An input sensitive online algorithm for LCS computation

Heikki Hyyrö

Department of Computer and Information Sciences, University of Tampere, Finland.
heikki.hyyro@cs.uta.fi

Abstract. We consider the classic problem of computing (the length of) the longest common subsequence (LCS) between two strings A and B with lengths m and n, respectively. There are several input sensitive algorithms for this problem, such as the $\mathcal{O}(\sigma n + \min\{Lm, L(n-L)\})$ algorithms by Rick [15] and Goeman and Clausen [5] and the $\mathcal{O}(\sigma n + \min\{\sigma d, Lm\})$ algorithms by Chin and Poon [4] and Rick [15]. Here L is the length of the LCS and d is the number of dominant matches between A and B, and σ is the alphabet size. These algorithms require $O(\sigma n)$ time preprocessing for both A and B. We propose a new fairly simple $\mathcal{O}(\sigma m + \min\{Lm, L(n-L)\})$ time algorithm that works in online manner: It needs to preprocess only A, and it can process B one character at a time, without knowing the whole string B beforehand. The algorithm also adapts well to the linear space[1] scheme of Hirschberg [6] for recovering the LCS, which was not as easy with the above-mentioned algorithms. In addition, our scheme fits well into the context of incremental string comparison [12,10]. The original algorithm of Landau et al. [12] for this problem uses $\mathcal{O}(\sigma m + Lm)$ space. By using our scheme instead, the space usage becomes $\mathcal{O}(\sigma m + \min\{Lm, L(n-L)\})$.

Keywords: string algorithms, longest common subsequences, incremental string comparison

1 Introduction

We use the following conventions and notation in this paper. Σ is a finite alphabet of size σ. Strings are composed of a finite (possibly length-zero) sequence of characters from Σ. The length of a string A is denoted by $|A|$. When $1 \le i \le |A|$, A_i denotes the ith character of A. The notation $A_{i..h}$, where $i \le h$, denotes the substring of A that begins at character A_i and ends at character A_h. Hence $A = A_{1..|A|}$. String A is a subsequence of string B if B can be transformed into A by deleting zero or more characters from it. That is, the characters of A must appear in B in the same order as in A, but they do not need to appear consecutively.

The length of a longest common subsequence (LCS) between two strings is a classic measure of string similarity. Given two strings A and B, we denote the set of their longest common subsequences by $LCS(A,B)$. The length of each longest common subsequence is denoted by $LLCS(A,B)$. For example if A = "string" and B = "writing", then $LCS(A,B) = \{$"ring", "ting"$\}$ and $LLCS(A,B) = 4$. Throughout this paper we use the traditional conventions that m denotes the length of string A, n denotes the length of string B, and $m \le n$.

The problem of LCS/LLCS computation has been studied extensively (see e.g. [3]). Wagner and Fischer [17] have given a basic $\mathcal{O}(mn)$ time LCS algorithm based on dynamic programming. In terms of theoretical results, it has been proven that the time complexity of the LCS problem has a general lower bound of $\Omega(n \log m)$ [8], and

[1] Here, as well as in [5] and [16], the alphabet size σ is assumed to be a constant.

that the quadratic $\mathcal{O}(mn)$ worst case time complexity of the basic dynamic programming algorithm cannot be improved by any algorithm that is based on individual "equal/nonequal" comparisons between characters [1]. Currently the theoretically fastest algorithm in the worst case is the $\mathcal{O}(mn/\log n)$ "Four Russians" algorithm of Masek and Paterson [13].

There are several input sensitive algorithms for the LCS problem whose running times depend on the properties of the input strings. For example the algorithm of Hunt and Szymanski [9] has a running time $O(r \log n)$, where r is the number of matches $A_i = B_j$ over all $i = 1, \ldots, m$ and $j = 1, \ldots, n$. In similar fashion both Chin and Poon [4] and Rick [15] have proposed $\mathcal{O}(\sigma n + \min\{\sigma d, Lm\})$ time algorithms, where d is the number of so-called dominant matches and $L = LLCS(A, B)$. Algorithms whose running time depends on L are typically more efficient than basic dynamic programming either when L is low, like for example the $\mathcal{O}(Ln + n \log n)$ algorithm of Hirschberg [7], or when L is high, like for example the $\mathcal{O}(n(m - L))$ algorithm of Wu et al. [18], but not in both cases simultaneously. Exceptions to this rule are the $\mathcal{O}(\sigma n + \min\{Lm, L(n-L)\})$ algorithms by Rick [15] and later by [5]. Rick's algorithm was the first algorithm that is efficient with both low and high values of L, and it has also been found to be very efficient in practice [3].

The input sensitive algorithms typically rely on a preprocessing phase that is possibly costly. For example the term σn in the two algorithms of Rick [15] and the algorihms of Goeman and Clausen [5] and Chin and Poon [4] comes from the preprocessing phase. It is furher often the case that the preprocessing needs to be done for both A and B before the actual computation. This is true for example in the case of each of the four above mentioned algorithms. This may be significant for example within the setting of one-against-many type of comparison, e.g. when comparing a single pattern string A against each string B in some string database. In such a setting it would be desirable that the preprocessing phase would not need to be repeated for each different string B, that is, if it would be enough to only preprocess the string A once before the comparisons.

In addition to preprocessing only A, a further sometimes desirable property is that the LCS algorithm should work in online manner. By this we mean that the algorithm is able to process the string B one character at a time, without relying on knowledge about the yet unprocessed characters. That is, the algorithm can first read B_1 and compute $LLCS(A, B_1)$, then read B_2 and update the previous solution to correspond to $LLCS(A, B_{1..2})$, and so on until $LLCS(A, B_{1..n})$. This is useful for example if we wish to generate the set of all strings B for which it is true that $LLCS(A, B) \geq \alpha$, where α is some threshold. Such a setting is feasible for example within the context of indexed approximate matching, as proposed by Myers [14][2]. For a pattern (piece) A, the method of Myers generates a set of interesting strings by performing a depth-first search (DFS) over a conceptional trie that contains all possible strings. During the DFS, A is always compared to $B_{1..j}$, the string that corresponds to the current node in the trie. Maintaining this information in an efficient manner essentially requires an online algorithm: when stepping from the node of $B_{1..j}$ to one of its child nodes, which corresponds to some $B_{1..j+1}$, the comparison information about $LLCS(A, B_{1..j})$ should be updated to correspond to $LLCS(A, B_{1..j+1})$.

[2] Myers considered edit distance, but a similar scheme can be used with LCS.

Typically the space complexities of LCS algorithms coincide with their time complexities if we wish to construct a string from the set $LCS(A,B)$[3]. The divide-and-conquer scheme of Hirschberg [6] is a classic method to save space. It can be used with several LCS algorithms in such manner that the value $LLCS(A,B)$ and a string from $LCS(A,B)$ can be computed in linear space while the original asymptotic time complexity of the LCS algorithm is preserved. This space saving scheme is not simple to use with Rick's algorithm. Goeman and Clausen [5] proposed their own $\mathcal{O}(\sigma n + \min\{Lm, L(n-L)\})$ variant of Rick's algorithm and showed how to modify the algorithm to use $\mathcal{O}(\sigma n)$ space, ie. linear space when the alphabet size σ is constant. Their linear space scheme, however, changed the time complexity to $\mathcal{O}(\sigma n + \min\{Lm, m\log m + L(n-L)\})$. Finally, Rick [16] proposed another linear space variant that was able to maintain the $\mathcal{O}(\sigma n + \min\{Lm, L(n-L)\})$ time complexity of his original algorithm.

In this paper we propose an LCS algorithm that has $\mathcal{O}(\sigma m + \min\{Lm, L(n-L)\})$ time complexity, same as the algorithms by Rick [15] and Goeman and Clausen [5]. Our algorithm, however, has some advantages. First of all we find it a bit more simple than the previous two, which may be an important consideration in practice. Perhaps the most significant difference is that our algorithm needs to preprocess only the string A, and it furhermore works in online manner. As discussed above, there are situations where these properties are important. The proposed algorithm is also straight-forward to use within the linear-space divide-and-conquer scheme of Hirschberg while preserving the $\mathcal{O}(\sigma m + \min\{Lm, L(n-L)\})$ time complexity. And as last we mention that the underlying principle behind our algorithm can also be used quite directly within the setting of incremental string comparison [12,10].

2 Preliminaries

2.1 Dynamic programming

The basic $\mathcal{O}(mn)$ dynamic programming solution for the LCS problem is based on filling an $(m+1)\times(n+1)$ dynamic programming matrix D in such manner, that eventually each cell $D[i,j]$ holds the value $D[i,j] = LLCS(A_{1..i}, B_{1..j})$. This can be done using the well-known rules that are shown in Recurrence 1.

Recurrence 1.

$$\text{When } 0 \le i \le m \text{ and } 0 \le j \le n:$$
$$D[i,j] = \begin{cases} 0, \text{ if } i=0 \text{ or } j=0, \\ D[i-1,j-1]+1, \text{ if } A_i = B_j, \text{ and otherwise} \\ \max\{D[i-1,j], D[i,j-1]\}. \end{cases}$$

In the end, the desired LCS length $LLCS(A,B)$ is found in the cell $D[m,n]$. The matrix D is usually filled either in column- or rowwise order. If we are interested only in the value $LLCS(A,B) = D[m,n]$, for example a rowwise filling process needs to store only the currently filled row i and the previous row $i-1$, which means that only linear space is needed. A string in $LCS(A,B)$ can be constructed by backtracking along legal values from the cell $D[m,n]$ to the cell $D[0,0]$ in the filled matrix D. Any such legal path from $D[m,n]$ to $D[0,0]$ represents a string in $LCS(A,B)$. The

[3] If only the value $LLCS(A,B)$ is required, most algorithms can be modified to use much less space.

characters of the string are determined in reverse order by the diagonal steps along the path from $D[i,j]$ to $D[i-1,j-1]$ (meaning that $A_i = B_j$ is included in the subsequence). This backtracking process needs $\mathcal{O}(mn)$ space to recover a string in $LCS(A,B)$ as it needs to store the whole matrix D.

2.2 Linear space construction of a longest common subsequence

Hirschberg [6] proposed a divide-and-conquer scheme that can construct a string in $LCS(A,B)$ in linear space. Let $\overleftarrow{A}$ and $\overleftarrow{B}$ denote the reverse strings of A and B. That is, $\overleftarrow{A}_i = A_{m-i+1}$ and $\overleftarrow{B}_j = B_{n-j+1}$ for $1 \le i \le m$ and $1 \le j \le n$. Also let $\overleftarrow{D}$ denote dynamic programming matrix that has been filled using strings $\overleftarrow{A}$ and $\overleftarrow{B}$ instead of A and B. The method finds the middle point of a backtracking path, and then proceeds recursively. The first step is to compute the the values $D[\lfloor \frac{m}{2} \rfloor, j]$ and $\overleftarrow{D}[\lceil \frac{m}{2} \rceil, j]$ for $j = 1, \ldots, n$, where $\lfloor \frac{m}{2} \rfloor$ is a chosen middle row. This information can be computed in $\mathcal{O}(m+n)$ space with rowwise filling order. It can be shown that a backtracking path goes through those cells $D[\lfloor \frac{m}{2} \rfloor, k]$ for which the sum $D[\lfloor \frac{m}{2} \rfloor, k] + \overleftarrow{D}[\lceil \frac{m}{2} \rceil, n-k+1]$ is maximal. Finding one such k takes linear time. After that the recursion proceeds to find the midpoints in the two submatrices that correspond to comparing the string $A_{1..\lfloor \frac{m}{2} \rfloor}$ with $B_{1..k}$ and the string $A_{\lfloor \frac{m}{2} \rfloor+1..m}$ with $B_{k+1..n}$, respectively. The subsequence can be constructed during this process (see [6]). The total work is directly proportional to the total number of cells filled in the dynamic programming matrices. And this number is $\approx \Sigma_{h=0}^{\log_2 m} \frac{1}{2^h} mn < 2mn = \mathcal{O}(mn)$, ie. the total work is at most roughly twice as much as in the basic dynamic programming algorithm.

2.3 Incremental encoding of the dynamic programming matrix

Lemma 1 states well-known properties of adjacent values in D.

Lemma 1. *Let D be a dynamic programming matrix that contains the values $D[i,j] = LLCS(A_{1..i}, B_{1..j})$ for $0 \le i \le m$ and $0 \le j \le n$. Then the following three properties hold for $1 \le i \le m$ and $1 \le j \le n$:*

1. *D[i,j] = D[i-1,j] or D[i,j] = D[i-1,j] + 1*
2. *D[i,j] = D[i,j-1] or D[i,j] = D[i,j-1] + 1*
3. *D[i,j] = D[i-1,j-1] or D[i,j] = D[i-1,j-1] + 1*

Hunt and Szymanski [9] gave an $\mathcal{O}(M \log L + n \log \sigma)$ algorithm that uses these properties. Here M denotes the number of match points between A and B, ie. $M = |\{(i,j) \mid A_i = B_j, 1 \le i \le m, 1 \le j \le n\}|$. Two relevant variants of the algorithm of Hunt And Szymanski are the $\mathcal{O}(\sigma n + Lm)$ algorithm of Apostolico and Guerra [2] and the $\mathcal{O}(M + Lm + n \log \sigma)$ algorithm of Kuo and Cross [11].

All these algorithms represent the dynamic programming matrix D in an incremental manner. When we move from the cell $D[i,j-1]$ to the cell $D[i,j]$, Lemma 1 states that the value of the current cell either remains the same or grows by one. Let us define $R_i[k]$ as the smallest column j where $D[i,j] = k$. Such a column j exists for $k = 0, \ldots, D[i,n]$. It is convenient to define also special sentinel values $R_i[k] = n+1$ for $k > D[i,n]$. Now when $0 \le k \le D[i,n]$, the values $R_i[k]$ represent the values $D[i,j]$ according to the relationship $D[i,j] = k$ for $j = R_i[k], \ldots, R_i[k+1]-1$. In addition, the equality $D[i, R_i[k]] = k = D[i, R_i[k]-1] + 1$ holds when $1 \le k \le D[i,n]$. Due to this latter rule, we may view the values $R_i[k]$ as *increment points*, although this is awkward in the case of the first point $R_i[0] = 0$, which does not have a previous value "$D[i, -1] = -1$" to increment. The left side of Fig. 1 shows an example of increment points.

The values $R_i[k]$ may be computed according to Recurrence 2.

Recurrence 2. (Based on Hunt and Szymanski [9])

$$\text{When } 0 \le i \le m:$$

$$R_i[k] = \begin{cases} 0, \text{ if } k = 0, \\ n+1, \text{ if } i = 0 \text{ and } k > 0, \text{ and otherwise} \\ \min\{\, j \mid (A_i = B_j \text{ and } R_{i-1}[k-1] < j < R_{i-1}[k]) \text{ or } j = R_{i-1}[k]\}. \end{cases}$$

Several LCS algorithms use either a *MatchList* or a *NextMatch*[4] auxiliary data structure (see e.g. [12]).

MatchList is a vector of length m, where the entry *MatchList*$[i]$ points to a linked list that contains in sorted order indices j where $A_i = B_j$. This data structure takes overall $\mathcal{O}(m+n)$ space and can be constructed in $\mathcal{O}(m \log \sigma)$ time.

NextMatch is an $n \times \sigma$ matrix. For a given character $a \in \Sigma$, the entry *NextMatch*$[i, a]$ gives smallest k that is larger than i and where $B_k = a$. It is convenient to use $n+1$ as a sentinel value if such k does not exist. So more formally *NextMatch*$[i, a] = \min\{k \mid (k > i \text{ and } B_k = a) \text{ or } k = n+1\}$. *NextMatch* can be constructed in $\mathcal{O}(\sigma n)$ time and space.

The algorithm of Hunt and Szymanski [9] uses the *MatchList* data structure and stores the increment points $R_i[k]$ of row i consecutively and in sorted order in an array. Note that one row has at most $\mathcal{O}(L)$ increment points. The algorithm processes row i after row $i-1$. In order to compute the values $R_i[k]$, the list *MatchList*$[i]$ is processed sequentially. At each match column $j \in$ *MatchList*$[i]$, the algorithm uses an $\mathcal{O}(\log L)$ binary search in the array of the values $R_{i-1}[k]$ to check if the condition $R_{i-1}[k-1] < j \le R_{i-1}[k]$ of Recurrence 2 holds for some k, and then updates the increment points accordingly. This process takes overall $\mathcal{O}(M \log L + n \log \sigma)$ time, which includes preprocessing the *MatchList* data structure.

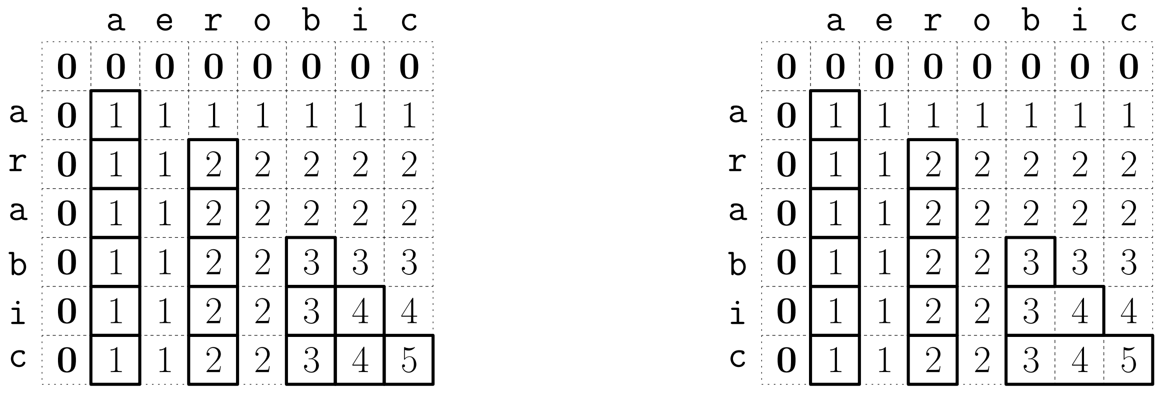
The algorithm of Kuo and Cross [11] is quite similar to the algorithm of Hunt and Szymanski. The difference is that the condition $R_{i-1}[k-1] < j \le R_{i-1}[k]$ of Recurrence 2 is checked at each step while going through the list *MatchList*$[i]$ and a sorted list of values $R_{i-1}[k]$ in parallel. The overall number of steps in this process is limited by the total number of match points and increment points. Since the former number equals M and the latter number is at most Lm, the overall time is $\mathcal{O}(M + Lm + n \log \sigma)$ when also preprocessing *MatchList* is included.

The algorithm of Apostolico and Guerra [2] uses the *NextMatch* matrix. When processing row i, the values $R_{i-1}[k]$ are considered in increasing order. For given $R_{i-1}[k-1]$ and $R_{i-1}[k]$, the existence of j that fulfills the conditions $A_i = B_j$ and $R_{i-1}[k-1] < j \le R_{i-1}[k]$ of Recurrence 2 can be checked in $\mathcal{O}(1)$ time by consulting the value *NextMatch*$[R_{i-1}[k-1], A_i]$[5]. Now the number of steps is limited only by the number of increment points, and the overall time, including constructing *NextMatch*, is $\mathcal{O}(\sigma n + Lm)$.

[4] The *NextMatch* data structure is sometimes called *Closest*.

[5] It can be seen from Recurrence 2 that the condition $R_{i-1}[k-1] < j \le R_{i-1}[k]$ needs to be checked only once for each pair $R_{i-1}[k-1]$ and $R_{i-1}[k]$.

3 An algorithm using block encoding of the increment points



		a	e	r	o	b	i	c
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
a	**0**	1	1	1	1	1	1	1
r	**0**	1	1	2	2	2	2	2
a	**0**	1	1	2	2	2	2	2
b	**0**	1	1	2	2	3	3	3
i	**0**	1	1	2	2	3	4	4
c	**0**	1	1	2	2	3	4	5

		a	e	r	o	b	i	c
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
a	**0**	1	1	1	1	1	1	1
r	**0**	1	1	2	2	2	2	2
a	**0**	1	1	2	2	2	2	2
b	**0**	1	1	2	2	3	3	3
i	**0**	1	1	2	2	3	4	4
c	**0**	1	1	2	2	3	4	5

Figure 1. An example with A = "arabic" and B = "aerobic". The left side shows each cell $D[i,j]$, which has an increment point $R_i[k]$, in bold. The right side shows the cells $D[i,j]$ within a block β_i enclosed in a bold rectangle.

We propose to store the increment points $R_i[k]$ using a kind of block encoding. The idea is in some sense similar to the well-known run-length encoding used in data compression. We will define β_i as a list that stores the values $R_i[k]$ of row i using block encoding. Each block item in β_i is a pair of integers (s,e). This value tells that there exists some k for which $R_i[k] = s$ and $R_i[k+h] = s+h$ for $h = 0,\ldots,e-s$. That is, there is a block of consecutive increment points starting from column $s = R_i[k]$ and ending at column $e = R_i[k+e-s]$. We also require that each block (s,e) is maximal: if $k > 0$, then $R_i[k-1] < s-1$, and if $k+e-s < D[i,n]$, then $R_i[k+e-s+1] > e$. The right side of Fig. 1 shows an example.

It is convenient to define that each list β_i has in its end a special sentinel block $(n+1,n+1)$. The sentinel delimits the end of row i and does not correspond to any real increment point. The sentinel blocks also never change, and they for example cannot be merged with another block if column n contains an increment point. A constructive definition of the list β_i is as follows:

1. The initial case $k = 0$: Set the pair $(R_i[0], 0) = (0,0)$ as the first item in β_i.
2. The general case $1 \le k \le D[i,n]$: Let (s,e) be the last item in the list β_i before processing $R_i[k]$.
   a) If $R_i[k] = e+1$, replace the item (s,e) with $(s,e+1)$ in the list β_i.
   b) If $R_i[k] > e+1$, insert the new item $(R_i[k], R_i[k])$ to the end of the list β_i.
3. The sentinel corresponding to $k = D[i,n]+1$: Insert the pair $(n+1,n+1)$ to the end of the list β_i.

From here on we will use the notation (s^i_q, e^i_q) to denote the qth item in the list β_i. Using this notation, a list β_i with r items may be expressed as $\beta_i = ((s^i_1, e^i_1),\ldots,(s^i_r, e^i_r))$. Note that a complete list β_i always has at least two blocks. In addition we will use the notation (s^i_ℓ, e^i_ℓ) to denote the last block in the current, possibly only partially completed list β_i.

Initially at row 0 we know that $\beta_0 = ((0,0),(n+1,n+1))$. Algorithm 1 describes how the list β_i can be constructed from the list β_{i-1}.

Algorithm 1. Assume that we are given the list $\beta_{i-1} = ((s_1^{i-1}, e_1^{i-1}), \ldots, (s_r^{i-1}, e_r^{i-1}))$ that contains r items and represents all increment points $R_{i-1}[k]$ of row $i-1$. Then the list β_i that represents all increment points $R_i[k]$ of row i can be formed correctly by using the following steps:

```
1. Initially set the list $\beta_i$ to be empty.
2. The first case $q = 1$:
   Insert the item $(s_1^{i-1}, e_1^{i-1}) = (0, e_1^{i-1})$ to $\beta_i$.
3. For $q = 2, \ldots, r$:
   Set $j = \mathit{NextMatch}[e_{q-1}^{i-1}, A_i]$. There are the following subcases:
   a) If $e_{q-1}^{i-1} < j < s_q^{i-1}$, then:
        i) If $j > e_\ell^i$, insert the block $(j, j)$ to the end of $\beta_i$.
       ii) If $j = e_\ell^i + 1$, replace $(s_\ell^i, e_\ell^i)$ with $(s_\ell^i, e_\ell^i + 1)$ in the list $\beta_i$.
      iii) After processing case $i$ or $ii$, insert the block
             $(\min\{n+1, s_q^{i-1}+1\}, e_q^{i-1})$ to the end of $\beta_i$ if $\min\{n+1, s_q^{i-1}+1\} \le e_q^{i-1}$.
   b) If $j \ge s_q^{i-1}$ and $q < r$, insert the block $(s_q^{i-1}, e_q^{i-1})$ to the end of $\beta_i$.
```

Theorem 2. *Algorithm 1 builds the list β_i correctly.*

Proof. It is not difficult to show that Algorithm 1 follows the principles of Recurrence 2. Fig. 2 illustrates the process.

The case $q = 1$ can be seen to be correct. When $q > 1$ and Algorithm 1 begins processing the block (s_q^{i-1}, e_q^{i-1}), it can be shown that within the column interval $(e_\ell^i, \ldots, e_{q-1}^{i-1})$ and (s_q^{i-1}, e_q^{i-1}), β_i should contain increment points in the column $j = \mathit{NextMatch}[e_{q-1}^{i-1}, A_i]$ and in the columns $j = s_q^{i-1}+1, \ldots, e_q^{i-1}$ (if $s_q^{i-1}+1 \le e_q^{i-1}$). Figs. 2*b*, 2*c* and 2*d*, correspond, respectively, to the case 3*b*, the subcase *i* of the case 3*a*, and the subcase *ii* of the case 3*a* in Algorithm 1.

Note that when processing the next q, the newly created block (s_ℓ^i, e_ℓ^i) may be appended to contain one more increment point in the subcase *ii* of the case 3*a* in Algorithm 1. This is reflected in Figs. 2*b* - 2*d* in how the value in column $e_{q-1}^{i-1}+1$ remains undecided.

We omit a more detailed proof from this version of the paper. □

Theorem 3. *The time complexity of Algorithm 1 is* $\mathcal{O}(\min\{Lm, L(n-L)\})$.

Proof. Fig. 3 illustrates the proof. Since the time complexity of Algorithm 1 is directly proportional to the total number of blocks that it processes, we will find an upper bound for the number of the blocks.

We begin by considering some column interval $j = u, \ldots, v$ on row i, where $0 \le u \le v \le n$. Let $\#_i(u,v)$ denote the number of increment points and $\overline{\#}_i(u,v)$ denote the number of non-increment points within these columns. Clearly $\#_i(u,v) + \overline{\#}_i(u,v) = v-u+1$, since each column j either does or does not contain an increment point $R_i[k]$ for some k.

Since each maximal block of consecutive increment points contains at least one increment point, the number of blocks that appear, even partially, within columns $j = u, \ldots, v$ is bounded by $\#_i(u,v)$. On the other hand, each maximal block is followed by a non-increment point or the end of the considered region. Hence the number of blocks within columns $j = u, \ldots, v$ is bounded also by $\overline{\#}_i(u,v)+1$.

Let us first analyse the first $z = m-L$ rows. Row i has $\#_i(0,n) = D[i,n]$ increment points, and from Lemma 1 we know that $D[i,n] \le i$. Hence β_i contains at most i

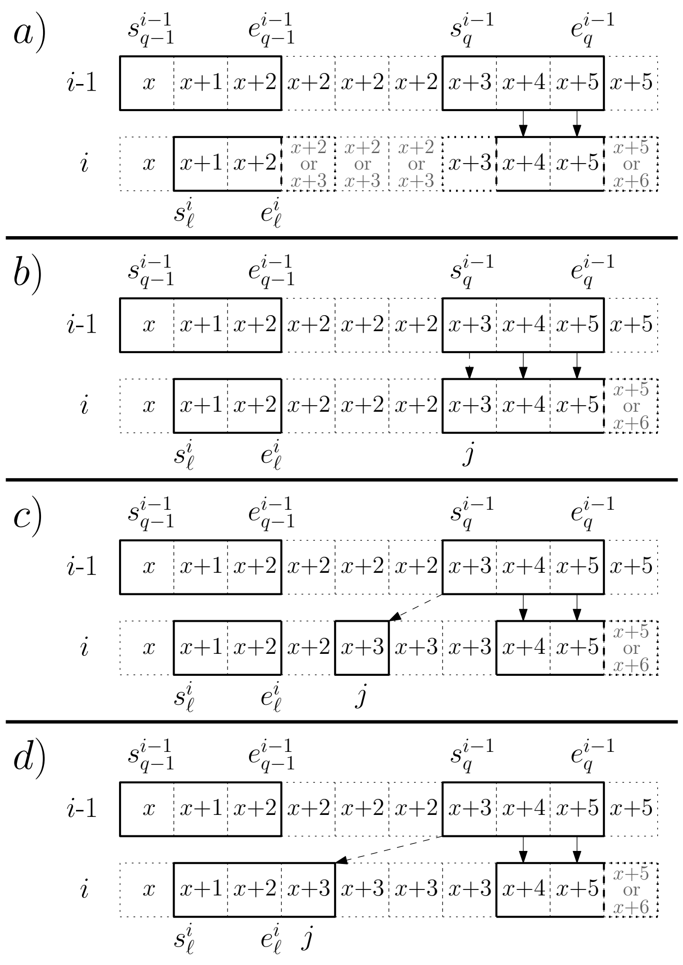


Figure 2. Figures a) - d) illustrate processing the block (s^{i-1}_q, e^{i-1}_q). Here x denotes the value $D[i-1, s^{i-1}_q]$, and the bold rectangles enclose blocks of consecutive increment points. The solid arrows show increment blocks that must be inherited to row i in columns $s^{i-1}_q+1, \ldots, e^{i-1}_q$. The dashed arrow shows where the increment point at s^{i-1}_q moves.

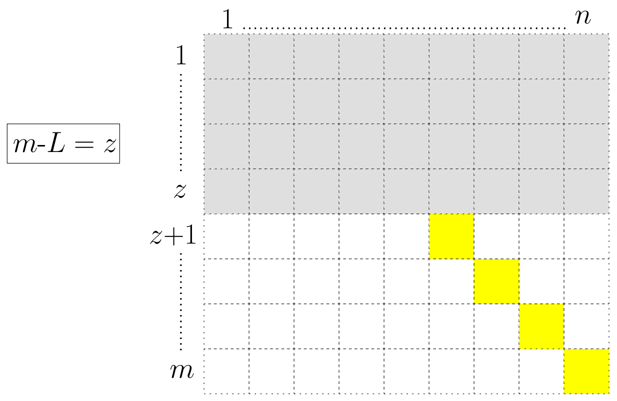


Figure 3. The figure illustrates the time complexity analysis of Algorithm 1.

blocks. This means that the lists β_i for rows $i = 0, \ldots, z$ contain at most a total of $\Sigma_{i=0}^{z} i = \frac{z(z+1)}{2}$ blocks.

Let us then consider the rows $z+i$ for $i = 1, \ldots, L$. Each such row contains at most $\#_{z+i}(0, n) \le z + i$ increment points. On the other hand, at least i of the increment points must be located within the first $n-L+i$ columns, ie. $\#_i(0, n-L+i) \ge i$. This is because the condition $LLCS(A, B) = L$ requires that $LLCS(A_{1..m-L+i}, B_{1..n-L+i}) = D[z + i, n - L + i] \ge i$ (this is not difficult to prove by using Lemma 1). Now $\overline{\#}_i(0, n - L + i) = n - L + i + 1 - \#_i(0, n - L + i) \le n - L + 1$, so columns $j = 0, \ldots, n-L+i$ of row $z+i$ contain at most $n-L+1$ blocks. If $i < L$, the remaining columns $j = n - L + i + 1, \ldots, n$ of row i contain at most $\#_{z+i}(n - L + i + 1, n) = \#_{z+i}(0, n) - \#_{z+i}(0, n - L + i) \le z$ blocks. Hence the total number of blocks on row $z + i$ is bounded by $n - L + 1 + z$.

Based on the preceding discussion, the total number of blocks on all rows $i = 0, \ldots, m$ is at most $\frac{z(z+1)}{2} + L(n - L + 1 + z)$. We note that $\frac{z(z+1)}{2} \le m(m - L) = \mathcal{O}(m(n - L))$, and that $L(n - L + 1 + z) = L(n - L + 1 + m - L) \le L(2n - 2L + 1) = \mathcal{O}(L(n - L)) = \mathcal{O}(m(n - L))$.

On the other hand $\#_i(0, n) \le L$ for all $i = 0, \ldots, m$, and so the total number of blocks has also the bound $\mathcal{O}(Lm)$.

By combining the previous two bounds, we have that the total number of blocks is bounded by $\mathcal{O}(\min\{Lm, m(n - L)\}$. If $n - L < L$, then $\frac{m}{2} \le \frac{n}{2} < L$, ie. $m = \mathcal{O}(L)$. This implies that $\mathcal{O}(\min\{Lm, m(n - L)\} = \mathcal{O}(\min\{Lm, L(n - L)\}$, since the choice $m(n - L)$ is smaller than Lm only when $m = \mathcal{O}(L)$.

Hence we have reached the conclusion that the asymptotic time complexity of Algorithm 1 may be stated in the form $\mathcal{O}(\min\{Lm, L(n - L)\}$. When also preprocessing of *NextMatch* is taken into account, the time complexity becomes $\mathcal{O}(\sigma n + \min\{Lm, L(n - L)\}$. □

3.1 Constructing a longest common subsequence in $\mathcal{O}(\sigma n)$ space

We briefly sketch how to use Algorithm 1 in the divide-and-conquer scheme discussed in Section 2.2 in order to construct a string from the set $LCS(A, B)$ using $\mathcal{O}(\sigma n + \min\{Lm, L(n - L)\}$ time and $\mathcal{O}(\sigma n)$ space. The required space is determined by *NextMatch* and is linear for constant σ.

Let $\overleftarrow{\beta}_i$ denote the block list for row i of $\overleftarrow{D}$ that corresponds to the reverse strings $\overleftarrow{A}$ and $\overleftarrow{B}$. Algorithm 1 can produce both middle-row lists $\beta_{\lfloor \frac{m}{2} \rfloor}$ and $\overleftarrow{\beta}_{\lceil \frac{m}{2} \rceil}$ in $\mathcal{O}(\min\{Lm, m(n-L)\}$ time. A column k where the sum $D[\lfloor \frac{m}{2} \rfloor, k] + \overleftarrow{D}[\lceil \frac{m}{2} \rceil, n-k+1]$ is maximal can be found in $\mathcal{O}(L)$ time by merging the size-$\mathcal{O}(L)$ lists $\beta_{\lfloor \frac{m}{2} \rfloor}$ and $\overleftarrow{\beta}_{\lceil \frac{m}{2} \rceil}$ in such manner that the other list is processed in reverse order. Overall we may state that the process has an upper bound of $c \min\{Lm, m(n - L)\}$ operations for some constant c.

Then the divide and conquer scheme does the same process for the string-pairs $(A_{1..\lfloor \frac{m}{2} \rfloor},\ B_{1..k})$ and $(A_{\lfloor \frac{m}{2} \rfloor+1..m},\ B_{k+1..n})$. Handling these takes at most $c(\min\{L_1 \frac{m}{2}, \frac{m}{2}(k - 1 - L_1)\} + c(\min\{L_2 \frac{m}{2}, \frac{m}{2}(n - k + 1 - L_2)\}$ operations, where $L_1 = LLCS((A_{1..\lceil \frac{m}{2} \rceil-1}, B_{1..k-1})$, $L_2 = LLCS(A_{\lceil \frac{m}{2} \rceil+1..m}, B_{k+1..n})$, and $L_1 + L_2 = L$. The sum of minimal choices in these two min-clauses is obviously never larger than a sum of two arbitrary choices. Therefore the overall value is limited above by both $c(L_1 \frac{m}{2} + L_2 \frac{m}{2}) = cL\frac{m}{2} = \frac{1}{2}cLm$ and $c(\frac{m}{2}(k - 1 - L_1) + \frac{m}{2}(n - k + 1 - L_2)) = c(\frac{m}{2}(k - 1 - L_1 + n - k + 1 - L_2)) = \frac{1}{2}cm(n - L)$, which results in the upper bound

$\frac{1}{2}c\min\{Lm, m(n-L)\}$ for the operations done in the second stage. By continuing the same analysis, the result is that the overall number of operations done during the divide-and-conquer scheme has an asymptotic limit $\Sigma_{h=0}^{\log_2 m}\frac{1}{2^h}c\min\{Lm, m(n-L)\} < 2c\min\{Lm, m(n-L)\} = \mathcal{O}(\min\{Lm, m(n-L)\}) = \mathcal{O}(\min\{Lm, L(n-L)\}$. By employing simple index-readjustments, each stage can use the same *NextMatch* built for the complete strings A and B (and a sorresponding $\overleftarrow{NextMatch}$ for $\overleftarrow{A}$ and $\overleftarrow{B}$). Hence the preprocessing needs to be done only once, using $\mathcal{O}(\sigma n)$ time and space.

3.2 A remark on incremental string comparison

Landau et al. [12] proposed an algorithm (that Ishida et al. [10] later extended), which can handle an incremental version of the LCS problem: After computing $LLCS(A, B)$, we should next be able to compute either $LLCS(A, Bb)$ or $LLCS(A, bB)$, ie. a character b may be added to either end of B. The algorithm uses *NextMatch* and maintains all increment points of D that corresponds to comparing the current A and B. We do not go into more details in this paper but just briefly note that our block encoding can be used also in this setting with very few modifications. Then the overall space usage becomes $\mathcal{O}(\sigma n + \min\{Lm, L(n-L)\})$ instead of $\mathcal{O}(\sigma n + Lm)$, the time for computing $LLCS(A, bB)$ after $LLCS(A, B)$ remains the same, and the time for computing $LLCS(A, Bb)$ after $LLCS(A, B)$ becomes $\mathcal{O}(\min\{L, (n-L)\})$ instead of the $\mathcal{O}(L)$ time of the original scheme [10].

References

1. A. V. Aho, D. S. Hirschberg, and J. D. Ullman: *Bounds on the complexity of the longest common subsequence problem.* Journal of the ACM, 23(1) 1976, pp. 1–12.
2. A. Apostolico and C. Guerra: *The longest common subsequence problem revisited.* Algorithmica, 2 1987, pp. 316–336.
3. L. Bergroth, H. Hakonen, and T. Raita: *A survey of longest common subsequence algorithms*, in Proc. 7th String Processing and Information Retrieval (SPIRE 2000), 2000, pp. 39–48.
4. F. Y. L. Chin and C. K. Poon: *A fast algorithm for computing longest common subsequences of small alphabet size.* Journal of Information Processing, 13(4) 1990, pp. 463–469.
5. H. Goeman and C. Clausen: *A new practical linear space algorithm for the longest common subsequence problem.* Kybernetika, 38(1) 2002, pp. 45–66.
6. D. S. Hirschberg: *A linear space algorithm for computing maximal common subsequences.* Communications of the ACM, 18(6) 1975, pp. 341–343.
7. D. S. Hirschberg: *Algorithms for the longest common subsequence problem.* Journal of the ACM, 24(4) 1977, pp. 664–675.
8. D. S. Hirschberg: *An information-theoretic lower bound for the longest common subsequence problem.* Information Processing Letters, 7(1) 1978, pp. 40–41.
9. J. W. Hunt and T. G. Szymanski: *A fast algorithm for computing longest subsequences.* Communications of the ACM, 20(5) 1977, pp. 350–353.
10. Y. Ishida, S. Inenaga, A. Shinohara, and M. Takeda: *Fully incremental lcs computation*, in Proc. 15th Fundamentals of Computation Theory (FCT 2005), vol. 3623 of Lecture Notes in Computer Science, 2005, pp. 563–574.
11. S. Kuo and G. R. Cross: *An improved algorithm to find the length of the longest common subsequence of two strings.* ACM SIGIR Forum, 23(3-4) 1989, pp. 89–99.
12. G. M. Landau, E. W. Myers, and M. Ziv-Ukelson: *Two algorithms for lcs consecutive suffix alignment.* Journal of Computer and System Sciences, 73(7) 2007, pp. 1095–1117.
13. W. J. Masek and M. Paterson: *A faster algorithm for computing string edit distances.* Journal of Computer and System Sciences, 20 1980, pp. 18–31.
14. E. W. Myers: *A sublinear algorithm for approximate keyword searching.* Algorithmica, 12(4) 1994, pp. 345–374.

15. C. Rick: *A new flexible algorithm for the longest common subsequence problem*, in Proc. 6th Combinatorial Pattern Matching (CPM'95), vol. 937 of Lecture Notes in Computer Science, 1995, pp. 340–351.
16. C. Rick: *Simple and fast linear space computation of longest common subsequences.* Information Processing Letters, 75(6) 2000, pp. 275–281.
17. R. A. Wagner and M. J. Fischer: *The string-to-string correction problem.* Journal of the ACM, 21(1) 1974, pp. 168–173.
18. S. Wu, U. Manber, G. Myers, and W. Miller: *An $\mathcal{O}(NP)$ sequence comparison algorithm.* Information Processing Letters, 35 1990, pp. 317–323.