Theoretical Computer Science 687 (2017) 79–92

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

A space efficient algorithm for the longest common subsequence in k-length substrings

Daxin Zhu [a], Lei Wang [b], Tinran Wang [c], Xiaodong Wang [d,*]

[a] Quanzhou Normal University, Quanzhou, China
[b] Facebook, 1 Hacker Way, Menlo Park, CA 94052, USA
[c] School of Mathematical Sciences, Peking University, Beijing, China
[d] Fujian University of Technology, Fuzhou, China

A R T I C L E   I N F O

Article history:
Received 9 February 2017
Received in revised form 17 May 2017
Accepted 19 May 2017
Available online 26 May 2017
Communicated by R. Giancarlo

Keywords:
Longest common subsequence
Similarity of strings
Edit distance
Dynamic programming

A B S T R A C T

Two space efficient algorithms to solve the $LCSk$ problem and $LCS_{\geq}k$ problem are presented in this paper. The algorithms improve the time and space complexities of the algorithms of Benson et al. [4]. The space cost of the first algorithm to solve the $LCSk$ problem is reduced from $O(n^2)$ to $O(kn)$, if the size of the two input sequences are both n. The time and space costs of the second algorithm to solve the $LCS_{\geq}k$ problem are both improved. The time cost is reduced from $O(kn^2)$ to $O(n^2)$, and the space cost is reduced from $O(n^2)$ to $O(kn)$. In the case of $k = O(1)$, the two algorithms are both linear space algorithms.

1. Introduction

The longest common subsequence (LCS) problem is a classic problem in computer science [8,17]. Given two sequences A and B, the longest common subsequence (LCS) problem is to find a subsequence of A and B whose length is the longest among all common subsequences of the two given sequences. The problem has numerous applications in many apparently unrelated fields ranging from file comparison, pattern matching and computational biology [11]. The LCS problem has many variants, such as LCS alignment [13–15], constrained LCS [2,5,16,18,19], weighted LCS [1], restricted LCS [7] and LCS approximation [12].

In past years, many related sequence similarity problems, often motivated by computational biology, have also been studied. One of them, proposed very recently by Benson et al. [3,4], is the longest common subsequence in k-length substrings problem, in which the common subsequence is required to consist of k or at least k length substrings.

The $LCSk$ problem can be characterized as follows.

Definition 1. Given two sequences $A = a_1a_2 \cdots a_n$ and $B = b_1b_2 \cdots b_m$, and an integer k, the $LCSk$ problem is to find the maximal length l such that there are l substrings, $a_{i_1} \cdots a_{i_1+k-1}, \cdots, a_{i_l} \cdots a_{i_l+k-1}$, identical to $b_{j_1} \cdots b_{j_1+k-1}, \cdots, b_{j_l} \cdots b_{j_l+k-1}$ where $\{a_{i_t}\}$ and $\{b_{j_t}\}$ are in increasing order for $1 \leq t \leq l$ and any two k-length substrings in the same sequence, do not overlap.

* Corresponding author.
E-mail address: wangxd139@139.com (X. Wang).

http://dx.doi.org/10.1016/j.tcs.2017.05.015
0304-3975/© 2017 Elsevier B.V. All rights reserved.

A similar problem is the LCS at least k problem ($LCS_{\geq}k$). In this problem, the demand of matching substrings of length exactly k is relaxed to the length of the matched substrings to be at least k. The length of the common substrings is further limited by $2k-1$ since a longer common substring contains two substrings each of length k or more.

The $LCS_{\geq}k$ problem can be defined as follows.

Definition 2. Given two sequences $A=a_1a_2\cdots a_n$ and $B=b_1b_2\cdots b_m$, and an integer k, the $LCS_{\geq}k$ problem is to find substrings with maximal total length such that $a_{i_p}\cdots a_{i_p+k+t}$ is identical to $b_{j_p}\cdots b_{j_p+k+t}$ for $-1\leq t\leq k-2$ where $\{a_{i_t}\}$ and $\{b_{j_t}\}$ are in increasing order for $1\leq t\leq l$ and any two substrings in the same sequence, do not overlap.

In the case of $n=m$, Benson et al. [3] presented a dynamic programming algorithm to solve the $LCSk$ problem using $O(kn^2)$ time and $O(n^2)$ space. In the case of $k=O(1)$, the time complexity of the algorithm becomes $O(n^2)$. For unbounded k, the time complexity was further improved from $O(kn^2)$ to $O(n^2)$ [4,6]. If only the length of $LCSk$ has to be computed, the space cost of their algorithm can be reduced to $O(kn)$, but if an $LCSk$ has to be constructed, the whole table is needed in their algorithm, implying $O(n^2)$ space requirement. Applying the sparse dynamic programming paradigm of Hunt and Szymanski [10], Deorowicz et al. [6] presented an $O(n+r\log l)$ time and $O(r)$ space algorithm, where r is the number of matches, $l\leq n/k$ is the solution length. If the number of matches in the dynamic programming matrix is large, an $O(n^2/k+n(k\log n)^{2/3})$ and $O(nl)$ space algorithm was also presented in [6] by using the observation that matches forming a longest common subsequence must be separated with gaps of size at least k. Its variant based on the van Emde Boas tree was also briefly discussed. Finally, a tabulation-based algorithm was presented, using $O(n^2/\log n)$ time and $O(n^2/\log n)$ space.

For the $LCS_{\geq}k$ problem, Benson et al. [4] presented a first dynamic programming algorithm to solve the problem using $O(kn^2)$ time and $O(n^2)$ space. If only the length of $LCS_{\geq}k$ has to be computed, the space cost of their algorithm can be reduced to $O(kn)$, but if an $LCS_{\geq}k$ has to be constructed, their algorithm requires $O(n^2)$ space.

Very recently, Ueki et al. presented a similar dynamic programming algorithm to solve the problem [16]. The algorithm was described very concisely in their paper since the main topic in their paper is to find the LCS in at least k length order-isomorphic substrings problem. Their algorithm composes of two parts. In the first part, the longest common suffix problem for the same input strings is solved. Then, in the second part of the algorithm, a dynamic programming formula can be established by utilizing the solution array L obtained in the first part. If the sizes of the two input strings are m and n respectively, their algorithm requires $O(mn)$ time and $O(mn)$ space since three tables of size $(m+1)(n+1)$ are used. If only the length of an $LCS_{\geq}k$ is required, the space complexity can be easily reduced to $O(kn)$. The algorithm presented in this paper for the same problem requires also $O(mn)$ time, but uses only $O(kn)$ space.

In this paper, we focus on the space efficient algorithms to solve the $LCSk$ problem and $LCS_{\geq}k$ problem. We present two new algorithms to solve the problems. The first algorithm is a dynamic programming algorithm to solve the $LCSk$ problem, using $O(mn)$ time and $O(kn)$ space, if the sizes of the input sequences are n and m respectively. In the case of $k=O(1)$, the algorithm is a linear space algorithm.

The second algorithm is an improved algorithm of Benson et al. to solve the $LCS_{\geq}k$ problem. The time complexity is reduced from $O(kmn)$ to $O(mn)$, and the space complexity is reduced from $O(mn)$ to $O(kn)$. In the case of $k=O(1)$, the algorithm is also a linear space algorithm.

The organization of the paper is as follows.

In the following 3 sections, we describe our improved algorithms of Benson et al to solve the $LCSk$ and $LCS_{\geq}k$ problems. In Section 2, we present an $O(kn)$ space algorithm for solving the $LCSk$ problem. In Section 3, the time and space costs of the algorithm of Benson et al to solve the $LCS_{\geq}k$ problem are reduced to $O(mn)$ and $O(kn)$. Some concluding remarks are placed in Section 4.

2. An $O(kn)$ space algorithm for solving the $LCSk$ problem

2.1. The description of the algorithm

As stated in [8], $LCSk$ can be solved by using a dynamic programming algorithm. Let $d(i,j)$ denote the length of the longest match between the prefixes of $A[1:i]=a_1a_2\cdots a_i$ and $B[1:j]=b_1b_2\cdots b_j$. Then, $d(i,j)$ can be computed recursively as follows.

$$d(i,j)=\begin{cases}1+d(i-1,j-1) & \text{if } a_i=b_j,\\ 0 & \text{otherwise}\end{cases} \tag{1}$$

Let $f(i,j)$ denote the number of k matchings in the longest common subsequence, consisting of k matchings in the prefixes $A[1:i]$ and $B[1:j]$. Then, $f(i,j)$ can be computed recursively as follows.

$$f(i,j)=\max\begin{cases}f(i-1,j)\\ f(i,j-1)\\ f(i-k,j-k)+\delta(d(i,j))\end{cases} \tag{2}$$

Where, δ is a simple piecewise linear function defined by:

$$\delta(i) = \begin{cases} 1 & \text{if } i \geq k, \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Based on the formula (2), the table $f(i,j)$ for the given input sequences $A = a_1a_2\cdots a_n$ and $B = b_1b_2\cdots b_m$ of size n and m respectively, can be computed in $O(mn)$ time and $O(mn)$ space by a standard dynamic programming algorithm.

Algorithm 1: *LCSk*.

```
Input: $A, B$
Output: $f(i,j)$, the number of $k$ matchings in the longest common subsequence of $A$ and $B$
1 for i=1 to n do
2     for j=1 to m do
3         if $a_i = b_j$ then $d(i,j) \leftarrow 1 + d(i-1, j-1)$;
4         $f(i,j) \leftarrow \max\{f(i-1,j), f(i,j-1), f(i-k,j-k) + \delta(d(i,j))\}$;
5     end
6 end
7 return $f(n,m)$
```

It is clear that the time and space complexities of the algorithm are both $O(mn)$.

When computing a particular row of the dynamic programming table, no rows before the previous k rows are required. Thus only $k+1$ rows have to be kept in memory at a time. Without loss of generality, we can assume $m = O(n)$ in the following discussion. Thus, we need only $(k+1)m = O(kn)$ entries to compute the table.

If a longest common subsequence in k-length substrings has to be constructed, not just its length, then the information provided by the $(k+1)m$ entries is not enough for this purpose. The $O(kn)$ space algorithm proposed in this section is also based on Hirschberg's divide-and-conquer method of solving the LCS problem in linear space [9]. In order to use the divide-and-conquer method, we need to extend the definition of $LCSk$ to a more general definition as follows.

Definition 3. For the two substrings $A[i_0 : i_1] = a_{i_0}a_{i_0+1}\cdots a_{i_1}$ and $B[j_0 : j_1] = b_{j_0}b_{j_0+1}\cdots b_{j_1}$, $1 \leq i_0 \leq i_1 \leq n, 1 \leq j_0 \leq j_1 \leq m$, the set of all $LCSk$s of $A[i_0 : i_1]$ and $B[j_0 : j_1]$ is denoted by $LCSk(i_0, i_1, j_0, j_1)$. The length of an $LCSk$ in $LCSk(i_0, i_1, j_0, j_1)$ is denoted by $f(i_0, i_1, j_0, j_1)$.

Similarly, the set of all $LCSk$s of the two reversed substrings $\overline{A}[i_0 : i_1] = a_{i_1}a_{i_1-1}\cdots a_{i_0}$ and $\overline{B}[j_0 : j_1] = b_{j_1}b_{j_1-1}\cdots b_{j_0}$ is denoted by $LCSkR(i_0, i_1, j_0, j_1)$. The length of an $LCSk$ in $LCSkR(i_0, i_1, j_0, j_1)$ is denoted by $g(i_0, i_1, j_0, j_1)$.

In the special case of $i_0 = 1$ and $j_0 = 1$, $f(1, i, 1, j)$ is abbreviated to $f(i,j)$ for $1 \leq i \leq n, 1 \leq j \leq m$. Similarly, in the special case of $i_1 = n$ and $j_1 = m$, $g(i, n, j, m)$ is abbreviated to $g(i,j)$ for $1 \leq i \leq n, 1 \leq j \leq m$. It is clear that in the special case of $i_0 = 1, i_1 = n$ and $j_0 = 1, j_1 = m$, we have $f(n,m) = g(1,1)$.

The following algorithm $\xi(i_0, i_1, j_0, j_1, L)$ uses an array $L[0:k][1:m]$ to store the $(k+1)m$ entries required to compute the current table. The row i $(1 \leq i \leq n)$ of f is mapped to the row $\lambda(i)$ $(1 \leq \lambda(i) \leq k)$ of L, where $\lambda(i)$ is defined as

$$\lambda(i) = (i-1) \bmod k + 1. \tag{4}$$

A space efficient algorithm $\xi(i_0, i_1, j_0, j_1, L)$ to solve the $LCSk$ problem for the input sequences $A[i_0 : i_1]$ and $B[j_0 : j_1]$ can be described as follows.

Algorithm 2: $\xi(i_0, i_1, j_0, j_1, L)$.

```
Input: $A[i_0 : i_1] = a_{i_0}a_{i_0+1}\cdots a_{i_1}$; $B[j_0 : j_1] = b_{j_0}b_{j_0+1}\cdots b_{j_1}$
Output: $L$, the map of $f(i,j)$
1 for $i = i_0$ to $i_1$ do
2     for $j = j_0$ to $j_1$ do
3         $L(\lambda(i-1), j) \leftarrow L(0, j)$;
4         if $a_i = b_j$ then $d(\lambda(i), j) \leftarrow 1 + d(\lambda(i-1), j-1)$;
5         $L(0, j) \leftarrow \max\{L(\lambda(i-1), j), L(0, j-1), L(\lambda(i-k), j-k) + \delta(d(\lambda(i), j))\}$;
6     end
7 end
8 for $j = j_0$ to $j_1$ do $L(\lambda(i_1), j) \leftarrow L(0, j)$;
9 return $L(\lambda(i_1), j_1)$
```

In the algorithm above, the array L of size $(k+1)m$ is utilized to hold the appropriate entries of f. At the time $f(i,j)$ to be computed, L will hold the following entries:

- $L(\lambda(t), j) = f(t, j)$ for $1 \le t < i, 1 \le j \le m$;
- $L(0, j) = f(i, j)$ for $1 \le j \le m$.

Therefore, a total of $(k+1)m$ entries is used in the algorithm. The time complexity of the algorithm is obviously $O((i_1 - i_0)(j_1 - j_0))$. In the case of $i_0 = 1, i_1 = n$ and $j_0 = 1, j_1 = m$, the algorithm $\xi(1, n, 1, m, L)$ can find $f(n, m)$, the number of k matchings in the longest common subsequence of A and B, in $O(mn)$ time using $O(km)$ space.

Similarly, the space efficient algorithm $\eta(i_0, i_1, j_0, j_1, L)$ to solve the $LCSkR$ problem for the input sequences $A[i_0 : i_1]$ and $B[j_0 : j_1]$ can be described as follows.

Algorithm 3: $\eta(i_0, i_1, j_0, j_1, L)$.

```
Input: $A[i_0 : i_1] = a_{i_0} a_{i_0+1} \cdots a_{i_1}$; $B[j_0 : j_1] = b_{j_0} b_{j_0+1} \cdots b_{j_1}$
Output: $L$, the map of $g(i, j)$
1 for $i = i_1$ downto $i_0$ do
2     for $j = j_1$ downto $j_0$ do
3         $L(\lambda(i+1), j) \leftarrow L(0, j)$;
4         if $a_i = b_j$ then $d(\lambda(i), j) \leftarrow 1 + d(\lambda(i+1), j+1)$;
5         $L(0, j) \leftarrow \max\{L(\lambda(i+1), j), L(0, j+1), L(\lambda(i+k), j+k) + \delta(d(\lambda(i), j))\}$;
6     end
7 end
8 for $j = j_0$ to $j_1$ do $L(\lambda(i_0), j) \leftarrow L(0, j)$;
9 return $L(\lambda(i_0), j_0)$
```

In the following discussion, we will use $\oplus$ to denote the concatenation of two strings.

For $k < t \le n$, let

$$M(t) = \max_{\substack{t-k+1 \le i \le t \\ 0 \le j \le m}} \{f(i, j) + g(i+1, j+1)\} \tag{5}$$

Theorem 1. *For $k < t \le n$, $M(t) = f(n, m)$.*

Proof. Let $M(t) = f(i, j) + g(i+1, j+1)$ for some i, j. Let $Z(i, j) \in LCSk(1, i, 1, j)$ and $Z'(i, j) \in LCSkR(i+1, n, j+1, m)$. Then, $C = Z(i, j) \oplus Z'(i, j)$, the concatenation of the two $LCSk$s, is a common subsequence of A and B in k-length substrings of length $M(t)$. Therefore, $M(t) \le f(n, m)$.

On the other hand, let $Z(n, m) \in LCSk(1, n, 1, m)$, and $l = f(n, m)$.

In the case of $l = 0$, it is clear that $M(t) = f(n, m) = 0$ for any $k < t \le n$.

In the case of $l > 0$, let

$$Z(n, m) = \begin{pmatrix} A[i_1 : i_1 + k - 1], \cdots, A[i_l : i_l + k - 1] \\ B[j_1 : j_1 + k - 1], \cdots, B[j_l : j_l + k - 1] \end{pmatrix}.$$

For $k < t \le n$, there are some cases to be distinguished.

(1) $t \notin [i_p, i_p + k - 1]$, for $1 \le p \le l$.

In this case, there are also some subcases.

(1.1) $t < i_1$. In this case, let $i = t$ and $j = j_1 - 1$, then $t - k + 1 \le i \le t$, $0 \le j \le m$, and $f(i, j) = 0$, $g(i+1, j+1) = f(n, m)$, and thus $M(t) \ge f(i, j) + g(i+1, j+1) = f(n, m)$.

(1.2) $t > i_l + k - 1$. In this case, let $i = t$ and $j = j_l + k - 1$, then $t - k + 1 \le i \le t$, $0 \le j \le m$, and $f(i, j) = f(n, m)$, $g(i+1, j+1) = 0$, and thus $M(t) \ge f(i, j) + g(i+1, j+1) = f(n, m)$.

(1.3) There exists $1 \le p < l$ such that $i_p + k - 1 < t < i_{p+1}$. In this case, let $i = t$ and $j = j_p + k - 1$, then $t - k + 1 \le i \le t$, $0 \le j \le m$, and $f(i, j) = p$, $g(i+1, j+1) = l - p$, and thus $M(t) \ge f(i, j) + g(i+1, j+1) = f(n, m)$.

(2) There exists $1 \le p \le l$ such that $t \in [i_p, i_p + k - 1]$.

(2.1) $t = i_p + k - 1$. In this case, let $i = i_p + k - 1$ and $j = j_p + k - 1$, then $t - k + 1 \le i \le t$, $0 \le j \le m$, and $f(i, j) = p$, $g(i+1, j+1) = l - p$, and thus $M(t) \ge f(i, j) + g(i+1, j+1) = f(n, m)$.

(2.2) $i_p \le t < i_p + k - 1$. In this case, let $i = i_p - 1$ and $j = j_p - 1$, then $t - i = t - i_p + 1 < i_p + k - 1 - i_p + 1 = k$, and thus $t - k + 1 \le i \le t$, $0 \le j \le m$, and $f(i, j) = p - 1$, $g(i+1, j+1) = l - p + 1$. Thus, $M(t) \ge f(i, j) + g(i+1, j+1) = f(n, m)$.

Finally we have, $M(t) = f(n, m)$.

The proof is completed. □

Based on Hirschberg's divide-and-conquer method of solving the LCS problem in linear space [6], we now can use the above theorem recursively to design a divide-and-conquer algorithm to find an $LCSk$ of A and B as follows.

Algorithm 4: $D\&C(i_0, i_1, j_0, j_1)$.

```
1 if $i_1 - i_0 + 1 < 2k$ then return;
2 $l \leftarrow \lfloor (i_1 - i_0 + 1 + k)/2 \rfloor$;
3 $\xi(i_0, i_0 + l - 1, j_0, j_1, L_1)$;
4 $\eta(i_0 + l - k + 1, i_1, j_0, j_1, L_2)$;
5 $split(k_1, k_2, l_1, l_2, s_1, s_2, i_0 + l - 1, j_0, j_1)$;
6 if $l_1 > 1$ then $D\&C(i_0, k_1, j_0, k_2)$;
7 else if $l_1 = 1$ then print $b_{s_1} b_{s_1+1} \cdots b_{s_1+k-1}$;
8 if $l_2 > 1$ then $D\&C(k_1 + 1, i_1, k_2 + 1, j_1)$;
9 else if $l_2 = 1$ then print $b_{s_2} b_{s_2+1} \cdots b_{s_2+k-1}$;
```

The algorithm is invoked in the condition of $n \geq 2k$. In the case of $n < 2k$, the problem can be solved by using the standard dynamic programming algorithm of Benson et al. [3,4]. The space cost in this case is also $O(km)$.

Once the above algorithm is invoked, the sub-algorithm $split(k_1, k_2, l_1, l_2, s_1, s_2, i_1, j_0, j_1)$ is used to find the split points k_1 and k_2 by using Theorem 1.

The two sequences $A[i_0 : i_1]$ and $B[j_0 : j_1]$ are spitted into $A[i_0 : k_1]$, $A[k_1 + 1 : i_1]$ and $B[j_0 : k_2]$, $B[k_2 + 1 : j_1]$ such that

$$\begin{cases} l_1 = f(i_0, k_1, j_0, k_2) \\ l_2 = g(k_1 + 1, i_1, k_2 + 1, j_1) \\ f(i_0, i_1, j_0, j_1) = l_1 + l_2 \end{cases} \tag{6}$$

In the case of $l_1 = 1$ ($l_2 = 1$), the first k-sting start from s_1 (s_2) can be found by

$$\begin{cases} s_1 = \min\limits_{j_0 \leq j \leq k_2} \{j | f(i_0, k_1, j_0, j) = 1\} - k + 1 & \text{if } l_1 = 1 \\ s_2 = \max\limits_{k_2+1 \leq j \leq j_1} \{j | g(k_1 + 1, i_1, j, j_1) = 1\} & \text{if } l_2 = 1 \end{cases} \tag{7}$$

It is clear that in the case of $l_1 = 1$, if $s_1 \geq j_0$ then $B[s_1 : s_1 + k - 1]$ is an $LCSk$ of $A[i_0 : k_1]$ and $B[j_0 : k_2]$. Similarly, in the case of $l_2 = 1$, if $s_2 \geq k_2 + 1$ then $B[s_2 : s_2 + k - 1]$ is an $LCSk$ of $A[k_1 + 1 : i_1]$ and $B[k_2 + 1 : j_1]$.

In the cases of $l_1 > 1$ and $l_2 > 1$, the problem can be solved recursively.

Algorithm 5: $split(k_1, k_2, l_1, l_2, s_1, s_2, i_1, j_0, j_1)$.

```
Output: $k_1, k_2, l_1, l_2, s_1, s_2$
$s_1, s_2, tmp \leftarrow 0$;
for $i = i_1 - k + 1$ to $i_1$ do
    for $j = j_0 - 1$ to $j_1$ do
        $t \leftarrow L_1(\lambda(i), j) + L_2(\lambda(i + 1), j + 1)$;
        if $t > tmp$ then $tmp \leftarrow t, k_1 \leftarrow i, k_2 \leftarrow j$;
    end
end
$l_1 \leftarrow L_1(\lambda(k_1), k_2); l_2 \leftarrow L_2(\lambda(k_1 + 1), k_2 + 1)$;
if $l_1 = 1$ then $s_1 \leftarrow \min_{j_0 \leq j \leq j_1} \{j | L_1(\lambda(k_1), j) = 1\} - k + 1$;
if $l_2 = 1$ then $s_2 \leftarrow \max_{j_0 \leq j \leq j_1} \{j | L_2(\lambda(k_1 + 1), j) = 1\}$;
```

2.2. Correctness of the algorithm

We now prove that if the above algorithm is applied to the given sequences A and B, $D\&C(1, n, 1, m)$ will produce an $LCSk$ of A and B.

Proof. The claim can be proven by induction on n and m, the sizes of the input sequence A and B respectively. In the case of $n < 2k$ and any $m > 0$, we have $l \leq 1$ in the algorithm.

In the cases of $l_1 = 1$ and $l_2 = 1$, let $B[s_1 : s_1 + k - 1]$ and $B[s_2 : s_2 + k - 1]$ be the k-strings start from s_1 and s_2 respectively, where

$$\begin{cases} s_1 = \min\limits_{1 \leq j \leq k_2} \{j | f(1, k_1, 1, j) = 1\} - k + 1 \\ s_2 = \max\limits_{k_2+1 \leq j \leq m} \{j | g(k_1 + 1, n, j, m) = 1\} \end{cases} \tag{8}$$

Then, it can be verified directly that the k-strings $B[s_1 : s_1+k-1] \in LCSk(1, k_1, 1, k_2)$ and $B[s_2 : s_2+k-1] \in LCSk(k_1+1, n, k_2+1, m)$, since $l_1 = 1$ and $l_2 = 1$.

In the cases of $l_1 = 0$ and $l_2 = 0$, it is clear that $LCSk(1, k_1, 1, k_2) = \emptyset$ and $LCSk(k_1+1, n, k_2+1, m) = \emptyset$, and nothing is done by the algorithm in the cases.

Therefore, the claim applies in the case of $n < 2k$.

Suppose the claim is true when the size of the input sequence A is less than n. We now show that when the size of the input sequence A is n, the claim is also true. In this case, A is divided into two subsequences $A[1 : \lfloor (n+k)/2 \rfloor]$ and $A[\lfloor (n+k)/2 \rfloor + 1 : n]$. Then, in line 3–4 of the algorithm, the length of an $LCSk$ in $LCSk(1, \lfloor (n+k)/2 \rfloor, 1, m)$ is computed by $\xi(1, \lfloor (n+k)/2 \rfloor, 1, m)$ and the result is stored in L_1. The length of an $LCSk$ in $LCSkR(\lfloor (n+k)/2 \rfloor + 1, n, 1, m)$ is also computed by $\eta(\lfloor (n+k)/2 \rfloor + 1, n, 1, m)$ and the result is stored in L_2.

$$M(\lfloor (n+k)/2 \rfloor) = \max_{\substack{\lfloor (n+k)/2 \rfloor - k+1 \le i \le \lfloor (n+k)/2 \rfloor \\ 0 \le j \le m}} \{f(i, j) + g(i+1, j+1)\}$$

is then computed by the algorithm $split(k_1, k_2, l_1, l_2, s_1, s_2, \lfloor (n+k)/2 \rfloor, 1, m)$ in line 5, using the results in L_1 and L_2. The split points k_1 and k_2 are found such that

$$\begin{cases} l_1 = f(1, k_1, 1, k_2) \\ l_2 = g(k_1+1, n, k_2+1, m) \\ M(\lfloor (n+k)/2 \rfloor) = l_1 + l_2 \end{cases} \tag{9}$$

In the cases of $l_1 > 1$ and $l_2 > 1$, the $LCSk$s $Z_1 \in LCSk(1, k_1, 1, k_2)$ and $Z_2 \in LCSk(k_1+1, n, k_2+1, m)$ are found recursively, in lines 6 and 8 of the algorithm.

Thus, $Z = Z_1 \oplus Z_2$, the concatenation of the subsequences Z_1 and Z_2, is a common subsequence of A and B. It follows from (9) and Theorem 1 that $|Z| = M(\lfloor (n+k)/2 \rfloor) = f(n, m)$. Therefore, Z, the common subsequence of A and B produced by the algorithm is an $LCSk$ of A and B.

Finally, we can conclude by induction that the algorithm $D\&C(1, n, 1, m)$ can produce an $LCSk$ of A and B.

The proof is completed. □

2.3. Time analysis of the algorithm

We have proved that a call $D\&C(1, n, 1, m)$ of the algorithm produces an $LCSk$ of A and B. Let the time cost of the algorithm be $T(n, m)$ if the sizes of the input sequences are n and m respectively. The problem is divided into two smaller subproblems of finding $LCSk$s in $LCSk(1, k_1, 1, k_2))$ and $LCSk(k_1+1, n, k_2+1, m)$ by a call of $split(k_1, k_2, l_1, l_2, s_1, s_2, \lfloor (n+k)/2 \rfloor, 1, m)$ and two calls of $\xi(1, \lfloor (n+k)/2 \rfloor, 1, m)$ and $\eta(\lfloor (n+k)/2 \rfloor + 1, n, 1, m)$. It is clear that the time costs of ξ and η are both $O(mn)$, and the time cost of $split$ is $O(km)$. Thus $T(n, m)$ satisfies the following equation.

$$T(n, m) = \begin{cases} T(k_1, k_2) + T(n-k_1, m-k_2) + O(mn) & \text{if } n \ge k, \\ O(1), & \text{if } n < k. \end{cases} \tag{10}$$

Where, $\lfloor (n+k)/2 \rfloor - k + 1 \le k_1 \le \lfloor (n+k)/2 \rfloor$ and $0 \le k_2 \le m$.

It can be proved by induction that (10) has a solution $T(n, m) = O(mn)$.

Proof. The claim is trivially true for $n < 2k$.

In the case of $n \ge 2k$, it follows that $n/4 \le k_1 \le 3n/4$. Assume $T(n, m)$ is bounded by $c_1 \cdot mn$, and the $O(mn)$ term in (10) is bounded by $c_2 \cdot mn$. It follows from (10) that $T(k_1, k_2) + T(n-k_1, m-k_2) + O(mn)$ is bounded by

$$\begin{aligned} & c_1 \cdot (k_1 k_2 + (n-k_1)(m-k_2)) + c_2 \cdot mn \\ & \le c_1 \cdot (3n/4)(k_2 + m - k_2) + c_2 \cdot mn \qquad \text{只放缩了k1} \\ & = c_1 \cdot 3mn/4 + c_2 \cdot mn \\ & = (3c_1/4 + c_2) \cdot mn \end{aligned}$$

To be consistent with the assumption on the time bound of $T(n, m)$, we must have $3c_1/4 + c_2 \le c_1$, which is realizable by letting $c_1 \ge 4c_2$. It follows from (10) and by induction on n that $T(n, m) \le c_1 \cdot mn$.

The proof is completed. □

2.4. Space analysis of the algorithm

We assume that the input sequences A and B are in common storage using $O(m+n)$ space. In the execution of the algorithm $D\&C$, the temporary arrays L_1 and L_2 are used in the execution of the algorithms ξ and η. It is clear that

$|L_1| \le (k+1)m$ and $|L_2| \le (k+1)m$. It is seen that the execution of the algorithm $D\&C$ uses $O(1)$ temporary space, and the recursive calls to $D\&C$ are exclusive. There are at most $2n-1$ calls to the algorithm $D\&C$ (it can be proved analogously in [9]), and thus the space cost of the algorithm $D\&C$ is proportional to $(k+1)m$, i.e. $O(kn)$.

3. An $O(kn)$ space algorithm for solving the $LCS_{\ge}k$ problem

As stated in [4,8], $LCS_{\ge}k$ can be solved by using a dynamic programming algorithm. A dynamic programming algorithm using $O(kn^2)$ time and $O(n^2)$ space to solve the problem was presented by Benson et al. If only the length of $LCS_{\ge}k$ has to be computed, the space cost of their algorithm can be reduced to $O(kn)$, but if an $LCS_{\ge}k$ has to be constructed, their algorithm requires $O(n^2)$ space. Very recently, Ueki et al. presented a similar dynamic programming algorithm to solve the problem [16]. Their algorithm requires $O(mn)$ time and $O(mn)$ space. If only the length of an $LCS_{\ge}k$ is required, the space complexity can be easily reduced to $O(kn)$.

In the following section, a totally different algorithm for solving the $LCS_{\ge}k$ is presented. A very simple recursive formula can be built for our purpose in Theorem 2, which enables us to compute each item recursively in $O(1)$ time, and finally the total computing time can be reduced to $O(mn)$.

The main contribution of this section is that the space cost of the new algorithm can be reduced from $O(mn)$ to $O(kn)$, and an $LCS_{\ge}k$ can be output as a result of computing, not just its length. This is a new feature has never been reported yet by the previous algorithms.

3.1. Reduce the time cost to $O(mn)$

As stated above, $LCS_{\ge}k$ can be solved by using a dynamic programming algorithm. The recursive rule for $LCS_{\ge}k$ is a modification of (2) for $LCSk$.

Let $p(i,j)$ denote the length of the longest common subsequence consisting of at least k length matchings in the prefixes $A[1:i]$ and $B[1:j]$. For each fixed pair of (i,j), $k \le i \le n, 1 \le j \le m$, let

$$\begin{cases} \mu(i,j,t) = p(i-t, j-t) + t \\ \nu(i,j) = \min\{d(i,j), 2k-1\} \end{cases} \tag{11}$$

$$\beta(i,j) = \begin{cases} \max\limits_{k \le t \le \nu(i,j)} \{\mu(i,j,t)\} & \text{if } \nu(i,j) \ge k \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

Where $d(i,j)$, the length of the longest match between the prefixes of $A[1:i]$ and $B[1:j]$ is defined by (1). Then, $p(i,j)$ can be computed recursively as follows [8].

$$p(i,j) = \max \begin{cases} p(i-1,j) \\ p(i,j-1) \\ \beta(i,j) \end{cases} \tag{13}$$

Based on the formula (13), the table $p(i,j)$ for the given input sequences $A = a_1a_2\cdots a_n$ and $B = b_1b_2\cdots b_m$ of size n and m respectively, can be computed in $O(kmn)$ time and $O(mn)$ space by a standard dynamic programming algorithm, since $\beta(i,j)$ can be computed by $O(k)$ comparisons.

In this section, we can show that $\mu(i,j,t)$ is a very special function of t, and thus $\beta(i,j)$ can be recursively computed in $O(1)$ time for each pair of (i,j).

Let $Z = s_1 \bigoplus \cdots \bigoplus s_r \in LCS_{\ge}k(1,i,1,j)$ be a longest common subsequence consisting of r at least k length matchings in the prefixes $A[1:i]$ and $B[1:j]$, where

$$s_t = \begin{pmatrix} a_{i_t}, \cdots, a_{i_t+l_t-1} \\ b_{j_t}, \cdots, b_{j_t+l_t-1} \end{pmatrix} \tag{14}$$

The length of s_t is l_t, and $k \le l_t \le 2k-1$, $1 \le t \le r$, and thus

$$p(i,j) = \sum_{t=1}^{r} l_t \tag{15}$$

Any two substrings in the same sequence do not overlap, and thus, for all $1 \le t < r$,

$$\begin{cases} i_t + l_t \le i_{t+1} \\ j_t + l_t \le j_{t+1} \end{cases} \tag{16}$$

In the following, the suffix matching of $A[1:i]$ and $B[1:j]$ with length t, $1 \le t \le d(i,j)$, is defined as

$$\alpha(i,j,t) = \begin{pmatrix} a_{i-t+1}, \cdots, a_i \\ b_{j-t+1}, \cdots, b_j \end{pmatrix} \tag{17}$$

Lemma 1. *For any pair of (i,j), if $d(i,j) \ge 2k-1$, then $p(i,j) = \beta(i,j)$.*

Proof. Let $Z = s_1 \bigoplus \cdots \bigoplus s_r \in LCS_{\ge}k(1,i,1,j)$. It is clear that $l_r \le 2k-1$. It follows from $i_r + l_r - 1 \le i$ and $j_r + l_r - 1 \le j$ that $i_r \le i - l_r + 1$ and $j_r \le j - l_r + 1$, and thus $s_1 \bigoplus \cdots \bigoplus s_{r-1} \bigoplus \alpha(i,j,l_r) \in LCS_{\ge}k(1,i,1,j)$, since $l_r \le 2k-1 \le d(i,j)$. It follows that $p(i,j) = p(i-l_r, j-l_r) + l_r \le \beta(i,j)$.

It follows from (13) that $p(i,j) \ge \beta(i,j)$, and finally we have $p(i,j) = \beta(i,j)$.

The proof is completed. □

Lemma 2. *For any pair of (i,j), if $d(i,j) \ge 2k$, then $p(i,j) = 1 + p(i-1,j-1)$.*

Proof. It follows from Lemma 1 that $p(i,j) = \beta(i,j)$ and $p(i-1,j-1) = \beta(i-1,j-1)$, since $d(i,j) \ge 2k$.

We now prove Lemma 2 in the following two steps.

(1) We first show that

$$p(i,j) \le 1 + p(i-1,j-1) \tag{18}$$

Let $s_1 \bigoplus \cdots \bigoplus s_r \in LCS_{\ge}k(1,i,1,j)$. It follows from Lemma 1 that we can assume that $s_r = \alpha(i,j,l_r)$.

There are two cases to be distinguished.

(1.1) $l_r > k$.

In this case, $s_1 \bigoplus \cdots \bigoplus s_{r-1} \bigoplus \alpha(i-1,j-1,l_r-1)$ must be a common subsequence consisting of at least k length matchings in the prefixes $A[1:i-1]$ and $B[1:j-1]$ with length $p(i,j)-1$.

It follows that $p(i-1,j-1) \ge p(i,j)-1$.

(1.2) $l_r = k$.

In this case, we have $i_{r-1} + l_{r-1} = i_r$ or $j_{r-1} + l_{r-1} = j_r$. Otherwise, $s_1 \bigoplus \cdots \bigoplus s_{r-1} \bigoplus \alpha(i,j,l_r+1)$ would be a longer common subsequence.

We check the case $i_{r-1} + l_{r-1} = i_r$. The other case $j_{r-1} + l_{r-1} = j_r$ can be proved analogously.

There are also two different subcases.

(1.2.1) $l_{r-1} > k$.

In this case, let $s' = \begin{pmatrix} a_{i_{r-1}}, \cdots, a_{i_{r-1}+l_{r-1}-2} \\ b_{j_{r-1}}, \cdots, b_{j_{r-1}+l_{r-1}-2} \end{pmatrix}$. Then, $|s'| = l_{r-1} - 1 \ge k$, and $s_1 \bigoplus \cdots \bigoplus s_{r-2} \bigoplus s' \bigoplus \alpha(i-1,j-1,l_r)$ must be a common subsequence consisting of at least k length matchings in the prefixes $A[1:i-1]$ and $B[1:j-1]$ with length $p(i,j)-1$.

It follows that $p(i-1,j-1) \ge p(i,j)-1$.

(1.2.2) $l_{r-1} = k$.

In this case, $s_1 \bigoplus \cdots \bigoplus s_{r-2} \bigoplus \alpha(i-1,j-1,2k-1)$ must be a common subsequence consisting of at least k length matchings in the prefixes $A[1:i-1]$ and $B[1:j-1]$ with length $p(i,j)-1$.

It follows that $p(i-1,j-1) \ge p(i,j)-1$.

(2) We now show

$$p(i,j) \ge 1 + p(i-1,j-1) \tag{19}$$

Let $s_1 \bigoplus \cdots \bigoplus s_r \in LCS_{\ge}k(1,i-1,1,j-1)$. It follows from Lemma 1 that we can assume that $s_r = \alpha(i-1,j-1,l_r)$.

There are two cases to be distinguished.

(1) $l_r < 2k-1$.

In this case, $s_1 \bigoplus \cdots \bigoplus s_{r-1} \bigoplus \alpha(i,j,l_r+1)$ must be a common subsequence consisting of at least k length matchings in the prefixes $A[1:i]$ and $B[1:j]$ with length $p(i-1,j-1)+1$.

It follows that $p(i,j) \ge p(i-1,j-1)+1$.

(2) $l_r = 2k-1$.

In this case, $|\alpha(i,j,l_r+1)| = 2k$, and thus $\alpha(i,j,l_r+1)$ is a concatenation of two at least k length matchings. It follows that $s_1 \bigoplus \cdots \bigoplus s_{r-1} \bigoplus \alpha(i,j,l_r+1)$ must be a common subsequence consisting of at least k length matchings in the prefixes $A[1:i]$ and $B[1:j]$ with length $p(i-1,j-1)+1$.

It follows that $p(i,j) \ge p(i-1,j-1)+1$.

Finally, it follows from (18) and (19) that $p(i,j) = 1 + p(i-1,j-1)$.

The proof is completed. □

It follows from Lemma 1 and Lemma 2 that if $d(i,j) \ge 2k$, then

$$\beta(i,j) = 1 + \beta(i-1,j-1) \tag{20}$$

Lemma 3. *For any pair of (i, j), if $d(i, j) < 2k$, then*

$$\beta(i,j) = \max \begin{cases} p(i-k, j-k) + k \\ 1 + \beta(i-1, j-1) \end{cases} \tag{21}$$

Proof. It follows from $d(i, j) < 2k$ that $\nu(i, j) = d(i, j)$. It is clear that $\mu(i, j, t)$ can also be defined recursively as follows.

$$\mu(i,j,t) = \begin{cases} 1 + \mu(i-1, j-1, t-1) & \text{if } k < t \\ p(i-k, j-k) + k & \text{otherwise} \end{cases} \tag{22}$$

It follows from (12) and (22) that

$$\begin{aligned} \beta(i,j) &= \max_{k \le t \le d(i,j)} \{\mu(i,j,t)\} \\ &= \max \begin{cases} \mu(i,j,k) \\ \max\limits_{k < t \le d(i,j)} \{1 + \mu(i-1, j-1, t-1)\} \end{cases} \\ &= \max \begin{cases} \mu(i,j,k) \\ 1 + \max\limits_{k \le t \le d(i,j)-1} \{\mu(i-1, j-1, t)\} \end{cases} \\ &= \max \begin{cases} \mu(i,j,k) \\ 1 + \max\limits_{k \le t \le d(i-1,j-1)} \{\mu(i-1, j-1, t)\} \end{cases} \\ &= \max \begin{cases} p(i-k, j-k) + k \\ 1 + \beta(i-1, j-1) \end{cases} \end{aligned}$$

The proof is completed. □

A new recursive formula to compute $\beta(i, j)$ in $O(1)$ time can be built as follows.

Theorem 2.

$$\beta(i,j) = \begin{cases} \max \begin{cases} p(i-k, j-k) + k \\ 1 + \beta(i-1, j-1) \end{cases} & \textit{if } d(i,j) \ge k \\ 0 & \textit{otherwise} \end{cases} \tag{23}$$

Proof. It remains to show that (21) is also true for the case of $d(i, j) \ge 2k$. It is clear that $p(i-k, j-k) \ge k$, since $d(i, j) \ge 2k$. Let $Z = s_1 \bigoplus \cdots \bigoplus s_r \in LCS_{\ge}k(1, i-k, 1, j-k)$, and

$$s_r = \begin{pmatrix} a_{i_r}, \cdots, a_{i_r+l_r-1} \\ b_{j_r}, \cdots, b_{j_r+l_r-1} \end{pmatrix}.$$

We can show that

$$p(i-1, j-1) \ge p(i-k, j-k) + k - 1 \tag{24}$$

There are some cases to be distinguished.

(1) $i_r + l_r - 1 < k$ and $j_r + l_r - 1 < k$.

In this case, $Z \bigoplus \alpha(i-1, j-1, k)$ is a common subsequence consisting of at least k length matchings in the prefixes $A[1 : i-1]$ and $B[1 : j-1]$ with length $p(i-k, j-k) + k$, and thus $p(i-1, j-1) \ge p(i-k, j-k) + k$.

(2) $i_r + l_r - 1 = k$ or $j_r + l_r - 1 = k$.

We check the case $i_r + l_r - 1 = k$. The other case $j_r + l_r - 1 = k$ can be proved analogously.

(2.1) $l_r > k$.

In this case, $s_1 \bigoplus \cdots \bigoplus s_{r-1} \begin{pmatrix} a_{i_r}, \cdots, a_{i_r+l_r-2} \\ b_{j_r}, \cdots, b_{j_r+l_r-2} \end{pmatrix} \bigoplus \alpha(i-1, j-1, k)$ is a common subsequence consisting of at least k length matchings in the prefixes $A[1 : i-1]$ and $B[1 : j-1]$ with length $p(i-k, j-k) + k - 1$, and thus $p(i-1, j-1) \ge p(i-k, j-k) + k - 1$.

(2.2) $l_r = k$.

In this case, $s_1 \bigoplus \cdots \bigoplus s_{r-1} \bigoplus \alpha(i-1, j-1, 2k-1)$ is a common subsequence consisting of at least k length matchings in the prefixes $A[1:i-1]$ and $B[1:j-1]$ with length $p(i-k, j-k)+k-1$, and thus $p(i-1, j-1) \geq p(i-k, j-k)+k-1$.

It follows that in all cases (24) is true.

It follows from Lemma 1 that

$$1 + p(i-1, j-1) = 1 + \beta(i-1, j-1) \geq p(i-k, j-k) + k$$

Thus

$$1 + \beta(i-1, j-1) = \max \begin{cases} p(i-k, j-k) + k \\ 1 + \beta(i-1, j-1) \end{cases}$$

It follows from (20) that

$$\beta(i, j) = 1 + \beta(i-1, j-1) = \max \begin{cases} p(i-k, j-k) + k \\ 1 + \beta(i-1, j-1) \end{cases}$$

The proof is completed. □

Based on Theorem 2, the table $p(i, j)$ for the given input sequences $A = a_1 a_2 \cdots a_n$ and $B = b_1 b_2 \cdots b_m$ of size n and m respectively, can be computed in $O(mn)$ time and $O(mn)$ space by a standard dynamic programming algorithm.

Algorithm 6: $LCS_{\geq}k$.

```
Input: $A, B$
Output: $p(i, j)$, the length of $Z \in LCS_{\geq}k(1, i, 1, j)$
1 for $i$=1 to $n$ do
2     for $j$=1 to $m$ do
3         if $a_i = b_j$ then $d(i, j) \leftarrow 1 + d(i-1, j-1)$;
4         if $d(i, j) \geq k$ then $\beta(i, j) \leftarrow \max\{p(i-k, j-k)+k, 1+\beta(i-1, j-1)\}$
          $p(i, j) \leftarrow \max\{p(i-1, j), p(i, j-1), \beta(i, j)\}$;
5     end
6 end
7 return $p(n, m)$
```

It is clear that the time and space complexities of the algorithm are both $O(mn)$.

3.2. *Reduce the space cost to $O(kn)$*

In the algorithm above, when a particular row of the table is to compute, no rows before the previous k rows are required. Thus only $k+1$ rows have to be kept in memory at a time. Thus, we need only $O(kn)$ entries to compute the table.

If a longest common subsequence in at least k-length substrings has to be constructed, not just its length, then the $O(mn)$ entries of L is required for this purpose. Based on Hirschberg's divide-and-conquer method, the space cost can be reduced to $O(kn)$.

In order to use the divide-and-conquer method, we need to extend the definition of $LCS_{\geq}k$ to a more general definition as follows.

Definition 4. For the two substrings $A[i_0 : i_1] = a_{i_0} a_{i_0+1} \cdots a_{i_1}$ and $B[j_0 : j_1] = b_{j_0} b_{j_0+1} \cdots b_{j_1}$, $1 \leq i_0 \leq i_1 \leq n, 1 \leq j_0 \leq j_1 \leq m$, the set of all $LCS_{\geq}k$s of $A[i_0 : i_1]$ and $B[j_0 : j_1]$ is denoted by $LCS_{\geq}k(i_0, i_1, j_0, j_1)$. The length of an $LCS_{\geq}k$ in $LCS_{\geq}k(i_0, i_1, j_0, j_1)$ is denoted by $p(i_0, i_1, j_0, j_1)$.

Similarly, the set of all $LCS_{\geq}k$s of the two reversed substrings $\overline{A}[i_0 : i_1] = a_{i_1} a_{i_1-1} \cdots a_{i_0}$ and $\overline{B}[j_0 : j_1] = b_{j_1} b_{j_1-1} \cdots b_{j_0}$ is denoted by $LCS_{\geq}kR(i_0, i_1, j_0, j_1)$. The length of an $LCS_{\geq}k$ in $LCS_{\geq}kR(i_0, i_1, j_0, j_1)$ is denoted by $q(i_0, i_1, j_0, j_1)$.

In the special case of $i_0 = 1$ and $j_0 = 1$, $p(1, i, 1, j)$ is abbreviated to $p(i, j)$ for $1 \leq i \leq n, 1 \leq j \leq m$. Similarly, in the special case of $i_1 = n$ and $j_1 = m$, $q(i, n, j, m)$ is abbreviated to $q(i, j)$ for $1 \leq i \leq n, 1 \leq j \leq m$. It is clear that in the special case of $i_0 = 1, i_1 = n$ and $j_0 = 1, j_1 = m$, we have $p(n, m) = q(1, 1)$.

The following algorithm $\xi(i_0, i_1, j_0, j_1, L)$ uses an array $L[0 : 2k][1 : m]$ to store the $(2k+1)m$ entries to compute the current table. The row $i (1 \leq i \leq n)$ of f is mapped to the row $\lambda(i) (1 \leq \lambda(i) \leq k)$ of L, where $\lambda(i)$ is defined somewhat different from (4).

$$\lambda(i) = (i-1) \bmod 2k+1. \tag{25}$$

A space efficient algorithm $\xi(i_0, i_1, j_0, j_1, L)$ to solve the $LCS_{\geq}k$ problem for the input sequences $A[i_0 : i_1]$ and $B[j_0 : j_1]$ can be described as follows.

Algorithm 7: $\xi(i_0, i_1, j_0, j_1, L)$.

```
Input: $A[i_0 : i_1] = a_{i_0}a_{i_0+1}\cdots a_{i_1}$; $B[j_0 : j_1] = b_{j_0}b_{j_0+1}\cdots b_{j_1}$
Output: $L$, the map of $f(i, j)$
for $i = i_0$ to $i_1$ do
    for $j = j_0$ to $j_1$ do
        $L(\lambda(i-1), j) \leftarrow L(0, j)$;
        if $a_i = b_j$ then $d(\lambda(i), j) \leftarrow 1 + d(\lambda(i-1), j-1)$;
        if $d(\lambda(i), j) \geq k$ then $\beta(\lambda(i), j) \leftarrow \max\{k + L(\lambda(i-k), j-k), 1 + \beta(\lambda(i-1), j-1)\}$;
        $L(0, j) \leftarrow \max\{L(\lambda(i-1), j), L(0, j-1), \beta(\lambda(i), j)\}$;
    end
end
for $j = j_0$ to $j_1$ do $L(\lambda(i_1), j) \leftarrow L(0, j)$;
return $L(\lambda(i_1), j_1)$
```

In the algorithm above, the array L of size $(2k+1)m$ is used to hold the appropriate entries of p. At the time $p(i, j)$ to be computed, L will hold the following entries:

- $L(\lambda(t), j) = p(t, j)$ for $1 \leq t < i, 1 \leq j \leq m$;
- $L(0, j) = p(i, j)$ for $1 \leq j \leq m$.

Therefore, a total of $(2k+1)m$ entries is used in the algorithm. The time complexity of the algorithm is obviously $O((i_1 - i_0)(j_1 - j_0))$. In the case of $i_0 = 1, i_1 = n$ and $j_0 = 1, j_1 = m$, the algorithm $\xi(1, n, 1, m, L)$ can find $p(n, m)$ in $O(mn)$ time using $O(kn)$ space.

Similarly, the space efficient algorithm $\eta(i_0, i_1, j_0, j_1, L)$ to solve the $LCS_{\geq}kR$ problem for the input sequences $A[i_0 : i_1]$ and $B[j_0 : j_1]$ can be described as follows.

Algorithm 8: $\eta(i_0, i_1, j_0, j_1, L)$.

```
Input: $A[i_0 : i_1] = a_{i_0}a_{i_0+1}\cdots a_{i_1}$; $B[j_0 : j_1] = b_{j_0}b_{j_0+1}\cdots b_{j_1}$
Output: $L$, the map of $g(i, j)$
for $i = i_1$ downto $i_0$ do
    for $j = j_1$ downto $j_0$ do
        $L(\lambda(i+1), j) \leftarrow L(0, j)$;
        if $a_i = b_j$ then $d(\lambda(i), j) \leftarrow 1 + d(\lambda(i+1), j+1)$;
        if $d(\lambda(i), j) \geq k$ then $\beta(\lambda(i), j) \leftarrow \max\{k + L(\lambda(i+k), j+k), 1 + \beta(\lambda(i+1), j+1)\}$;
        $L(0, j) \leftarrow \max\{L(\lambda(i+1), j), L(0, j+1), \beta(\lambda(i), j)\}$;
    end
end
for $j = j_0$ to $j_1$ do $L(\lambda(i_0), j) \leftarrow L(0, j)$;
return $L(\lambda(i_0), j_0)$
```

For $2k \leq x \leq n$, let

$$M(x) = \max_{\substack{x-2k+1 \leq i \leq x \\ 0 \leq j \leq m}} \{p(i, j) + q(i+1, j+1)\} \tag{26}$$

Theorem 3. *For $2k \leq x \leq n$, $M(x) = p(n, m)$.*

Proof. Let $M(x) = p(i, j) + q(i+1, j+1)$ for some i, j. Let $Z \in LCS_{\geq}k(1, i, 1, m)$ and $Z' \in LCS_{\geq}kR(i+1, n, j+1, m)$. Then, $C = Z \oplus Z'$, the concatenation of the two $LCS_{\geq}k$s, is a common subsequence of A and B in k-length substrings of length $M(x)$. Therefore, $M(x) \leq p(n, m)$.

On the other hand, let $Z = s_1 \bigoplus \cdots \bigoplus s_r \in LCS_{\geq}k(1, n, 1, m)$, where

$$s_t = \begin{pmatrix} a_{i_t}, \cdots, a_{i_t+l_t-1} \\ b_{j_t}, \cdots, b_{j_t+l_t-1} \end{pmatrix}.$$

The length of s_t is l_t, and $k \leq l_t \leq 2k-1$, $1 \leq t \leq r$. It is clear that $p(n, m) = \sum_{t=1}^{r} l_t$.

In the case of $Z = \emptyset$, it is clear that $M(x) = p(n, m) = 0$ for any $2k \leq x \leq n$.

In the case of $r > 0$, there are some cases to be distinguished.

(1) $x \notin [i_t, i_t + l_t - 1]$, for $1 \le t \le r$.

In this case, there are also some subcases.

(1.1) $x < i_1$. In this case, let $i = x$ and $j = j_1 - 1$, then $x - 2k + 1 \le i \le x$, $0 \le j \le m$, and $p(i, j) = 0$, $q(i+1, j+1) = p(n, m)$, and thus $M(x) \ge p(i, j) + q(i + 1, j + 1) = p(n, m)$.

(1.2) $x > i_r + l_r - 1$. In this case, let $i = x$ and $j = j_r + l_r - 1$, then $t - 2k + 1 \le i \le x$, $0 \le j \le m$, and $p(i, j) = p(n, m)$, $q(i + 1, j + 1) = 0$, and thus $M(x) \ge p(i, j) + q(i + 1, j + 1) = p(n, m)$.

(1.3) There exists $1 \le t < r$ such that $i_t + l_t - 1 < x < i_{t+1}$. In this case, let $i = x$ and $j = j_t + l_t - 1$, then $x - 2k + 1 \le i \le x$, $0 \le j \le m$, and $p(i, j) = \sum_{u=1}^{t} l_u$, $q(i + 1, j + 1) = \sum_{u=t+1}^{r} l_u$, and thus $M(x) \ge p(i, j) + q(i + 1, j + 1) = p(n, m)$.

(2) There exists $1 \le t \le r$ such that $x \in [i_t, i_t + l_t - 1]$.

(2.1) $x = i_t + l_t - 1$. In this case, let $i = i_t + l_t - 1$ and $j = j_t + l_t - 1$, then $x - 2k + 1 \le i \le x$, $0 \le j \le m$, and $p(i, j) = \sum_{u=1}^{t} l_u$, $q(i + 1, j + 1) = \sum_{u=t+1}^{r} l_u$, and thus $M(x) \ge p(i, j) + q(i + 1, j + 1) = p(n, m)$.

(2.2) $i_t \le x < i_t + l_t - 1$. In this case, let $i = i_t - 1$ and $j = j_t - 1$, then $x - i = x - i_t + 1 < l_t \le 2k - 1$, and thus $x - 2k + 1 \le i \le x$, $0 \le j \le m$, and $p(i, j) = \sum_{u=1}^{t-1} l_u$, $q(i + 1, j + 1) = \sum_{u=t}^{r} l_u$. Thus, $M(t) \ge p(i, j) + q(i + 1, j + 1) = p(n, m)$.

Finally we have, $M(x) = p(n, m)$.

The proof is completed. □

Based on Hirschberg's divide-and-conquer method of solving the LCS problem in linear space [6], we now can use the above theorem recursively to design a divide-and-conquer algorithm to find an $LCS_{\ge}k$ of A and B as follows.

Algorithm 9: $D\&C(i_0, i_1, j_0, j_1)$.

```
1 if $i_1 - i_0 + 1 < 2k$ then return;
2 $l \leftarrow \max\{2k - 1, \lfloor (i_1 - i_0 + 1)/2 \rfloor + k\}$;
3 $\xi(i_0, i_0 + l - 1, j_0, j_1, L_1)$;
4 $\eta(\max\{i_0, i_0 + l - 2k\} + 1, i_1, j_0, j_1, L_2)$;
5 $split(k_1, k_2, l_1, l_2, s_1, s_2, i_0, i_0 + l - 1, j_0, j_1)$;
6 if $l_1 \ge 2k$ then $D\&C(i_0, k_1, j_0, k_2)$;
7 else if $s_1 > 0$ then print $b_{s_1} b_{s_1+1} \cdots b_{s_1+l_1-1}$;
8 if $l_2 \ge 2k$ then $D\&C(k_1 + 1, i_1, k_2 + 1, j_1)$;
9 else if $s_2 > 0$ then print $b_{s_2} b_{s_2+1} \cdots b_{s_2+l_2-1}$;
```

The algorithm is invoked in the condition of $n \ge 2k$. In the case of $n < 2k$, the problem can be solved by using the standard dynamic programming algorithm of Benson et al. [3,4]. The space costs in this case is also $O(kn)$.

Once the above algorithm is invoked, the sub-algorithm $split(k_1, k_2, l_1, l_2, s_1, s_2, i_0, i_1, j_0, j_1)$ is used to find the split points k_1 and k_2 by using Theorem 3.

The two sequences $A[i_0 : i_1]$ and $B[j_0 : j_1]$ are spitted into $A[i_0 : k_1]$, $A[k_1 + 1 : i_1]$ and $B[j_0 : k_2]$, $B[k_2 + 1 : j_1]$ such that

$$\begin{cases} l_1 = p(i_0, k_1, j_0, k_2) \\ l_2 = q(k_1 + 1, i_1, k_2 + 1, j_1) \\ p(i_0, i_1, j_0, j_1) = l_1 + l_2 \end{cases} \tag{27}$$

In the case of $l_1 < 2k$ ($l_2 < 2k$), the first at least k length matching start from s_1 (s_2) can be found by

$$\begin{cases} s_1 = \min\limits_{j_0 \le j \le k_2} \{j | p(i_0, k_1, j_0, j) = l_1\} - l_1 + 1 & \text{if } k \le l_1 \\ s_2 = \max\limits_{k_2+1 \le j \le j_1} \{j | q(k_1 + 1, i_1, j, j_1) = l_2\} & \text{if } k \le l_2 \end{cases} \tag{28}$$

In the cases of $l_1 \ge 2k$ and $l_2 \ge 2k$, the problem can be solved recursively.

Algorithm 10: $split(k_1, k_2, l_1, l_2, s_1, s_2, i_0, i_1, j_0, j_1)$.

```
Output: $k_1, k_2, l_1, l_2, s_1, s_2$
$s_1, s_2, tmp \leftarrow 0$;
for $i = \max\{i_0, i_1 - 2k + 1\}$ to $i_1$ do
    for $j = j_0 - 1$ to $j_1$ do
        $t \leftarrow L_1(\lambda(i), j) + L_2(\lambda(i + 1), j + 1)$;
        if $t \ge tmp$ then $tmp \leftarrow t, k_1 \leftarrow i, k_2 \leftarrow j$;
    end
end
$l_1 \leftarrow L_1(\lambda(k_1), k_2); l_2 \leftarrow L_2(\lambda(k_1 + 1), k_2 + 1)$;
if $k \le l_1 < 2k$ then $s_1 \leftarrow \min_{j_0 \le j \le j_1} \{j | L_1(\lambda(k_1), j) = l_1\} - l_1 + 1$;
if $k \le l_2 < 2k$ then $s_2 \leftarrow \max_{j_0 \le j \le j_1} \{j | L_2(\lambda(k_1 + 1), j) = l_2\}$;
```

3.3. Correctness of the algorithm

We now prove that if the above algorithm is applied to the given sequences A and B, $D\&C(1, n, 1, m)$ will produce an $LCS_{\geq}k$ of A and B.

Proof. The claim can be proven by induction on n and m, the sizes of the input sequence A and B respectively. In the case of $n < 2k$ and any $m > 0$, we have $0 \leq l \leq 2k - 1$ in the algorithm.

In the cases of $l_1 = 0$ and $l_2 = 0$, it is clear that $LCS_{\geq}k(1, k_1, 1, k_2) = \emptyset$ and $LCS_{\geq}k(k_1 + 1, n, k_2 + 1, m) = \emptyset$, and nothing is done by the algorithm in the cases.

If $l_1 > 0$ ($l_2 > 0$) then $k \leq l_1 \leq 2k - 1$ ($k \leq l_2 \leq 2k - 1$).

In the cases of $l_1 \geq k$ ($l_2 \geq k$), let $B[s_1 : s_1 + l_1 - 1]$ ($B[s_2 : s_2 + l_2 - 1]$) be the at least k length matching start from s_1 (s_2), defined by (28). Then, it can be verified directly that the strings $B[s_1 : s_1 + l_1 - 1] \in LCS_{\geq}k(1, k_1, 1, k_2)$ and $B[s_2 : s_2 + l_2 - 1] \in LCS_{\geq}k(k_1 + 1, n, k_2 + 1, m)$.

Therefore, the claim applies in the case of $n < 2k$.

Suppose the claim is true when the size of the input sequence A is less than n. We now show that when the size of the input sequence A is n, the claim is also true. In this case, A is divided into two subsequences $A[1 : \lfloor n/2 \rfloor + k]$ and $A[\lfloor n/2 \rfloor + k + 1 : n]$. Then, in line 3–4 of the algorithm, the length of an $LCS_{\geq}k$ in $LCS_{\geq}k(1, \lfloor n/2 \rfloor + k, 1, m)$ is computed by $\xi(1, \lfloor n/2 \rfloor + k, 1, m)$ and the result is stored in L_1. The length of an $LCS_{\geq}k$ in $LCS_{\geq}kR(\lfloor n/2 \rfloor + k + 1, n, 1, m)$ is also computed by $\eta(\lfloor n/2 \rfloor + k + 1, n, 1, m)$ and the result is stored in L_2.

$$M(\lfloor n/2 \rfloor + k) = \max_{\substack{\lfloor n/2 \rfloor - k + 1 \leq i \leq \lfloor n/2 \rfloor + k \\ 0 \leq j \leq m}} \{p(i, j) + q(i + 1, j + 1)\}$$

is then computed by the algorithm $split(k_1, k_2, l_1, l_2, s_1, s_2, 1, \lfloor n/2 \rfloor + k, 1, m)$ in line 5, using the results in L_1 and L_2. The split points k_1 and k_2 are found such that

$$\begin{cases} l_1 = p(1, k_1, 1, k_2) \\ l_2 = q(k_1 + 1, n, k_2 + 1, m) \\ M(\lfloor n/2 \rfloor + k) = l_1 + l_2 \end{cases} \tag{29}$$

In the cases of $l_1 \geq 2k$ and $l_1 \geq 2k$, the $LCS_{\geq}k$s $Z_1 \in LCS_{\geq}k(1, k_1, 1, k_2)$ and $Z_2 \in LCS_{\geq}k(k_1 + 1, n, k_2 + 1, m)$ are found recursively, in lines 6 and 8 of the algorithm.

Thus, $Z = Z_1 \oplus Z_2$, the concatenation of the subsequences Z_1 and Z_2, is a common subsequence of A and B. It follows from (29) and Theorem 3 that $|Z| = M(\lfloor n/2 \rfloor + k) = p(n, m)$. Therefore, Z, the common subsequence of A and B produced by the algorithm is an $LCS_{\geq}k$ of A and B.

Finally, we can conclude by induction that the algorithm $D\&C(1, n, 1, m)$ can produce an $LCS_{\geq}k$ of A and B.

The proof is completed. □

3.4. Time analysis of the algorithm

We have proved that a call $D\&C(1, n, 1, m)$ of the algorithm produces an $LCS_{\geq}k$ of A and B. Without loss of generality, we can assume $m \leq n$. Let the time cost of the algorithm be $T(n, m)$ if the sizes of the input sequences are n and m respectively. The problem is divided into two smaller subproblems of finding $LCS_{\geq}k$s in $LCS_{\geq}k(1, k_1, 1, k_2))$ and $LCS_{\geq}k(k_1 + 1, n, k_2 + 1, m)$ by a call of $split(k_1, k_2, l_1, l_2, s_1, s_2, 1, \lfloor n/2 \rfloor + k, 1, m)$ and two calls of $\xi(1, \lfloor n/2 \rfloor + k, 1, m)$ and $\eta(\lfloor n/2 \rfloor + k + 1, n, 1, m)$. It is clear that the time costs of ξ and η are both $O(mn)$, and the time cost of $split$ is $O(kn)$. Thus $T(n, m)$ satisfies the following equation.

$$T(n, m) = \begin{cases} T(k_1, k_2) + T(n - k_1, m - k_2) + O(mn) & \text{if } n \geq k, \\ O(1), & \text{if } n < k. \end{cases} \tag{30}$$

Where, $\lfloor n/2 \rfloor - k + 1 \leq k_1 \leq \lfloor n/2 \rfloor + k$ and $0 \leq k_2 \leq m$.

It can be proved by induction that (30) has a solution $T(n, m) = O(mn)$.

Proof. (1) We first show that the claim is true in the case of $n < 3k$. Let $Z = s_1 \bigoplus \cdots \bigoplus s_r \in LCS_{\geq}k(1, n, 1, m)$. It is clear that $r \leq 2$ since $n < 3k$. Thus, the algorithm $split$ can be called at most once, and its time cost is $O(km)$. Output of Z will cost $O(k)$ time. Therefore, the total time cost of the algorithm is $O(km)$ in the case of $n < 3k$.

(2) In the case of $n \geq 3k$, it follows that $n/6 \leq k_1 \leq 5n/6$. Assume $T(n, m)$ is bounded by $c_1 \cdot mn$, and the $O(mn)$ term in (30) is bounded by $c_2 \cdot mn$. It follows from (30) that $T(k_1, k_2) + T(n - k_1, m - k_2) + O(mn)$ is bounded by

$$\begin{aligned} & c_1 \cdot (k_1 k_2 + (n - k_1)(m - k_2)) + c_2 \cdot mn \\ & \leq c_1 \cdot (5n/6)(k_2 + m - k_2) + c_2 \cdot mn \end{aligned}$$

$$= c_1 \cdot 5mn/6 + c_2 \cdot mn$$
$$= (5c_1/6 + c_2) \cdot mn$$

To be consistent with the assumption on the time bound of $T(n,m)$, we must have $5c_1/6 + c_2 \le c_1$, which is realizable by letting $c_1 \ge 6c_2$. It follows from (30) and by induction on n that $T(n,m) \le c_1 \cdot mn$.

The proof is completed. □

3.5. Space analysis of the algorithm

We assume that the input sequences A and B are in common storage using $O(m+n)$ space. In the execution of the algorithm $D\&C$, the temporary arrays L_1 and L_2 are used in the execution of the algorithms ξ and η. It is clear that $|L_1| \le (2k+1)m$ and $|L_2| \le (2k+1)m$. It is seen that the execution of the algorithm $D\&C$ uses $O(1)$ temporary space, and the recursive calls to $D\&C$ are exclusive. There are at most $2n-1$ calls to the algorithm $D\&C$ (it can be proved analogously in [9]), and thus the space cost of the algorithm $D\&C$ is proportional to $(2k+1)m$, i.e. $O(kn)$.

4. Concluding remarks

We have presented two space efficient algorithms to solve the $LCSk$ problem and $LCS_{\ge}k$ problem. The space cost of the first algorithm to solve the $LCSk$ problem is reduced from $O(n^2)$ to $O(kn)$, if the size of the two input sequences are both n. The time and space costs of the second algorithm to solve the $LCS_{\ge}k$ problem are both improved. The time cost is reduced from $O(kn^2)$ to $O(n^2)$, and the space cost is reduced from $O(n^2)$ to $O(kn)$. In the case of $k = O(1)$, the two algorithms are both linear space algorithms.

The experiments to compare the behavior of $LCSk$ versus LCS on real data have been performed in [4]. The choice of the value k interestingly affects the similarity percentage in the experiments. The experiments were carried out only for small k. It is not clear that the large k, for example $k = O(n)$, is meaningful to the similarity measure.

References

[1] A. Amir, Z. Gotthilf, R. Shalom, Weighted LCS, J. Discrete Algorithms 8 (3) (2010) 273–281.
[2] A. Amir, T. Hartman, O. Kapah, R. Shalom, D. Tsur, Generalized LCS, Theoret. Comput. Sci. 409 (3) (2008) 438–449.
[3] G. Benson, A. Levy, B.R. Shalom, Longest common subsequence in k length substrings, Lecture Notes in Comput. Sci. 8199 (2013) 257–265.
[4] G. Benson, A. Levy, S. Maimoni, D. Noifeld, B.R. Shalom, LCSk: a refined similarity measure, Theoret. Comput. Sci. 638 (2016) 11–26.
[5] Y.C. Chen, K.M. Chao, On the generalized constrained longest common subsequence problems, J. Comb. Optim. 21 (3) (2011) 383–392.
[6] S. Deorowicza, S. Grabowski, Efficient algorithms for the longest common subsequence in k-length substrings, Inform. Process. Lett. 114 (2014) 634–638.
[7] Z. Gotthilf, D. Hermelin, G.M. Landau, M. Lewenstein, Restricted LCS, in: Proc. 17th Symposium on String Processing and Information Retrieval, SPIRE, 2010, pp. 250–257.
[8] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, Cambridge, UK, 1997.
[9] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, Commun. ACM 18 (6) (1975) 341–343.
[10] J.W. Hunt, T.G. Szymanski, A fast algorithm for computing longest subsequences, Commun. ACM 20 (5) (1977) 350–353.
[11] N.C. Jones, P.A. Pevzner, An Introduction to Bioinformatics Algorithms, MIT Press, 2004.
[12] G.M. Landau, A. Levy, I. Newman, LCS approximation via embedding into locally non-repetitive strings, Inform. and Comput. 209 (4) (2011) 705–716.
[13] G.M. Landau, E.W. Myers, M. Ziv-Ukelson, Two algorithms for LCS consecutive suffix alignment, in: Proc. 15th Annual Symposium on Combinatorial Pattern Matching, CPM, 2004, pp. 173–193.
[14] G.M. Landau, B. Schieber, M. Ziv-Ukelson, Sparse LCS common substring alignment, in: Proc. 14th Annual Symposium on Combinatorial Pattern Matching, CPM, 2003, pp. 225–236.
[15] F. Pavetic, G. Zuzic, M. Sikic, LCSk++: practical similarity metric for long strings, https://arxiv.org/abs/1407.2407.
[16] Y. Ueki Diptarama, M. Kurihara, Y. Matsuoka, K. Narisawa, R. Yoshinaka, H. Bannai, S. Inenaga, A. Shinohara, Longest common subsequence in at least k length order-isomorphic substrings, in: SOFSEM 2017, in: Lecture Notes in Comput. Sci., vol. 10139, 2017, pp. 363–374.
[17] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173.
[18] L. Wang, X. Wang, Y. Wu, D. Zhu, A dynamic programming solution to a generalized LCS problem, Inform. Process. Lett. 113 (2013) 723–728.
[19] Y. Wu, L. Wang, D. Zhu, X. Wang, An efficient dynamic programming algorithm for the generalized LCS problem with multiple substring exclusive constraints, J. Discrete Algorithms 26 (2014) 98–105.