On Finding Longest Palindromic Subsequences using Longest Common Subsequences

Gerth Stølting Brodal
Aarhus University, Denmark

Rolf Fagerberg
University of Southern Denmark, Odense, Denmark

Casper Moldrup Rysgaard
Aarhus University, Denmark

Abstract

Two standard textbook problems illustrating dynamic programming are to find the longest common subsequence (LCS) between two strings and to find the longest palindromic subsequence (LPS) of a single string. A popular claim is that the longest palindromic subsequence in a string can be computed as the longest common subsequence between the string and the reversed string. We prove that the correctness of this claim depends on how the longest common subsequence is computed. In particular, we prove that the classical dynamic programming solution by Wagner and Fischer [JACM 1974] for finding an LCS in fact does find an LPS, while a slightly different LCS backtracking strategy makes the algorithm fail to always report a palindrome.

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

Keywords and phrases Palindromic subsequence, longest common subsequence, dynamic programming

Digital Object Identifier 10.4230/LIPIcs.ESA.2024.30

Funding *Gerth Stølting Brodal*: Independent Research Fund Denmark, grant 9131-00113B.
Rolf Fagerberg: Independent Research Fund Denmark, grant 0135-00420B. EU Horizon Europe, MSC grant 101072930.
Casper Moldrup Rysgaard: Independent Research Fund Denmark, grant 9131-00113B.

1 Introduction

This paper addresses the frequently appearing statement that the longest palindromic subsequence of a string can be computed as the longest common subsequence between the string and the reversed string—a statement also given by AI assistants (see Appendix A). We show that the correctness of this statement depends on how the longest common subsequence is computed.

The statement appears in the context of dynamic programming, a standard algorithmic design technique taught in most undergraduate courses on algorithms. In textbooks, the longest common subsequence (LCS) problem on two input strings is the prevailing example of a problem solvable by dynamic programming. The LCS problem asks to find a longest possible string that is a subsequence of two given input strings, or equivalently, to delete the fewest possible symbols from the two strings, such that they become equal. Figure 1 (left) illustrates that the two strings `abcaabcabc` and `bcabbbac` have `bcabbc` as an LCS. The LCS is in general not unique—for instance, `bcabac` is also an LCS for these two strings. Solving the LCS problem on two strings of length m and n, respectively, takes $O(mn)$ time using dynamic programming. In the standard textbook exposition, the lengths of optimal solutions for subproblems are tabulated first, and an actual string solution is found as a postprocessing step backtracking through the tabulated results.

The longest palindromic subsequence (LPS) problem on a single input string is sometimes presented (e.g., in an exercise) as another example of a problem solvable by dynamic programming. The LPS problem asks to find a longest subsequence of a string that is a palindrome, i.e., the subsequence should read the same forwards and backwards. Solving the LPS problem on a string of length n using textbook dynamic programming takes time $O(n^2)$.

The arguments behind the recurrences for the two problems have some analogies, but the resulting recurrences are distinct. For instance, the recurrence for LCS fills out a table of rectangular shape, while the recurrence for LPS uses a triangular tabular area. Still, there is a strong connection: the *length* of LPS solutions on a string s is the same as the length of LCS solutions on s and the reversed string s^R. This statement can be found repeated several places on the web, but usually without a proof. Indeed, a proof is complicated by the fact that the *set* of LPS solutions on a string s is *not* the same as the set of LCS solutions on s and the reversed string s^R: members of the first set are easily seen to be members of the second set, but the reverse inclusion is not always true. One example of this is $s =$ `abccab`, where `accb` is an LCS of s and s^R, but `accb` is not a palindrome. Figure 1 (right) contains another example.

So, a natural question is, how to arrive at an LPS solution via the LCS route. As mentioned, textbook explanations of LCS often include a method for producing an actual solution. The method consists of backtracking over the LCS table, following optimal choices in the LCS recurrence. The recurrence for LCS can actually take on a few variations, but almost all textbooks (including, e.g., [11, pages 397–398]) use the same variant and hence the same backtracking algorithm for producing a solution, namely the one from the classical paper by Wagner and Fischer [28].

The starting point of this paper is the empirical observation that the above backtracking algorithm *always* finds a palindromic solution to LCS on s and s^R, while some other variants of it do not. In this paper, we set out to find out why.

1.1 Previous work on LCS

The LCS problem and its applications have been studied intensely in the literature. Here, we briefly review some of the main results and refer the reader to the surveys by Bergroth, Hakonen, and Raita [4] and Navarro [25] for further references on the LCS problem.

Wagner and Fischer [28] presented the classical $O(mn)$ time dynamic programming algorithm for finding an LCS between two strings s and t of length m and n, respectively. Their solution requires space $O(mn)$. The same time bound is also achieved by Sankoff [27] and the bound is mentioned by Chvatal, Karner and Knuth [10, Problem 35], who ask if one can do better. Navarro [25] gives an overview of several concurrent inventions of the algorithm. Hirschberg [17] improved the space usage to $O(m + n)$. Masek and Paterson [23] improved the running time to $O(n \cdot \max(1, m/\log n))$, when $m \leq n$ and the alphabet Σ for the strings is finite, by applying the "the four Russians" tabulation technique to the algorithm by Wagner and Fischer. Subsequently, Bille and Farach-Colton [5] and Grabowski [14] improved and generalized the result to arbitrary alphabet sizes achieving $O(n + mn(\log\log n)^2/\log^2 n)$ time and $O(mn \log\log n/\log^2 n)$ time, respectively.

Hunt and Szymanski [19] presented an algorithm with running time $O((r + n)\log n)$ for two strings of length n, where $0 \leq r \leq n^2$ is the number of pairs of positions at which the two strings match, i.e., the running time is between $O(n \log n)$ and $O(n^2 \log n)$, depending on r. For $m \leq n$ and k denoting the length of an LCS, Hirschberg [18] presented two algorithms with running time $O(kn + n \log n)$ and $O(k(m + 1 - k)\log n + m \log n)$, respectively, and Nakatsu, Kambayashi and Yajima [24] presented an algorithm with running time $O((m + 1 - k)n)$.

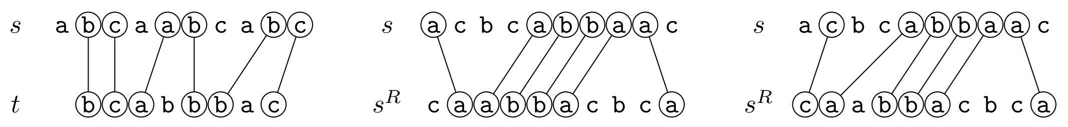


Figure 1 (left) the LCS `bcabbc` between two strings s and t; (center) the LCS `aabbaa` between s and s^R is a palindrome; (right) the LCS `cabbaa` between s and s^R is *not* a palindrome; (center) and (right) are obtained by the backtracking strategies $\langle\nwarrow,\uparrow,\leftarrow\rangle$ and $\langle\uparrow,\leftarrow,\nwarrow\rangle$, respectively (see Section 3 for the definition of a backtracking strategy).

Aho, Hirschberg and Ullman [2] considered decision tree lower bounds for the LCS problem. Abboud, Backurs and Williams [1], and independently Bringmann and Künnemann [7], showed that the LCS problem cannot be solved in time $O\big(n^{2-\varepsilon}\big)$, for any $\varepsilon > 0$, under the *strong exponential time hypothesis* (SETH). This explains the lack of essential improvements over the $O(n^2)$ bound from Wagner and Fischer. Due to the SETH lower bound, recent work on the LCS problem has been in the direction of approximation algorithms, see, e.g., recent work by Bringmann, Cohen-Addad and Das [6].

1.2 Previous work on LPS

Algorithms for finding palindromes in strings is discussed in many textbooks on string algorithms, like Gusfield [15] and Crochemore and Rytter [12], although the focus is on finding longest palindromic substrings instead of subsequences, and variations like approximate palindromes [26] and gapped palindromes [21]. Whereas the LCS problem is covered in detail in textbooks as a canonical example of dynamic programming, the LPS problem appears more rarely, and if so often as an exercise—for instance, Cormen, Leiserson, Rivest and Stein *et al.* [11, Problem 14-2] ask for an efficient algorithm for the LPS problem and Dasgupta, Papadimitriou and Vazirani [13, Problem 6.7] ask explicitly for an $O(n^2)$ algorithm.

Manacher [22] described a linear time algorithm to identify all palindromic prefixes of a string. As observed by, e.g., Apostolico, Breslauer and Galil [3], Manacher's algorithm also identifies for each position in the string the longest palindromic substring centered at this position, i.e., solves the longest palindromic substring problem in linear time. Chowdhury, Hasan, Iqdal and Rahman [8] initiated the study of finding the longest common palindromic subsequence in two strings (without reference to the simpler LPS problem for a single string).

Hasan, Islam, Rahman and Sen [16] described an automaton (denoted a palindromic subsequence automaton) that is a compact representation of all palindromic subsequences of a string of length n. Their construction uses space $O(n^2)$, improving upon the exponential space used by Chuang, Lee and Hunag [9]. As an application, Hasan *et al.* showed how to use their automaton to compute the longest common palindromic subsequence for two strings of length n in worst-case time $O\big(n^4|\Sigma|\big)$, but the algorithm can be faster depending on the size of the automatons for the two strings. Their paper makes crucial use of the following claim [16, page 221] (rephrased to use our notation): "*Intuitively, we can always obtain a longest palindromic subsequence of s by first taking the longest common subsequence (LCS) t of s and s^R and then "reflecting" the first half of the result onto the second half; that is, if t has k symbols, then we replace the last $\lfloor k/2\rfloor$ symbols of t by the reverse of the first $\lfloor k/2\rfloor$ symbols of t.*" In Section 2 we provide a proof of this statement and the implication that the length of an LCS for s and s^R equals the length of an LPS for s.

The LCS and LPS problems are equally hard. This follows from the facts that we can use an LCS algorithm to compute an LPS (Lemma 3) and that an LCS between two strings

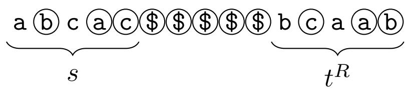


Figure 2 The reduction of LCS(s,t) to LPS$(s\$^n t^R)$. The circled symbols are an LPS of $s\$^n t^R$, and the circled symbols of s are the corresponding LCS of s and t.

s and t of length n can be computed by finding an LPS of $s\$^n t^R$ (see Figure 2). Inenaga and Hyyrö [20, Section 3] gave a generalized version of this reduction for the case of finding the longest common subsequence among four strings to finding a longest common palindromic subsequence in two strings. Note that while a longest palindromic substring can be found in $O(n)$ time [22], finding a longest palindromic subsequence requires worst-case time $\Omega(n^{2-\varepsilon})$, for any $\varepsilon > 0$, under the strong exponential time hypothesis (SETH), by the above reduction and the conditional lower bound for the LCS problem [1, 7].

1.3 Results of this paper

In this paper we perform a detailed study of how to find an LPS of a string s by computing an LCS of s and s^R.

In Section 2, we provide proofs for the following two facts. These facts are folklore, but proofs appear to be scarce in the literature. For completeness, we fill this gap.

1. The length an LCS equals the length of an LPS (Lemma 2).
2. Reflecting an LCS in the middle (as described by Hasan *et al.* [16]) gives an LPS (Lemma 3).

Interestingly, one does not always need to reflect the found LCS to obtain an LPS. In particular, our starting point for this paper was the empirical observation that the Wagner and Fischer algorithm [28] seems to *always* find a palindrome. In Section 3, we prove the following new results:

3. The Wagner and Fischer algorithm [28] always reports a palindrome (Theorem 14).
4. The backtracking in a dynamic programming based solution for LCS, like the Wagner and Fischer algorithm [28], can be ordered in six different ways. Four of these always return a palindrome (Theorem 14), whereas two fail to do so (Lemma 15). See Table 1.

The proof of our main result (Theorem 14) is composed of the following steps: We first prove that the standard textbook algorithm [28] for constructing an LCS path always produces the same solution—although not necessarily via the same path in the LCS table—as the algorithm whose path is the "right-most extreme path". We then prove that for the LCS of s and s^R, the set of edges from all paths in the table must be symmetric around a table diagonal. From this follows that the right-most extreme path itself must be symmetric, which implies that it spells out a palindrome. Hence, the textbook algorithm must always return a palindrome.

1.4 Preliminaries

We let $s = s_1 s_2 \cdots s_n$ denote a string of length $|s| = n$ with symbols s_i from an alphabet Σ. The reversed string is denoted $s^R = s_n s_{n-1} \cdots s_2 s_1$. For $1 \le i \le j \le n$, $s[i..j] = s_i s_{i+1} \cdots s_j$ denotes a substring of s of length $j - i + 1$. For $i > j$, $s[i..j]$ denotes the empty string. A *subsequence* of s is a string $s_{i_1} s_{i_2} \cdots s_{i_k}$, where $0 \le k \le n$ and $1 \le i_1 < i_2 < \cdots < i_k \le n$.

Table 1 The six different strategies to deterministically backtrack when computing LCS(s, s^R). E.g., $\langle \nwarrow, \uparrow, \leftarrow \rangle$ denotes that the prioritized backtracking order of moving from (i, j) is to $(i-1, j-1)$, $(i-1, j)$, $(j-1, j)$ (this corresponds to the Wagner and Fischer algorithm [28]). We prove in this paper that the four strategies on the left always report a palindrome, whereas the two strategies on the right might fail to report a palindrome.

LCS(s, s^R) is always a palindrome				LCS(s, s^R) can be non-palindromic	
$\langle \nwarrow, \uparrow, \leftarrow \rangle$	$\langle \nwarrow, \leftarrow, \uparrow \rangle$	$\langle \uparrow, \nwarrow, \leftarrow \rangle$	$\langle \leftarrow, \nwarrow, \uparrow \rangle$	$\langle \uparrow, \leftarrow, \nwarrow \rangle$	$\langle \leftarrow, \uparrow, \nwarrow \rangle$

For strings s and t, LCS(s, t) denotes the length of a *longest common subsequence* of the two strings, i.e., the maximal length over all strings which are subsequences of both s and t. For two strings s and t of length m and n, respectively, and $0 \le i \le m$ and $0 \le j \le n$, we let LCS$(i, j) = $ LCS$(s[1..i], t[1..j])$, when s and t are obvious from the context. A string s is a *palindrome* if $s = s^R$. We let LPS(s) denote the length of a *longest palindromic subsequence* of s.

The six LCS dynamic programming algorithms that we study are all captured by the following recurrence. Let s and t be two strings of length m and n, respectively, and let $\mathrm{lcs}_{i,j}$ denote an LCS of the first i and j symbols in s and t, respectively. Finding an LCS of s and t is then to find $\mathrm{lcs}_{m,n}$. One way to compute $\mathrm{lcs}_{i,j}$ is by the recurrence

$$\mathrm{lcs}_{i,j} = \begin{cases} \varepsilon & \text{if } i = 0 \lor j = 0 \\ \mathrm{longest}(\mathrm{lcs}_{i-1,j},\ \mathrm{lcs}_{i,j-1}) & \text{if } i > 0 \land j > 0 \land s_i \neq t_j \\ \mathrm{longest}(\mathrm{lcs}_{i-1,j},\ \mathrm{lcs}_{i,j-1},\ \mathrm{lcs}_{i-1,j-1} \cdot s_i) & \text{if } i > 0 \land j > 0 \land s_i = t_j\ , \end{cases} \tag{1}$$

where ε denotes the empty string, $\cdot$ denotes string concatenation, and longest$(\ldots)$ returns a longest string among a list of strings. What LCS is identified as $\mathrm{lcs}_{m,n}$ only depends on what string longest$(\ldots)$ returns in case of multiple strings of equal length. These choices correspond to the choices one has to do when backtracking a dynamic programming table to produce an LCS string. Typical dynamic programming solutions observe that the last case in the recurrence can be simplified to $\mathrm{lcs}_{i-1,j-1} \cdot s_i$, limiting the possible LCSs that can be produced.

2 Relationship between LCS and LPS

In this section, we furnish proofs of two core statements on the relationship between LCS(s, s^R) and LPS(s), namely Lemma 2 and Lemma 3. Potential novelty in this section lies in the exposition, not in the results stated.[1]

▶ **Lemma 1.** *Let t be a common subsequence of s and s^R, where $|t| = k$. Then there exist two (possibly identical) palindromic subsequences of s of combined length $2k$.*

Proof. A symbol of t is a match between a symbol s_i in s and a symbol s_j in s^R. In this proof, we let both indices i and j refer to positions in s (see Figure 3), i.e., the match is between the ith *first* symbol in s and the jth *last* symbol in s^R. Let "<", "=", and ">" designate the sets of matchings in t for which $i < j$, $i = j$, and $i > j$, respectively.

[1] We do note, though, that we have so far been unable to find solid arguments for these statements in the literature. The only close contender found is [29] (which does not consider the length of the underlying string when reversing the first half of the LCS, hence the reversing may fail—the proof can be completed by arguing that the reversing can only fail in one of the directions).

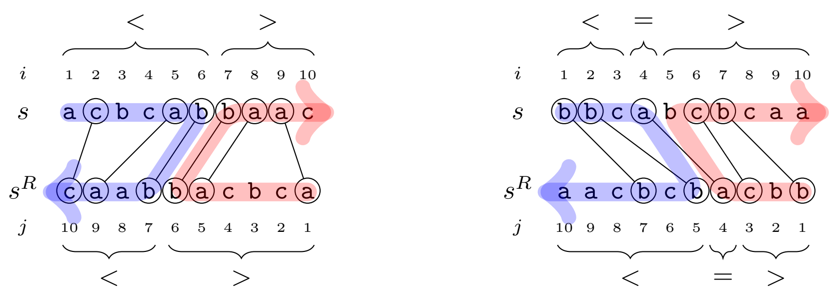


Figure 3 Proof of Lemma 1: A common subsequence between s and s^R (shown as circled symbols) and its partition into two palindromic subsequences (blue and red), depending on whether an index is matched to itself (right, index 4) or not (left).

If the set "=" is empty, consider the subsequence of symbols in s identified by the matchings of the set "<", as pointed out by the blue arrow in Figure 3 (left). More precisely, consider the symbols in circles passed by the blue arrow, taken in the order of that arrow. These symbols constitute a subsequence of s (as their indices in s are increasing) and constitute a palindrome of even length by the matchings of t. The same holds for the subsequence of symbols in s identified by the matchings of the set ">", as pointed out in a similar manner by the red arrow. Combined, these two subsequences cover all symbols of s and s^R which are matched by t, hence the two subsequences have a combined length of $2k$.

If the set "=" is non-empty, it must have size one, as indices in s of the matchings of t increase in the top row and decrease in the bottom row of Figure 3. The matchings of the two sets "<" and ">", each together with one end of the single matching from the set "=", give two palindromic subsequences with the same properties (except for having odd lengths), as pointed out by the blue and red arrows in Figure 3 (right). ◀

From Lemma 1 we can derive that the length of an LPS of a string s equals the length of an LCS of s and s^R.

▶ **Lemma 2.** *For all strings s, we have* $\mathrm{LCS}(s, s^R) = \mathrm{LPS}(s)$.

Proof. Consider a string s and a palindrome p that is a subsequence of s. Then p^R is a subsequence of s^R. Since $p^R = p$ we have that p is a common subsequence of s and s^R. This shows $\mathrm{LCS}(s, s^R) \geq \mathrm{LPS}(s)$.

Conversely, consider a common subsequence t between s and s^R of length k. By Lemma 1, we can find at least one palindromic subsequence of s of length at least k, which shows $\mathrm{LCS}(s, s^R) \leq \mathrm{LPS}(s)$. Combining the two inequalities, we get $\mathrm{LCS}(s, s^R) = \mathrm{LPS}(s)$. ◀

Given an LCS solution, an LPS solution can be produced using Lemma 1 by reflecting the LCS string in the middle.

▶ **Lemma 3.** *Let t be an LCS of s and s^R, where $|t| = k$. Then both $p_1 = t\big[1..\lceil \frac{k}{2} \rceil\big] \cdot t\big[1..\lfloor \frac{k}{2} \rfloor\big]^R$ and $p_2 = t\big[\lfloor \frac{k+2}{2} \rfloor..k\big]^R \cdot t\big[\lceil \frac{k+2}{2} \rceil..k\big]$ are LPS of s.*

Proof. Since t is an LCS of s and s^R, the two palindromic subsequences from Lemma 1 must both have length k, as otherwise maximality of t would be contradicted. In that situation, these two palindromic subsequences, i.e., the two subsequences in blue and red in Figure 3 (left or right, depending on whether k is even or odd), are exactly p_1 and p_2. ◀

As an example, Lemma 3 applied to the LCS `accb` of $s =$ `abccab` and s^R gives the two palindromic LCSs `acca` and `bccb`.

3 Properties of LCS algorithms

In this section we consider different backtracking based LCS algorithms and prove that two of the algorithms, AlgMatch and AlgContour, report identical subsequences, while a third algorithm, AlgSkip, not always reports the same LCS subsequences as the first two.

The standard textbook recurrence for calculating the length of an LCS between two strings s and t of length m and n, respectively, is the following (see, e.g., [11, page 396])

$$\mathrm{LCS}(i,j) = \begin{cases} 0 & \text{if } i = 0 \lor j = 0 \\ \max\{\mathrm{LCS}(i-1,j),\ \mathrm{LCS}(i,j-1)\} & \text{if } i > 0 \land j > 0 \land s_i \neq t_j \\ 1 + \mathrm{LCS}(i-1,j-1) & \text{if } i > 0 \land j > 0 \land s_i = t_j \ , \end{cases} \tag{2}$$

where $0 \leq i \leq m$, $0 \leq j \leq n$, and $\mathrm{LCS}(i,j)$ is the length of an LCS of $s[1..i]$ and $t[1..j]$. By tabulating the formula over all indexes i and j, a table of the LCS for all combinations of prefixes of the input strings is obtained. See Figure 4 for an example of such a table.

LCS strings can be produced by backtracking a path in the table starting in the lower-right corner cell (m,n). At each cell (i,j), one has the option to go up ($\uparrow$) to cell $(i-1,j)$ if $i > 0$ and $\mathrm{LCS}(i-1,j) = \mathrm{LCS}(i,j)$, go left ($\leftarrow$) to cell $(i,j-1)$ if $j > 0$ and $\mathrm{LCS}(i,j-1) = \mathrm{LCS}(i,j)$, and diagonal up-left ($\nwarrow$) to cell $(i-1,j-1)$ if $i > 0$ and $j > 0$ and $s_i = t_j$. When going diagonal up, the symbol s_i is reported. These options correspond exactly to the possible choices in the recurrence of Equation 1 (in Section 1.4).

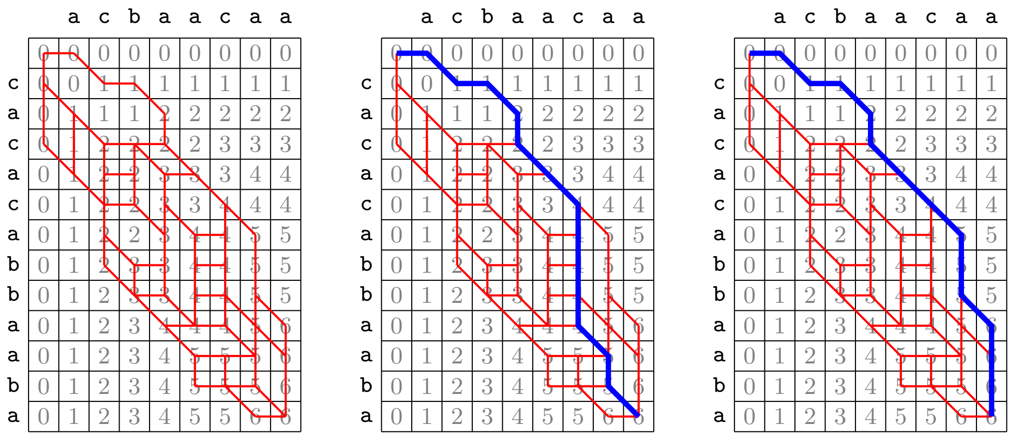
In cells where there is a choice between multiple options, different prioritization among them gives rise to different backtracking algorithms. These will follow different paths, but each will produce an LCS. Indeed, the set of oriented paths from cell (m,n) to cell $(0,0)$ which only contain edges corresponding to the options above exactly captures all LCSs, hence we call such paths *LCS paths.*

▶ **Lemma 4.** *Any LCS path will produce an LCS, and for any LCS there is an LCS path producing it.*

Proof. Any LCS path starts in cell (m,n) with cell value $\mathrm{LCS}(m,n)$ and ends in cell $(0,0)$ with cell value zero. Cell values along the path change only when traversing $\nwarrow$ edges, and this change is a decrease by one in the cell value. Hence, such edges are traversed exactly $\mathrm{LCS}(m,n)$ times, which will produce a common subsequence (in backwards order) of this length.

Conversely, let ℓ be any LCS of s and t. Then ℓ is a sequence of matchings of symbols of s and t. An LCS path producing ℓ can be created by having a $\nwarrow$ edge for all indices (i,j) where s_i and t_j is a matching in ℓ, and then connecting these by edges of the types $\leftarrow$'s and $\uparrow$. We now prove that the latter is possible. Let the kth matching of ℓ be between s_i and t_j, and let the $(k+1)$th matching in ℓ be between $s_{i'}$ and $t_{j'}$, where $i < i'$ and $j < j'$. We have $\mathrm{LCS}(i,j) = k$, since the first k matchings of ℓ constitute a common subsequence of $s[1..i]$ and $t[1..j]$ and must be optimal for those strings (or ℓ would not be optimal). Similarly, $\mathrm{LCS}(i',j') = k+1$. By the third line of Equation 2 we have $\mathrm{LCS}(i'-1,j'-1) = k$. As it follows from Equation 2 that cell values are non-increasing when moving up or left, we can conclude that the rectangular subset of the table having cell (i,j) as the upper left corner and cell $(i'-1,j'-1)$ as the lower right corner all will have cell values equal to k. Thus, this rectangle can be traversed by any combination of edges of the types $\leftarrow$ and $\uparrow$, which was what we wanted to prove. ◀

We denote the union of all LCS paths the *profile* of the LCS table. See Figure 4a for an illustration of a profile (directions of edges omitted for clarity).



(a) Profile of the LCS table in red.

(b) LCS path produced by AlgMatch in blue.

(c) Contour of the profile in blue. This is also the LCS path produced by AlgContour.

Figure 4 LCS table of the strings $s =$ `cacacabbaaba` and $t =$ `acbaacaa`, with the LCS values in gray and the profile overlayed.

▶ **Definition 5** (profile). *The* profile *is the union of all edges of all LCS paths.*

The following lemma shows that the profile contains no further paths than the LCS paths present by construction.

▶ **Lemma 6.** *L is an LCS path $\iff$ L is a path from cell (m, n) to cell $(0, 0)$ in the profile.*

Proof. The $\Rightarrow$ direction follows from the definition of the profile from Definition 5. For the $\Leftarrow$ direction, each edge of the profile exists in some LCS path, and must therefore fulfill the conditions on cell contents and string symbols associated with the edge. Hence, a path from (m, n) to $(0, 0)$, using only edges in the profile, is an LCS path. ◀

To produce an LCS solution, a natural method is to fix a deterministic choice in case of ties in the recurrence (implying nodes in the profile with outdegree more than one). For instance, based on the recursive LCS definition in Equation 2, a very common choice (including in [11]) is the one of AlgMatch[2] in Figure 5 (left). The particular algorithm chooses to match symbols whenever possible, i.e., following the edge $\nwarrow$ whenever it can. If it cannot follow this edge, then it chooses to follow $\uparrow$ and then $\leftarrow$. This can be summarized as the *strategy* of AlgMatch being $\langle \nwarrow, \uparrow, \leftarrow \rangle$. See Figure 4b for an example of the path chosen by AlgMatch.

Any strategy for the order on the edges of the profile to follow produces an LCS by Lemma 6. The strategy $\langle \uparrow, \nwarrow, \leftarrow \rangle$ follows the upper right most part of the profile, as choosing to go up as long as possible and waiting with going left as long as possible leaves no other LCS path above or to the right of it. This strategy is defined as the AlgContour algorithm, as it follows the *contour* of the profile. The contour is defined in Definition 7. See Figure 5 (right) for an implementation of AlgContour and Figure 4c for an illustration of the contour.

[2] Note this is the same algorithm as that of Wagner and Fischer [28]. The renaming in this paper is to clearly show that this is the backtracking strategy which prioritizes to *match* symbols first.

```
AlgMatch($s, t$)
Compute table $L$ by LCS
$i \leftarrow m$, $j \leftarrow n$
lcs $\leftarrow \varepsilon$
while $i > 0$ and $j > 0$
    if $s_i = t_i$
        // Follow $\nwarrow$
        lcs $\leftarrow s_i \cdot$ lcs
        $i \leftarrow i - 1$, $j \leftarrow j - 1$
    elseif $L(i, j) = L(i - 1, j)$
        // Follow $\uparrow$
        $i \leftarrow i - 1$
    else
        // Follow $\leftarrow$
        $j \leftarrow j - 1$
return lcs
```

```
AlgContour($s, t$)
Compute table $L$ by LCS
$i \leftarrow m$, $j \leftarrow n$
lcs $\leftarrow \varepsilon$
while $i > 0$ and $j > 0$
    if $L(i, j) = L(i - 1, j)$
        // Follow $\uparrow$
        $i \leftarrow i - 1$
    elseif $s_i = t_i$
        // Follow $\nwarrow$
        lcs $\leftarrow s_i \cdot$ lcs
        $i \leftarrow i - 1$, $j \leftarrow j - 1$
    else
        // Follow $\leftarrow$
        $j \leftarrow j - 1$
return lcs
```

Figure 5 Pseudocode implementations of (left) AlgMatch, which is the backtracking of the LCS table using strategy $\langle \nwarrow, \uparrow, \leftarrow \rangle$ and (right) AlgContour using strategy $\langle \uparrow, \nwarrow, \leftarrow \rangle$.

▶ **Definition 7** (Contour). *The* contour *of a profile is the upper right most path of the profile.*

▶ **Corollary 8.** *The contour of a profile is an LCS path.*

Proof. The contour is a path in the profile, which by Lemma 6 must be an LCS path. ◀

We are now ready to prove that the two algorithms AlgMatch and AlgContour compute the same output LCS strings.

▶ **Lemma 9.** *For all pairs of input strings s and t, algorithms* AlgMatch *and* AlgContour *produce the same LCS string.*

Proof. Recall that the strategies of AlgMatch and AlgContour are $\langle \nwarrow, \uparrow, \leftarrow \rangle$ and $\langle \uparrow, \nwarrow, \leftarrow \rangle$, respectively, cf. the pseudocode in Figure 5. Any LCS path must cross exactly n columns. This happens via the $\nwarrow$ and $\leftarrow$ steps of the path. A path outputs symbols of the LCS exactly when moving by a $\nwarrow$ step, hence paths can only output symbols when changing column. Define M_j to be the row where the LCS path of AlgMatch first enters column j and C_j to be the row where the LCS path of AlgContour first enters column j. Clearly, $\mathrm{M}_j \geq \mathrm{C}_j$ for all columns j, as AlgContour is the upper right most LCS path of all LCS paths. Also note that in any LCS table, it follows from Equation 2 that cell values are non-increasing when moving up or left.

The key element of the proof is the following *claim*: If the table values $\mathrm{LCS}(\mathrm{M}_j, j)$ and $\mathrm{LCS}(\mathrm{C}_j, j)$ are equal, then

1. The LCS path of AlgMatch must leave column j by the same type of step ($\nwarrow$ or $\leftarrow$) as the LCS path of AlgContour leaves column j by.
2. The table values $\mathrm{LCS}(\mathrm{M}_{j-1}, j-1)$ and $\mathrm{LCS}(\mathrm{C}_{j-1}, j-1)$ are also equal.

Since both LCS paths start in the same table cell (m, n), the condition of the claim trivially holds in the beginning. Hence, by induction on (decreasing values of) j, the two

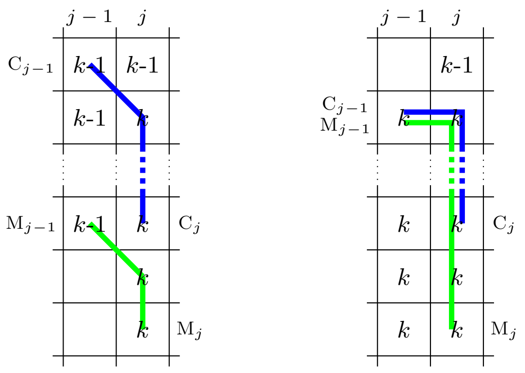


Figure 6 Behavior of AlgContour (in blue, starting in row C_j) and AlgMatch (in green, starting in row M_j), when (left) AlgContour leave column j by $\nwarrow$, and (right) AlgContour leave by $\leftarrow$.

paths leave exactly the same columns by $\nwarrow$ steps, hence they produce the same LCS string, assuming the claim is true.

To prove the claim, we make a case analysis on the step type by which AlgContour leaves column j. Let k denote the shared table value $k = \text{LCS}(M_j, j) = \text{LCS}(C_j, j)$.

Case 1: AlgContour leaves column j by a $\nwarrow$ step. The situation is depicted in Figure 6 (left). Before this step, AlgContour takes ℓ steps of type $\uparrow$, for $\ell \geq 0$. All ℓ table cells passed must also have the value k, by the condition for making $\uparrow$ steps. Hence, the cell $(C_{j-1}, j-1)$ arrived at in column $j-1$ by the $\nwarrow$ step must have value $k-1$, by the condition for making $\nwarrow$ steps. By the precondition $k = \text{LCS}(M_j, j) = \text{LCS}(C_j, j)$ of the claim and the vertical monotonicity of cell values, the cells in column j between cell (M_j, j) and the cell $(C_j - \ell, j)$ where the LCS path of AlgContour made the $\nwarrow$ step must all have the value k. By the strategy of AlgMatch, this algorithm will never take a $\leftarrow$ step if $\uparrow$ steps are possible, which is the case in column j by the vertical repetition of the cell value k. Hence, the LCS path of AlgMatch can only leave column j by a $\nwarrow$ step. This must happen either before it reaches the cell $(C_j - \ell, j)$ where the LCS path of AlgContour made the $\nwarrow$ step, or at the latest when it reaches that cell, since the LCS path of AlgContour is the upper right most path. By the conditions for making a $\nwarrow$ step, the cell arrived at in column $j-1$ by the LCS path of AlgMatch must have value $k-1$. This establishes the claim in this case.

Case 2: AlgContour leaves column j by a $\leftarrow$ step. The situation is depicted in Figure 6 (right). Again, AlgContour may first take ℓ steps of type $\uparrow$, for $\ell \geq 0$, and all cells passed during these must also have the value k. By the condition for making $\leftarrow$ steps, the cell $(C_j - \ell, j)$ arrived at in column $j-1$ by the $\leftarrow$ step must have value k. By monotonicity, the cells in column $j-1$ from cell $(C_{j-1}, j-1)$ and down to cell $(M_j, j-1)$ must all have value k. As the LCS path of AlgMatch must leave column j before it reaches the cell $(C_j - \ell, j)$ where the LCS path of AlgContour made the $\leftarrow$ step, or at the latest when it reaches that cell, it cannot leave by a $\nwarrow$ step, as this would need to end up in a cell in column $j-1$ of value $k-1$, by the conditions for such steps. Hence, it must leave column j by a $\leftarrow$ step, and must end up in a cell of value k. As the strategy of AlgMatch priorities $\uparrow$ over $\leftarrow$, it actually must leave column j from the same cell as AlgContour. This establishes the claim in the second case. ◀

Another strategy is $\langle\uparrow, \leftarrow, \nwarrow\rangle$, i.e., to choose to skip symbols over matching them. We denote the algorithm using this strategy as AlgSkip, see Figure 7 for the pseudocode for this backtracking algorithm. Using this strategy may seen at a first glance like a non-intuitive way of implementing a traversal of the LCS table. However, in favor of this algorithm is the observation that it only performs comparisons of values in the LCS table, while comparisons of string symbols are not needed. Thus, if only the LCS indexes in s and t are to be reported, access to the strings is never needed.

```
AlgSkip($s, t$)
Compute table $L$ by LCS
$i \leftarrow m$, $j \leftarrow n$
lcs $\leftarrow \varepsilon$
while $i > 0$ and $j > 0$
    if $L(i, j) = L(i-1, j)$
        // Follow $\uparrow$
        $i \leftarrow i - 1$
    elseif $L(i, j) = L(i, j-1)$
        // Follow $\leftarrow$
        $j \leftarrow j - 1$
    else
        // Follow $\nwarrow$
        lcs $\leftarrow s_i \cdot$ lcs
        $i \leftarrow i - 1$, $j \leftarrow j - 1$
return lcs
```

Figure 7 Pseudocode implementation AlgSkip, which is the backtracking of the LCS table using strategy $\langle\uparrow, \leftarrow, \nwarrow\rangle$.

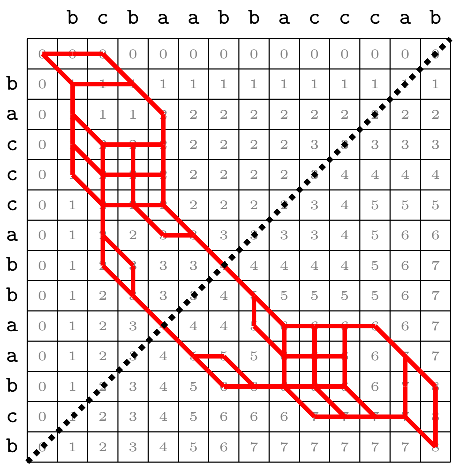
The AlgSkip algorithm is not guaranteed to output the same LCS string as AlgContour. Trying to prove Lemma 9 breaks for AlgSkip in the case where AlgSkip and AlgContour start on the same cell, and AlgSkip is allowed to leave the column by $\leftarrow$, with AlgContour leaving by $\nwarrow$. Concretely, on input strings $s =$ `cacacabbaaba` and $t =$ `acbaacaa` from Figure 4, AlgSkip produces the LCS string `acacaa`, while AlgContour produces the string `caacaa`.

This covers the three basic strategies. Three other strategies can be obtained by switching the order of $\uparrow$ and $\leftarrow$ in the three presented algorithms. Switching $\uparrow$ and $\leftarrow$ is equal to switching the input strings s and t in the LCS table. Therefore, the switched AlgContour algorithm, strategy $\langle\leftarrow, \nwarrow, \uparrow\rangle$, instead follows the lower left contour of the profile. The switched AlgMatch algorithm, strategy $\langle\nwarrow, \leftarrow, \uparrow\rangle$, outputs the same indexes in s as the switched AlgContour algorithm, leading it to agree with a contour of the profile. The switched AlgSkip algorithm, strategy $\langle\leftarrow, \uparrow, \nwarrow\rangle$, similarly has no guarantee to output the same string as the switched AlgContour.

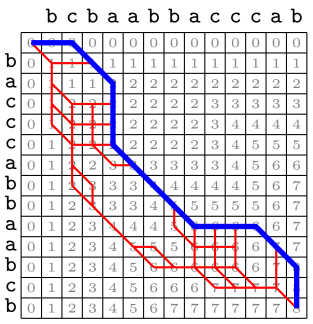
4 Computing an LPS using an LCS algorithm

In Section 3 it was shown that AlgMatch outputs the same string as AlgContour. In this section we show that AlgContour(s, s^R) outputs an LPS of a string s, which implies that AlgMatch(s, s^R) outputs an LPS of s.

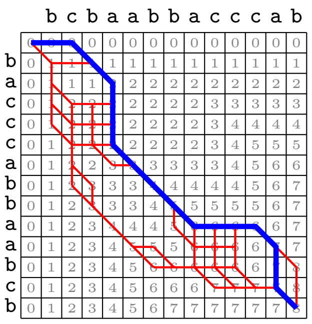
To argue that AlgContour outputs a palindrome, we observe that the profile is *symmetric* when the LCS is over some string s and the reversed string s^R. In Figure 8a, the



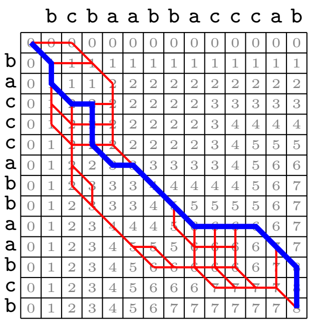
(a) Symmetry of the profile over the top-right to the bottom-left diagonal (dotted).



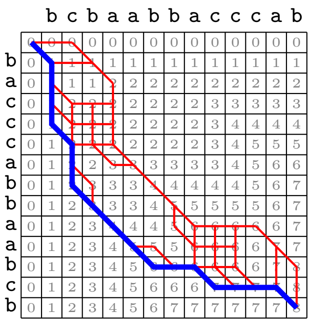
(b) Path using ⟨↑, ↖, ←⟩, output string: `baabbaab`.



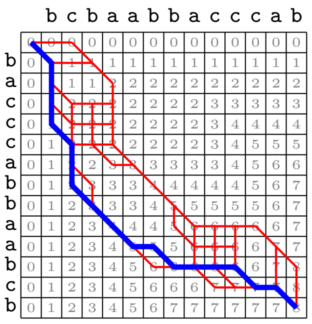
(c) Path using ⟨↖, ↑, ←⟩, output string: `baabbaab`.



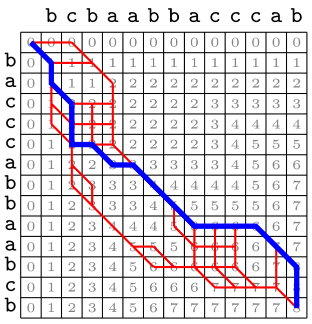
(d) Path using ⟨↑, ←, ↖⟩, output string: `bcabbaab`.



(e) Path using ⟨←, ↖, ↑⟩, output string: `bcbaabcb`.



(f) Path using ⟨↖, ←, ↑⟩, output string: `bcbaabcb`.



(g) Path using ⟨←, ↑, ↖⟩, output string: `bcabbaab`.

Figure 8 LCS table of the string `bacccabbaabcb` and the reversed string, including the profile (in red) with (top) highlighted mirror line and (bottom) different LCS paths (in blue) dependent on strategy.

profile is its own mirror over the dotted diagonal from the top-right to the bottom-left corner, when not considering the underlying orientation of the edges. In this mirroring, a cell (i, j) is *mirrored* to cell $(n-j, n-i)$. The individual edges are only of type $\nwarrow$, $\uparrow$, and $\leftarrow$, and under the mirroring new types of edges should not be created. An edge $(i, j) \rightarrow (i', j')$ is therefore mirrored to $(n-j', n-i') \rightarrow (n-j, n-i)$, i.e., the endpoints of the edge are mirrored and then the orientation is reversed, to preserve the set of types of edges. This results in $\nwarrow$ mirrored to $\nwarrow$, $\uparrow$ mirrored to $\leftarrow$, and $\leftarrow$ mirrored to $\uparrow$. Note that a $\nwarrow$ originating from cell (i, j) in the mirror becomes a $\nwarrow$ originating from cell $(n+1-j, n+1-i)$. In the following, define for an LCS path L the *mirror* L^R, as the edges of L mirrored and their order reversed.

▶ **Definition 10** (Symmetric). *A profile is defined to be* symmetric *when the set of edges in the profile is equal to the set of mirrored edges. An LCS path L is defined to be* symmetric *when $L = L^R$, i.e., it is equal to its own mirror.*

▶ **Lemma 11.** *The profile of the LCS table of s and s^R is symmetric.*

Proof. Any edge in the profile must be included in some LCS path L. By showing that L^R is in the profile, it shows that the mirrored edge is in the profile.

If a cell (i, j) contains a $\nwarrow$, then it must hold that $s_i = s^R_j$. By definition of reversing a string it must hold that $s_i = s^R_{n+1-i}$, i.e., that the i'th symbol of s is equal to the i'th last symbol of s^R. Therefore it also holds that $s_{n+1-j} = s^R_{n+1-i}$, which results in cell $(n+1-j, n+1-i)$ also contains a $\nwarrow$.

All $\nwarrow$ in L become $\nwarrow$ in L^R, and by the above argument, all $\nwarrow$ of L^R exist in the table. All other steps of L^R are $\uparrow$ or $\leftarrow$, which must exist as subpaths between pairs of $\nwarrow$. As L^R contains the same number of $\nwarrow$ as L, then using a similar argument as in Lemma 4, these subpaths exist, and therefore L^R is an LCS path, concluding that L^R is in the profile. ◀

▶ **Lemma 12.** *Any symmetric LCS path L outputs a palindromic string.*

Proof. The proof goes by induction on the number of $\nwarrow$ in L. If there is at most one $\nwarrow$ in L, then the output string contains at most one symbol, and is therefore a palindrome.

Assume that L contains two or more $\nwarrow$. Traverse L until the first $\nwarrow$ is reached in cell (i, j). By symmetry of L, then the last $\nwarrow$ of L is in cell $(n+1-j, n+1-i)$. The first symbol L outputs is s_i and the last symbol is s^R_{n+1-i}, which, by definition of s^R, are equal. By induction on the remaining of L, the output string is a palindrome. ◀

▶ **Lemma 13.** *The contour of the profile of the LCS table of s and s^R is symmetric.*

Proof. Assume for contradiction that the contour is not symmetric. Define the contour as C. By Lemma 11 it holds that the mirror C^R of the contour C is contained in the profile. As C is not symmetric, then there must be some part of C and C^R which is non overlapping. By the mirroring, this part exists with C^R above and/or to the right of C. Therefore C is not the top right most part of the profile, leading to the contradiction that C is not the contour. Therefore, the contour is symmetric. ◀

▶ **Theorem 14.** *The string produced by* AlgMatch *of s and s^R is a palindrome.*

Proof. By Lemma 9, the string produced by AlgMatch is equal to the string produced by AlgContour on any pairs of input strings, and therefore also on input s and s^R. By Lemma 13 and Lemma 12 the contour outputs a palindromic string. Therefore, the string produced by AlgMatch is a palindrome. ◀

This implies that the first four strategies from Table 1 always outputs a palindromic string, as the first two strategies, $\langle \nwarrow, \uparrow, \leftarrow \rangle$ and $\langle \nwarrow, \leftarrow, \uparrow \rangle$, corresponds to the AlgMatch and the switched AlgMatch algorithms, with the next two strategies, $\langle \uparrow, \nwarrow, \leftarrow \rangle$ and $\langle \leftarrow, \nwarrow, \uparrow \rangle$, being the AlgContour and the switched AlgContour algorithms. To show that the last two strategies can output non-palindromic strings, Lemma 15 below shows this by example, using the string from Figure 8.

▶ **Lemma 15.** *The string produced by* AlgSkip *can be non-palindromic.*

Proof. AlgSkip follows the strategy $\langle \uparrow, \leftarrow, \nwarrow \rangle$. On input string $s =$ `bacccabbaabcb` and the reversed string s^R the output LCS string is `bcabbaab`, which is non-palindromic. See Figure 8d. ◀

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 59–78. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.14.

2 Alfred V. Aho, Daniel S. Hirschberg, and Jeffrey D. Ullman. Bounds on the complexity of the longest common subsequence problem. *Journal of the ACM*, 23(1):1–12, 1976. doi:10.1145/321921.321922.

3 Alberto Apostolico, Dany Breslauer, and Zvi Galil. Parallel detection of all palindromes in a string. *Theoretical Computer Science*, 141(1&2):163–173, 1995. doi:10.1016/0304-3975(94)00083-U.

4 Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In Pablo de la Fuente, editor, *Seventh International Symposium on String Processing and Information Retrieval, SPIRE 2000, A Coruña, Spain, September 27-29, 2000*, pages 39–48. IEEE Computer Society, 2000. doi:10.1109/SPIRE.2000.878178.

5 Philip Bille and Martin Farach-Colton. Fast and compact regular expression matching. *Theoretical Computer Science*, 409(3):486–496, 2008. doi:10.1016/J.TCS.2008.08.042.

6 Karl Bringmann, Vincent Cohen-Addad, and Debarati Das. A linear-time $n^{0.4}$-approximation for longest common subsequence. *ACM Transactions on Algorithms*, 19(1):9:1–9:24, 2023. doi:10.1145/3568398.

7 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 79–97. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.15.

8 Shihabur Rahman Chowdhury, Md. Mahbubul Hasan, Sumaiya Iqbal, and M. Sohel Rahman. Computing a longest common palindromic subsequence. In S. Arumugam and W. F. Smyth, editors, *Combinatorial Algorithms, 23rd International Workshop, IWOCA 2012, Tamil Nadu, India, July 19-21, 2012, Revised Selected Papers*, volume 7643 of *Lecture Notes in Computer Science*, pages 219–223. Springer, 2012. doi:10.1007/978-3-642-35926-2_24.

9 K. Chuang, R. Lee, and C. Huang. Finding all palindrome subsequences in a string. In *The 24th Workshop on Combinatorial Mathematics and Computation Theory*, 2007.

10 V. Chvatal, D. A. Karner, and D.E. Knuth. Selected combinatorial research problems. Technical Report STAN-CS-72-292, Computer Science Department, Stanford University, June 1972. URL: http://i.stanford.edu/pub/cstr/reports/cs/tr/72/292/CS-TR-72-292.pdf.

11 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Fourth Edition.* MIT Press, 2022. URL: https://mitpress.mit.edu/9780262046305.

12 Maxime Crochemore and Wojciech Rytter. *Jewels of stringology.* World Scientific, 2002. doi:10.1142/4838.

13 Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2006. URL: https://www.mheducation.com/highered/product/M9780073523408.html.

14 Szymon Grabowski. New tabulation and sparse dynamic programming based techniques for sequence similarity problems. *Discrete Applied Mathematics*, 212:96–103, 2016. doi:10.1016/J.DAM.2015.10.040.

15 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

16 Md. Mahbubul Hasan, A. S. M. Sohidull Islam, M. Sohel Rahman, and Ayon Sen. Palindromic subsequence automata and longest common palindromic subsequence. *Mathematics in Computer Science*, 11(2):219–232, 2017. doi:10.1007/S11786-016-0288-7.

17 Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. doi:10.1145/360825.360861.

18 Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24(4):664–675, 1977. doi:10.1145/322033.322044.

19 James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest subsequences. *Communications of the ACM*, 20(5):350–353, 1977. doi:10.1145/359581.359603.

20 Shunsuke Inenaga and Heikki Hyyrö. A hardness result and new algorithm for the longest common palindromic subsequence problem. *Information Processing Letters*, 129:11–15, 2018. doi:10.1016/J.IPL.2017.08.006.

21 Roman Kolpakov and Gregory Kucherov. Searching for gapped palindromes. *Theoretical Computer Science*, 410(51):5365–5373, 2009. doi:10.1016/J.TCS.2009.09.013.

22 Glenn K. Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, 1975. doi:10.1145/321892.321896.

23 William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980. doi:10.1016/0022-0000(80)90002-1.

24 Narao Nakatsu, Yahiko Kambayashi, and Shuzo Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18:171–179, 1982. doi:10.1007/BF00264437.

25 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001. doi:10.1145/375360.375365.

26 Alexandre H. L. Porto and Valmir Carneiro Barbosa. Finding approximate palindromes in strings. *Pattern Recognition*, 35(11):2581–2591, 2002. doi:10.1016/S0031-3203(01)00179-0.

27 David Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences of the United States of America*, 69(1):4–6, 1972. URL: http://www.jstor.org/stable/60967.

28 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974. doi:10.1145/321796.321811.

29 Woburn Collegiate Institute Programming Enrichment Group (PEG) Wiki. Longest palindromic subsequence. http://wcipeg.com/wiki/Longest_palindromic_subsequence?#LCS-based_approach. Accessed: 2024.04.20.

A AI assistants

A frequent statement on the internet[3] is that an LPS of a string can be computed as an LCS of the string and the reversed string[4,5], which is reflected in the suggestions from AI assistants like GitHub Copilot[6], Google Gemini[7], and Open AI ChatGPT[8]. The Python 3.12 code in Figure 9 was given to the free versions of GitHub Copilot, Google Gemini, and OpenAI ChatGPT 3.5, without the implementation for `palindrome` but including its docstring. All three AI assistants suggested `return lcs(x, x[::-1])` as the completion of the code, i.e., to return the LCS of the string and the reversed string. This suggestion would be correct if `lcs(x, y)` was implemented using Wagner and Fischer [28], but in the code provided to the AI assistants, the `lcs` function uses a dynamic programming implementation with strategy $\langle\uparrow,\leftarrow,\nwarrow\rangle$ as the backtracking prioritization, and the reported subsequence is not a palindrome.

Identical suggestions were obtained on lmsys.org[9] for ChatGPT 4 (gpt-4o-2024-05-13), Meta Llama 3 (llama-3-70b-instruct), Claude 3.5 Sonnet (version claude-3-5-sonnet-20240620), Databricks DBRX (dbrx-next), Microsoft Phi-3 mini (phi-3-mini-4k-instruct), NVIDIA Nenotron-4 340B (nemotron-4-340b), Alibaba Cloud Qwen2 (qwen2-72b-instruct) and others (deepseek-coder-v2, glm-4-0520, reka-flash-preview-20240611, yi-large-preview).

```python
from functools import cache

@cache
def lcs(x, y):
    '''Longest common subsequence of strings x and y.'''

    if not x or not y:
        return ''
    if x[-1] == y[-1]:
        return max(lcs(x[:-1], y), lcs(x, y[:-1]),
                   lcs(x[:-1], y[:-1]) + x[-1], key=len)
    return max(lcs(x[:-1], y), lcs(x, y[:-1]), key=len)

def palindrome(x):
    '''Longest palindromic subsequence of string x.'''

    # Suggestion by GitHub Copilot, Google Gemini, OpenAI ChatGPT
    return lcs(x, x[::-1])

print(f"{palindrome('acbcabbaac') = }")
```

Output

```
palindrome('acbcabbaac') = 'cabbaa'
```

Figure 9 Implementation of `palindrome` according to GitHub Copilot, Google Gemini, and OpenAI ChatGPT 3.5.

[3] All links in this section were accessed June 25, 2024.
[4] https://www.geeksforgeeks.org/print-longest-palindromic-subsequence/
[5] https://www.educative.io/answers/longest-palindromic-subsequence-algorithm
[6] https://github.com/features/copilot/
[7] https://gemini.google.com/
[8] https://chat.openai.com/
[9] https://chat.lmsys.org/