Faster Algorithms for Computing Longest Common Increasing Subsequences

Martin Kutz*[1], Gerth Stølting Brodal**[2], Kanela Kaligosi[1], and Irit Katriel[3]

[1] Max-Plank-Institut für Informatik, Saarbrücken, Germany.
kaligosi@mpi-inf.mpg.de.
[2] MADALGO***, Aarhus University, Aarhus, Denmark.
gerth@cs.au.dk.
[3] Aarhus University, Aarhus, Denmark.
irit@cs.au.dk.

Abstract. We present algorithms for finding a longest common increasing subsequence of two or more input sequences. For two sequences of lengths n and m, where $m \geq n$, we present an algorithm with an output-dependent expected running time of $O((m + n\ell)\log\log\sigma + \mathit{Sort})$ and $O(m)$ space, where ℓ is the length of an LCIS, σ is the size of the alphabet, and *Sort* is the time to sort each input sequence. For $k \geq 3$ length-n sequences we present an algorithm which improves the previous best bound by more than a factor k for many inputs. In both cases, our algorithms are conceptually quite simple but rely on existing sophisticated data structures. Finally, we introduce the problem of longest common weakly-increasing (or non-decreasing) subsequences (LCWIS), for which we present an $O(\min\{m + n\log n, m\log\log m\})$-time algorithm for the 3-letter alphabet case. For the extensively studied longest common subsequence problem, comparable speedups have not been achieved for small alphabets.

Keywords: Common subsequences; Increasing subsequences ; Small alphabets; van Emde Boas trees.

1 Introduction

Algorithms that search for the longest common subsequence (LCS) of two input sequences or the longest increasing subsequence (LIS) of one input sequence date back several decades.

Formally, given two sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_m)$ with elements from an alphabet Σ and with $m \geq n$, a *common subsequence* of A and B is a subsequence $(a_{j_1} = b_{\kappa_1}, a_{j_2} = b_{\kappa_2}, \ldots a_{j_\ell} = b_{\kappa_\ell})$, where $j_1 < j_2 < \cdots < j_\ell$ and $\kappa_1 < \kappa_2 < \cdots < \kappa_\ell$. Given one sequence $A = (a_1, \ldots, a_n)$ where the a_i's are drawn from a totally ordered set, an *increasing subsequence* of A is a subsequence $(a_{j_1}, a_{j_2}, \ldots, a_{j_\ell})$ such that $j_1 < j_2 < \cdots < j_\ell$ and $a_{j_1} < a_{j_2} < \cdots < a_{j_\ell}$.

A classic algorithm by Wagner and Fischer [13] solves the LCS problem using dynamic programming in $O(mn)$ time and space. Hirschberg [7] reduced the space complexity to $O(n)$, using a divide-and-conquer approach. The fastest known algorithm by Masek and Paterson [9] runs in $O(n^2/\log n)$ time. Faster algorithms are known for special cases, such as when the input consists of permutations or when the output is known to be very long or very short. Hunt and Szymanski [8] studied the complexity of the LCS problem in terms of matching index pairs, i.e., they defined r

* Our friend and colleague Martin Kutz died in tragic circumstances while this manuscript was under review.

** Supported by the Danish Natural Science Research Council (grant # 21-04-0389).

*** Center for Massive Data Algorithmics - a Center of the Danish National Research Foundation.

to be the number of index-pairs (i, j) with $a_i = b_j$ (such a pair is called a *match*) and designed an algorithm that finds the LCS of two sequences in $O(r \log n)$ time. For a survey on the LCS problem see [2].

Fredman [5] showed how to compute an LIS of a length-n sequence in optimal $O(n \log n)$ time. When the input sequence is a permutation of $\{1, \ldots, n\}$, Hunt and Szymanski [8] designed an $O(n \log \log n)$-time solution, which was later simplified by Bespamyatnikh and Segal [3]. The expected length of a longest increasing subsequence of a random permutation has been shown (after successive improvements) to be $2\sqrt{n} - o(\sqrt{n})$; for a survey see [1].

Note that after sorting both input sequences we can in linear time remove symbols that do not appear in both sequences and rename the remaining symbols of the alphabet to $\{1, 2, \ldots, \sigma\}$. We can therefore assume that this preprocessing stage was performed and hence the size of the alphabet, σ, is at most n. In the following we let $Sort_\Sigma(m)$ denote the time required to sort a length-m input sequence drawn from the alphabet Σ.

Recently, Yang et al. [15] combined the two concepts and defined a *common increasing subsequence* (CIS) of two sequences A and B, i.e., an increasing sequence that is a subsequence of both A and B. They designed a dynamic programming algorithm that finds a *longest CIS* (an LCIS, for short) of A and B using $\Theta(mn)$ time and space. Sakai [11] showed that Hirschberg's technique for LCS [7] can be adapted to reduce the space complexity to $O(m)$.

Subsequently, Chan et al. [4] obtained an upper bound of $O(\min\{r \log \sigma, m\sigma + r\} \log \log m + Sort_\Sigma(m))$. The number of matches r is in the worst case $\Omega(mn)$, but in some important cases it is much smaller. For instance, when A and B are permutations of $\{1, \ldots, n\}$ then $r = O(n)$. They then proceeded to generalize their algorithm to find an LCIS of $k \geq 3$ length-n sequences. They show that this can be done in $O(\min\{kr^2, kr \log \sigma \log^{k-1} r\} + kSort_\Sigma(n))$ time, where r is again the number of matches, i.e., k-coordinate vectors that contain an index from each input sequence, all with the same symbol.

1.1 Our results

In this paper we present three new upper bounds for the LCIS problem. The first is an output-dependent algorithm which runs in $O((m + n\ell) \log \log \sigma + Sort_\Sigma(m))$ expected time and $O(m)$ worst-case space, where ℓ is the length of an LCIS. Whenever $n = \Omega(\log \log \sigma + Sort_\Sigma(m)/m)$ and either $m = \Omega(n \log \log \sigma)$ or $\ell = o(n/ \log \log n)$, it is faster than Yang et al.'s $\Theta(mn)$-time algorithm.

For a strictly-increasing subsequence we have $\ell \leq \sigma$. However, in the weakly-increasing (i.e. non-decreasing) variant, the length of the output can be arbitrarily larger than the size of the alphabet. We show that a *longest common weakly-increasing subsequence* (LCWIS) can be found in linear time for an alphabet of size two and in $O(\min\{m + n \log n, m \log \log m\})$ time for an alphabet of size three. These results are interesting because they pinpoint what seems to be a fundamental difference between the LCS and LWCIS problems. The approaches we use to get the two bounds for 3-letter alphabets cannot be applied to LCS, and to date, comparable speedups have not been achieved for LCS with small alphabets.

Finally, we consider the case of $k \geq 3$ length-n sequences. The upper bound of Chan et al. is achieved by two algorithms; the first is a simple $O(kr^2 + kSort_\Sigma(n))$ time algorithm and the second is a more complex implementation of the same approach, which runs in $O(kr \log \sigma \log^{k-1} r + kSort_\Sigma(n))$ time. We describe an algorithm which is significantly simpler than the latter and obtain a running time of $O(\min\{kr^2, r \log^{k-1} r \log \log r\} + kSort_\Sigma(n))$.

Symbol	Meaning
n, m	Lengths of input sequences (we assume $m \geq n$).
ℓ	Length of the LCIS/LCWIS.
k	Number of input sequences.
σ	Size of the alphabet (number of different symbols).
r	Number of matches in the input sequences.

Table 1. Parameters of the LCIS/LCWIS problems.

	Previous Results (LCIS)	New (LCIS and LCWIS)
$k = 2$	$O(mn)$ [15] $O(\min\{r \log \sigma, m\sigma + r\} \log\log m$ $+ Sort_{\Sigma}(m))$ [4]	$O((m + n\ell) \log\log \sigma + Sort_{\Sigma}(m))$ $O(m)$ when $\sigma = 2$ $O(\min\{m + n \log n, m \log\log m\})$ when $\sigma = 3$
$k \geq 3$	$O(\min\{kr^2, kr \log \sigma \log^{k-1} r\}$ $+kSort_{\Sigma}(n))$ [4]	$O(\min\{kr^2, r \log^{k-1} r \log\log r\}$ $+kSort_{\Sigma}(n))$

Table 2. Previous and new results. The new upper bounds apply to both LCIS and LCWIS.

Table 1 provides a list of the symbols used in the paper and Table 2 summarizes the previous and new results.

The rest of the paper is organized as follows. In Section 2 we describe a dynamic programming algorithm that uses a data structure based on van Emde Boas trees and runs in expected $O((m + n\ell) \log\log \sigma + Sort_{\Sigma}(m))$ time and $O(m)$ space. (In principle, the algorithm is deterministic, with the above running time. Randomization is only needed in order to obtain the linear space bound.)

In Section 3 we present our results on LCWIS with small alphabets, which use different techniques. Finally, in Section 4 we describe how to use a data structure by Gabow et al. [6] to obtain an algorithm for finding an LCIS or LCWIS of $k \geq 3$ sequences, which is simpler and faster than Chan et al.'s algorithm.

2 An Output-Dependent Upper Bound

2.1 Bounded heaps

In our output-dependent algorithm we need to access items that carry two integer parameters: *priorities* and *keys*. The basic query will be for the highest-priority element amongst all those whose keys are below a given threshold. We use a data structure, subsequently called a *bounded heap* (BH), that supports the following operations:

– $Insert(\mathcal{H}, s, p, d)$: Insert into the *BH* $\mathcal{H}$ the key s with priority p and associated data d.
– $DecreasePriority(\mathcal{H}, s, p, d)$: If the *BH* $\mathcal{H}$ does not already contain the key s, perform $Insert(\mathcal{H}, s, p, d)$. Otherwise, set this key's priority to $\min\{p, p'\}$, where p' is its previous priority.
– $BoundedMin(\mathcal{H}, s)$: Return the item that has minimum priority among all items in $\mathcal{H}$ with key smaller than s. If $\mathcal{H}$ does not contain any items with key smaller than s, return "invalid".

The priority search tree (PST) of McCreight [10] supports each of these operations in $O(\log n)$ time. However, the PST also allows deletions, which the BH is not required to support. Using van Emde Boas trees, we obtain a faster BH for integer keys:

Lemma 1. *There exists an implementation of bounded heaps that requires $O(n)$ space and supports each of the above operations in $O(\log \log n)$ amortized time, where keys are drawn from the set $\{1, \ldots, n\}$.*

Proof. The data structure applies standard techniques, such as those described in Section 3 of [6]. We rely on the fact that a snapshot of the heap, at any point in time, can be represented as a decreasing step function. More precisely, let $BM(s)$ be the value that would be returned by a $BoundedMin(\mathcal{H}, s)$ query. Then $BM(s) \le BM(s')$ whenever $s > s'$, i.e., the function BM is non-increasing in s (see Figure 1).

key s	1	2	3	4	5	6	7	8	9	10
priority	7	10	6	8	5	3	2	4	1	9
$BM(s)$	∞	7	7	6	6	5	3	2	2	1

Fig. 1. Example of *BM* values.

Assume that the keys are $s_1, s_2, \ldots$ with $s_i \le s_{i+1}$ for all i. To answer *BoundedMin* queries, it suffices to maintain a search structure that contains the $BM(s_i)$ value for every s_i at which the function BM changes, i.e., $BM(s_i) < BM(s_{i-1})$. Then, we answer a $BoundedMin(\mathcal{H}, s)$ by searching the data structure for the largest key which is at most s and returning its BM value. With a van Emde Boas tree [12] as search structure, this takes $O(\log \log n)$ time.

It remains to show how to support *Insert* and *DecreasePriority* operations in $O(\log \log n)$ amortized time. When the priority of a key s_i decreases to a new value of p, the following occurs:

1. $s_i + 1$ is inserted into the tree if $p < BM(s_{i^-})$, where s_{i^-} is the largest key in the tree which is smaller than s_i.
2. s_j is removed from the tree if $j > i$ and $BM(s_j) > p$.

With van Emde Boas trees, the two steps are handled, respectively, as follows.

1. Searching for s_{i^-}, checking whether s_i should be inserted and inserting it if so, takes $O(\log \log n)$ time.
2. Beginning at s_i, we repeatedly find the next item s_j in the tree (i.e., the smallest key larger than the current one) and remove it from the tree if $BM(s_j) > p$. The total time is $O(d \log \log n)$, where d is the number of items that were removed. Since the total number of items deleted by *DecreasePriority* operations is upper bounded by the total number of *Insert* operations, we can charge the cost of each deletion to the insertion of the same item, and obtain that the amortized cost of each operation is $O(\log \log n)$. □

2.2 An $O((m + n\ell) \log \log \sigma + Sort_{\Sigma}(m))$ time algorithm

Our output-dependent algorithm for the LCIS problem begins with a preprocessing step, where it removes from each sequence all elements that do not appear in the other sequence; this is easy after the sequences are sorted. For every remaining element s, it generates a sorted list Occ_s that contains ∞ and the indices of all occurrences of s in B.

Then, in n iterations the algorithm identifies common increasing subsequences (CISs) of increasing lengths: In iteration i it identifies length-i CISs (using the results of iteration $i-1$). More precisely, for every element a_j in A, it identifies the minimum index κ in B such that there is a length-i CIS which ends at a_j in A and at b_κ in B. The index κ is stored in $L_i[j]$.

To compute the array $L_1[1\ldots n]$, the algorithm traverses A and for each a_j, sets $L_1[j]$ to be the minimum index in the list Occ_{a_j}, i.e., the earliest occurrence of a_j in B. Note that due to the preprocessing, there exists such an index in B.

For $i > 1$, the ith iteration proceeds as follows. The algorithm traverses A again, and for every a_j, it checks whether a_j (together with some b_κ) can extend a length-$(i-1)$ CIS to a length-i CIS, and if so, identifies the minimum such κ. For this purpose, the algorithm maintains a bounded heap $\mathcal{H}$. When it begins processing a_j, $\mathcal{H}$ contains all elements $a_t \in \{a_1, \ldots, a_{j-1}\}$ for which $L_{i-1}[t] \neq \infty$. The key of a_t in $\mathcal{H}$ is a_t itself and its priority is $L_{i-1}[t]$, i.e., the minimum index of the endpoint in B of a length-$(i-1)$ CIS which ends, in A, at index t. The algorithm performs the query $\mathit{BoundedMin}(\mathcal{H}, a_j)$ to find the leftmost endpoint (in B) of a length-$(i-1)$ CIS, which contains only elements smaller than a_j. Let κ' be this endpoint. Then, $L_i[j]$ is set to the first occurrence of a_j in B which lies behind κ'; we prove that this is the leftmost endpoint in B of a length-i CIS which ends, in A, at a_j. A formal description of the algorithm is given in Figure 2.

We emphasize that $\mathcal{H}$ is built anew for every single pass. The only information saved between different scans of A and B is maintained in the arrays L_i.

The arrays $\mathit{Link}_1, \mathit{Link}_2, \ldots$ are used to save the information we need in order to construct the LCIS: Whenever we detect that the index pair (j, κ) can extend a length-$(i-1)$ CIS which ends at the index pair (j', κ'), we set $\mathit{Link}_i[j] = j'$. Finally, if there is a length-$(i-1)$ CIS which ends at a_j, then a_j is inserted into $\mathcal{H}$ with priority $L_{i-1}[a_j]$; it may later be extended into a length-i CIS by some $a_{j'}$ with $j' > j$.

Correctness: The correctness of the algorithm relies on the following lemma, which states that if there is a solution then the algorithm finds it. It is straightforward to show that the algorithm will not produce an invalid sequence.

Lemma 2. *Let A and B be two sequences that have a length-ℓ CIS which ends in A at index j and in B at index κ. Then at the end of the iteration in which $i = \ell$, $L_\ell[j] \leq \kappa$.*

Proof. By induction on ℓ. For $\ell = 1$, the claim is obvious. Assume that it holds for any length-$(\ell-1)$ CIS and that we are given A and B which have a length-ℓ CIS $c_1, \ldots, c_\ell$ that is located in A as $a_{j_1}, \ldots, a_{j_\ell}$ and in B as $b_{\kappa_1}, \ldots, b_{\kappa_\ell}$.

By the induction hypothesis, at the end of the $\ell-1$ iteration, $L_{\ell-1}$ contains entries that are not equal to ∞. Hence, the algorithm will proceed to perform iteration ℓ. Again by the induction hypothesis, $L_{\ell-1}[j_{\ell-1}] \leq \kappa_{\ell-1}$.

Since $a_{j_{\ell-1}} < a_{j_\ell}$, it is guaranteed that when $j = j_\ell$, $\mathcal{H}$ contains an item with key $a_{j_{\ell-1}}$, priority $\kappa' \leq \kappa_{\ell-1}$, and $d = (j_{\ell-1}, \kappa')$. So the $\mathit{BoundedMin}$ operation will return a valid value. If the value returned is $(j_{\ell-1}, \kappa_{\ell-1})$, then the smallest occurrence of a_ℓ in B after $\kappa_{\ell-1}$ is not beyond κ_ℓ. So the algorithm will set $L_\ell[j_\ell] \leq \kappa_\ell$. On the other hand, if the value returned is not $(j_{\ell-1}, \kappa_{\ell-1})$, then it is $(j_{\ell-1}, \kappa')$ for some $\kappa' \leq \kappa_{\ell-1}$. Since $a_{j'} < a_\ell$, again we get that the smallest occurrence of a_ℓ in B after $\kappa_{\ell-1}$ is not beyond κ_ℓ. So the algorithm will set $L_\ell[j_\ell] \leq \kappa_\ell$. □

```
Function LCIS($A = (a_1, \dots a_n), B = (b_1, \dots b_m)$)
    Preprocess (* Clean $A$ and $B$ and build $\mathrm{Occ}_s$ for every $s$ *)
    $i \leftarrow 1$

    (* Compute $L_1[1 \dots n]$ *)
    for $j = 1$ to $n$ do $L_1[j] \leftarrow \mathit{MinimumKey}(\mathrm{Occ}_{a_j})$

    (* Main loop *)
    do
        $\mathcal{H} \leftarrow []$ (* Empty Bounded Heap *)
        $i \leftarrow i + 1$
        for $j = 1$ to $n$ do
            $L_i[j] \leftarrow \infty$
            $(j', \kappa') \leftarrow \mathit{BoundedMin}(\mathcal{H}, a_j)$
            if $(j', \kappa') \neq$ "invalid" then
                $L_i[j] \leftarrow \min\{\kappa : \kappa \in \mathrm{Occ}_{a_j} \wedge \kappa > \kappa'\}$
                $\mathit{Link}_i[j] = j'$
            endif
            if $L_{i-1}[j] \neq \infty$ then
                (* Recall that DecreasePriority inserts $a_j$ if it is not already there *)
                $\mathit{DecreasePriority}(\mathcal{H}, a_j, L_{i-1}[j], (j, L_{i-1}[j]))$
            endif
        endfor
    while $i < n$ and $L_i[j] \neq \infty$ for some $j$

    (* Generate an LCIS in reverse order *)
    if $L_i[j] = \infty$ for all $j$ then $i \leftarrow i - 1$
    $j \leftarrow$ an index such that $L_i[j] \neq \infty$
    while $i > 0$ do
        output $a_j$
        $j \leftarrow \mathit{Link}_i[j]$
        $i \leftarrow i - 1$
    end while
end
```

Fig. 2. An $O((m + n\ell) \log\log\sigma + \mathit{Sort}_\Sigma(m))$ time LCIS algorithm for $k = 2$ sequences.

Time complexity: The preprocessing phase takes $O(\mathit{Sort}_\Sigma(m))$ time, to sort each of the sequences A and B. The construction of the Occ_s's takes $O(m)$ time. The array A is traversed $O(\ell)$ times. During each traversal, $O(n)$ operations are performed on the bounded heap, each of which takes $O(\log\log\sigma)$ amortized time, and the Occ_s lists are queried at most n times.

A naive implementation of the Occ_s lists would require $\Theta(\sigma m)$ time and space. We now sketch a possible randomized implementation that reduces this complexity to $O(m)$ time, and space.

Partition the range $\{1, \dots, m\}$ into m/σ blocks of σ consecutive locations and for every $1 \leq i \leq m/\sigma$ we denote by b_i the block containing locations $(i-1)\sigma + 1, \dots, i\sigma$. For each i and each $s \in \Sigma$ we create a data structure that represents occurrences of s in the block b_i and is based on Willard's y-fast tries [14], which use randomization. In addition, for each block we store the first occurrence of s succeeding the block. To answer a query for κ' in Occ_s, we first identify the block $b_{\lceil \kappa'/\sigma \rceil}$ containing the query point κ' in constant time. We then search for the smallest index larger than the query point κ' in the y-fast trie for this block in time $O(\log\log\sigma)$. If we found one, we are done. Otherwise, we return the first s succeeding the block, using the stored information. Initializing the m y-fast tries with a total of m elements takes $O(m \log\log\sigma)$ expected time. Note that this initialization step needs to be carried out only once.

In total, the main loop takes $O(m + n\ell \log\log\sigma)$ time. Finally, constructing the LCIS takes $O(\ell)$ time. We get that the total expected running time of the algorithm is $O((m + n\ell)\log\log\sigma + \mathit{Sort}_\Sigma(m))$.

Space complexity: As for space complexity, note that in the main loop we only use L_{i-1} and L_i. Therefore, we do not need to save the previous L's. In order to construct the LCIS, the algorithm as described requires $O(n\ell)$ space for the *Link* arrays.

However, we can reduce the space complexity to $O(m)$ with the technique developed by Hirschberg [7] for LCS. First, we run the algorithm once to compute ℓ (without constructing the *Link* arrays). Then we run a recursive version of the algorithm that construct the LCIS. The top recursive level invokes the usual algorithm, except that this time we remember only some of the *Link* information: Each match in the second half of a CIS knows the location in A and B of the $\lfloor \ell/2 \rfloor$-th match of the CIS that it was appended to. This information is found in the $\lfloor \ell/2 \rfloor$-th iteration of the main loop and propagated by the later iterations while the L arrays are constructed. Then, we know for every LCIS the location (i, j) in A and B of the middle match. We select one LCIS and recursively run the same algorithm to find the length-$\lfloor \ell/2 \rfloor - 1$ LCIS of $(a_1, \dots, a_{i-1})$ $(b_1, \dots, b_{j-1})$ and the length-$\lceil \ell/2 \rceil$ LCIS of $(a_{i+1}, \dots, a_n)$ and $(b_{j+1}, \dots, b_m)$. The base case is when we look for a constant-size LCIS. Then we run the original algorithm in linear space. To achieve that the time complexity remains unchanged we need to limit the work done processing B during the recursion. For the preprocessing for the outermost recursion we need time $\mathit{Sort}_\Sigma(m)$. For the remaining recursive calls we do not need to sort the arrays again and the preprocessing time is $O(m)$. The computation of a middle match considers at most matches involving $n\ell$ entries from B. These entries in B can be marked during the computation of the middle match, and only this subsequence of B is provided to the recursive calls. The thinning of B is done before each recursive call. Let $T(m, n, \ell)$ be the running time of the recursion on two sequences of lengths n and m with a length-ℓ LCIS and $m \leq n\ell$. Assume that the middle match is (n_1, m_1). Then $T(m, n, \ell) \leq n\ell \log\log\sigma + n\ell + T(m_1, n_1, \ell/2) + T(m_2, n_2, \ell/2)$, where $n_1 + n_2 + 1 = n$ and $m_1 + m_2 + 1 \leq m$. This recurrence solves to $O(n\ell \log\log\sigma)$. The total running time becomes $O((m + n\ell)\log\log\sigma + \mathit{Sort}_\Sigma(m))$. It is easy to see that the amount of space we need is $O(m)$.
In conclusion, we have shown:

Theorem 1. *An LCIS of two sequences of lengths m and n with $m \geq n$ can be found in $O((m + n\ell)\log\log\sigma + \mathit{Sort}_\Sigma(m))$ expected time and $O(m)$ worst-case space where ℓ is the length of the output and $\mathit{Sort}_\Sigma(m)$ is the time required to sort a length-m input sequence.*

3 Weakly-Increasing Subsequences

We now turn to longest common non-decreasing or *weakly-increasing* subsequences (LCWIS) for small alphabets. By simply replacing $<$ by $\leq$ in the *BoundedMin* operation in our algorithm for the LCIS problem, it is straightforward to verify that the algorithm solves the LCWIS problem in $O((m + n\ell)\log\log\sigma + \mathit{Sort}(m))$ expected time. But while the LCIS problem can be solved in linear time for alphabets of bounded size σ, simply because the length of the solution is then also bounded by σ, it is not clear how this fact should carry over to LCWIS, where the output size need not relate to σ at all.

We show how to solve LCWIS for the 2-letter alphabet in linear time and for the 3-letter alphabet in $O(\min\{m + n\log n, m\log\log m\})$ time; the latter complexity is achieved by two algorithms, each of which runs in time bounded by one of the terms inside the min. Since the complexities of the two algorithms are incomparable, the bounds they imply are both interesting. However, our main motivation for including both of them is that they use very different techniques, and it is not clear which of the approaches, if any, can be generalized to solve the problem over larger alphabets.

The results of this section come in contrast to the classic LCS problem, where already the 2-letter case seems to be essentially as hard as the general problem. In fact, it seems that LCWIS behaves very different from both LCIS and LCS.

3.1 Preprocessing

Let us use as our alphabet the Greek letters $\Sigma = \{\alpha, \beta, \gamma\}$ in their standard order: $\alpha < \beta < \gamma$. For both tasks, the 2-letter and 3-letter cases, we prepare arrays $\mathtt{Num}_{A,\alpha}$,$\mathtt{Num}_{B,\alpha}$,$\mathtt{Num}_{A,\beta}$, $\ldots$,$\mathtt{Num}_{B,\gamma}$ that count the number of αs, βs and γs, respectively, in prefixes of A and B. For example, the number of γs in A up to position 9 (inclusively) is stored in $\mathtt{Num}_{A,\gamma}[9]$. We also create arrays $\mathtt{Pos}_{A,\alpha}$ through $\mathtt{Pos}_{B,\gamma}$, which provide us with the position of the ith occurrence of α, β, or γ in A or B. These arrays can clearly be prepared in $O(m)$ time.

3.2 The 2-letter case is simple

After the preprocessing, the 2-letter case becomes trivial. For each i, where $0 \le i \le \min\{\mathtt{Num}_{A,\alpha}[n]$, $\mathtt{Num}_{B,\alpha}[m]\}$, we determine the position of the ith α in A and B and then the number of βs that come after those positions in the two sequences. This gives us, for every i, the length of an LCWIS of type $\alpha^i\beta^*$. The longest of them over all i are the LCWISs of the two sequences. The total time is $O(m)$.

3.3 The 3-letter case in $O(m + n\log n)$ time

The naïve extension of the above approach to three letters would have to deal with a quadratic number of tentative exponent pairs (i, j) for subsequences of type $\alpha^i\beta^j\gamma^*$. We somehow need to avoid the testing of all such pairs. The basis of our near-linear-time algorithm for a 3-letter alphabet are what we like to call "split-diagrams," a data structure that stores information about parts of the given sequences in a compact way.

Assume we were only interested in subsequences of A that have all their αs up to some fixed position s and all their γs strictly after s. Likewise, we only consider subsequences in B with all αs up to some position t and all γs strictly after that. We shall see that under these conditions, with a fixed *split* between αs and γs, it is possible to find an LCWIS in linear time.

Say, we try and see how long a sequence we can build if we started with exactly i many αs. We determine the ith pair of αs from the left and then count the number of βs in A and B up to the split (s, t). There are $p = \mathtt{Num}_{A,\beta}[s] - \mathtt{Num}_{A,\beta}[\mathtt{Pos}_{A,\alpha}[i]]$ such βs in A and $q = \mathtt{Num}_{B,\beta}[t] - \mathtt{Num}_{B,\beta}[\mathtt{Pos}_{B,\alpha}[i]]$ in B. See Figure 3.

Assume $p \le q$ for the moment. For the three values i, p, q, we define a piecewise-linear function $f_i^{s,t}$ consisting of a slope-1 segment from $(0, i + p)$ to $(q - p, i + q)$ and a horizontal extension from that point to infinity as shown in the left diagram of Figure 4.

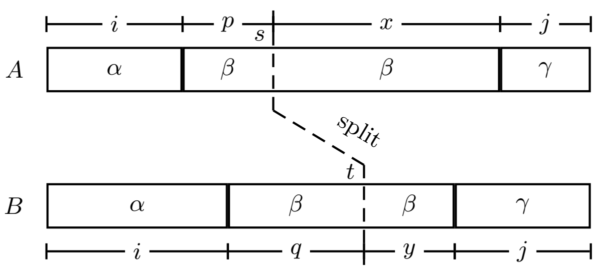


Fig. 3. Splitting a pair of sequences.

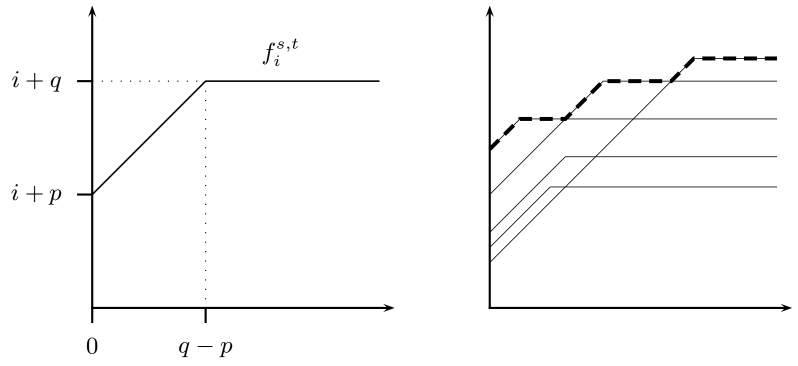


Fig. 4. Split diagrams.

What is the purpose of this function? Assume we tried to find a long common subsequence by matching exactly j many γs in the two sequences. We would align these j pairs as far to the right as possible in order to gain as many βs as possible. So count the number of βs between position s and the leftmost matched γ in A and likewise in B. Say, there are x such βs in A and y in the respective part of B. (See Figure 3 again.) We can now use our function $f_i^{s,t}$ to obtain the length of an LCWIS of type $\alpha^i\beta^*\gamma^j$: Compute the surplus $z = x - y$ of unmatchable βs in A on the right (assuming $x \geq y$ for the moment) and read off the function value of $f_i^{s,t}$ for that argument. The value $f_i^{s,t}(z)$ tells us exactly how long a subsequence we can build to the left of the split if we throw in a surplus of z βs into A.

For example, with no extra βs from the right, we only get $\min(p,q) = p$ many pairs of βs, which together with the i αs yield a sequence of length $f_i^{s,t}(0) = i + p$. If we have $q - p$ free βs on the right, we could get a sequence of length $f(q - p) = i + q$. More βs would not bring an advantage, which is expressed in the stagnation of the function f beyond $q - p$. The case $p > q$, which we had originally excluded for cleaner presentation, is simply covered by an analogous function $\bar{f}_i^{s,t}$, defined in the obvious way to handle free βs on the right of the split in sequence B.

Of course, we have not gained anything yet from the function $f_i^{s,t}$. The trick is now to draw the functions $f_i^{s,t}$ for all values of i into *one* diagram. Their pointwise maximum $f^{s,t}$, the *upper envelope* of their plots, indicated in the right of Figure 4, gives us the best possible length to the left of the split for any surplus of βs from the right.

Lemma 3. *Amongst all subsequences that have all their αs to the left and all γs to the right of a fixed split (s,t), we can find an LCWIS in linear time.*

Proof. For $0 \leq i \leq \min(\texttt{Num}_{A,\alpha}[s], \texttt{Num}_{B,\alpha}[t])$, "draw" all functions $f_i^{s,t}$ into one split diagram. One can build an array of function values of the upper envelope left-to-right in $O(n)$ time as follows: Sort all triples (i, p, q) corresponding to the $f_i^{s,t}$ functions with respect to decreasing $i + p$ value; draw the slope-1 segment of the first $f_i^{s,t}$ function; repeatedly advance to the next function with a piece of its slope-1 segment on the upper envelope (functions completely below the previous function on the upper envelope are skipped). See Figure 4.

After that, test for each right-aligned match of γs, how many βs match to the right of the split (s, t) and evaluate the envelope function for the respective surplus of βs. Actually constructing an LCWIS once its length is known is an easy task. □

In order to turn the split technique into a fast algorithm for the general case, where we do not have any pre-knowledge about good splits, we will have to refine it a little further. If we know that there is an LCWIS with many βs, we can apply Lemma 3 immediately.

Theorem 2. *For two length-n sequences over three letters $\alpha < \beta < \gamma$, we can find an LCWIS that contains at least rn many βs ($r \in (0,1)$) in $O(n/r^2)$ time.*

Proof. Put a marker every rn positions in A and also in B. Test all $\lceil 1/r \rceil^2$ candidate splits at marker pairs. Any $\alpha^*\beta^{\lceil rn \rceil}\beta^*\gamma^*$ subsequence must cover at least one of those pairs with its β-section. Hence we will find it. □

A hierarchy of splits: In the general case, when we need to make sure that we identify subsequences with only a few βs, we need a few tricks to further reduce the number of splits. To this end, recall that we may always restrict attention to left-aligned common subsequences. In particular, any such subsequence with exactly i many αs has all its αs to the left of the cut $(\texttt{Pos}_{A,\alpha}[i], \texttt{Pos}_{B,\alpha}[i])$ and all its γs to the right of that cut. Hence, we may restrict attention to the collection $\mathcal{S}$ of all cuts of this type.

Note that $\mathcal{S}$ comes with a natural linear order since no two of its splits cross and hence, $|\mathcal{S}| = O(n)$. We could now, naively, draw all α-prefix information from the left into each cut of $\mathcal{S}$ simultaneously, and afterwards check each γ-postfix against each cut. That would cost quadratic time and space.

The key observation behind our $O(n \log n)$-time algorithm is that the naive approach stores multiple copies of the same information in different split diagrams. For instance, all diagrams will store information and will be queried regarding the possibility of a βs-only LCWIS. We can reduce the amount of work by drawing only a partial diagram at each split, while spreading all necessary information among the different diagrams. We assign levels to the splits in $\mathcal{S}$: let the level of the ith split (counting from left) be the index of the least significant bit equal to one in the binary representation of i. This scheme has the nice property that between any two splits on the same level there lies another split on a higher level. Our algorithm will use this property to ensure that for any LCWIS L, there exists at least one split diagram into which we insert the α information of L and from which we read off the γ part of L.

Our algorithm proceeds in two sweeps over the sequences. In the first sweep it constructs a split diagram for each of the splits in $\mathcal{S}$. Then, some left-side configurations are entered into some diagrams. For each integer i, match the first i αs from A and B and enter the corresponding functions into the split diagram of the closest split (s, t) to the right on each level. This means that the effect of starting with exactly i αs is entered into $O(\log |\mathcal{S}|) = O(\log n)$ diagrams. After all diagrams are prepared, the algorithm makes a second sweep of the sequences forming all right-aligned matches of γs. For each such partial subsequence we then query the split diagrams for the closest split to the left on each level to obtain the maximum length of an LCWIS with these many γs. A formal description of the algorithm is given in Figure 5.

```
Function LCWIS_3 $(A = (a_1, \dots, a_n), B = (b_1, \dots, b_m))$
  Preprocess    (* create arrays $\mathtt{Num}_{\cdot,\cdot}[]$ and $\mathtt{Pos}_{\cdot,\cdot}[]$ *)
  $t \leftarrow \min(\mathtt{Num}_{A,\alpha}[n], \mathtt{Num}_{B,\alpha}[m]$    (* the size of $\mathcal{S}$ *)
  $h := \lfloor \log_2 t \rfloor$    (* the highest level *)
  for $i = 1$ to $t$ do create empty split diagrams $D_i$ and $\bar{D}_i$
    for position $(\mu_i, \nu_i) \leftarrow (\mathtt{Pos}_{A,\alpha}[i], \mathtt{Pos}_{B,\alpha}[i])$ with level $\max\{r : 2^r | i\}$
  (* First sweep: filling the diagrams *)
  for $i = 1$ to $t$ do
    for $r = 0$ to $h$ do
      $d \leftarrow$ index of closest level-$r$ diagram to the right ($\geq$)
            of $(\mathtt{Pos}_{A,\alpha}[i], \mathtt{Pos}_{B,\alpha}[i])$
      $p \leftarrow \mathtt{Num}_{A,\beta}[\mu_d] - \mathtt{Num}_{A,\beta}[\mathtt{Pos}_{A,\alpha}[i]]$
      $q \leftarrow \mathtt{Num}_{B,\beta}[\nu_d] - \mathtt{Num}_{B,\beta}[\mathtt{Pos}_{B,\alpha}[i]]$
      enter triple $(i, p, q)$ into $D_d$ and triple $(i, q, p)$ into $\bar{D}_d$
    od
  od
  Preprocess all $D_i$ and $\bar{D}_i$ for quick look-up
  (* Second sweep: reading the diagrams *)
  $best \leftarrow 0$
  for $i = 0$ to $\min(\mathtt{Num}_{A,\gamma}[n], \mathtt{Num}_{B,\gamma}[m]) - 1$ do
    for $r = 0$ to $h$ do
      $d \leftarrow$ index of closest level-$r$ diagram to the left ($<$)
            of $(\mathtt{Pos}_{A,\gamma}[n-i], \mathtt{Pos}_{B,\gamma}[m-i])$
      $x \leftarrow \mathtt{Num}_{A,\beta}[\mathtt{Pos}_{A,\gamma}[n-i]] - \mathtt{Num}_{A,\beta}[\mu_d]$
      $y \leftarrow \mathtt{Num}_{B,\beta}[\mathtt{Pos}_{B,\gamma}[m-i]] - \mathtt{Num}_{B,\beta}[\nu_d]$
      $length \leftarrow i + \min(x, y) + \max(D_d(x, y), \bar{D}_d(y, x))$
      if $length > best$ then $best \leftarrow length$
    od
  od
  return $best$
```

Fig. 5. The $O(m + n \log n)$-time 3-letter LCWIS algorithm.

Lemma 4. *Let L be an LCWIS for (A, B) with exactly i many αs and j many γs. Then the algorithm above maintains a split diagram that receives i left-aligned αs and which is afterwards queried with j right-aligned γs.*

Proof. Consider the set $\mathcal{S}'$ of all split diagrams between the first i many αs and the last j many γs. By the hierarchical structure of $\mathcal{S}$, there is a unique diagram on the highest level of $\mathcal{S}'$. This diagram obviously has the desired property. □

This lemma tells us that any LCWIS will be found by the algorithm because some split diagram receives its left-aligned α-part and is queried by its right-aligned γ-part.

The two sweeps can be implemented to run in $O(m + n \log n)$ time as follows. During the first sweep we simply create a list of $O(n \log n)$ quadruples (i, p, q, s) that represent the contents of the $O(n)$ splitters: s is the identity of a splitter and (i, p, q) are the parameters that define one of the functions illustrated in the left of Figure 4. Similarly, during the second sweep we construct a list of $O(n \log n)$ quadruples (i, p, q, s) where (i, p, q) is a query and s is the splitter on which it is to be performed. After bucket-sorting each list, all queries can be answered by a simultaneous linear scan of the lists.

Theorem 3. *We can find an LCWIS of two 3-letter sequences of lengths m and n, with $m \geq n$, in $O(m + n \log n)$ time.*

3.4 The 3-letter case in $O(m \log \log m)$ time

We consider again the 3-letter case and show an algorithm that solves the problem in time $O(m \log \log m)$. This algorithm is based on a new approach to the 2-letter case. In particular, we show how to solve the 2-letter case in an online manner, where one symbol of a sequence is revealed at a time. We denote as $(A[1 \dots a], B[1 \dots b])$ a subproblem where the first a symbols of A and the first b symbols of B have been revealed. Given the solution to such a subproblem we show how to solve the subproblem $(A[1 \dots a + 1], B[1 \dots b])$ or $(A[1 \dots a], B[1 \dots b + 1])$.

Assuming that such a procedure for the 2-letter case exists, then the 3-letter case is simple: For each k, where k takes values from $\min\{\mathtt{Num}_{A,\gamma}[n], \mathtt{Num}_{B,\gamma}[m]\}$ down to 0, we determine the positions a_k and b_k of the kth γ in A and B respectively, counting from the end. Namely, $a_k = \mathtt{Pos}_{A,\gamma}[\mathtt{Num}_{A,\gamma}[n] - \mathtt{Num}_{A,\gamma}[k] + 1]$ and similarly $b_k = \mathtt{Pos}_{B,\gamma}[\mathtt{Num}_{B,\gamma}[m] - \mathtt{Num}_{B,\gamma}[k] + 1]$. We solve the 2-letter subproblem $(A[1 \dots a_k], B[1 \dots b_k])$ by using the 2-letter online procedure. Given the solution to the 2-letter subproblem $(A[1 \dots a_{k+1}], B[1 \dots b_{k+1}])$ that we solved in the previous iteration, we reveal to the procedure the symbols $B[b_{k+1} + 1], \dots, B[b_k]$ and $A[a_{k+1} + 1], \dots, A[a_k]$ one after the other obtaining each time the solution to the new subproblem and finally obtain the solution to the subproblem $(A[1 \dots a_k], B[1 \dots b_k])$. Where for $k = \min\{\mathtt{Num}_{A,\gamma}[n], \mathtt{Num}_{B,\gamma}[m]\}$ we have initialized b_{k+1} and a_{k+1} to zero. In this way we obtain for every k the length of a CWIS of type $\alpha^* \beta^* \gamma^k$. The longest of them over all k are the LCWISs of the two sequences. If each call to the 2-letter procedure takes time $O(\log \log m)$, the algorithm has running time $O((m + n) \log \log m) = O(m \log \log m)$.

Solving the 2-letter case: We now describe how to solve the 2-letter problem in an online fashion. Recall the simple algorithm described in section 3.2. The correctness of this algorithm is based on the fact that it considers all possible CWISs that can potentially be extended to an LCWIS. Namely, for each $0 \leq i \leq \min\{\mathtt{Num}_{A,\alpha}[n], \mathtt{Num}_{B,\alpha}[m]\}$ it considers a CWIS containing i αs and as many βs as possible.

The new approach maintains this property too. Namely, for the current subproblem it maintains all currently possible CWISs and the longest one is the solution to the subproblem. So, if $(A[1 \dots a], B[1 \dots b])$ is the current subproblem, then for all $0 \leq i \leq \min\{\mathtt{Num}_{A,\alpha}[a], \mathtt{Num}_{B,\alpha}[b]\}$ we maintain a CWIS containing i αs; the i leftmost pairs of αs and as many βs as possible. We denote such a CWIS by $\langle i \rangle$.

The general idea is as follows. The algorithm maintains a set $\mathcal{P}$ of at most n partial solutions, each of which is a CWIS that contains a different number of αs. Assume that we have a solution to the current subproblem and a new symbol of one of the sequences is revealed. It is either an α or a β. If it is an α, assume that it is the ith α in its sequence. The algorithm checks whether there are i αs in the prefix of the other sequence that has been revealed, and if so it inserts the sequence α^i into $\mathcal{P}$, as the sequence $\langle i \rangle$. If the new letter is a β, the algorithm updates those CWISs that can be extended with one more pair of βs. Namely, those CWISs which have at least one free β in the other sequence. The solution to the new subproblem is the currently longest CWIS. If we implement the above algorithm naively then each time we consider a β we may spend $\Theta(n)$ time, since there can be up to n CWISs in $\mathcal{P}$.

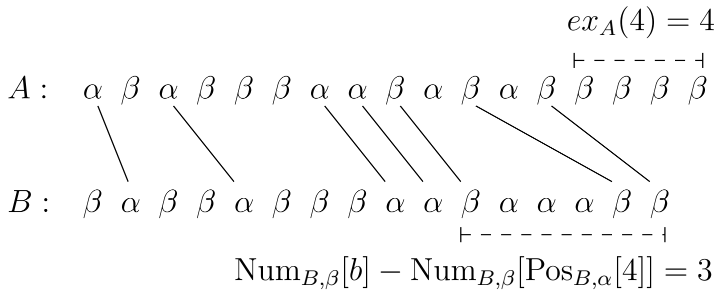
A speedup via priority queues: Our aim is to handle every new symbol in $O(\log\log m)$ time. In order to do so we need the following simple observations. Assume we have maintained all currently possible CWISs. Among those consider the CWISs which have a non-negative number of free βs in A. Namely, the set of CWISs with

$$(\texttt{Num}_{A,\beta}[a] - \texttt{Num}_{A,\beta}[\texttt{Pos}_{A,\alpha}[i]]) - (\texttt{Num}_{B,\beta}[b] - \texttt{Num}_{B,\beta}[\texttt{Pos}_{B,\alpha}[i]]) > 0,$$

i.e., those for which the number of βs in A, after the ith α in A is at least the number of βs in B, after the ith α in B. We call this number the excess of $\langle i \rangle$ in A and denote it by $ex_A(i)$ (see Figure 6). Observe that the length of such a CWIS $\langle i \rangle$, call it $\ell(i)$, can be expressed, at any point, as i plus the number of βs in B, after the position of the ith α in B, i.e.,

$$\ell(i) = i + \texttt{Num}_{B,\beta}[b] - \texttt{Num}_{B,\beta}[\texttt{Pos}_{B,\alpha}[i]].$$

The term $\texttt{Num}_{B,\beta}[b]$ does not depend on i and it is common to all these CWISs. Therefore, if we need to keep some ordering on their length it is sufficient to do so by using the term $i - \texttt{Num}_{B,\beta}[\texttt{Pos}_{B,\alpha}[i]]$. The advantage is that the value of this term does not change as new symbols of the sequences are revealed, as is the case for the term $\texttt{Num}_{B,\beta}[b]$.



$ex_A(4) = 4$
$A:$ $\alpha\ \beta\ \alpha\ \beta\ \beta\ \beta\ \alpha\ \alpha\ \beta\ \alpha\ \beta\ \alpha\ \beta\ \beta\ \beta\ \beta\ \beta$
$B:$ $\beta\ \alpha\ \beta\ \beta\ \alpha\ \beta\ \beta\ \beta\ \alpha\ \alpha\ \beta\ \alpha\ \alpha\ \alpha\ \beta\ \beta$
$\mathrm{Num}_{B,\beta}[b] - \mathrm{Num}_{B,\beta}[\mathrm{Pos}_{B,\alpha}[4]] = 3$

Fig. 6. Example of the revealed prefixes of A and B and their CWIS $\langle 4 \rangle$.

So, consider those CWISs with $ex_A(i) > 0$ in non-decreasing order of their current length. When a new β appears in B the length of all these CWISs increases by one since they have at least one free β in A to match this new β. Therefore, their relative order does not change. On the other hand, when a new β appears in A the length of none of these CWISs changes since they have no free βs in B and therefore again their relative order does not change. So, if we maintain them in a maximum priority queue with priority their length, the structure of the priority queue will not change when a new β is revealed.

The priority queues L_A and L_B ordered by length: Given the above observations, the algorithm maintains a maximum priority queue L_A keeping CWISs with $ex_A(i) > 0$ as follows. The elements of the priority queue are (key, data) pairs, where the key is a number that corresponds to the length of one or more CWISs and the data is a doubly linked list containing the CWISs of that same length. For a given CWIS the key is: $k(i) = i - \mathtt{Num}_{B,\beta}[\mathtt{Pos}_{B,\alpha}[i]] + m$. (We added m in order to avoid negative priorities.) So, whenever we need the length of a CWIS contained in this priority queue we compute it by $\ell(i) = k(i) + \mathtt{Num}_{B,\beta}[b] - m$.

Moreover, in order to be able to find and delete elements of the doubly linked lists fast, we keep an array $P[0 \ldots \min\{\mathtt{Num}_{A,\alpha}[n], \mathtt{Num}_{B,\alpha}[m]\}]$, where $P[i]$ corresponds to CWIS $\langle i \rangle$ and contains a pointer to the element of the doubly linked list corresponding to CWIS $\langle i \rangle$.

Insertion of a CWIS $\langle i \rangle$ in L_A works as follows: Given i, compute $k(i)$ with the above formula. If $k(i)$ is not already in L_A then add the pair ($k(i)$, new doubly linked list). Create a list element with value i and insert it at end of the list. Set $P[i]$ to point at the newly inserted element of the list.

Deletion of a CWIS from L_A works as follows: Given i, delete from the list the element that $P[i]$ points to. If the list becomes empty, then compute $k(i)$ and delete it from L_A.

Since the keys we insert this way are integers in the range $\{1, \ldots, m+n\}$, we can use as a priority queue a van Emde Boas tree [12] which supports each operation in $O(\log\log m)$ time.

The complete algorithm uses three additional priority queues, all of which hold keys which are positive integers of value at most $m+n$, and therefore we can use van Emde Boas trees for all queues. Moreover, in all of them we also add a suitable term in $O(m+n)$, simply to avoid negative priorities, and we will always have to subtract this term when computing a particular priority. Finally, in all of them we omit an additive term which although changes as a and b increase, at any particular point it is the same for all elements in the priority queue and has to be added whenever a particular priority is computed.

Similar properties to the above hold for the set of CWISs that have a non-negative number of free βs in B. Their relative order does not change when a new β is revealed in A or B. So, we keep them in a priority queue L_B. The length of such a CWIS $\langle i \rangle$ can be expressed as

$$\ell(i) = i + \mathtt{Num}_{A,\beta}[a] - \mathtt{Num}_{A,\beta}[\mathtt{Pos}_{A,\alpha}[i]].$$

Therefore, similarly to above, it is sufficient to keep them in L_B with priority $i - \mathtt{Num}_{A,\beta}[\mathtt{Pos}_{A,\alpha}[i]] + n$, where n is added in order to avoid negative priorities. The priorities we insert in this way are integers in the range $\{1, \ldots 2n\}$.

Interaction between the queues: Any CWIS $\langle i \rangle$ has either free βs in A, in which case it belongs in L_A or free βs in B, in which case it belongs in L_B or it has no free βs. With some care we are going to be able to put a CWIS of the latter case in either L_A or L_B. Since all possible CWISs are included in L_A and L_B the maximum between the maximum in L_A and the maximum in L_B is the current solution. Note that as symbols are revealed, the status of a CWIS can change from having a positive number of free βs in A, to not having any free βs in A to having a positive number of free βs in B, and vice versa. This means that there are cases in which we need to move CWISs between L_A and L_B.

The priority queues EX_A and EX_B ordered by excess Next, we show how to handle the above mentioned movements. If $\langle i \rangle$ has free βs in A, recall that we call their number the *excess* of

$\langle i \rangle$ in A and denote it by $ex_A(i) = (\texttt{Num}_{A,\beta}[a] - \texttt{Num}_{A,\beta}[\texttt{Pos}_{A,\alpha}[i]]) - (\texttt{Num}_{B,\beta}[b] - \texttt{Num}_{B,\beta}[\texttt{Pos}_{B,\alpha}[i]])$. Observe that when a new β is revealed in A, the excess of all CWISs in L_A is increased by one, while, when a new β is revealed in B their excess is decreased by one. Therefore when a new β is revealed in B we need to check which CWISs got negative excess, i.e., need to be moved to L_B. For this purpose we keep a minimum priority queue EX_A where we keep the same CWISs as in L_A but with priority their excess. Observe that it is sufficient to keep them with priority $\texttt{Num}_{B,\beta}[\texttt{Pos}_{B,\alpha}[i]] - \texttt{Num}_{A,\beta}[\texttt{Pos}_{A,\alpha}[i]] + m$, since the other two terms are the same for every $\langle i \rangle$, where m is added to avoid negative priorities.

Similarly to EX_A we keep EX_B which is a priority queue, that holds the CWISs of L_B sorted by their excess of βs in B.

So, when a new β is revealed in B we need to check whether the minimum in EX_A became negative, in which case we remove it from L_A and EX_A and insert it into L_B and EX_B.

Symmetrically, when a new β is revealed in A we check whether the minimum excess in EX_B became negative, in which case we remove the corresponding CWIS from L_B and EX_B and insert it into L_A and EX_A.

Eliminating duplicates: In order to have that each new revealed symbol causes only a constant number of priority queue operations, we must ensure that in each of the queues EX_A and EX_B, each key appears at most once. In this way when a new β is revealed in A or B at most one element in EX_A and at most one element in EX_B gets negative excess and has to be moved.

We describe how this can be achieved when we wish to insert a CWIS $\langle i \rangle$ into L_A and EX_A. The strategy is similar when inserting a CWIS into L_B and EX_B. We first check whether there is a CWIS of the same excess in EX_A. If not, we insert $\langle i \rangle$ into L_A and EX_A as described before. Otherwise, there is a CWIS $\langle k' \rangle$ in EX_A with $ex_A(k') = ex_A(i)$. We decide according to the following rule. If $\ell(i) > \ell(k')$, delete $\langle k' \rangle$ from L_A and EX_A and insert $\langle i \rangle$. Otherwise, discard $\langle i \rangle$. We can do this because both CWISs have the same excess, so they will always have the same growth and therefore, the larger of the two will always remain the larger.

The complete 2-letter algorithm: Putting everything together, we have the following algorithm. Keep four priority queues L_A, EX_A, L_B, EX_B as described and a variable that holds the *current solution*, i.e., the currently best CWIS. Create a CWIS with zero αs, i.e., the $\langle 0 \rangle$ CWIS. Insert $\langle 0 \rangle$ in L_A and EX_A and set it to be the current solution. When a new α is revealed in A or B and it is the ith one in its sequence, check whether there are i αs in the other sequence and if so, create a new CWIS $\langle i \rangle = \alpha^i$. Compute its excess in A and if it is non-negative insert $\langle i \rangle$ into L_A and EX_A. If the excess in A is negative, then the excess in B is positive, so, insert $\langle i \rangle$ into L_B and EX_B. Moreover, check whether it has length greater than the current solution, in which case set it to be the current solution. When a new β is revealed in A, check whether the minimum of EX_A now has negative excess and if so, delete it from L_A and EX_A and insert it into L_B and EX_B. Next, check whether the maximum between the maximum in L_A and the maximum in L_B has length greater than the current solution and if so, set it to be the current solution. When a new β is revealed in B, check whether the minimum excess in EX_B became negative and if so, delete it from L_B and EX_B and insert it into L_A and EX_A. Also, check whether the maximum between the maximum in L_A and the maximum in L_B is longer than the current solution in which case we set it to be the current solution.

The time we spent in the above algorithm each time we encounter a new symbol is $O(\log\log m)$, so we have

Theorem 4. *We can find an LCWIS of two 3-letter sequences of lengths m and n, with $m \geq n$, in $O(m \log\log m)$ time.*

4 Multiple Sequences

In this section we consider the problem of finding an LCIS of k length-n sequences, for $k \geq 3$. We will denote the sequences by $A^1 = (a^1_1, \ldots, a^1_n)$, $A^2 = (a^2_1, \ldots, a^2_n)$, …, $A^k = (a^k_1, \ldots, a^k_n)$. A *match* is a vector $(i_1, i_2, \ldots, i_k)$ of indices such that $a^1_{i_1} = a^2_{i_2} = \cdots = a^k_{i_k}$. Let r be the number of matches. Chan et al. [4] showed that an LCIS can be found in $O(\min(kr^2, kr \log\sigma \log^{k-1} r) + k Sort_\Sigma(n))$ time (they present two algorithms, each corresponding to one of the terms in the min). We present a simpler solution which replaces the second term by $O(r \log^{k-1} r \log\log r)$.

We denote the ith coordinate of a vector v by $v[i]$, and the alphabet symbol corresponding to the match described by a vector v will be denoted $s(v)$. A vector v *dominates* a vector v' if $v[i] > v'[i]$ for all $1 \leq i \leq k$, and we write $v' < v$. Clearly, an LCIS corresponds to a sequence $v_1, \ldots, v_\ell$ of matches such that $v_1 < v_2 < \cdots < v_\ell$ and $s(v_1) < s(v_2) < \cdots < s(v_\ell)$.

To find an LCIS, we use a data structure by Gabow et al. [6, Theorem 3.3], which stores a fixed set of n vectors from $\{1, \ldots, n\}^k$. Initially all vectors are *inactive*. The data structure supports the following two operations:

1. *Activate* a vector with an associated priority.
2. A query of the form "what is the maximum priority of an active vector that is dominated by a vector p?"

A query takes $O(\log^{k-1} n \log\log n)$ time and the total time for at most n activations is $O(n \log^{k-1} n \log\log n)$. The data structure requires $O(n \log^{k-1} n)$ preprocessing time and space.

Each of the r matches $v = (v_1, \ldots, v_k)$ corresponds to a vector. The priority of v will be the length of the longest LCIS that ends at the match v. We will consider the matches by non-decreasing order of their symbols. For each symbol s of the alphabet, we first compute the priority of every match v with $s(v) = s$. This is equal to 1 plus the maximum priority of a vector dominated by v. Then, we activate these vectors in the data structure with the priorities we have computed; they should be there when we compute the priorities for matches v with $s(v) > s$.

The algorithm applies to the case of a common weakly-increasing subsequence by the following modification: The matches will be considered by non-decreasing order of $s(v)$ as before, but within each symbol also in non-decreasing lexicographic order of v. For each match, we compute its priority and immediately activate it in the data structure (so that it is active when considering other matches with the same symbol). The lexicographic order ensures that if $v > v'$ then v' is in the data structure when v is considered.

Theorem 5. *An LCIS or LCWIS of k length-n sequences can be computed in $O(r \log^{k-1} r \log\log r)$ time, where r counts the number of match vectors.*

5 Outlook

The central question about the LCS problems is, whether it can be solved in $O(n^{2-\epsilon})$ time in general. It seems that with LCIS we face the same frontier. Our new algorithms are fast in many situations, but in general, we do not obtain subquadratic running-time, either.

On the other hand, LCWIS seems to behave very different from the other two problems. Our result shows that it behaves somewhat like a mixture of LCS and LCIS. While already the 2-letter problem is unsolved for LCS, finite alphabets are trivial for LCIS. With LCWIS, we present near-linear-time solutions for alphabets with up to three letters, while it is unclear whether similar results can be obtained for all finite alphabets.

References

1. D. Aldous and P. Diaconis. Longest increasing subsequences: From patience sorting to the Baik-Deift-Johansson theorem. *Bulletin of the AMS*, 36(4):413–432, 1999.
2. L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE '00)*, pages 39–48. IEEE Computer Society, 2000.
3. S. Bespamyatnikh and M. Segal. Enumerating longest increasing subsequences and patience sorting. *Information Processing Letters*, 76(1-2):7–11, 2000.
4. W.-T. Chan, Y. Zhang, S. P.Y. Fung, D. Ye, and H. Zhu. Efficient Algorithms for Finding A Longest Common Increasing Subsequence. *Journal of Combinatorial Optimization*, 13(3):277–288, 2007.
5. M.L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29–35, 1975.
6. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing (STOC '84)*, pages 135–143. ACM Press, 1984.
7. D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.
8. J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.
9. W.J. Masek and M.S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20:18–31, 1980.
10. E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.
11. Y. Sakai. A linear space algorithm for computing a longest common increasing subsequence. *Information Processing Letters*, 99(5):203–207, 2006.
12. P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.
13. R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.
14. D. E. Willard. Log-logarithmic worst-case range queries are possible in space Theta(N). *Information Processing Letters*, 17(2):81–84, 1983.
15. I.-H. Yang, C.-P. Huang, and K.-M. Chao. A fast algorithm for computing a longest common increasing subsequence. *Information Processing Letters*, 93(5):249–253, 2005.