Longest Common Subsequence in at Least k Length Order-Isomorphic Substrings

Yohei Ueki[1(✉)], Diptarama[1], Masatoshi Kurihara[1], Yoshiaki Matsuoka[2], Kazuyuki Narisawa[1], Ryo Yoshinaka[1], Hideo Bannai[2], Shunsuke Inenaga[2], and Ayumi Shinohara[1]

[1] Graduate School of Information Sciences, Tohoku University, Sendai, Japan
{yohei_ueki,diptarama,masatoshi_kurihara}@shino.ecei.tohoku.ac.jp,
{narisawa,ry,ayumi}@ecei.tohoku.ac.jp

[2] Department of Informatics, Kyushu University, Fukuoka, Japan
{yoshiaki.matsuoka,bannai,inenaga}@inf.kyushu-u.ac.jp

Abstract. We consider the longest common subsequence (LCS) problem with the restriction that the common subsequence is required to consist of at least k length substrings. First, we show an $O(mn)$ time algorithm for the problem which gives a better worst-case running time than existing algorithms, where m and n are lengths of the input strings. Furthermore, we mainly consider the LCS in at least k length order-isomorphic substrings problem. We show that the problem can also be solved in $O(mn)$ worst-case time by an easy-to-implement algorithm.

Keywords: Longest common subsequence · Dynamic programming · Order-isomorphism · Order-preserving matching

1 Introduction

The *longest common subsequence (LCS)* problem is fundamental and well studied in computer science. The most common application of the LCS problem is measuring similarity between strings, which can be used in many applications such as the `diff` tool, the time series data analysis [12], and in bioinformatics.

One of the major disadvantages of LCS as a measure of similarity is that LCS cannot consider consecutively matching characters effectively. For example, for strings $X =$ `ATGG`, $Y =$ `ATCGGC` and $Z =$ `ACCCTCCCGCCCG`, `ATGG` is the LCS of X and Y, which is also the LCS of X and Z. Benson *et al.* [2] introduced the *longest common subsequence in k length substrings (LCS$_k$)* problem, where the subsequence needs to be a concatenation of k length substrings of given strings. For example, for strings $X =$ `ATCTATAT` and $Y =$ `TAATATCC`, `TAAT` is an LCS$_2$ since $X[4:5] = Y[1:2] =$ `TA` and $X[7:8] = Y[5:7] =$ `AT`, and no longer one exists. They showed a quadratic time algorithm for it, and Deorowicz and Grabowski [7] proposed several algorithms, such as a quadratic worst-case time algorithm for unbounded k and a fast algorithm on average.

Pavetić *et al.* [15] considered the *longest common subsequence in at least k length substrings (LCS$_{k+}$)* problem, where the subsequence needs to be a concatenation of *at least k* length substrings of given strings. They argued that LCS_{k+} would be more appropriate than LCS_k as a similarity measure of strings. For strings $X = \texttt{ATTCGTATCG}$, $Y = \texttt{ATTGCTATGC}$, and $Z = \texttt{AATCCCTCAA}$, $LCS_2(X,Y) = LCS_2(X,Z) = 4$, where $LCS_2(A,B)$ denotes the length of an LCS_2 between A and B. However, it seems that X and Y are more similar than X and Z. Instead, if we consider LCS_{2+}, we have $LCS_{2+}(X,Y) = 6 > 4 = LCS_{2+}(X,Z)$, that better fits our intuition. The notion of LCS_{k+} is applied to bioinformatics [16].

Pavetić *et al.* showed that LCS_{k+} can be computed in $O(m + n + r\log r + r\log n)$ time, where m, n are lengths of the input strings and r is the total number of matching k length substring pairs between the input strings. Their algorithm is fast on average, but in the worst case, the running time is $O(mn\log(mn))$. Independently, Benson *et al.* [2] proposed an $O(kmn)$ worst-case time algorithm for the LCS_{k+} problem.

In this paper, we first propose an algorithm to compute LCS_{k+} in $O(mn)$ worst-case time by a simple dynamic programming. Secondly, we introduce the *longest common subsequence in at least k length order-isomorphic substrings (op-LCS_{k+})* problem. Order-isomorphism is a notion of equality of two numerical strings, intensively studied in the *order-preserving matching* problem[1] [13,14]. op-LCS_{k+} is a natural definition of similarity between numerical strings, and can be used in time series data analysis. The op-LCS_{k+} problem cannot be solved as simply as the LCS_{k+} problem due to the properties of the order-isomorphism. However, we will show that the op-LCS_{k+} problem can also be solved in $O(mn)$ worst-case time by an easy-to-implement algorithm, which is one of the main contributions of this paper. Finally, we report experimental results.

2 Preliminaries

We assume that all strings are over an *alphabet* Σ. The length of a string $X = (X[1], X[2], \cdots, X[n])$ is denoted by $|X| = n$. A *substring* of X beginning at i and ending at j is denoted by $X[i:j] = (X[i], X[i+1], \cdots, X[j-1], X[j])$. We denote $X\langle i, +l\rangle = X[i : i+l-1]$ and $X\langle j, -l\rangle = X[j-l+1 : j]$. Thus $X\langle i, +l\rangle = X\langle i+l-1, -l\rangle$. We write $X[: i]$ and $X[j :]$ to denote the *prefix* $X[1 : i]$ and the *suffix* $X[j : n]$ of X, respectively. Note that $X[: 0]$ is the empty string. The reverse of a string X is denoted by X^{R}, and the operator $\cdot$ denotes the concatenation. We simply denote a string $X = (X[1], X[2], \cdots, X[n])$ as $X = X[1]X[2]\cdots X[n]$ when clear from the context.

We formally define the LCS_{k+} problem as follows.

Definition 1 (LCS_{k+} problem [2,15][2]). *Given two strings X and Y of length m and n, respectively, and an integer $k \geq 1$, we say that Z is a* common

[1] Since the problem is motivated by the order-preserving matching problem, we abbreviate it to the op-LCS_{k+} problem.

[2] The formal definition given by Pavetić *et al.* [15] contains a minor error, i.e., they do not require that each chunk is identical, while Benson *et al.* [2] and we do (confirmed by F. Pavetić, personal communication, October 2016).

subsequence in at least k length substrings *of X and Y, if there exist $i_1, \cdots, i_t$ and $j_1, \cdots, j_t$ such that $X\langle i_s, +l_s\rangle = Y\langle j_s, +l_s\rangle = Z\langle p_s, +l_s\rangle$ and $l_s \geq k$ for $1 \leq s \leq t$, and $i_s + l_s \leq i_{s+1}$, $j_s + l_s \leq j_{s+1}$ and $p_{s+1} = p_s + l_s$ for $1 \leq s < t$, $p_1 = 1$ and $|Z| = p_t + l_t - 1$. The* longest common subsequence in at least k length substrings (LCS$_{k^+}$) *problem asks for the length of an LCS$_{k^+}$ of X and Y.*

Remark that the LCS$_{1^+}$ problem is equivalent to the standard LCS problem. Without loss of generality, we assume that $n \geq m$ through the paper.

Example 1. For strings X = `acdbacbc` and Y = `aacdabca`, Z = `acdbc` is the LCS$_{2^+}$ of X and Y, since $X\langle 1, +3\rangle = Y\langle 2, +3\rangle =$ `acd` $= Z\langle 1, +3\rangle$ and $X\langle 7, +2\rangle = Y\langle 6, +2\rangle =$ `bc` $= Z\langle 4, +2\rangle$. Note that the standard LCS of X and Y is `acdabc`.

The main topic of this paper is to give an efficient algorithm for computing the longest common subsequence *under order-isomorphism*, defined below.

Definition 2 (Order-isomorphism[13,14]**).** *Two strings S and T of the same length l over an ordered alphabet are* order-isomorphic *if $S[i] \leq S[j] \iff T[i] \leq T[j]$ for any $1 \leq i, j \leq l$. We write $S \approx T$ if S is order-isomorphic to T, and $S \not\approx T$ otherwise.*

Example 2. For strings $S = (32, 40, 4, 16, 27)$, $T = (28, 32, 12, 20, 25)$ and $U = (33, 51, 10, 22, 42)$, we have $S \approx T$, $S \not\approx U$, and $T \not\approx U$.

Definition 3 (op-LCS$_{k^+}$ problem). *The* op-LCS$_{k^+}$ *problem is defined as the problem obtained from Definition 1 by replacing the matching relation $X\langle i_s, +l_s\rangle = Y\langle j_s, +l_s\rangle = Z\langle p_s, +l_s\rangle$ with order-isomorphism $X\langle i_s, +l_s\rangle \approx Y\langle j_s, +l_s\rangle \approx Z\langle p_s, +l_s\rangle$.*

Example 3. For strings $X = (14, 84, 82, 31, 74, 68, 87, 11, 20, 32)$ and $Y = (21, 64, 2, 83, 73, 51, 5, 29, 7, 71)$, $Z = (1, 3, 2, 31, 74, 68, 87)$ is an op-LCS$_{3^+}$ of X and Y since $X\langle 1, +3\rangle \approx Y\langle 3, +3\rangle \approx Z\langle 1, +3\rangle$ and $X\langle 4, +4\rangle \approx Y\langle 7, +4\rangle \approx Z\langle 4, +4\rangle$.

The op-LCS$_{k^+}$ problem does not require that $(X\langle i_1, +l_1\rangle \cdot X\langle i_2, +l_2\rangle \cdot \cdots \cdot X\langle i_t, +l_t\rangle) \approx (Y\langle j_1, +l_1\rangle \cdot Y\langle j_2, +l_2\rangle \cdot \cdots \cdot Y\langle j_t, +l_t\rangle)$. Therefore, the op-LCS$_{1^+}$ problem makes no sense. Note that the op-LCS$_{k^+}$ problem with this restriction is **NP**-hard already for $k = 1$ [3].

3 The LCS$_{k^+}$ Problem

In this section, we show that the LCS$_{k^+}$ problem can be solved in $O(mn)$ time by dynamic programming. We define $Match(i, j, l) = 1$ if $X\langle i, -l\rangle = Y\langle j, -l\rangle$, and 0 otherwise. Let $C[i, j]$ be the length of an LCS$_{k^+}$ of $X[: i]$ and $Y[: j]$, and $A_{i,j} = \{C[i - l, j - l] + l \cdot Match(i, j, l) : k \leq l \leq \min\{i, j\}\}$. Our algorithm is based on the following lemma.

Lemma 1 ([2]). *For any $k \le i \le m$ and $k \le j \le n$,*

$$C[i,j] = \max\left(\{C[i,j-1], C[i-1,j]\} \cup A_{i,j}\right), \tag{1}$$

and $C[i,j] = 0$ otherwise.

The naive dynamic programming algorithm based on Eq. (1) takes $O(m^2n)$ time, because for each i and j, the naive algorithm for computing $\max A_{i,j}$ takes $O(m)$ time assuming $n \ge m$. Therefore, we focus on how to compute $\max A_{i,j}$ in constant time for each i and j in order to solve the problem in $O(mn)$ time. It is clear that if $Match(i,j,l_1) = 0$ then $Match(i,j,l_2) = 0$ for all valid $l_2 \ge l_1$, and $C[i',j'] \ge C[i'-l', j'-l']$ for all valid i', j' and $l' > 0$. Therefore, in order to compute $\max A_{i,j}$, it suffices to compute $\max_{k \le l \le L[i,j]}\{C[i-l, j-l] + l\}$, where $L[i,j] = \max\{l : X\langle i, -l\rangle = Y\langle j, -l\rangle\}$.

We can compute $L[i,j]$ for all $0 \le i \le m$ and $0 \le j \le n$ in $O(mn)$ time by dynamic programming because the following equation clearly holds:

$$L[i,j] = \begin{cases} L[i-1,j-1] + 1 & (\text{if}\, i,j > 0 \,\text{and}\, X[i] = Y[j]) \\ 0 & (\text{otherwise}). \end{cases} \tag{2}$$

Next, we show how to compute $\max_{k \le l \le L[i,j]}\{C[i-l, j-l] + l\}$ in constant time for each i and j. Assume that the table L has already been computed. Let $M[i,j] = \max_{k \le l \le L[i,j]}\{C[i-l, j-l] + l\}$ if $L[i,j] \ge k$, and -1 otherwise.

Lemma 2. *For any $0 \le i \le m$ and $0 \le j \le n$, if $L[i,j] > k$ then $M[i,j] = \max\{M[i-1,j-1] + 1, C[i-k, j-k] + k\}$.*

Proof. Let $l = L[i,j]$. Since $L[i,j] > k$, we have $L[i-1,j-1] = l - 1 \ge k$, and $M[i-1,j-1] \ne -1$. Therefore, $M[i-1,j-1] = \max_{k \le l' \le l-1}\{C[i-1-l', j-1-l'] + l'\} = \max_{k+1 \le l' \le l}\{C[i-l', j-l'] + l'\} - 1$. Hence, $M[i,j] = \max_{k \le l' \le l}\{C[i-l', i-l'] + l'\} = \max\{M[i-1,j-1]+1, C[i-k,j-k]+k\}$. □

By Lemma 2 and the definition of $M[i,j]$, we have

$$M[i,j] = \begin{cases} \max\{M[i-1,j-1] + 1, C[i-k,j-k] + k\} & (\text{if}\, L[i,j] > k) \\ C[i-k,j-k] + k & (\text{if}\, L[i,j] = k) \\ -1 & (\text{otherwise}). \end{cases} \tag{3}$$

Equation (3) shows that each $M[i,j]$ can be computed in constant time if $L[i,j]$, $M[i-1,j-1]$, and $C[i-k,j-k]$ have already been computed.

We can fill in tables C, L and M of size $(m+1) \times (n+1)$ based on Eqs. (1), (2) and (3) in $O(mn)$ time by dynamic programming. An example of computing LCS_{3+} is shown in Fig. 1(a). We note that LCS_{k+} itself (not only its length) can be extracted from the table C in $O(m+n)$ time, by tracing back in the same way as the standard dynamic programming algorithm for the standard LCS problem. Our algorithm requires $O(mn)$ space since we use three tables of size $(m+1) \times (n+1)$. Note that if we want to compute only the length of an LCS_{k+}, the space complexity can be easily reduced to $O(km)$. Hence, we get the following theorem.

			a	b	c	b	c	d	e	f
		0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
a	1	0	0	0	0	0	0	0	0	0
b	2	0	0	0	0	0	0	0	0	0
c	3	0	0	0	3	3	3	3	3	3
d	4	0	0	0	3	3	3	3	3	3
e	5	0	0	0	3	3	3	3	4	4
f	6	0	0	0	3	3	3	3	4	6

(a) Table C for the LCS_{3+} problem

			4	0	9	6	2	0	3	1
		0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0	0
1	2	0	0	2	2	2	2	2	2	2
3	3	0	0	2	2	2	2	2	2	2
8	4	0	0	2	2	2	2	2	4	4
7	5	0	0	2	2	4	4	4	4	5
2	6	0	0	2	2	4	5	5	5	5

(b) Table C for the op-LCS_{2+} problem

Fig. 1. Examples of computing LCS_{3+} and op-LCS_{2+}

Theorem 1. *The LCS_{k+} problem can be solved in $O(mn)$ time and $O(km)$ space.*

4 The op-LCS$_{k+}$ Problem

In this section, we show that the op-LCS_{k+} problem can be solved in $O(mn)$ time as well as the LCS_{k+} problem. We redefine $C[i,j]$ to be the length of an op-LCS_{k+} of $X[: i]$ and $Y[: j]$, and $Match(i,j,l) = 1$ if $X\langle i,-l\rangle \approx Y\langle j,-l\rangle$, and 0 otherwise. It is easy to prove that Eq. (1) also holds with respect to the order-isomorphism. However, the op-LCS_{k+} problem cannot be solved as simply as the LCS_{k+} problem because Eqs. (2) and (3) do not hold with respect to the order-isomorphism, as follows. For two strings A, B of length l such that $A \approx B$, and two characters a, b such that $A \cdot a \not\approx B \cdot b$, the statement "$(A \cdot a)[i :] \not\approx (B \cdot b)[i :]$ for all $1 \le i \le l+1$" is not always true. For example, for strings $A = (32, 40, 4, 16, 27)$, $B = (28, 32, 12, 20, 25)$, $A' = A \cdot (41)$ and $B' = B \cdot (26)$, we have $A \approx B$, $A' \not\approx B'$, and $A'[3 :] \approx B'[3 :]$. Moreover, for $A'' = A \cdot (15)$ and $B'' = B \cdot (22)$, we have $A''[5 :] \approx B''[5 :]$. These examples show that Eqs. (2) and (3) do not hold with respect to the order-isomorphism. Therefore, we must find another way to compute $\max_{k \le l' \le l}\{C[i-l', j-l'] + l'\}$, where $l = \max\{l' : X\langle i,-l'\rangle \approx Y\langle j,-l'\rangle\}$ in constant time.

First, we consider how to find $\max\{l : X\langle i,-l\rangle \approx Y\langle j,-l\rangle\}$ in constant time. We define the *order-preserving longest common extension (op-LCE)* query on strings S_1 and S_2 as follows.

Definition 4 (op-LCE query). *Given a pair (S_1, S_2) of strings, an* op-LCE query *is a pair of indices i_1 and i_2 of S_1 and S_2, respectively, which asks $opLCE_{S_1,S_2}[i_1, i_2] = \max\{l : S_1\langle i_1, +l\rangle \approx S_2\langle i_2, +l\rangle\}$.*

Since $\max\{l : X\langle i,-l\rangle \approx Y\langle j,-l\rangle\} = opLCE_{X^{\mathrm{R}},Y^{\mathrm{R}}}[m-i+1, n-j+1]$, we can find $\max\{l : X\langle i,-l\rangle \approx Y\langle j,-l\rangle\}$ by using op-LCE queries on X^{R} and Y^{R}. Therefore, we focus on how to answer op-LCE queries on S_1 and S_2 in constant time with at most $O(|S_1||S_2|)$ time preprocessing. Hereafter we write $\mathit{opLCE}[i_1, i_2]$ for $\mathit{opLCE}_{S_1,S_2}[i_1, i_2]$ fixing two strings S_1 and S_2.

If S_1 and S_2 are strings over a polynomially-bounded integer alphabet $\{1, \cdots, (|S_1| + |S_2|)^c\}$ for an integer constant c, op-LCE queries can be answered in $O(1)$ time and $O(|S_1| + |S_2|)$ space with $O((|S_1| + |S_2|)\log^2 \log(|S_1| + |S_2|)/\log\log\log(|S_1| + |S_2|))$ time preprocessing, by using the *incomplete generalized op-suffix-tree* [6] of S_1 and S_2 and finding the *lowest common ancestor (LCA)* [1] in the op-suffix-tree. The proof is similar to that for LCE queries in the standard setting [10].

However, implementing the incomplete generalized op-suffix-tree is quite difficult. Therefore, we introduce another much simpler method to answer op-LCE queries in $O(1)$ time with $O(|S_1||S_2|)$ time preprocessing. In a preprocessing step, our algorithm fills in the table $\mathit{opLCE}[i_1, i_2]$ for all $1 \leq i_1 \leq |S_1|$ and $1 \leq i_2 \leq |S_2|$ in $O(|S_1||S_2|)$ time. Then, we can answer op-LCE queries in constant time.

In the preprocessing step, we use the *Z-algorithm* [10,11] that calculates the following table efficiently.

Definition 5 (Z-table). *The* Z-table Z_S *of a string* S *is defined by* $Z_S[i] = \max\{l : S\langle 1, +l\rangle \approx S\langle i, +l\rangle\}$ *for each* $1 \leq i \leq |S|$.

By definition, we have

$$\mathit{opLCE}[i_1, i_2] = \min\bigl\{Z_{(S_1 \cdot S_2)_{i_1}}[|S_1| - i_1 + i_2 + 1],\ |S_1| - i_1 + 1\bigr\}. \qquad (4)$$

If we use the Z-algorithm and Eq. (4) naively, it takes $O((|S_1| + |S_2|)^2 \log(|S_1| + |S_2|))$ time to compute $\mathit{opLCE}[i_1, i_2]$ for all $1 \leq i_1 \leq |S_1|$ and $1 \leq i_2 \leq |S_2|$, because the Z-algorithm requires $O(|S| \log |S|)$ time to compute Z_S for a string S. We extend the Z-algorithm to compute Z_{S_i} for *all* $1 \leq i \leq |S|$ totally in $O(|S|^2)$ time.

In order to verify the order-isomorphism in constant time with preprocessing, Hasan *et al.* [11] used tables called Prev_S and Next_S. For a string S where all the characters are distinct[3], Prev_S and Next_S are defined as

$$\mathit{Prev}_S[i] = j \text{ if there exists } j = \operatorname*{argmax}_{1 \leq k < i}\{S[k] : S[k] < S[i]\}, \text{ and } \quad -\infty \text{ otherwise}$$

$$\mathit{Next}_S[i] = j \text{ if there exists } j = \operatorname*{argmin}_{1 \leq k < i}\{S[k] : S[k] > S[i]\}, \text{ and } \quad \infty \text{ otherwise}$$

for all $1 \leq i \leq |S|$. Their algorithm requires $O(|S| \log |S|)$ time to compute the tables Prev_S and Next_S, and all operations except computing the tables take only $O(|S|)$ time. Therefore, if we can compute tables Prev_{S_i} and Next_{S_i} for each $1 \leq i \leq |S|$ in $O(|S|)$ time with $O(|S| \log |S|)$ time preprocessing, Z_{S_i} for

[3] Hasan *et al.* [11] assume that characters in a string are distinct. If the assumption is false, use Lemma 4 in [4] in order to verify the order-isomorphism, that is, modify line 10 of Algorithm 4 in [11] and line 7 and 12 in Algorithm 1. Note that *Prev* and *Next* are denoted as *LMax* and *LMin* in [4], respectively, with slight differences.

Algorithm 1. The algorithm for computing op-LCE queries

```
Function preprocess($S$, $i$, $S'$, $S''$)
    Let $s$ and $t$ be empty stacks that support push, top, and pop operations;
    Let $Prev$ and $Next$ be tables of size $|S| - i + 1$;
    for $j \leftarrow 1$ to $|S|$ do
        if $S'[j] \geq i$ then
            while $s \neq \emptyset$ and $s$.top() $> S'[j]$ do $s$.pop();
            if $s = \emptyset$ then $Prev[S'[j] - i + 1] \leftarrow -\infty$;
            else $Prev[S'[j] - i + 1] \leftarrow s$.top() $- i + 1$;
            $s$.push($S'[j]$);
        if $S''[j] \geq i$ then
            while $t \neq \emptyset$ and $t$.top() $> S''[j]$ do $t$.pop();
            if $t = \emptyset$ then $Next[S''[j] - i + 1] \leftarrow \infty$;
            else $Next[S''[j] - i + 1] \leftarrow t$.top() $- i + 1$;
            $t$.push($S''[j]$);
    return $(Prev, Next)$;
Function Z-function($S, i_1, S', S''$)
    $(Prev, Next) \leftarrow$ preprocess($S, i_1, S', S''$); $S \leftarrow S[i_1:]$;
    Do the same operations described in line 3-17 of Algorithm 4 in [11];
    return $Z$;
Function preprocess-opLCE($S_1, S_2$)
    Let $opLCE$ be a table of size $|S_1| \times |S_2|$; $S \leftarrow S_1 \cdot S_2$;
    Let $S'$ and $S''$ be stably sorted positions of $S$ with respect to their elements
      in ascending and descending order, respectively;
    for $i_1 \leftarrow 1$ to $|S_1|$ do
        $Z \leftarrow$ Z-function($S, i_1, S', S''$);
        for $i_2 \leftarrow 1$ to $|S_2|$ do
            $opLCE[i_1, i_2] \leftarrow \min\left\{Z[|S_1| - i_1 + i_2 + 1], |S_1| - i_1 + 1\right\}$;
    return $opLCE$;
```

all $1 \leq i \leq |S|$ can be computed in $O(|S|^2)$ time. We also assume that all the characters in S are distinct (see footnote 3).

In order to compute the tables $Prev_{Si}$ and $Next_{Si}$, we modify a sort-based algorithm presented in Lemma 1 in [14] instead of the algorithm in [11] that uses a balanced binary search tree. First, for computing $Prev_S$ (resp. $Next_S$), we stably sort positions of S with respect to their elements in ascending (resp. descending) order. We can compute $Prev_{Si}$ and $Next_{Si}$ for each $1 \leq i \leq |S|$ in $O(|S|)$ time by using the sorted tables and the stack-based algorithm presented in [14], ignoring all elements of the sorted tables less than i.

Algorithm 1 shows the pseudocode of the op-LCE algorithm based on the Z-algorithm. The push(x) operation inserts x on the top of the stack, top() returns the top element in the stack, and pop() removes it. Algorithm 1 takes $O(|S_1||S_2|)$ time as discussed above. The total space complexity is $O(|S_1||S_2|)$

Algorithm 2. The algorithm for the op-LCS$_{k+}$ problem

```
Input: A string $X$ of length $m$, a string $Y$ of length $n$, and an integer $k$
Output: The length of an op-LCS$_{k+}$ between $X$ and $Y$
1  Let $C$ be a table of size $(m+1) \times (n+1)$ initialized by 0;
2  Let $R_i$ for $-n+k \leq i \leq m-k$ be semi-dynamic RMQ data structures;
3  $opLCE \leftarrow$ preprocess-opLCE($X^{\mathrm{R}}, Y^{\mathrm{R}}$);
4  for $i \leftarrow 0$ to $m-k$ do
5      if $i < k$ then $n' \leftarrow n-k$;
6      else $n' \leftarrow k-1$;
7      for $j \leftarrow 0$ to $n'$ do $R_{i-j}$.prepend($C[i,j] - \min\{i,j\}$);
8  for $i \leftarrow k$ to $m$ do
9      for $j \leftarrow k$ to $n$ do
10         $l \leftarrow opLCE[m-i+1, n-j+1]$;
11         if $l \geq k$ then $M \leftarrow R_{i-j}$.rmq($k, l$) $+ \min\{i,j\}$;
12         else $M \leftarrow 0$;
13         $C[i,j] \leftarrow \max\{C[i,j-1], C[i-1,j], M\}$;
14         $R_{i-j}$.prepend($C[i,j] - \min\{i,j\}$);
15 return $C[m,n]$;
```

because the Z-algorithm requires linear space [11], and the table *opLCE* needs $O(|S_1||S_2|)$ space. Hence, we have the following lemma.

Lemma 3. *op-LCE queries on S_1 and S_2 can be answered in $O(1)$ time and $O(|S_1||S_2|)$ space with $O(|S_1||S_2|)$ time preprocessing.*

Let $\texttt{opLCE}(i,j)$ be the answer to the op-LCE query on X^{R} and Y^{R} with respect to the index pair (i,j). We consider how to find the maximum value of $C[i-l, j-l]+l$ for $k \leq l \leq \texttt{opLCE}(m-i+1, n-j+1)$ in constant time. We use a *semi-dynamic range maximum query (RMQ)* data structure that maintains a table A and supports the following two operations:

- $\texttt{prepend}(x)$: add x to the beginning of A in $O(1)$ amortized time.
- $\texttt{rmq}(i_1, i_2)$: return the maximum value of $A[i_1 : i_2]$ in $O(1)$ time.

The details of the semi-dynamic RMQ data structure will be given in Sect. 5.

By using the semi-dynamic RMQ data structures and the following obvious lemma, we can find $\max_{k \leq l \leq \texttt{opLCE}(m-i+1, n-j+1)}\{C[i-l, j-l]+l\}$ for all $1 \leq i \leq m$ and $1 \leq j \leq n$ in totally $O(mn)$ time.

Lemma 4. *We may assume that $i \geq j$ without loss of generality. Let $A[l] = C[i-l, j-l]+l$ and $A'[l] = C[i-l, j-l]-j+l$ for each $1 \leq l \leq j$. For any $1 \leq i_1, i_2 \leq |A|$, we have $\max_{i_1 \leq l \leq i_2} A[l] = (\max_{i_1 \leq l \leq i_2} A'[l]) + j$ and $\operatorname{argmax}_{i_1 \leq l \leq i_2} A[l] = \operatorname{argmax}_{i_1 \leq l \leq i_2} A'[l]$.*

Algorithm 2 shows our algorithm to compute op-LCS$_{k+}$. An example of computing op-LCS$_{2+}$ is shown in Fig. 1(b). As discussed above, the algorithm runs

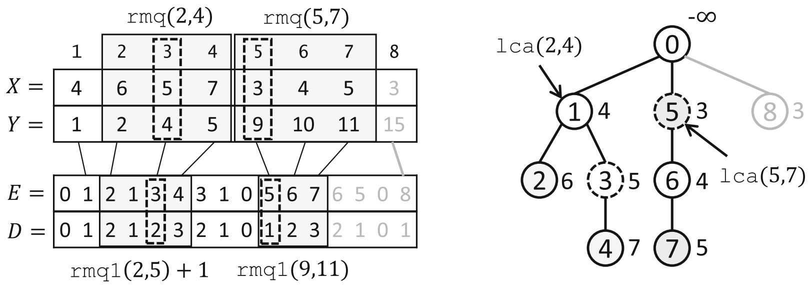


Fig. 2. An example of searching for the RMQ by using a 2d-Min-Heap and the ±1RMQ algorithm [1]. The tree shows the 2d-Min-Heap of $X = (4, 6, 5, 7, 3, 4, 5, 3)$ represented by arrays E and D. The gray node 8 in the tree and gray numbers in the table are added when the last character $X[8] = 3$ is processed. The boxes with the dashed lines show the answers of RMQs $\texttt{rmq}(2, 4)$ and $\texttt{rmq}(5, 7)$.

in $O(mn)$ time. Each semi-dynamic RMQ data structure requires linear space and a total of $O(mn)$ elements are maintained by the semi-dynamic RMQ data structures. Therefore, the total space of semi-dynamic RMQ data structures is $O(mn)$. Consequently, the total space complexity is $O(mn)$. Hence, we have the following theorem.

Theorem 2. *The op-LCS$_{k^+}$ problem can be solved in $O(mn)$ time and space.*

5 The Semi-dynamic Range Minimum/Maximum Query

In this section we will describe the algorithm that solves the semi-dynamic RMQ problem with $O(1)$ query time and amortized $O(1)$ prepend time. To simplify the algorithm, we consider the prepend operation as appending a character into the end of array. In order to solve this problem, Fischer [8] proposed an algorithm that uses a 2d-Min-Heap [9] and dynamic LCAs [5]. However, the algorithm for dynamic LCAs is very complex to implement. Therefore, we propose a simple semi-dynamic RMQ algorithm that can be implemented easily if the number of characters to be appended is known beforehand. This algorithm uses a 2d-Min-Heap and the ±1RMQ algorithm proposed by Bender and Farach-Colton [1].

Let X be a string of length n and let $X[0] = -\infty$. The 2d-Min-Heap H of X is an ordered tree of $n+1$ nodes $\{0, 1, \cdots, n\}$, where 0 is the root node, and the parent node of node $i > 0$ is $\max\{j < i : X[j] < X[i]\}$. Moreover, the order of the children is chosen so that they increase from left to right (see Fig. 2 for instance). Note that the vertices are inevitably aligned in preorder. Actually, the tree H is represented by arrays E and D that store the sequences of nodes and their depths visited in an Euler tour of H, respectively. In addition, let Y be an array defined as $Y[i] = \min\{j : E[j] = i\}$ for each $1 \le i \le n$.

For two positions $1 \le i_1 \le i_2 \le n$ in X, $\texttt{rmq}(i_1, i_2)$ can be calculated by finding $\texttt{lca}(i_1, i_2)$, the LCA of the nodes i_1 and i_2 in H. If $\texttt{lca}(i_1, i_2) = i_1$, then $\texttt{rmq}(i_1, i_2) = i_1$. Otherwise, $\texttt{rmq}(i_1, i_2) = i_3$ such that i_3 is a child of

$\texttt{lca}(i_1, i_2)$ and an ancestor of i_2. The $\texttt{lca}(i_1, i_2)$ can be computed by performing the ±1RMQ query $\texttt{rmq1}(Y[i_1], Y[i_2])$ on D, because $D[j+1] - D[j] = \pm 1$ for every j. It is known that ±1RMQs can be answered in $O(1)$ time with $O(n)$ time preprocessing [1]. Therefore, we can calculate $\texttt{rmq}(i_1, i_2)$ as follows,

$$\texttt{rmq}(i_1, i_2) = \begin{cases} E[\texttt{rmq1}(Y[i_1], Y[i_2])] & (\text{if } E[\texttt{rmq1}(Y[i_1], Y[i_2])] = i_1) \\ E[\texttt{rmq1}(Y[i_1], Y[i_2]) + 1] & (\text{otherwise}). \end{cases}$$

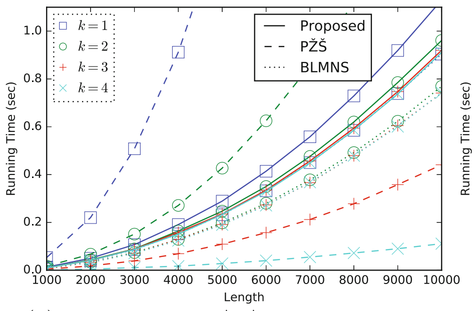
Figure 2 shows an example of calculating the RMQ. From the property of a 2d-Min-Heap, arrays E and D are always extended to the end when a new character is appended. Moreover, the ±1RMQ algorithm can be performed semi dynamically if the size of sequences is known beforehand, or by increasing the arrays size exponentially. Therefore, this algorithm can be performed online and can solve the semi-dynamic RMQ problem, as we intended.

6 Experimental Results

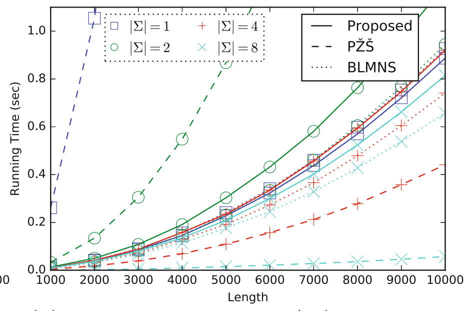
In this section, we present experimental results. We compare the running time of the proposed algorithm in Sect. 3 to the existing algorithms [2,15]. Furthermore, we show the running time of Algorithm 2. We used a machine running Ubuntu 14.04 with Core i7 4820 K and 64 GB RAM. We implemented all algorithms in C++ and compiled with gcc 4.8.4 with `-O2` optimization. We used an implementation of the algorithm proposed by Pavetić *et al.*, available at github.com/fpavetic/lcskpp. We denote the algorithm proposed by Pavetić *et al.* [15] and the algorithm proposed by Benson *et al.* [2] as PŽŠ and BLMNS, respectively.

We tested the proposed algorithm in Sect. 3, PŽŠ, and BLMNS in the following three conditions: (1) random strings over an alphabet of size $|\Sigma| = 4$ with $n = m = 1000, 2000, \cdots, 10000$ and $k = 1, 2, 3, 4$ (2) random strings over alphabets of size $|\Sigma| = 1, 2, 4, 8$ with $n = m = 1000, 2000, \cdots, 10000$ and $k = 3$ (3) DNA sequences that are available at www.ncbi.nlm.nih.gov/nuccore/346214858 and www.ncbi.nlm.nih.gov/nuccore/U38845.1, with $k = 1, 2, 3, 4, 5$. The experimental results under the conditions (1), (2) and (3) are shown in Fig. 3(a), (b), and (c), respectively.

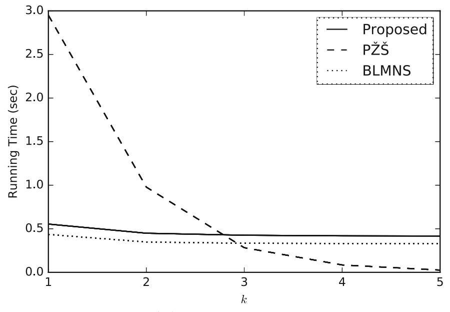
The proposed algorithm in Sect. 3 runs faster than PŽŠ for small k or small alphabets. This is due to that PŽŠ strongly depends on the total number of matching k length substring pairs between input strings, and for small k or small alphabets there are many matching pairs. In general BLMNS runs faster than ours. The proposed algorithm runs a little faster for small k or small alphabets, except $|\Sigma| = 1$. We think that this is because for small k or small alphabets the probability that $L[i, j] \geq k$ is high, and this implies that we need more operations to compute $M[i, j]$ by definition. In Fig. 3(b), it is observed that the proposed algorithm with $|\Sigma| = 1$ runs faster than with $|\Sigma| = 2$. Since $|\Sigma| = 1$ implies that $X = Y$ if X and Y have the same length, $L[i, j] > k$ almost always holds, which leads to reduce branch mispredictions and speed up execution.



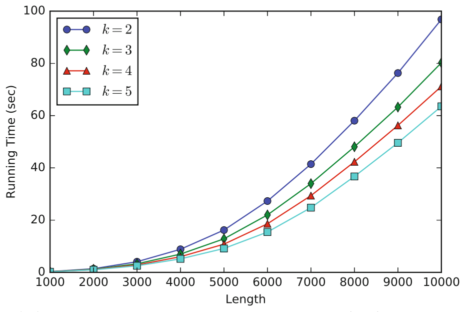
(a) Random data; $|\Sigma| = 4; k = 1, 2, 3, 4$



(b) Random data; $k = 3; |\Sigma| = 1, 2, 4, 8$



(c) DNA data



(d) Algorithm 2, random data, $|\Sigma| = 100$

Fig. 3. Running times of the proposed algorithm in Sect. 3, PŽŠ, and BLMNS (Fig. 3(a), (b) and (c)), and Algorithm 2 (Fig. 3(d)). In Fig. 3(a), (b), and (c), the line styles denote algorithms. The line markers in Fig. 3(a) and (b) represent the parameter k and the alphabet size, respectively.

We show the running time of Algorithm 2 in Fig. 3(d). We tested Algorithm 2 on random strings over $\Sigma = \{1, 2, \cdots, 100\}$ with $n = m = 1000, 2000, \cdots, 10000$ and $k = 2, 3, 4, 5$. It is observed that the algorithm runs faster as the parameter k is smaller. We suppose that the hidden constant of the RMQ data structure described in Sect. 5 is large. Therefore, the running time of Algorithm 2 depends on the number of times the `rmq` operation is called, and for small k the number of them increases since the probability that $l \geq k$ is high.

7 Conclusion

We showed that both the LCS$_{k+}$ problem and the op-LCS$_{k+}$ problem can be solved in $O(mn)$ time. Our result on the LCS$_{k+}$ problem gives a better worst-case running time than previous algorithms [2,15], while the experimental results showed that the previous algorithms run faster than ours on average. Although the op-LCS$_{k+}$ problem looks much more challenging than the LCS$_{k+}$, since the former cannot be solved by a simple dynamic programming due to the properties of order-isomorphisms, the proposed algorithm achieves the same time complexity as the one for the LCS$_{k+}$.

Acknowledgements. This work was funded by ImPACT Program of Council for Science, Technology and Innovation (Cabinet Office, Government of Japan), Tohoku University Division for Interdisciplinary Advance Research and Education, and JSPS KAKENHI Grant Numbers JP24106010, JP16H02783, JP26280003.

References

1. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000). doi:10.1007/10719839_9
2. Benson, G., Levy, A., Maimoni, S., Noifeld, D., Shalom, B.: LCSk: a refined similarity measure. Theor. Comput. Sci. **638**, 11–26 (2016)
3. Bouvel, M., Rossin, D., Vialette, S.: Longest common separable pattern among permutations. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 316–327. Springer, Heidelberg (2007). doi:10.1007/978-3-540-73437-6_32
4. Cho, S., Na, J.C., Park, K., Sim, J.S.: A fast algorithm for order-preserving pattern matching. Inf. Process. Lett. **115**(2), 397–402 (2015)
5. Cole, R., Hariharan, R.: Dynamic LCA queries on trees. SIAM J. Comput. **34**(4), 894–923 (2005)
6. Crochemore, M., Iliopoulos, C.S., Kociumaka, T., Kubica, M., Langiu, A., Pissis, S.P., Radoszewski, J., Rytter, W., Waleń, T.: Order-preserving indexing. Theor. Comput. Sci. **638**, 122–135 (2016)
7. Deorowicz, S., Grabowski, S.: Efficient algorithms for the longest common subsequence in k-length substrings. Inf. Process. Lett. **114**(11), 634–638 (2014)
8. Fischer, J.: Inducing the LCP-array. In: Dehne, F., Iacono, J., Sack, J.-R. (eds.) WADS 2011. LNCS, vol. 6844, pp. 374–385. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22300-6_32
9. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. **40**(2), 465–492 (2011)
10. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York (1997)
11. Hasan, M.M., Islam, A., Rahman, M.S., Rahman, M.: Order preserving pattern matching revisited. Pattern Recogn. Lett. **55**, 15–21 (2015)
12. Khan, R., Ahmad, M., Zakarya, M.: Longest common subsequence based algorithm for measuring similarity between time series: a new approach. World Appl. Sci. J. **24**(9), 1192–1198 (2013)
13. Kim, J., Eades, P., Fleischer, R., Hong, S.H., Iliopoulos, C.S., Park, K., Puglisi, S.J., Tokuyama, T.: Order-preserving matching. Theor. Comput. Sci. **525**(13), 68–79 (2014)
14. Kubica, M., Kulczynski, T., Radoszewski, J., Rytter, W., Walen, T.: A linear time algorithm for consecutive permutation pattern matching. Inf. Process. Lett. **113**(12), 430–433 (2013)
15. Pavetić, F., Žužić, G., Šikić, M.: *LCSk*++: practical similarity metric for long strings (2014). CoRR 1407.2407
16. Sović, I., Šikić, M., Wilm, A., Fenlon, S.N., Chen, S., Nagarajan, N.: Fast and sensitive mapping of nanopore sequencing reads with GraphMap. Nat. Commun. **7**, Article No. 11307 (2016). doi:10.1038/ncomms11307