Faster Algorithms for Computing Longest Common Increasing Subsequences

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

[1] BRICS***, University of Aarhus, Århus, Denmark.
{gerth,irit}@daimi.au.dk.

[2] Max-Plank-Institut für Informatik, Saarbrücken, Germany
{kaligosi,mkutz}@mpi-inf.mpg.de.

Abstract. We present algorithms for finding a longest common increasing subsequence of two or more input sequences. For two sequences of lengths m and n, 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 $O(m + n \log n)$-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.

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 $a_{j_1} < a_{j_2} < \cdots < a_{j_\ell}$.

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

** Supported by the Danish Research Agency (grant # 272-05-0081).

*** Basic Research in Computer Science, funded by the Danish National Research Foundation.

A classic algorithm by Wagner and Fischer [12] 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 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 to the alphabet $\{1, 2, \ldots, \sigma\}$. 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. [14] 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.

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.

Symbol	Meaning
m, n	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	New
$k = 2$	$O(mn)$ [14] $O(\min\{r \log \sigma, m\sigma + r\} \log \log m$ $+ Sort_\Sigma(m))$ [4]	$O((m + n\ell) \log \log \sigma + Sort_\Sigma(m))$ $O(m)$ when $\sigma = 2$ $O(m + n \log n)$ 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.

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 $O(m + n \log n)$ 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 approach we use 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 $O(\min\{kr^2, r \log^{k-1} r \log \log r\} + kSort_\Sigma(n))$.

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 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*$(\mathcal{H}, k, p, d)$: Insert into the *BH* $\mathcal{H}$ the key k with priority p and associated data d.
- *DecreasePriority*$(\mathcal{H}, k, p, d)$: If the *BH* $\mathcal{H}$ does not already contain the key k, perform *Insert*$(\mathcal{H}, k, p, d)$. Otherwise, set this key's priority to $\min\{p, p'\}$, where p' is its previous priority.
- *BoundedMin*$(\mathcal{H}, k)$: Return the item that has minimum priority among all items in $\mathcal{H}$ with key smaller than k. If $\mathcal{H}$ does not contain any items with key smaller than k, 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 for the set $\{1, \ldots, n\}$.*

Proof (sketch). 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 $BM(s) \leq BM(s')$ whenever $s > s'$, i.e., the function BM is non-increasing in s (see Figure 1).

key k	1	2	3	4	5	6	7	8	9	10
priority	7	10	6	8	5	3	2	4	1	9
$BM(k)$	∞	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 $s_i \leq s_{i+1}$ 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 $BoundedMin(s)$ 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 [11] as search structure, this takes $O(\log\log n)$ time. The implementation of Insert and DecreasePriority are described in the full version of this paper. □

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 queries $\mathcal{H}$ 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 the full version of this paper.

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 $Link_1, Link_2, \ldots$ 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 $Link_i[j] = j'$. 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 $c_1, \dots, c_\ell$ that is located in A as $a_{j_1}, \dots, a_{j_\ell}$ and in B as $b_{\kappa_1}, \dots, b_{\kappa_\ell}$.

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

Time complexity The preprocessing phase takes $O(Sort_\Sigma(m))$ 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. We now sketch a possible implementation of the Occ_s lists.

We 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 [13]. In addition, for each block we store the first occurrence of s succeeding the block. To answer a query in Occ_s, we first identify the block containing the query point in constant time. We then search for the smallest index larger than the query point 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 $O((m + n\ell)\log\log\sigma + Sort_\Sigma(m))$.

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 $(a_1, \ldots, a_{i-1})$ $(b_1, \ldots, b_{j-1})$ and the length-$\lceil \ell/2 \rceil$ LCIS of $(a_{i+1}, \ldots, a_n)$ and $(b_{j+1}, \ldots, b_m)$. 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 $Sort_{\Sigma}(m)$. 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 $O((m + n\ell)\log\log\sigma + Sort_{\Sigma}(m))$. 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 $O((m + n\ell)\log\log\sigma + Sort_{\Sigma}(m))$ expected time and $O(m)$ worst-case space where ℓ is the length of the output and $Sort_{\Sigma}(m)$ 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 $O((m + n\ell)\log\log\sigma + Sort(m))$ time. But while the LCIS problem can be solved in linear time for alphabets of bounded size t, simply because the length of the solution is then also bounded by t, it is not clear how this fact should carry over to LCWIS, where the output size need not relate to t at all.

We show how to solve LCWIS for the 2– and the 3-letter alphabet in linear respectively $O(m + n \log n)$ time. This is 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 $\texttt{Num}_{A,\alpha}$,$\texttt{Num}_{B,\alpha}$,$\texttt{Num}_{A,\beta}$, …,$\texttt{Num}_{B,\gamma}$ that count the number of αs, βs and γs, respectively, in prefixes of A and B. For example, $\texttt{Num}_{A,\gamma}[9]$ contains the number of γs in A up to position 9 (inclusively). We also create arrays $\texttt{Pos}_{A,\alpha}$ through $\texttt{Pos}_{B,\gamma}$, 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 $0 \leq i \leq \min\{\texttt{Num}_{A,\alpha}[n], \texttt{Num}_{B,\alpha}[m]\}$, 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 three-letter case—split diagrams

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 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 $p = \texttt{Num}_{A,\beta}[s] - \texttt{Num}_{A,\beta}[\texttt{Pos}_{A,\alpha}[i]]$ such βs in A and $q = \texttt{Num}_{B,\beta}[t] - \texttt{Num}_{B,\beta}[\texttt{Pos}_{B,\alpha}[i]]$ in B.

Assume $p \leq q$ 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 2.

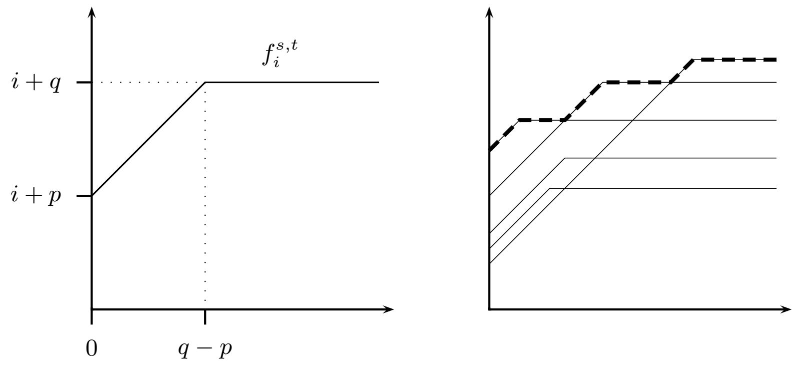

Fig. 2. Split diagrams.

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. 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 $q > p$, which we had originally excluded for cleaner presentation, is simply covered by a 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 point-wise maximum $f^{s,t}$, the *upper envelope* of their plots, indicated in the right of Figure 2, 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.*

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. □

3.4 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, first note that we may restrict attention to splits (s, t) that are given by left-aligned α-matches: The collection $\mathcal{S}$ of all splits of the form $(\texttt{Pos}_{A,\alpha}[i], \texttt{Pos}_{B,\alpha}[i])$ suffices to find an LCWIS.

Note that $\mathcal{S}$ comes with a natural linear order since no two of its splits cross and hence, $|\mathcal{S}| = O(n)$. Yet, if we drew a complete split diagram for every split in $\mathcal{S}$, we would still face a quadratic running-time. To reduce the work, we avoid drawing complete diagrams for all splits but spread information over splits. Therefore, 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.

Conceptually, our algorithm proceeds in two sweeps over the sequences. In the first sweep it constructs a split diagram for each of the splits in $\mathcal{S}$. However, not all left-side configurations are entered into all 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 the full version of this paper.

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 2. 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 three-letter sequences of lengths m and n, with $m \geq n$, in $O(m + n \log n)$ 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 $O(\min(kr^2, kr \log \sigma \log^{k-1} r) + kSort_\Sigma(n))$ 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 4. *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 now, 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. *Bull. AMS*, 36(4):413–432, 1999.
2. L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *SPIRE '00*, pages 39–48. IEEE Computer Society, 2000.
3. S. Bespamyatnikh and M. Segal. Enumerating longest increasing subsequences and patience sorting. *Inf. Process. Lett.*, 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. In *ISAAC '05*, 2005.
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 *STOC '84*, pages 135–143. ACM Press, 1984.
7. D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.
8. J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977.
9. W.J. Masek and M.S. Paterson. A faster algorithm computing string edit distances. *J. Comput. System Sci.*, 20:18–31, 1980.
10. E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.
11. P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.
12. R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.
13. D. E. Willard. Log-logarithmic worst-case range queries are possible in space Theta(N). *Inf. Process. Lett.*, 17(2):81–84, 1983.
14. I.-H. Yang, C.-P. Huang, and K.-M. Chao. A fast algorithm for computing a longest common increasing subsequence. *Inf. Process. Lett.*, 93/5:249–253, 2005.