Longest Square Subsequence Problem Revisited

Takafumi Inoue\textsuperscript{1}, Shunsuke Inenaga\textsuperscript{1,2}, and Hideo Bannai\textsuperscript{3}

\textsuperscript{1} Department of Informatics, Kyushu University, Fukuoka, Japan
\textsuperscript{2} PRESTO, Japan Science and Technology Agency, Kawaguchi, Japan
\textsuperscript{3} M&D Data Science Center, Tokyo Medical and Dental University, Tokyo, Japan

\texttt{inenaga@inf.kyushu-u.ac.jp}
\texttt{hdbn.dsc@tmd.ac.jp}

Abstract

The longest square subsequence (LSS) problem consists of computing a longest subsequence of a given string $S$ that is a square, i.e., a longest subsequence of form $XX$ appearing in $S$. It is known that an LSS of a string $S$ of length $n$ can be computed using $O(n^2)$ time [Kosowski 2004], or with (model-dependent) polylogarithmic speed-ups using $O(n^2(\log \log n)^2/\log^2 n)$ time [Tiskin 2013]. We present the first algorithm for LSS whose running time depends on other parameters, i.e., we show that an LSS of $S$ can be computed in $O(r \min\{n, M\} \log \frac{n}{r} + M \log n)$ time with $O(M)$ space, where $r$ is the length of an LSS of $S$ and $M$ is the number of matching points on $S$.

1 Introduction

Subsequences of a string $S$ with some interesting properties have caught much attention in mathematics and algorithmics. The most well-known of such kinds is the longest increasing subsequence (LIS), which is a longest subsequence of $S$ whose elements appear in lexicographically increasing order. It is well known that an LIS of a given string $S$ of length $n$ can be computed in $O(n \log n)$ time with $O(n)$ space [8]. Other examples are the longest palindromic subsequence (LPS) and the longest square subsequence (LSS). Since an LPS of $S$ is a longest common subsequence (LCS) of $S$ and its reversal, an LPS can be computed by a classical dynamic programming for LCS, or by any other LCS algorithms.

Computing an LSS of a string is not as easy, because a reduction from LSS to LCS essentially requires to consider $n - 1$ partition points on $S$. Kosowski [6] was the first to tackle this problem, and showed an $O(n^2)$-time $O(n)$-space LSS algorithm. Computing LSS can be motivated by e.g. finding an optimal partition point on a given string so that the corresponding prefix and suffix are most similar. Later, Tiskin [9] presented a (model-dependent) $O(n^2(\log \log n)^2/\log^2 n)$-time LSS algorithm, based on his semi-local string comparison technique applied to the $n - 1$ partition points (i.e. $n - 1$ pairs of prefixes and suffixes.) Since strongly sub-quadratic $O(n^{2-\epsilon})$-time LSS algorithms do not exist for any $\epsilon > 0$ unless the SETH is false [2], the aforementioned solutions are almost optimal in terms of $n$. 
In this paper, we present the first LSS algorithm whose running time depends on other parameters, i.e., we show that an LSS of $S$ can be computed in $O(r \min\{n, M\} \log \frac{n}{r} + M \log n)$ time with $O(M)$ space, where $r$ is the length of an LSS of $S$ and $M$ is the number of matching points on $S$. This algorithm outperforms Tiskin’s $O(n^2(\log \log n)^2/\log^2 n)$-time algorithm when $r = o(n(\log \log n)^2/\log^2 n)$ and $M = o(n^2(\log \log n)^2/\log^3 n)$.

Our algorithm is based on a reduction from computing an LCS of two strings of total length $n$ to computing an LIS of an integer sequence of length at most $M$, where $M$ is roughly $n^2/\sigma$ for uniformly distributed random strings over alphabets of size $\sigma$. We then use a slightly modified version of the dynamic LIS algorithm \cite{3} for our LIS instances that dynamically change over $n-1$ partition points on $S$. A similar but more involved reduction from LCS to LIS is recently used in an intermediate step of a reduction from dynamic time warping (DTW) to LIS \cite{7}. We emphasize that our reduction (as well as the one in \cite{7}) from LCS to LIS should not be confused with a well-known folklore reduction from LIS to LCS.

2 Preliminaries

Let $\Sigma$ be an alphabet. An element $S$ of $\Sigma^*$ is called a string. The length of a string $S$ is denoted by $|S|$. For any $1 \leq i \leq |S|$, $S[i]$ denotes the $i$th character of $S$. For any $1 \leq i \leq j \leq |S|$, $S[i..j]$ denotes the substring of $X$ beginning at position $i$ and ending at position $j$.

A string $X$ is said to be a subsequence of a string $S$ if there exists a sequence $1 \leq i_1 < \cdots < i_{|X|} \leq |S|$ of increasing positions of $S$ such that $X = S[i_1] \cdots S[i_{|X|}]$. Such a sequence $i_1, \ldots, i_{|X|}$ of positions in $S$ is said to be an occurrence of $X$ in $S$.

A non-empty string $Y$ of form $XX$ is called a square. A square $Y$ is called a square subsequence of a string $S$ if square $Y$ is a subsequence of $S$. Let $\text{LSS}(S)$ denote the length of the longest square subsequence (LSS) of string $S$. This paper deals with the problem of computing $\text{LSS}(S)$ for a given string $S$ of length $n$.

For strings $A, B$, let $	ext{LCS}(A, B)$ denote the length of the longest common subsequence (LCS) of $A$ and $B$. For a sequence $T$ of numbers, a subsequence $X$ of $T$ is said to be an increasing subsequence of $T$ if $X[i] < X[i+1]$ for $1 \leq i < |X|$. Let $\text{LIS}(T)$ denote the length of the longest increasing subsequence (LIS) of $T$.

A pair $(i, j)$ of positions $1 \leq i < j \leq |S|$ is said to be a matching point if $S[i] = S[j]$. The set of all matching points of $S$ is denoted by $\mathcal{M}(S)$, namely, $\mathcal{M}(S) = \{(i, j) \mid 1 \leq i < j \leq |S|, S[i] = S[j]\}$. Let $M = |\mathcal{M}(S)|$.

3 Algorithm

We begin with a simple folklore reduction of computing $\text{LSS}(S)$ to computing the LCS of $n-1$ pairs of the prefix and the suffix of $S$.

Lemma 1 \cite{6} $\text{LSS}(S) = 2 \max_{1 \leq p < n} \text{LCS}(S[1..p], S[p+1..n])$.

Following Lemma \cite{6}, one can use the decremental LCS algorithm by Kim and Park \cite{5} for computing $\text{LSS}(S)$. Given two strings $A$ and $B$ of length $n$, Kim and Park proposed how
to update, in $O(n)$ time, an $O(n^2)$-space representation for the dynamic programming table for $LCS(A, B)$ when the leftmost character is deleted from $B$. Since their algorithm also allows to append a character to $A$ in $O(n)$ time, it turns out that $LSS(S)$ can be computed in $O(n^2)$ time and space. Kosowski [6] presented an $O(n^2)$-time $\Theta(n)$-space algorithm for computing $LSS(S)$, which can be seen as a space-efficient version of an application of Kim and Park’s algorithm to this specific problem of computing $LSS$. Tiskin [9] also considered the problem of computing $LSS(S)$, and showed that using his semi-local LCS method, $LSS(S)$ can be computed in $O(n^2(\log \log n)^2/\log^2 n)$ time. We remark that the log-shaving factor is model-dependent (i.e., Tiskin’s method uses the so-called “Four-Russian” technique).

Let $A = S[1..p]$, $A' = S[1..p + 1]$, $B = S[p + 1..n]$ and $B' = S[p + 2..n]$. For ease of explanations, suppose that the indices on $B$ and $B'$ begin with $p + 1$ and $p + 2$, respectively. Next, we further reduce computing $LCS(A', B')$ from (a representation of) $LCS(A, B)$, to computing an LIS of a dynamic integer sequence of length at most $M = |\mathcal{M}(S)|$.

For any integer pairs $(u, v)$ and $(x, y)$, let $(u, v) \prec (x, y)$ if (i) $u < x$, or (ii) $u = x$ and $v < y$. Consider the following integer sequence $T$: Let $\mathcal{P}$ be the set of integer pairs $(i, n - j)$ such that $A[i] = B[j]$. Then, we set $T[q] = j$ iff the integer pair $(i, n - j)$ is of rank $q$ in $\mathcal{P}$ w.r.t. $\prec$. See Fig. [1] for an example. Intuitively, $T$ is an integer sequence representation of the (transposed) matching points between $A$ and $B$, obtained by scanning the matching points between $A$ and $B$ from the bottom row to the top row, where each row is scanned from right to left. Thus, the length of the integer sequence $T$ is bounded by $M$.

**Lemma 2** Any common subsequence of $A$ and $B$ corresponds to an increasing subsequence of $T$ of the same length. Also, any increasing subsequence of $T$ corresponds to a common subsequence of $A$ and $B$ of the same length.

**Proof.** For any common subsequence $C$ of $A$ and $B$, let $i_1 < \cdots < i_{|C|}$ and $j_1 < \cdots < j_{|C|}$ be occurrences of $C$ in $A$ and $B$, respectively. For any $1 \leq k < |C|$, let $q_k$ and $q_{k+1}$ be the ranks of integer pairs $(i_k, n - j_k)$ and $(i_{k+1}, n - j_{k+1})$ in the set $\mathcal{P}$ w.r.t. $\prec$. By the definition of $T$, $q_k < q_{k+1}$ and $T[q_k] < T[q_{k+1}]$ hold. Hence, $C$ corresponds to an increasing subsequence of $T$ of the same length.

For any increasing subsequence $I$ in $T$, let $t_1 < \cdots < t_{|I|}$ be an occurrence of $I$ in $T$. For any $1 \leq k < |I|$, let $(i_k, n - j_k)$ and $(i_{k+1}, n - j_{k+1})$ be the integer pairs corresponding
Figure 2: Illustration on how points in the 2D plane (and elements in T) are to be deleted or inserted when A and B are updated to A' and B', respectively.

Let T' be the integer sequence for A' and B' defined analogously to T for A and B. Now the task is how to compute LIS(T') from (a data structure that represents) LIS(T). See Fig. 2 for an example. Observe that when the leftmost character is deleted from B (upper part of Fig. 2), then the lowest points are deleted from the 2D plane, and thus the smallest elements are deleted from T. Also, when the leftmost character of B is appended to A (upper part of Fig. 2), which gives us A' = S[1..p + 1], then a new point for every j with $A'[j] = B'[j]$ is inserted to the right end of the 2D plane in decreasing order of j, and thus j is appended to the right end of T in decreasing order of j, one by one. Thus, computing LCS(A', B') from LCS(A, B) reduces to the following sub-problem:

**Problem 1** For a dynamic integer sequence T, maintain a data structure that supports the following operations and queries:

- **Insertion**: Insert a new element to the right-end of T;
- **Batched Deletion**: Delete all the smallest elements from T;

By Lemma 2, computing LCS(A, B) can be reduced to computing LIS(T).
• Query: Return LIS(T).

We can use Chen et al.’s algorithm \[3\] for insertions. Let \( \ell = \text{LIS}(T) \). Their algorithm supports insertions at the right-end of \( T \) in \( O(\log |T|) \) time each. Since \( |T| \leq M \leq n^2 \), insertions at the right-end can be done in \( O(\log n) \) time.

Next, let us consider batched deletions. Chen et al. \[3\] showed that an insertion or deletion of a single element at an arbitrary position of \( T \) can be supported in \( O(\ell \log |T|) \) time each. However, since our batched deletion may contain \( O(|T|) \) characters in the worst case, a naïve application of a single-element deletion only leads to an inefficient \( O(\ell |T| \log \frac{n}{T}) \) \( \subseteq O(\ell M \log \frac{n}{T}) \) batched deletion. In what follows, we show how to support batched deletions in \( O(\ell \log \frac{n}{T}) \) time each, using Chen et al.’s data structure.

For any position \( 1 \leq t \leq |T| \) in sequence \( T \), let \( l(t) \) denote the length of an LIS of \( T[1..t] \) that has an occurrence \( i_1 < \cdots < i_l(t) = t \), namely, an occurrence that ends at position \( t \) in \( T \). The following observations are immediate:

**Lemma 3** (\[3\]) Let \( q \) be the second to last position of any occurrence of a length-\( l(t) \) LIS of \( T[1..t] \) ending at position \( t \). Then, \( l(q) = l(t) - 1 \).

**Lemma 4** (\[3\]) If \( q < t \) and \( l(q) = l(t) \), then \( T[q] \geq T[t] \).

For any \( 1 \leq k \leq \ell \), let \( \mathcal{L}_k \) be a list of pairs \( \langle t, T[t] \rangle \) such that \( l(t) = k \), sorted in increasing order of the first elements \( t \). See Fig. 3 for an example. It follows from Lemma 4 that this list is also sorted in non-increasing order of the second elements \( T[t] \). It is clear that \( \text{LIS}(T) = \max\{k \mid \mathcal{L}_k \neq \emptyset\} \). It is also clear that for any \( k > 1 \), if \( \mathcal{L}_k \neq \emptyset \), then \( \mathcal{L}_{k-1} \neq \emptyset \).

Thus, our task is to maintain a collection of the non-empty lists \( \mathcal{L}_1, \ldots, \mathcal{L}_\ell \) that are subject to change when \( T \) is updated to \( T' \). As in \[3\], we maintain each \( \mathcal{L}_k \) by a balanced binary search tree such as red-black trees \[4\] or AVL trees \[1\].

The following simple claim is a key to our batched deletion algorithm:

**Lemma 5** The pairs having the smallest elements of \( T \) are at the tail of \( \mathcal{L}_1 \).

Proof. Since the list \( \mathcal{L}_1 \) is sorted in non-increasing order of the second elements, the claim clearly holds. \(\square\)

**Lemma 6** We can perform a batched deletion of the smallest elements of \( T \) in \( O(\ell \log \frac{n}{T}) \) time, where \( \ell = \text{LIS}(T) \).

Proof. Due to Lemma 5 we can delete the smallest elements of \( T \) from the list \( \mathcal{L}_1 \) by splitting the balanced search tree into two, in \( O(\log |\mathcal{L}_1|) \) time.

The rest of our algorithm follows Chen et al.’s approach \[3\]: Note that the split operation on \( \mathcal{L}_1 \) can incur changes to the other lists \( \mathcal{L}_2, \ldots, \mathcal{L}_\ell \). Let \( l'(t) \) be the length of
an LIS of $T'[1..t]$ that has an occurrence ending at position $t$ in $T'$, and let $L'_k$ be the list of pairs $\langle t, T'[t]\rangle$ such that $l'(t) = k$ sorted in increasing order of the first elements $t$.

Let $Q_1$ be the list of deleted pairs corresponding to the smallest elements in $T$, and let $Q_k = \{ t \mid l(t) = k, l'(t) = k - 1 \}$ for $k \geq 2$. Then, it is clear that $L'_k = (L_k \setminus Q_k) \cup Q_{k+1}$.

Chen et al. [3] showed that $Q_k$ can be found in $O(\log |L_{k+1}|)$ time for each $k$, provided that $Q_k$ has been already computed. Since $Q_k$ is a consecutive sub-list of $L_k$, the split operation for $L_k \setminus Q_k$ can be done in $O(\log |L_k|)$ time, and the concatenation operation for $(L_k \setminus Q_k) \cup Q_{k+1}$ can be done in $O(\log |L_k| + \log |L_{k+1}|)$ time, by standard split and concatenation algorithms on balanced search trees.

Thus our batched deletion takes $O(\sum_{1 \leq k \leq \ell} \log |L_k|) = O(\log(|L_1| \cdots |L_\ell|))$ time, where $\ell = \text{LIS}(T)$. Since $\sum_{1 \leq k \leq \ell} |L_k| = |T|$ and $\log(|L_1| \cdots |L_\ell|)$ is maximized when $|L_1| = \cdots = |L_\ell|$, the above time complexity is bounded by $O(\ell \log \frac{|T|}{\ell}) \subseteq O(\ell \log \frac{n}{\ell})$ time.

We are ready to show our main theorem.

**Theorem 1** An LSS of a string $S$ can be computed in $O(r \min \{n, M\} \log \frac{n}{r} + M \log n)$ time with $O(M)$ space, where $n = |S|$, $r = \text{LSS}(S)$, and $M = |\text{M}(S)|$.

**Proof.** By Lemma [1] and Lemma [2] it suffices to consider the total number of insertions, batched deletions, and queries of Problem [1] for computing an LSS of our dynamic integer sequence $T$. Since each matching point in $\text{M}(S)$ is inserted to the dynamic sequence exactly once, the total number of insertions is exactly $M$. The total number of batched deletions is bounded by the number $n - 1$ of partition points $p$ that divide $S$ into $S[1..p]$ and $S[p+1..n]$. Also, it is clearly bounded by the number $M$ of matching points. Thus, the total number of batched deletions is at most $\min \{n, M\}$. We perform queries $n - 1$ times for all $1 \leq p < n$. Each query for $\text{LIS}(T)$ can be answered in $O(1)$ time, by explicitly maintaining and storing the value of $\text{LIS}(T)$ each time the dynamic integer sequence $T$ is updated. Thus, it follows from Lemma [3] that our algorithm returns $\text{LSS}(S)$ in $O(r \min \{n, M\} \log \frac{n}{r} + M \log n)$ time. By keeping the lists $L_k$ for a partition point $p$ that gives $2\ell = r = \text{LSS}(S)$, we can also return an LSS (as a string) in $O(r \log \frac{n}{r})$ time, by finding an optimal sequence elements from $L_{\ell}, L_{\ell-1}, \ldots, L_1$.

The space complexity is clearly linear in the total size of the lists $L_1, \ldots, L_\ell$, which is $|T| \in O(M)$.

We remark that our $O(r \min \{n, M\} \log \frac{n}{r} + M \log n)$-time algorithm outperforms Tiskin’s $O(n^2(\log \log n)^2 / \log^2 n)$-time solution [9] when $r = o(n(\log \log n)^2 / \log^3 n)$ and $M = o(n^2 (\log \log n)^2 / \log^3 n)$. The former condition $r = o(n(\log \log n)^2 / \log^3 n)$ implies that our algorithm can be faster than Tiskin’s algorithm (as well as Kosowski’s algorithm [8]) when the length $r$ of the LSS of the input string $S$ is relatively short. For uniformly distributed random strings of length $n$ over an alphabet of size $\sigma$, we have $M \approx n^2 / \sigma$. Thus, for alphabets of size $\sigma = \omega(\log^3 n / (\log \log n)^2)$, the latter condition $M = o(n^2 (\log \log n)^2 / \log^3 n)$ is likely to be the case for the majority of inputs.
References

[1] G. Adelson-Velsky and E. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences (in Russian)*, 146:263–266, 1962. English translation by Myron J. Ricci in Soviet Mathematics - Doklady, 3:1259–1263, 1962.

[2] K. Bringmann and M. Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *FOCS 2015*, pages 79–97, 2015. full version [https://arxiv.org/abs/1502.01063](https://arxiv.org/abs/1502.01063).

[3] A. Chen, T. Chu, and N. Pinsker. Computing the longest increasing subsequence of a sequence subject to dynamic insertion. *CoRR*, abs/1309.7724, 2013.

[4] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *FOCS 1978*, pages 8–21, 1978.

[5] S.-R. Kim and K. Park. A dynamic edit distance table. *J. Disc. Algo.*, 2:302–312, 2004.

[6] A. Kosowski. An efficient algorithm for the longest tandem scattered subsequence problem. In *Proc. SPIRE 2004*, pages 93–100, 2004.

[7] Y. Sakai and S. Inenaga. A reduction of the dynamic time warping distance to the longest increasing subsequence length. *CoRR*, abs/2005.09169, 2020.

[8] C. Schensted. Longest increasing and decreasing subsequences. *Canadian Journal of Mathematics*, 13:179–191, 1961.

[9] A. Tiskin. Semi-local string comparison: algorithmic techniques and applications. *CoRR*, abs/0707.3619, 2013.