $t = \max_{\substack{1 \le k < i \\ 1 \le l \le j}} \{T(k, l)\}$, if current row number is i. At the end of the algorithm, for each $P(t) \in S$, $t = r(X, Y[1 : j])$ for any $P(t) \le j < P(t+1)$. Especially, the maximal element of S is $P(L)$, and $L = r(X, Y)$.

Proof.

By the definition of P, we have, for each $P(t) \in S$, if $P(t) \le j < P(t+1)$, then $Q(j) = t$. Therefore, it follows from formula (7) that if current row number is i, then $t = \max_{\substack{1 \le k < i \\ 1 \le l \le j}} \{T(k, l)\}$

At the end of the algorithm, all the n rows are treated. At this time, for each $P(t) \in S$, $t = \max_{\substack{1 \le k \le n \\ 1 \le l \le j}} \{T(k, l)\} = r(X, Y[1 : j])$.

Especially, if $P(t')$ is the maximal element of S, then $P(t') \le n < P(t'+1) = n+1$, and thus,

$t' = r(X, Y[1 : n]) = r(X, Y) = L$, i.e., the maximal element of S is $P(L)$, and $L = r(X, Y)$.

The proof is completed. □

In the algorithm, if current match (i, j) is processed, then the value of $T(i, j)$ is changed to $1 + Q(j-1)$. Let $Q(j')$ be the successor of $Q(j-1)$ in Q, i.e.,

$$j' = \min_{j \le l \le n} \{l | Q(l) > Q(j-1)\} \tag{12}$$

Then $Q[j : j'-1]$ must be changed to $1 + Q(j-1)$, according to the definition of Q.

Similarly, Let $S(k)$ be the successor of $j-1$ in S, i.e.,

$$k = \min_{1 \le l \le n} \{l | S(l) > j-1\} \tag{13}$$

Then $S(k)$ must be changed to j, according to the definition of S.

In the case of $j-1$ has no successor in S, i.e., $j-1 \ge S(\alpha)$, then j must be added into S.

For example, if $Q = (0, 1, 2, 2, 2, 2, 2)$, $S = (2, 3)$, and current match $(4, 6)$ is processed, then $T(4, 6)$ is changed to $1 + Q(j-1) = 3$. In this case, $j = 6$ and $j' = 8$, and thus $Q[6 : 7]$ must be replaced by 3. The current values of Q becomes $Q = (0, 1, 2, 2, 2, 3, 3)$. At this point, $j - 1 = 5 > S(\alpha) = 3$, and thus j must be added to S. The current values of S becomes $S = (2, 3, 6)$.

C. The improved algorithm

It is readily seen from the discussions above that if we can use the ordered set S defined by formula (11) to calculate the function $RMQ(1, j-1)$, then the algorithm will be simplified substantially. The key point is the way to maintain the ordered set S efficiently.

Let $U = \{j | 1 \le j \le n\}$. Suppose DS be an abstract data type on an ordered positive integer set S. The abstract data type DS can support the following operations on S:

1) $Size(S)$
   A query on an ordered positive integer set S that returns the number of integers in S.
2) $Succ(S, x)$
   A query that, given a positive integer x whose key is from U, returns the next larger integer in S, or 0 if x is the maximum integer in S.
3) $Pred(S, x)$
   A query that, given a positive integer x whose key is from U, returns the next smaller integer in S, or 0 if x is the minimum integer in S.
4) $Update(S, x)$
   A modifying operation that, given a positive integer x, if $Succ(S, x-1) > 0$, then replace $Succ(S, x-1)$ with the integer x, otherwise augments the set S with a new integer x.

With this abstract data type, we can maintain the ordered positive integer set $S \subseteq U$ defined by formula (10) in our new algorithm LCS as follows.

Algorithm 2 LCS

```
1: S ← ∅
2: for i = 1 to n do
3:     for all j ∈ L_Y(x_i) do
4:         Update(S, j)
5:     end for
6: end for
7: return Size(S)
```

In above algorithm LCS, the list $L_Y(c), c \in \Sigma$ stores, in a decreasing order, the positions of c in Y. The $|\Sigma|$ lists can be constructed in $O(n)$ time, by simply scanning Y in turn. At the end of the algorithm, $L = r(X, Y) = Size(S)$, the length of $LCS(X, Y)$, is returned.

The efficiency of the new algorithm is depended heavily on the efficiency of the abstract data type DS, especially on the efficiency of its operation $Update(S, x)$.

We have found that the van Emde Boas tree is an elegant data structure for our purpose. Specifically, van Emde Boas trees support each of the following dynamic set operations, $Search$, $Insert$, $Delete$, Min, Max, $Succ$, and $Pred$ in $O(\log \log n)$ time. The operation $Update(S, x)$ can be easily implemented by combining at most two successive operations $delete$ and $insert$.

If we chose van Emde Boas tree as our data structure for S, the new algorithm can be described as follows.

The structure of algorithm $vEB - LCS$ is very simple. Although the algorithm can correctly return the length of $LCS(X, Y)$, it does not directly give $LCS(X, Y)$. If we want to compute $LCS(X, Y)$, but not just its length, we have to record more information. A commonly used method is to

Algorithm 3 vEB_LCS

```
S ← ∅
for i = 1 to n do
    for all j ∈ L_Y(x_i) do
        k ← Succ(S, j − 1)
        if k < Max(S) then
            Delete(S, k)
        end if
        Insert(S, j)
    end for
end for
return Size(S)
```

record the predecessor of each match $(i, j) \in M$ by a two dimensional array like $T(i, j)$. The two dimensional array returned by the algorithm allows us to quickly construct an LCS of X and Y. This method requires extra $O(n^2)$ space. For this purpose, we can design a more efficient method using only $O(R)$ space. We use two vectors B and C, both of length R, to record the predecessor's match number and the matched character for each match $(i, j) \in M$, respectively. The match numbers for all matches $(i, j) \in M$ are generated one after another in the algorithm.

Algorithm 4 vEB_LCS

```
m ← 0, S ← ∅
for i = 1 to n do
    for all j ∈ L_Y(x_i) do
        k ← Succ(S, j − 1)
        if k < Max(S) then
            Delete(S, k)
        end if
        Insert(S, j)
        p ← Pred(S, j)
        m ← m + 1
        B(m) ← D(p), C(m) ← j, D(j) ← m
    end for
end for
return Size(S)
```

In the above algorithm, the vector D is a temporary vector of length at most L used to store the match numbers for current set S. With the two vectors B and C computed in the above algorithm, the following recursive algorithm prints out $LCS(X, Y)$. The initial call is $Print - LCS(Size(S))$.

Algorithm 5 $Print\_LCS(k)$

```
if k ≤ 0 then
    return
else
    Print − LCS(B(k))
    print Y(C(k))
end if
```

It is obvious that $Print-LCS(Size(S))$ takes time $O(L)$, since it prints one character of $LCS(X, Y)$ in each recursive call. Finally, we can find that the following theorem holds.

Theorem 1: The algorithm vEB_LCS correctly computes $LCS(X, Y)$ in $O(R \log\log n + n)$ time and $O(R)$ space in the worst case, where R is the total number of matches for X and Y.

Proof. It follows from formulas (11) and (13) that the values of $RMQ(1, j-1)$ for each row i can be correctly computed by the set S maintained in the algorithm. Therefore, the algorithm can compute $LCS(X, Y)$ correctly as the original algorithm of Iliopoulos and Rahman. It is obvious that the computation of list $L_Y(c), c \in \Sigma$, costs $O(n)$ time and space. For each match $(i, j) \in M$, the algorithm executes each of the 4 operations $Succ$, $Succ$,$Succ$, and $Succ$ at most once, and each of the 4 operations costs $O(\log\log n)$. Therefore, the total time spent for these operations is $O(R \log\log n)$. The worst case time complexity of the algorithm is therefore $O(R \log\log n + n)$.

To maintain the van Emde Boas tree, $O(n)$ space is sufficient. To reconstruct the $LCS(X, Y)$, the algorithm uses two vectors B and C, both of size R. The space required by the algorithm is thus $O(R)$. The worst case space complexity of the algorithm is therefore $O(R)$.

The proof is completed. □

If we chose a balanced binary search tree such as red-black tree as our data structure for the ordered positive integer set S, the following dynamic set operations, $Search$, $Insert$, $Delete$, Min, Max, $Succ$, and $Pred$ can be implemented in $O(\log |S|)$ time. In this case, The time complexity of our algorithm becomes $O(R \log L)$, since $|S| \leq L$. We then can find that the following theorem holds.

Theorem 2: The longest common subsequence problem can be solved in $O(R \log L + n)$ time and $O(R)$ space in the worst case, where n, L and R are the length of input sequences X and Y, the length of $LCS(X, Y)$, and the total number of matches for X and Y, respectively.

The ordered positive integer set S in our algorithm can also be efficiently supported by an ordered vector s of length at most L as follows.

Algorithm 6 V_LCS

```
α ← 0, k ← −1
for i = 1 to n do
    for all j ∈ L_Y(x_i) do
        while k ≥ 0 and s(k) ≥ j do
            k ← k − 1
        end while
        s(k + 1) ← j
        if k = α then
            α ← 1 + α
        end if
        k ← α
    end for
end for
return α
```

In the above algorithm, for each row i, all columns $j \in L_Y(x_i)$ are processed in a decreasing order. The successors of all $j-1$ in s are searched and updated also in a decreasing order from the largest element $s(\alpha)$. It is obvious that the time spent for each row i is $O(\alpha)$. Therefore, we can conclude that the time complexity of the above algorithm is $O(nL)$, since $\alpha \leq L$.

Theorem 3: The longest common subsequence problem can be solved in $O(nL)$ time and $O(R)$ space in the worst

case, where n and L are the length of input sequences X and Y, and the length of $LCS(X, Y)$, respectively.

REFERENCES

[1] Bergroth, L., Hakonen,H., Raita,T., A survey of longest common subsequence algorithms, *SPIRE*, 2000, pp. 39-48.

[2] Crochemore M.,Hancart C., and Lecroq T., *Algorithms on strings*, Cambridge University Press, Cambridge, UK, 2007.

[3] Gusfield, D., *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, Cambridge, UK, 1997.

[4] Hirschberg, D.S., Algorithms for the longest common subsequence problem, *J. ACM* 24(4), 1977, pp. 664-675.

[5] Hunt, J.W., Szymanski, T.G., A fast algorithm for computing longest subsequences, *Commun. ACM*, 20(5), 1977, pp. 350-353.

[6] Iliopoulos, C.S., Rahman, M.S., A New Efficient Algorithm for Computing the Longest Common Subsequence, *Theor. Comput. Syst.*, 45, 2009, pp. 355-371.

[7] Masek, W.J., Paterson, M., A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1), 1980, pp. 18-31.

[8] Myers, E.W., An $O(nd)$ difference algorithm and its variations, *Algorithmica*, 1(2), 1986, pp. 251-266.

[9] Nakatsu, N., Kambayashi, Y., Yajima, S., A longest common subsequence algorithm suitable for similar text strings, *Acta Inf.*, 18, 1982, pp. 171-179.

[10] Rahman, M.S., Iliopoulos, C.S., Algorithms for computing variants of the longest common subsequence problem, *Lecture Notes in Computer Science*, 4288, 2006, pp. 399-408.

[11] Rahman, M.S., Iliopoulos, C.S., A new efficient algorithm for computing the longest common subsequence, *Lecture Notes in Computer Science*, 4508, pp. 82-90.

[12] van Emde Boas, P., Preserving order in a forest in less than logarithmic time and linear space, *Inf. Process. Lett.* 6, 1977, pp. 80-82.

[13] Wagner, R.A., Fischer, M.J., The string-to-string correction problem, *J. ACM*, 21(1), 1974, pp. 168-173.

[14] Zhu D., Wang L., Tian J., and Wang X., Efficient Algorithms for a Generalized Shuffling Problem, *IAENG International Journal of Computer Science*, vol. 41, no.4, 2014, pp237-248.

[15] Zhu D., Wang L., Tian J., and Wang X., A Simple Polynomial Time Algorithm for the Generalized LCS Problem with Multiple Substring Exclusive Constraints, *IAENG International Journal of Computer Science*, vol. 42, no.3, 2015, pp214-220.

[16] Zhu D., Wang L., Tian J., and Wang X., An Efficient Dynamic Programming Algorithm for a New Generalized LCS Problem, *IAENG International Journal of Computer Science*, vol. 43, no.2, 2016, pp204-211.