A Faster Subquadratic Algorithm for the Longest Common Increasing Subsequence Problem

Anadi Agrawal
Institute of Computer Science, University of Wrocław, Poland

Paweł Gawrychowski
Institute of Computer Science, University of Wrocław, Poland

Abstract

The Longest Common Increasing Subsequence (LCIS) is a variant of the classical Longest Common Subsequence (LCS), in which we additionally require the common subsequence to be strictly increasing. While the well-known "Four Russians" technique can be used to find LCS in subquadratic time, it does not seem directly applicable to LCIS. Recently, Duraj [STACS 2020] used a completely different method based on the combinatorial properties of LCIS to design an $\mathcal{O}(n^2(\log\log n)^2/\log^{1/6} n)$ time algorithm. We show that an approach based on exploiting tabulation (more involved than "Four Russians") can be used to construct an asymptotically faster $\mathcal{O}(n^2 \log\log n/\sqrt{\log n})$ time algorithm. As our solution avoids using the specific combinatorial properties of LCIS, it can be also adapted for the Longest Common Weakly Increasing Subsequence (LCWIS).

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Longest Common Increasing Subsequence, Four Russians

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2020.4

1 Introduction

In the well-known Longest Common Subsequence problem we aim to find the length of the longest subsequence common to two strings $A[1..n]$ and $B[1..n]$. A textbook exercise is to find it in $\mathcal{O}(n^2)$ time [17], and using the so-called "Four Russians" technique this has been brought down to $\mathcal{O}(n^2/\log^2 n)$ for constant alphabets [17] and $\mathcal{O}(n^2(\log\log n)^2/\log^2 n)$ [5] or even $\mathcal{O}(n^2 \log\log n/\log^2 n)$ [11] for general alphabets. Recently, there was some progress in providing explanation for why a strongly subquadratic $\mathcal{O}(n^{2-\epsilon})$ time algorithm is unlikely [1,7], and in fact even achieving $\mathcal{O}(n^2/\log^{7+\epsilon} n)$ would have some exciting unexpected consequences [2]. Other problems for which this technique has lead to subquadratic algorithms include Boolean matrix multiplication [3] and regular expression matching [6, 14]. Interestingly, for the well-known all-pairs shortest paths (APSP) problem, a long line of work brought nontrivial polylogarithmic improvements over the classical $\mathcal{O}(n^3)$ solution, until Williams designed an $\mathcal{O}(n^3/2^{\Omega(\sqrt{\log n})})$ time algorithm [18].

In this paper we consider a problem related to LCS defined as follows:

Problem: Longest Common Increasing Subsequence (LCIS)
Input: integer sequences $A[1..n]$ and $B[1..n]$
Output: largest ℓ such that there exist indices $i_1 < \ldots < i_\ell$ and $j_1 < \ldots < j_\ell$ with the property that (i) $A[i_k] = B[j_k]$, for every $k = 1, \ldots, \ell$, and (ii) $A[i_1] < \ldots < A[i_\ell]$.

While this is less obvious than for LCS, LCIS can be also solved in $\mathcal{O}(n^2)$ time [19] (and in linear space [16]), and it can be proved that a strongly subquadratic algorithm would refute SETH [10] (although faster algorithms are known for some special cases [13]). However, as opposed to LCS, the usual "Four Russians" approach, that roughly consists in

31st International Symposium on Algorithms and Computation (ISAAC 2020).
Editors: Yixin Cao, Siu-Wing Cheng, and Minming Li; Article No. 4; pp. 4:1–4:12
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

partitioning the DP table into blocks of size $\log n \times \log n$, does not seem directly applicable to LCIS. Very recently, Duraj [9] used a completely different approach based on some nice combinatorial properties specific to LCIS to design a subquadratic $\mathcal{O}(n^2(\log\log n)^2/\log^{1/6} n)$ time algorithm. This brings the challenge of determining if $1/6$ is the right exponent, or maybe we can shave more than that from the time complexity?

Our contribution. We design a faster subquadratic $\mathcal{O}(n^2 \log\log n/\sqrt{\log n})$ time algorithm for LCIS. Interestingly, instead of using the combinatorial properties of LCIS as in the previous work, we apply a technique based on exploiting tabulation (but differently than in the classical "Four Russians" approach). This allows our algorithm to be modified to solve the Longest Common Weakly Increasing Subsequence (LCWIS) problem (for which an $\mathcal{O}(n^{2-\epsilon})$ time algorithm is also known to refute SETH [15]). This does not seem to be the case for Duraj's approach based on bounding the number of so-called significant symbol matches, that for LCWIS might be $\Omega(n^2)$. Throughout the paper we assume that A and B are of the same length, and the goal is to calculate the length of LCIS. However, the algorithm can be easily modified to avoid this assumption and recover the subsequence itself.

Overview of the paper. Our algorithm is based on combining two different procedures. By appropriately selecting the parameters, the overall complexity becomes $\mathcal{O}(n^2 \log\log n/\sqrt{\log n})$ as explained in Section 5.

The first procedure described in Section 3 works fast when there are only few distinct elements in both sequences. We start with a solution based on dynamic programming working in $\mathcal{O}(t \cdot n^2)$ time, where t is the number of distinct elements in both sequences. Then, we exploit tabulation to decrease its running time to $\mathcal{O}(t \cdot n^2/\log n)$.

The second procedure described in Section 4 is efficient when there are not too many matching pairs, that is, pairs (i, j) such that $A[i] = B[j]$. The main idea is to calculate, for every such pair, LCIS of $A[1..i]$ and $B[1..j]$ that ends with $A[i] = B[j]$. This is done by applying an appropriate dynamic predecessor structure. This roughly follows the ideas of Duraj, except that instead of using van Emde Boas trees we notice that, in fact, one can plug in any balanced search trees with efficient split/merge.

In Section 6 we explain the necessary modification required to adapt our solution for LCWIS.

2 Preliminaries

We work with sequences consisting of integers. For such a sequence A, we write $A[i]$ to denote the i-th element, and $A[1..i]$ to denote the prefix of length i. $|A|$ is the length of A.

Let σ be the sequence consisting of all distinct integers present in A and B, arranged in the increasing order, and $\mathsf{cnt}(v)$ be the total number of occurrences of $\sigma[v]$ in A and B. Without loss of the generality we can assume that $\sigma[v] = v$, and write v instead of $\sigma[v]$.

We call a pair of indices (x, y) a *matching pair* when $A[x] = B[y]$. Further, we call it a *v-pair* when $A[x] = B[y] = v$.

We write $LCIS(i, j)$ to denote $LCIS(A[1..i], B[1..j])$, that is, the longest increasing common subsequence of $A[1..i]$ and $B[1..j]$. We write $LCIS^{\rightarrow}(i, j)$ to denote the longest increasing common subsequence of $A[1..i]$ and $B[1..j]$ which includes both $A[i]$ and $B[j]$ (so in particular, $A[i] = B[j]$).

Throughout the paper, $\log x$ denotes $\log_2 x$.

3 First Solution (Few Distinct Elements)

In this section we describe an algorithm for finding LCIS in $\mathcal{O}(|\sigma| \cdot n^2/\log n)$ time.

Let $dp_v[i][j]$ denote the largest possible length of a sequence C such that:

1. C is an increasing common subsequence of $A[1..i]$ and $B[1..j]$,
2. C consists of elements not larger than v.

Then, our goal is to compute $dp_{|\sigma|}[n][n]$.

All $|\sigma| \cdot n^2$ entries in dp can be calculated in $\mathcal{O}(1)$ time each using the following recurrence:

$$dp_{v+1}[i][j] = \begin{cases} dp_v[i-1][j-1]+1, & \text{if } A[i] = B[j] = v+1, \\ \max\{dp_v[i][j], dp_{v+1}[i-1][j], dp_{v+1}[i][j-1]\}, & \text{otherwise.} \end{cases}$$

In order to decrease the time we will speed up calculating dp_{v+1} from dp_v. Because calculating dp_{v+1} only requires the knowledge of dp_v, we will only keep the current dp_v and update all of its entries to obtain dp_{v+1}.

▶ **Lemma 1.** $0 \le dp_v[i][j] - dp_v[i][j-1] \le 1$ *and* $0 \le dp_v[i][j] - dp_v[i-1][j] \le 1$.

Proof. A subsequence of $B[1..(j-1)]$ is still a subsequence of $B[1..j]$, so $dp_v[i][j-1] \le dp_v[i][j]$. Consider a sequence C corresponding to $dp_v[i][j]$, and let C' be C without the last element. Because C is a subsequence of $B[1..j]$, C' is a subsequence of $B[1..(j-1)]$. So, C' is an increasing subsequence of $A[1..i]$ and $B[1..(j-1)]$, hence $|C'| \le dp_v[i][j-1]$. As $|C| = |C'|+1$, we conclude that $dp_v[i][j] \le dp_v[i][j-1]+1$. The second part of the lemma follows by a symmetrical reasoning. ◀

Instead of maintaining dp_v, we keep another table $dp'_v[i][j] = dp_v[i][j] - dp_v[i][j-1]$ (where $dp_v[i][j] = 0$ for $j < 1$). Due to Lemma 1, each entry of dp'_v is either 0 or 1. This allows us to store each row of dp'_v by partitioning it into $\mathcal{O}(n/b)$ blocks of length b, with every block represented by a bitmask of size b saved in a single machine word, where $b = \alpha \log n$ for some constant $\alpha \le 1$ to be fixed later. By definition, $dp_v[i][j] = \sum_{k=1}^{j} dp'_v[i][k]$. In addition to dp'_v, we store the value of $dp_v[i][j]$ for every block boundary, so $\mathcal{O}(n^2/b)$ values overall. This will allow us later to recover any $dp_v[i][j]$ in constant time by retrieving the value at the appropriate block boundary and adding the number of 1s in a prefix of some bitmask. We preprocess such prefix sums for every possible bitmask in $\mathcal{O}(2^b \cdot b)$ time and space. To implement updates efficiently we also need the following lemma[1].

▶ **Lemma 2.** $0 \le dp_{v+1}[i][j] - dp_v[i][j] \le 1$

Proof. Because allowing using more elements cannot decrease the length, $dp_v[i][j] \le dp_{v+1}[i][j]$. Let C be a sequence corresponding to $dp_{v+1}[i][j]$, and let C' be C without the last element. Because C is strictly increasing, the elements of C' are not larger than v, so $|C'| \le dp_v[i][j]$. Then, using $|C'|+1 = |C|$ we obtain that $dp_{v+1}[i][j] - 1 \le dp_v[i][j]$. ◀

We now describe how to obtain the table storing the values of dp'_{v+1} by modifying the table storing the values of dp'_v. To this end, we use the recursion for $dp_{v+1}[i][j]$ and process the rows one-by-one. We start by copying the corresponding i-th row of dp'_v, and then update the entries going from left to right. In the j-th step, we would like to have correctly determined the values of $dp'_{v+1}[i][1], dp'_{v+1}[i][2], \ldots, dp'_{v+1}[i][j]$ that together encode

[1] This lemma does not hold for LCWIS.

the values of $dp_{v+1}[i][1], dp_{v+1}[i][2], \ldots, dp_{v+1}[i][j]$. However, during this process we are no longer guaranteed that $dp_{v+1}[i][j] \leq dp_{v+1}[i][j+1]$, To overcome this issue, we immediately propagate each value to the right: after increasing $dp_{v+1}[i][j]$ (by one due to Lemma 2) we also increase every $dp_{v+1}[i][k]$ equal to the original value of $dp_{v+1}[i][j]$, for all $k > j$. This translates into setting $dp'_{v+1}[i][j]$ to 1 and setting $dp'_{v+1}[i][k]$ to 0, for the smallest $k > j$ such that $dp'_{v+1}[i][k] = 1$, if such k exists. To implement this efficiently, we maintain k while considering $j = 1, 2, \ldots, n$ in $\mathcal{O}(n)$ overall time. The details of this procedure are shown in Algorithm 1.

Algorithm 1 Calculate the i-th row of dp'_{v+1}.

procedure CalculateRow(v, i)
    $ptr \leftarrow 1$
    $cur\_value \leftarrow 0$
    $prv\_value \leftarrow 0$
    $prv\_phase \leftarrow 0$
    for $j = 1..n$ **do**
        $dp'_{v+1}[i][j] = dp'_v[i][j]$
    for $j = 1..n$ **do**
        if $ptr \leq j$ **then** $ptr \leftarrow j + 1$
        while $ptr \leq n$ **and** $dp'_{v+1}[i][ptr] = 0$ **do**
            $ptr \leftarrow ptr + 1$
        $cur\_value \leftarrow cur\_value + dp'_{v+1}[i][j]$
        $\triangleright$ $cur\_value = \sum_{j'=1}^{j} dp'_{v+1}[i][j'] = \max\{dp_v[i][j], dp_{v+1}[i][j-1]\}$
        $\triangleright$ $prv\_phase = dp_v[i-1][j-1]$
        if $A[i] = B[j] = v + 1$ **and** $cur\_value = prv\_phase$ **then**
            $dp'_{v+1}[i][j] \leftarrow 1$
            $cur\_value \leftarrow cur\_value + 1$
            if $ptr \leq n$ **then** $dp'_{v+1}[i][ptr] \leftarrow 0$
        $prv\_phase \leftarrow prv\_phase + dp'_v[i-1][j]$
        $prv\_value \leftarrow prv\_value + dp'_{v+1}[i-1][j]$
        $\triangleright$ $prv\_value = dp_{v+1}[i-1][j]$
        if $cur\_value < prv\_value$ **then**
            $cur\_value \leftarrow prv\_value$
            $dp'_{v+1}[i][j] \leftarrow 1$
            if $ptr \leq n$ **then** $dp'_{v+1}[i][ptr] \leftarrow 0$

We speed up Algorithm 1 by a factor of b by considering whole blocks of dp'_{v+1} instead of single entries. Consider a single block of dp'_{v+1} consisting of the values of $dp'_{v+1}[i][j], dp'_{v+1}[i][j+1], \ldots, dp'_{v+1}[i][j+b-1]$, and assume that they have been already partially updated by propagating the maximum. To calculate their correct values we need the following information:

1. $dp'_v[i-1][j], dp'_v[i-1][j+1], \ldots, dp'_v[i-1][j+b-1]$,
2. $dp'_{v+1}[i-1][j], dp'_{v+1}[i-1][j+1], \ldots, dp'_{v+1}[i-1][j+b-1]$,
3. $dp'_{v+1}[i][j], dp'_{v+1}[i][j+1], \ldots, dp'_{v+1}[i][j+b-1]$,
4. $dp_v[i-1][j-1]$,
5. $dp_{v+1}[i-1][j-1]$,
6. $dp_{v+1}[i][j-1]$,
7. for which indices $j, j+1, \ldots, j+b-1$ we have $A[i] = B[j] = v+1$.

In fact, we can rewrite the procedure so that instead of the values $dp_v[i-1][j-1]$, $dp_{v+1}[i-1][j-1]$, $dp_{v+1}[i][j-1]$ only the differences $dp_{v+1}[i-1][j-1] - dp_v[i-1][j-1]$ and $dp_{v+1}[i][j-1] - dp_{v+1}[i-1][j-1]$ are needed. By Lemma 1 and Lemma 2, both differences belong to $\{0,1\}$, so the whole information required for calculating the correct values consists of $4b+2$ bits. Blocks dp' are already stored in separate machine words, and we can prepare, for every v, an array with the j-th entry set to 1 when $B[j] = v$, partitioned into n/b blocks of length b, where each block is saved in a single machine word, in $\mathcal{O}(|\sigma| \cdot n)$ time. This allows us to gather all the required information in constant time and use a precomputed table of size $\mathcal{O}(2^{4b+2})$ that stores a single machine word encoding the correct values in a block for every possible combination. Additionally, the table stores the number of 1s to the right of the block that should be changed to 0. The table can be prepared in $\mathcal{O}(2^{4b+2} \cdot b)$ time by a straightforward modification of Algorithm 1. Now we can update a whole block in constant time by retrieving the precomputed answer, but then we still might need to remove some 1s on its right. Instead of removing them one-by-one we work block-by-block. In more detail, we maintain a pointer to the nearest block that might contain a 1. Let the number of 1s there be ℓ and the number of 1s that still need to be removed be s. As long as $s > 0$, we remove $\min\{\ell, s\}$ leftmost 1s from the current block in constant time using a precomputed table of size $\mathcal{O}(2^b \cdot b)$, decrease s by $\min\{\ell, s\}$, and move to the next block. This amortises to constant time per block over the row.

We set $b = \frac{\log n}{5}$ as to make the required preprocessing $o(n)$. Then, the overall complexity of the algorithm becomes $\mathcal{O}(|\sigma| \cdot n^2/\log n)$.

4 Second Solution (Rare Elements)

In this section we describe an algorithm for solving LCIS in $\mathcal{O}(\sum_{v=1}^{|\sigma|} (\mathsf{cnt}(v))^2(1+\log^2(n/\mathsf{cnt}(v))))$ time.

For every matching pair (x, y), we will compute $LCIS^{\rightarrow}(x, y)$, called the result for (x, y). The algorithm proceeds in phases corresponding to the elements of σ, and in the v-th step computes the results for all v-pairs. During this computation we maintain, for every $r = 1, 2, \ldots, n$, a structure $D(r)$ that allows us to quickly determine, given any (x, y), if there exists an already processed matching pair (x', y') with result r such that $x' < x$ and $y' < y$. Each $D(r)$ is implemented using the following lemma.

▶ **Lemma 3.** *We can maintain a set of points $S \subseteq [n] \times [n]$ under inserting a batch of $u \leq n$ points in amortised $\mathcal{O}(u(1 + \log \frac{n}{u}))$ time and answering a batch of $q \leq n$ queries of the form "given (x, y), is there $(x', y') \in S$ such that $x' < x$ and $y' < y$" in $\mathcal{O}(q(1 + \log \frac{n}{q}))$ time.*

Proof. We first describe a slower solution that achieves the claimed bounds only for $q = u = 1$, and then extend it to larger values of q and u. For the latter, we could have also used balanced search trees with dynamic finger property, such as the level linked (2,4)-trees [12]. However, this results in a somewhat complicated solution, and we opt for a self-contained description. We also note that the related question of implementing basic operations on two sets of size n and m, where $m \leq n$, in time $\mathcal{O}(m \log(n/m))$ goes back to the work of Brown and Tarjan [8].

We observe that if the current S contains two distinct points (x_i, y_i) and (x_j, y_j) with $x_i \leq x_j$ and $y_i \leq y_j$ then there is no need to keep (x_j, y_j). Thus, we keep in S only points that are not dominated. Let $(x_1, y_1), \ldots, (x_k, y_k)$ be these points arranged in the increasing order of x coordinates (observe that we cannot have two non-dominated points with the same x coordinate). So, $x_1 < x_2 < \ldots < x_k$, where $k \leq n$, and because the points are not dominated also $y_1 > y_2 > \ldots > y_k$. We store the x coordinates in a BST. This clearly allows us to answer a single query (x, y) in $\mathcal{O}(\log n)$ time by locating the predecessor of x. To insert a point (x, y), we first check that it is not dominated by locating the predecessor of x. Then, we might need to remove some of the subsequent x coordinates that correspond to points that are dominated by (x, y). This can be efficiently implemented by maintaining a doubly-linked list of all points, and linking each x coordinate with its corresponding point. Insertion takes $\mathcal{O}(\log n)$ time plus another $\mathcal{O}(\log n)$ for every removed point, so $\mathcal{O}(\log n)$ amortised time, and a query concerning (x, y) reduces to finding the predecessor of x among the x_is in $\mathcal{O}(\log n)$ time.

We first explain how to process a batch of q queries. We first sort them in $\mathcal{O}(q(1+\log(n/q)))$ time using radix sort with base q. We use a BST that allows split and merge in $\mathcal{O}(\log s)$ time, where s is the number of stored elements, for example AVL trees. Additionally, we store the size of the subtree in every node. Then we have the following easy proposition.

▶ **Proposition 4.** *We can split BST into at most b smaller BSTs containing $\Theta(s/b)$ elements each in $\mathcal{O}(b(1 + \log \frac{s}{b}))$ time.*

Proof. As long as there is a BST of size at least $2s/b$ we split it into two BSTs of (roughly) equal sizes. Assuming for simplicity that both s and b are powers of 2, this takes $\mathcal{O}(\sum_{i=0}^{\log b-1} 2^i \log(s/2^i))$ overall time, which can be bounded by calculating $\int_1^b \log(s/x)dx = \mathcal{O}(b(1 + \log(s/b)))$. ◀

We split the BST into at most q smaller BSTs containing $\Theta(s/q)$ elements each, where s is the number of stored elements, using Proposition 4. Because queries are sorted, we can determine for each of them the relevant BST by a linear scan, and then query the relevant BST in $\mathcal{O}(1 + \log(s/q))$ time, so $\mathcal{O}(q(1 + \log \frac{n}{q}))$ overall.

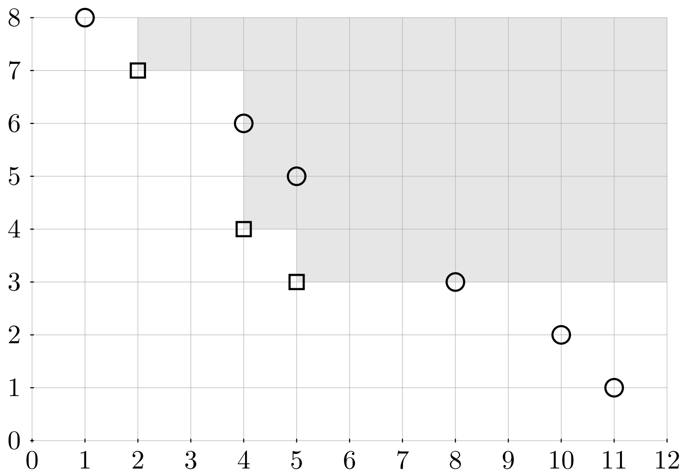

Figure 1 Processing a batch of 3 insertions, with the already existing and new points denoted by circles and squares, respectively.

We now explain how to process a batch of u insertions. We start with determining which of the new points are dominated by the already stored points in $\mathcal{O}(u(1 + \log(s/u))$ time using the above method. This also allows us to determine, for each new point (x, y), the range of already stored points $(x_i, y_i), (x_{i+1}, y_{i+1}), \ldots, (x_j, y_j)$ that should be removed from the structure because of inserting (x, y). See Figure 1. This takes additional $\mathcal{O}(\ell)$ time by traversing the doubly-linked list, where ℓ is the number of points to be removed. As in a query, we split the BST into at most u smaller BSTs containing $\Theta(s/u)$ elements each, and merge a sorted list of new points with the list of smaller BSTs in $\mathcal{O}(u(1+\log(s/u)))$ time. Then, each range of the points that should be removed is either fully contained in a single smaller BSTs, or consists of a prefix of a smaller BST, then a range of full smaller BSTs, and finally a suffix of a smaller BSTs. By splitting a smaller BST in $\mathcal{O}(\log(s/q))$ time and assigning a single credit to every stored element, we can hence implement all deletions in $\mathcal{O}(u(1+\log(s/u)))$ time. Finally, we insert each new point into the appropriate smaller BST. This might take more than $\mathcal{O}(\log(s/u))$ time per element if there are more than s/u insertions to the same smaller BST. In such case, we build an AVL tree containing all these $\ell \geq s/u$ new points in $\mathcal{O}(\ell)$ time, and then insert the $\Theta(s/q)$ already existing points there in $\mathcal{O}(s/q \log \ell) = \mathcal{O}(\ell \log(s/q))$ time, and discard the smaller BST. Finally, we merge the BSTs into pairs, quadruples, and so on. By the calculation from the proof of Proposition 4 this also takes $\mathcal{O}(u(1+\log(u/b)))$ time. ◀

Lemma 3 is already enough to binary search for the result of (x,y) in $\mathcal{O}(\log^2 n)$ time due to the following property.

▶ **Lemma 5.** *Consider any r and an already processed matching pair (x',y') with result r. Then either $r=1$ or there exists an already processed matching pair (x'',y'') with result $r-1$ such that $x''<x'$ and $y''<y'$.*

Proof. Assume that $r \geq 2$ and consider a sequence C which realises the result for (x',y'). Then $C[1..|C|-1]$ is an increasing subsequence of both $A[1..(x'-1)]$ and $B[1..(y'-1)]$. Let $A[x'']$ and $B[y'']$ be its last elements in A and B, respectively. Then $x''<x'$, $y''<y'$, and $A[x'']=B[y'']$, so (x'',y'') is a matching pair, and because C is strictly increasing this matching pair must have been already processed. ◀

However, our goal is to spend $\mathcal{O}(1+\log^2(n/\mathsf{cnt}(v)))$ time per every (x,y). We exploit the following property.

▶ **Lemma 6.** *Consider two v-pairs (x,y_1) and (x,y_2), where $y_1<y_2$. The result for (x,y_2) is at least as large as for (x,y_1).*

Proof. Consider a sequence C which realises $LCIS^{\rightarrow}(x,y_1)$. Then, replacing y_1 with y_2 we obtain a valid candidate for the value of $LCIS^{\rightarrow}(x,y_2)$. ◀

Consider all v-pairs with the same x coordinate $(x,y_1),(x,y_2),\ldots,(x,y_{\mathsf{cnt}(v)})$. We binary search for the result of (x,y_i) for $i=\mathsf{cnt}(v),\ldots,2,1$. By Lemma 6, in the i-th step we can start with the result found in the $(i+1)$-th step. Using exponential search [4], by convexity of the log function the overall complexity becomes $\mathcal{O}(\mathsf{cnt}(v)(1+\log(n/\mathsf{cnt}(v))))$. This is still too slow, as every step involves a separate invocation of Lemma 3 and takes $\mathcal{O}(\log n)$ time. To obtain the final speed up, we process all x coordinates $x_1,x_2,\ldots,x_{\mathsf{cnt}(v)}$ together. The high level idea is to synchronise all exponential searches and exploit the possibility of asking a batch of queries.

We start with modifying the proof of Lemma 3 to allow for more general queries: given x, we want to find the smallest y such that there exists $(x',y')\in S$ with $x'<x$ and $y'<y$ (or detect that there is none). The modification is straightforward and does not increase the time complexity. Now we can restate processing all pairs with the same x coordinates. We start with a counter c initially set to n and i set to $\mathsf{cnt}(v)$. As long as $i \geq 1$, we use exponential search starting at c to find the result for (x,y_i). Let c' be the found result. We use the modified Lemma 3 to determine the smallest y such that c' is the result for (x,y) and then keep decreasing i as long as $i \geq 1$ and $y_i > y$. Then, we decrease c' by 1 and repeat.

We further reformulate processing all pairs with the same x coordinate. Consider a conceptual complete binary tree on n leaves (without losing generality, n is a power of 2). Every node corresponds to an interval $[a, b]$, and by querying such a node we will understand querying structure $D(a)$ with the current (x, y_i). Consider the leaf corresponding to c. Calculating c' with exponential search can be phrased as starting at the leaf corresponding to c and going up as long as the query at the current node fails (we only need to ask a query if the previous node was the right child of the current node; otherwise, we can immediately jump to the nearest ancestor with such property). After having reached the first ancestor for which the query succeeds, we descend from its left child to the leaf corresponding to c' by repeating the following step: if querying the right child of the current node succeeds we descend to the right child, and otherwise we descend to the left child. See Figure 2.

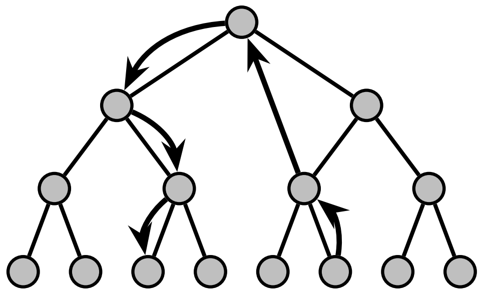

Figure 2 Exponential search for the next node phrased as traversing the binary tree.

Now we are able to synchronize the exponential searches as follows. We traverse the conceptual complete binary tree recursively: to traverse the subtree rooted at node u with children u_ℓ and u_r we (i) visit u, (ii) recursively traverse the subtree rooted at u_r, (iii) visit u again, (iv) recursively traverse the subtree rooted at u_ℓ. Thus, every node is visited twice. We claim that when visiting the nodes of the conceptual complete binary tree using this strategy, for any x coordinate we are always able to wait till we encounter the node that should be queried next. This is formalised in the following lemma.

▶ **Lemma 7.** *Let the result for (x, y_{i+1}) be c and the result for (x, y_i) be $c' < c$. All queries necessary to calculate c' can be answered during the traversal after the second visit to c and before the second visit to c'.*

Proof. The calculation consists of two phases. First, we need to ascend from the leaf corresponding to c, reaching its first ancestor u at which the query fails. Recall we only need to ask queries if the previous node is the right child of the current node. For each such node v we will be able to use second visit to v in the traversal. Thus, we will process all such queries after the second visit to u. Then, we need to descend from the left child of u. In every step, we query the right child v_r of the current node v, and continue either in the left or in the right subtree of v. To this end, we use the first visit to v_r in the traversal. ◀

For each x coordinate, by convexity of the log function, we need to query at most $\mathcal{O}(\mathsf{cnt}(v)(1+\log(n/\mathsf{cnt}(v))))$ nodes of the conceptual binary tree. Denoting by q_u the number of queries to a node u, we thus have $\sum_u q_u = s = \mathcal{O}(\mathsf{cnt}(v)^2(1 + \log(n/\mathsf{cnt}(v))))$. Invoking Lemma 3, the total time to answer all these queries is $\sum_u q_u(1 + \log(n/q_u))$. By convexity of the function $f(x) = x \log(n/x)$, this is maximised when all q_us are equal, but there are only n of them, making the total time :

$$\sum_u q_u(1 + \log(n/q_u)) \leq s(1 + \log(n^2/s)) \leq s(1 + \log(n^2/\mathsf{cnt}(v)^2))$$
$$= \mathcal{O}(\mathsf{cnt}(v)^2(1 + \log(n/\mathsf{cnt}(v)))^2).$$

5 Combining Solutions

Let c be a parameter to be fixed later. We call v *frequent* if $\frac{n}{c} < \mathsf{cnt}(v)$, and *rare* otherwise.

We partition the sequence σ into fragments. Each fragment is either a single frequent element or a maximal range of rare elements. By definition of a frequent element and maximality of fragments consisting of rare elements, we have $\mathcal{O}(c)$ fragments. We maintain the dp_v table as in the first solution, but we only update it after having processed a whole fragment. So, when considering a fragment starting at v we only assume that the values of dp_{v-1} can be accessed in constant time. For a fragment consisting of a single frequent element, we proceed exactly as in the first solution. In the remaining part of the description we describe how to process a fragment consisting of rare elements $v, v+1, \ldots$.

We consider all v'-pairs, for $v' = v, v+1, \ldots$. We will compute $LCIS^{\rightarrow}(x, y)$ for each such matching pair (x, y), and store it in the appropriate structure $D(r)$ implemented as described in Lemma 3. To compute the values of $LCIS^{\rightarrow}(x, y)$ for all v'-pairs, we use parallel exponential search as in the second solution with the following modification. To check if $LCIS^{\rightarrow}(x, y_i) > r$, we need to consider two possibilities for the corresponding sequence C ending at $A[x] = B[y_i] = v'$:

1. If $C[|C|-1]$ belongs to the same fragment then it is enough to check if $D(r)$ contains a pair (x', y') with $x' < x$ and $y' < y_i$.
2. Otherwise, it is enough to check if $dp_{v-1}[x][y_i] \geq r$.

Additionally, after having found c' we need to keep decreasing i as long as $i \geq 1$ and the answer for (x, y_i) is c', and this needs to be tested in constant time per each such i. We again need to consider two possibilities, and either compare y_i with the value of y' found by querying $D(c'-1)$ with x, or test if $dp_{v-1}[x][y_i] \geq r$ in constant time. Overall, this incurs only additional constant time per every step of the exponential search for every considered matching pair.

After having considered all v'-pairs for the last element v' in the current fragment, we need to compute $dp_{v'}$ from dp_{v-1} and the calculated values of $LCIS^{\rightarrow}$. Of course, we want to operate on $dp'_{v'}$ and dp'_{v-1} instead of $dp_{v'}$ and dp_{v-1}. This is done row-by-row. The i-th row is computed in two steps.

First, we need to set $dp_{v'}[i][j] = \max\{dp_{v'}[i-1][j], dp_{v-1}[i][j]\}$ for every $j = 1, 2, \ldots, n$. This is done by processing whole blocks in constant time and precomputing the result for every possible combination of the following information:

1. $dp'_{v'}[i-1][j], dp'_{v'}[i-1][j+1], \ldots, dp'_{v'}[i-1][j+b-1]$,
2. $dp'_{v-1}[i][j], dp'_{v-1}[i][j+1], \ldots, dp'_{v-1}[i][j+b-1]$,
3. $dp_{v'}[i-1][j-1]$,
4. $dp_{v-1}[i][j-1]$.

This can be preprocessed in $\mathcal{O}(4^b \cdot b^2)$ time after observing that, as in the first solution, only the difference $dp_{v'}[i-1][j-1] - dp_{v-1}[i][j-1]$ is relevant and, additionally, it can be capped at b (if it is bigger than b then we can set it to b). The time is $\mathcal{O}(n/b)$.

Second, we need to consider the values of $LCIS^{\rightarrow}(i, j)$ computed for the current fragment. If the result computed for a matching pair (i, j) is r then we need to update $dp_v[i][j'] = \max\{dp_v[i][j'], r\}$, for every $j' \geq j$. This can be done by simultaneously scanning all such js and the blocks. By maintaining the maximum r, we can update the value of $dp_v[i][j]$ at the beginning of the block. Then, we consider all other j's belonging to the same block, and consider its corresponding result r'. If $dp_v[i][j'] \geq r'$ then this result is irrelevant, and otherwise we must increase some of the values in the block by 1 (as $dp_v[i][j'-1]$ is assumed to have been already updated and due to Lemma 1). As in the first solution, this is implemented by setting $dp'_v[i][j'] = 1$ and changing the nearest 1 into 0. Overall, the time is bounded by the number of considered matching pairs plus additional $\mathcal{O}(n/b)$ time.

We set $b = \frac{\log n}{5}$ so that the preprocessing time is $o(n)$. For each frequent element we spend $\mathcal{O}(n^2/b)$ time, so $\mathcal{O}(n^2/b \cdot c)$ overall. For each fragment consisting of rare elements, the time is $\mathcal{O}(\mathsf{cnt}(v)^2 \log^2(n/\mathsf{cnt}(v)))$ for every v to compute the results, and then $\mathcal{O}(n^2/b)$ plus the number of results. Using $\mathsf{cnt}(v) \le n/c$, where c is sufficiently large, and calculating the derivative of $f(x) = x\log^2(n/x)$ we upper bound $\mathsf{cnt}(v)\log^2(n/\mathsf{cnt}(v)) \le n/c \cdot \log^2 c$ for every rare v, so the overall time is $\mathcal{O}(n^2/b \cdot c + n/c \cdot \log^2 c \sum_v \mathsf{cnt}(v)) = \mathcal{O}(n^2/b \cdot c + n^2/c \cdot \log^2 c)$.

Choosing $c = \sqrt{\log n}\log\log n$ we obtain an algorithm working in $\mathcal{O}(n^2 \log\log n/\sqrt{\log n})$ time.

6 Longest Common Weakly Increasing Subsequence

In this section we explain how to modify the algorithm to solve the weakly increasing version of the problem. We adapt both solutions without changing their complexity as explained below, and then combine them using the same threshold for the frequent/rare elements to arrive at $\mathcal{O}(n^2 \log\log n/\sqrt{\log n})$ complexity.

6.1 First solution

We define dp as in the algorithm for LCIS. It can be calculated using the following recurrence (slightly different than for LCIS):

$$dp_{v+1}[i][j] = \begin{cases} dp_{v+1}[i-1][j-1]+1, & \text{if } A[i]=B[j]=v+1, \\ \max\{dp_v[i][j], dp_{v+1}[i-1][j], dp_{v+1}[i][j-1]\}, & \text{otherwise.} \end{cases}$$

The proof of Lemma 1 still holds, so we can store a table dp' and retrieve any value of dp from dp' in constant time.

Algorithm 1 stays essentially the same so we skip a detailed explanation. The speed up is implemented by considering whole blocks of dp'_{v+1} instead of single entries. Consider a single block of dp'_{v+1} consisting of the values of $dp'_{v+1}[i][j], dp'_{v+1}[i][j+1], \ldots, dp'_{v+1}[i][j+b-1]$, and assume that they have been already partially updated by propagating the maximum. To calculate their correct values we need the following information:

1. $dp'_{v+1}[i-1][j], dp'_{v+1}[i-1][j+1], \ldots, dp'_{v+1}[i-1][j+b-1]$,
2. $dp'_{v+1}[i][j], dp'_{v+1}[i][j+1], \ldots, dp'_{v+1}[i][j+b-1]$,
3. $dp_{v+1}[i-1][j-1]$,
4. $dp_{v+1}[i][j-1]$,
5. for which indices $j, j+1, \ldots, j+b-1$ we have $A[i]=B[j]=v+1$.

Once again we can rewrite the procedure so that instead of the values $dp_{v+1}[i-1][j-1]$ and $dp_{v+1}[i][j-1]$ only the difference $dp_{v+1}[i][j-1] - dp_{v+1}[i-1][j-1]$ is needed. By Lemma 1, the difference belongs to $\{0,1\}$, so the whole information required for calculating the correct values consists of $3b+1$ bits. This allows us to update the whole table in $\mathcal{O}(n^2/b)$ as for LCIS.

We set $b = \frac{\log n}{4}$ as to make required preprocessing $o(n)$. Overall complexity of the algorithm becomes $\mathcal{O}(|\sigma|n^2/\log n)$.

6.2 Second solution

Calculating the result for each v-pair consists of two phases. In the first phase, for each v-pair (x,y), we calculate the result assuming that all previous elements in the subsequence are strictly smaller than v. In the second phase, we calculate the result assuming that the previous element is also equal to v. The first phase can be implemented exactly as for LCIS in $\mathcal{O}(\mathsf{cnt}(v)^2(1+\log^2(n/\mathsf{cnt}(v))))$ time. We now focus on explaining how to implement the second phase. Let $\mathsf{prev}_A[x]$ denote the greatest x' fulfilling $A[x'] = A[x]$, if there is no such then $\mathsf{prev}_A[x] = 0$. Similarly we define $\mathsf{prev}_B[y]$, both arrays can be prepared in negligible $\mathcal{O}(n \log n)$ time.

We analyze all v-pairs in the increasing order of rows and columns. Consequently, when analysing a pair (x, y), for all other v-pairs with $x' \leq x$, $y' \leq y$ we have already correctly calculated $LCWIS^{\rightarrow}(x', y')$. The proof of Lemma 6 still holds for LCWIS, and implies that among all other v-pairs (x', y') such that $x' \leq x$ and $y' \leq y$ the pair $(\mathsf{prev}_A[x], \mathsf{prev}_B[y])$ has the largest result. We can calculate $LCWIS^{\rightarrow}(x, y)$ as the maximum of the result computed in the first phase and $LCWIS^{\rightarrow}(\mathsf{prev}_A[x], \mathsf{prev}_B[y]) + 1$.

The second phase takes only $\mathcal{O}(\mathsf{cnt}(v)^2)$ time, so the overall complexity remains $\mathcal{O}(\mathsf{cnt}(v)^2(1+\log^2(n/\mathsf{cnt}(v))))$.

7 Conclusions

The $\mathcal{O}(n^2 \log\log n/\sqrt{\log n})$ complexity does not seem to be right answer yet, at least for LCIS. It seems to us that one can apply the combinatorial bound of Duraj on the number of significant pairs, and combine it with our approach, to achieve an even better complexity. However, as this does not seem to result in a clean bound of (say) $\mathcal{O}(n^2/\log n)$ yet, we leave determining the exact complexity for future work.

References

1. Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *56th FOCS*, pages 59–78, 2015.
2. Amir Abboud and Karl Bringmann. Tighter connections between formula-SAT and shaving logs. In *45th ICALP*, pages 8:1–8:18, 2018.
3. Vladimir Arlazarov, Yefim Dinitz, M. A. Kronrod, and Igor Faradžev. On economical construction of the transitive closure of an oriented graph. *Doklady Akademii Nauk*, 194:487–488, 1970.
4. Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3):82–87, 1976.
5. Philip Bille and Martin Farach-Colton. Fast and compact regular expression matching. *Theor. Comput. Sci.*, 409(3):486–496, 2008.
6. Philip Bille and Mikkel Thorup. Faster regular expression matching. In *36th ICALP*, pages 171–182, 2009.
7. Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *56th FOCS*, pages 79–97, 2015.
8. Mark R. Brown and Robert Endre Tarjan. A fast merging algorithm. *J. ACM*, 26(2):211–226, 1979.
9. Lech Duraj. A sub-quadratic algorithm for the longest common increasing subsequence problem. In *37th STACS*, pages 41:1–41:18, 2020.
10. Lech Duraj, Marvin Künnemann, and Adam Polak. Tight conditional lower bounds for longest common increasing subsequence. *Algorithmica*, 81(10):3968–3992, 2019.
11. Szymon Grabowski. New tabulation and sparse dynamic programming based techniques for sequence similarity problems. *Discret. Appl. Math.*, 212:96–103, 2016.
12. Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

13 Martin Kutz, Gerth Stølting Brodal, Kanela Kaligosi, and Irit Katriel. Faster algorithms for computing longest common increasing subsequences. *J. Discrete Algorithms*, 9(4):314–325, 2011.

14 Eugene W. Myers. A four Russians algorithm for regular expression pattern matching. *J. ACM*, 39(2):430–448, 1992.

15 Adam Polak. Why is it hard to beat $\mathcal{O}(n^2)$ for longest common weakly increasing subsequence? *Inf. Process. Lett.*, 132:1–5, 2018.

16 Yoshifumi Sakai. A linear space algorithm for computing a longest common increasing subsequence. *Inf. Process. Lett.*, 99(5):203–207, 2006.

17 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

18 R. Ryan Williams. Faster all-pairs shortest paths via circuit complexity. *SIAM J. Comput.*, 47(5):1965–1985, 2018.

19 I-Hsuan Yang, Chien-Pin Huang, and Kun-Mao Chao. A fast algorithm for computing a longest common increasing subsequence. *Inf. Process. Lett.*, 93(5):249–253, 2005.