Faster Algorithms for Shortest Unique or Absent Substrings

Panagiotis Charalampopoulos
King's College London, UK

Manal Mohamed
King's College London, UK

Solon P. Pissis
The Cyprus Institute, Nicosia, Cyprus
CWI, Amsterdam, The Netherlands
Vrije Universiteit Amsterdam, The Netherlands

Hilde Verbeek
CWI, Amsterdam, The Netherlands

Wiktor Zuba
University of Warsaw, Poland

Abstract

We revisit two well-known algorithmic problems on strings: computing a *shortest unique substring* (SUS) and a *shortest absent substring* (SAS) in a string S of length n. Both problems admit folklore $\mathcal{O}(n)$-time solutions using the suffix tree of S. However, for small alphabets, this complexity is not necessarily optimal in the word RAM model, where a string of length n over alphabet $[0, \sigma)$ can be stored in $\mathcal{O}(n \log \sigma / \log n)$ space and read in $\mathcal{O}(n \log \sigma / \log n)$ time.

We present an $\mathcal{O}(n \log \sigma / \sqrt{\log n})$-time algorithm for computing a SUS in S. This algorithm decomposes the problem according to the length and the period of the sought substring and uses several tools and techniques, such as synchronizing sets, the analysis of runs, and wavelet trees, to reduce the computation of a SUS to a simple geometric problem. Further, we adapt this algorithm and combine it with an efficient construction of de Bruijn sequences in order to obtain an $\mathcal{O}(n \log \sigma / \sqrt{\log n})$-time algorithm for computing a SAS in S.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string algorithms, unique substrings, absent substrings, absent words

Digital Object Identifier 10.4230/LIPIcs.SWAT.2026.13

Related Version *Full Version*: `https://arxiv.org/abs/2605.04826` [10]

Funding *Hilde Verbeek*: Supported by a Constance van Eeden Fellowship.

1 Introduction

Given a string S over an alphabet Σ, a string P is called *unique* in S if it occurs exactly once in S as a substring. A unique substring P of S is called a *shortest unique substring* (SUS) of S if there is no shorter string that is unique in S. Similarly, a string P over Σ is said to be *absent* from S if it does not occur in S as a substring. A string P that is absent from S is called a *shortest absent substring* (SAS) of S if no shorter string that is absent from S exists. For example, for string $S = \mathtt{gcattgcgtaggt}$ over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$, we have that $\mathtt{ag}$ is a SUS of S and $\mathtt{aa}$ is a SAS of S.

Shortest unique substrings (SUSs) and shortest absent substrings (SASs) have a wide range of applications in bioinformatics, information retrieval, and data compression. In bioinformatics, SUSs are employed in alignment-free sequence comparison methods [22],

whereas in information retrieval they are used to extract minimal text snippets from a document collection containing a query term [31]. Note that the notion of SUS used in [31] is position-dependent: for a given position in the input string, one seeks a SUS covering that position. In contrast, in this work we consider the global variant of the problem, where the goal is to compute a shortest substring that is unique in the entire string. SASs are likewise of significant importance in bioinformatics and data compression. They provide highly specific genomic signatures for pathogens such as SARS-CoV-2 [32], thereby facilitating the development of rapid diagnostic assays and targeted therapeutics [34]. Moreover, SASs are utilized in alignment-free sequence comparison methods [8] and constitute the foundational concept of compression schemes based on antidictionaries [13].

It is a classical exercise to compute a SUS or a SAS of a string S of length n in $\mathcal{O}(n)$ time using the suffix tree of S [37]. In particular, all SUSs and SASs are naturally encoded in the suffix tree of S; see Figure 1 for an illustration.

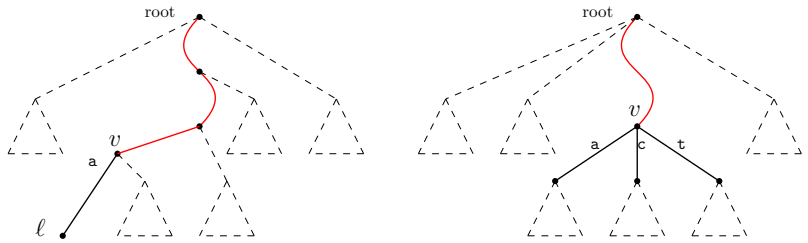


Figure 1 A schematic illustration of two suffix trees. Left, the root-to-v path-label, appended with letter `a`, is a *unique substring* that occurs as a prefix of the suffix represented by the leaf ℓ. Right, the root-to-v path-label, appended with letter $\mathtt{g} \notin \{\mathtt{a}, \mathtt{c}, \mathtt{t}\}$, yields an *absent substring*.

Here, we consider the unit-cost word RAM model of computation with word size $w = \Theta(\log n)$ for inputs of size n, and a standard instruction set including arithmetic operations, bitwise Boolean operations, and shifts. We measure the space complexity of our algorithms in terms of the number of machine words used. A *packed representation* of a string S over an integer alphabet $\Sigma = [0, \sigma)$ stores $\Theta(\log_\sigma n)$ letters per machine word (possibly apart from the last one), thus representing S in $\mathcal{O}(1 + |S|/\log_\sigma n)$ machine words. A string given in this representation is referred to as a *packed string*.

A large body of work exploits bit-level parallelism in the word RAM model to accelerate classical string processing tasks when the input consists of packed strings. In particular, the problems for which speed-ups have been obtained include pattern matching [1, 3, 4, 5, 7, 20, 21, 30], text indexing [6, 18, 36], computing palindromes [12], constructing longest common extension data structures [25], computing a longest common substring [9], constructing the BWT [25], computing Lempel-Ziv (LZ77) factorizations [16, 27], counting squares and locating runs [11], computing Lyndon arrays [2], computing covers [33], and constructing compressed suffix trees and compressed suffix arrays [26].

In this paper, we ask the following question for a string of length n:

Can a SUS (or a SAS) be computed in $o(n)$ time in the packed setting?

Our results. We answer this question in the affirmative, improving upon the folklore linear-time solutions for computing a single SUS or SAS in the packed setting.

▶ **Theorem 1** (SUS). *Given a packed string S of length n over an integer alphabet $[0, \sigma)$, a shortest unique substring of S can be computed in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time.*

▶ **Theorem 2** (SAS). *Given a packed string S of length n over an integer alphabet $[0, \sigma)$, a shortest absent substring of S can be computed in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time.*

Our techniques and paper organization. We obtain Theorem 1 by decomposing the problem based on the *length* and the *period* of the sought substring of S. For short substrings, we employ tabulation. For medium-length aperiodic substrings, we modify an existing technique for longest common substrings [9] that is based on an efficient construction of wavelet trees [29]. For long aperiodic substrings, we combine sampling via string synchronizing sets [17, 25] with a heavy-light decomposition [35] of two compacted tries T and T^R. Then, using pairs of heavy paths, we reduce the search to a 2D geometric problem that, as we show, underlies the computation of SUSs. For periodic substrings, we exploit the runs in S [23]: we group them by their Lyndon roots and sparse-Lyndon roots [11] and use this grouping to reduce the search to another instance of said geometric problem. See Section 3.

We obtain Theorem 2 using the same framework, augmented with a new efficient construction of de Bruijn sequences [14] in packed representation. See Section 4.

We start in Section 2 with the necessary preliminaries. Any materials omitted due to space constraints are provided in the full version [10].

Other related work. Kempa and Kociumaka [28] studied the hardness hierarchy for problems whose fastest known word-RAM algorithms run in $\mathcal{O}(n\sqrt{\log n})$ time on inputs of $\Theta(n)$ machine words. This class includes string processing problems such as constructing the BWT [25], computing a longest common substring [9], and computing LZ77 factorizations [27]. For these problems, they showed that the known $\mathcal{O}(n/\sqrt{\log n})$-time algorithms for *binary* strings – equivalently, $\mathcal{O}(n\sqrt{\log n})$-time algorithms when the input size is measured in machine words – are conditionally optimal, with conditional lower bounds established via non-trivial reductions to a variant of dictionary matching. Computing a SUS (Theorem 1) or a SAS (Theorem 2) for binary strings shares the same time complexity. It remains an interesting open problem whether one can improve upon our algorithms or prove that they are conditionally optimal. Let us note that proving a conditional lower bound for our problem using the framework of Kempa and Kociumaka [28] seems challenging. The reductions in said work highlight as the underlying hardness in the studied problems the task of looking for an occurrence of some (sub)string P in a string S (e.g., as in the longest common substring problem). The problems we study here are of a different flavor: in SUS we look for substrings that do not occur elsewhere in S while in SAS we look for strings that do not occur at all in S.

2 Preliminaries

Strings. An *alphabet* Σ is a finite set of elements called *letters*. A *string* $S = S[0]S[1]\cdots S[n-1]$ of *length* $|S| = n$ is a sequence of n letters from Σ. We refer to each $i \in [0, n)$ as a *position* of S. We consider throughout an *integer* alphabet $\Sigma = [0, \sigma)$ with $\sigma = n^{\mathcal{O}(1)}$. A string P is a *substring* of a string S if we have $P = S[i]\cdots S[i + |P| - 1]$ for some position i of S. In this case, we say that P *occurs* at position i of S. Such an occurrence is called a *fragment* of S; we denote it by either $S[i\,..\,i + |P|)$ or $S[i\,..\,i + |P| - 1]$. The set of the starting positions of the *occurrences* of a string P in S is denoted by $\mathrm{Occ}(P, S)$. A *prefix* of S is a fragment of the form $S[0\,..\,j)$, and a *suffix* of S is a fragment of the form $S[i\,..\,n)$. A substring P of S is called *proper* if $P \neq S$. The *reverse* of string S, which we denote by S^R, is defined as $S^R = S[n-1]S[n-2]\cdots S[0]$. The *concatenation* of two strings S and S' is denoted by $S \cdot S'$ and the concatenation of k copies of a string S is denoted by S^k.

▶ **Definition 3** (Period). *An integer $p > 0$ is a* period *of a string P if $P[i] = P[i + p]$, for all $i \in [0, |P| - p)$. The smallest period of P is called* the period *of P and is denoted by $\mathsf{per}(P)$. A string P is called* periodic *if $\mathsf{per}(P) \leq |P|/2$ and* aperiodic *otherwise.*

▶ **Example 4.** For string $P = \texttt{abaaabaaabaaaba}$, we have $\mathsf{per}(P) = 4$. Note that 8 is also a period of P. Since $\mathsf{per}(P) = 4 \leq |P|/2 = 15/2$, P is periodic.

▶ **Definition 5** (Run). *A fragment $F = S[i\,..\,j]$ of a string S is called a* run *of S if $\mathsf{per}(F) \leq |F|/2$, and extending F either to the left or to the right (if possible) would result in an increase of its period, that is, $S[i-1] \neq S[i-1+\mathsf{per}(F)]$ (or $i = 0$) and $S[j+1] \neq S[j+1-\mathsf{per}(F)]$ (or $j = |S|-1$).*

▶ **Definition 6** (Lyndon Root). *The* Lyndon root *of a run F, denoted by $\mathsf{Lroot}(F)$, is defined as the lexicographically smallest rotation of the prefix $F[0\,..\,\mathsf{per}(F))$ of F.*

▶ **Definition 7** (Lyndon Representation). *The Lyndon representation of a run F is a quadruple $\mathsf{Lrepr}(F) = (\lambda, e, \alpha, \beta)$ such that:*

- *$\lambda = \mathsf{Lroot}(F)$, and*
- *$F = P \cdot \lambda^e \cdot T$, where P is a (possibly empty) suffix of λ with $|P| = \alpha < |\lambda|$, and T is a (possibly empty) prefix of λ with $|T| = \beta < |\lambda|$.*

▶ **Example 8.** Let $S = \underline{\texttt{abaaabaaabaaaba}}\texttt{b}$. The *underlined* fragment $F = S[0\,..\,14] = \texttt{abaaabaaabaaaba}$ is a run with $\mathsf{per}(F) = 4 \leq |F|/2 = 15/2$. Observe that extending F to the right yields the fragment $S[0\,..\,15] = S$, whose period is 14; hence extending F increases its period. Moreover, we have $\mathsf{Lroot}(F) = \texttt{aaab}$ and $\mathsf{Lrepr}(F) = (\texttt{aaab}, 3, 2, 1)$.

▶ **Definition 9** (τ-Run). *A run R of a string S is called a* τ-run*, for an integer $\tau > 0$, if $|R| \geq 3\tau - 1$ and $\mathsf{per}(R) \leq \frac{1}{3}\tau$.*

▶ **Lemma 10** (Lemma 2.8 [9]). *Let S be a string of length n over an integer alphabet $[0, \sigma)$, with $\sigma = n^{\mathcal{O}(1)}$, and let $\tau > 0$ be an integer. Then S contains $\mathcal{O}(n/\tau)$ τ-runs. Moreover, if $\tau \leq \frac{1}{4}\log_\sigma n$, all τ-runs in S can be computed and grouped by their Lyndon roots in $\mathcal{O}(n/\tau)$ time. Within the same time bound, for each τ-run, we can compute the two leftmost occurrences of its Lyndon root.*

▶ **Definition 11** (Shortest Unique Substring (SUS)). *A string P is a* unique substring *of a string S if and only if $|\mathrm{Occ}(P, S)| = 1$. A unique substring P of S is called a* shortest unique substring *of S if there is no string P' with $|P'| < |P|$ that is a unique substring of S.*

▶ **Definition 12** (Shortest Absent Substring (SAS)). *A string P is* absent *from a string S if and only if $|\mathrm{Occ}(P, S)| = 0$. An absent string P of S is called a* shortest absent substring *of S, if there is no string P' with $|P'| < |P|$ that is absent from S.*

String synchronizing sets. We next define a powerful sampling mechanism.

▶ **Definition 13** (String Synchronizing Set [25]). *For a string S of length n and a positive integer $\tau \in [1, \lfloor \frac{n}{2} \rfloor]$, a set $\mathbf{Sync} \subseteq [0, n-2\tau]$ is a* τ-synchronizing set *of S if it satisfies:*

1. *Consistency: For $i, j \in [0, n-2\tau]$, if $S[i\,..\,i+2\tau) = S[j\,..\,j+2\tau)$, then $i \in \mathbf{Sync}$ if and only if $j \in \mathbf{Sync}$.*
2. *Density: For $i \in [0, n-3\tau+1]$, $\mathbf{Sync} \cap [i, i+\tau) = \emptyset$ if and only if $\mathsf{per}(S[i\,..\,i+3\tau-1)) \leq \frac{1}{3}\tau$.*

▶ **Theorem 14** ([17]). *A string $S \in [0, \sigma)^n$ can be preprocessed in $\mathcal{O}(\frac{n}{\log_\sigma n})$ time so that, given $\tau \in [1, \lfloor n/2 \rfloor]$, a τ-synchronizing set $\mathbf{Sync}$ of S of size $|\mathbf{Sync}| < \frac{70n}{\tau}$ can be constructed in $\mathcal{O}(n/\tau)$ time.*

Problem definitions. We now formally define the problems in scope.

Shortest Unique Substring
Input: A packed string S of length n over an integer alphabet $\Sigma = [0, \sigma)$.
Output: $i, j \in \mathbb{Z}_{\geq 0}$ such that $P = S[i\,..\,j]$ is a shortest unique substring of S.

Shortest Absent Substring
Input: A packed string S of length n over an integer alphabet $\Sigma = [0, \sigma)$.
Output: $i, j \in \mathbb{Z}_{\geq 0}$ and $c \in \Sigma$ such that $P = S[i\,..\,j] \cdot c$ is a shortest absent substring of S.

The folklore $\mathcal{O}(n)$-time solutions to the above problems are sketched in [10]. In [10], we also show that any Shortest Unique Substring instance of length n over alphabet $[0, \sigma)$ can be reduced in $\mathcal{O}(\frac{n}{\log_\sigma n})$ time to an instance of length $\mathcal{O}(n \log \sigma)$ over a binary alphabet. This implies that the binary alphabet constitutes a hardest case for computing a SUS. In particular, any algorithm designed for the binary case can be applied to general alphabets via this reduction, without any increase in the asymptotic running time.

In what follows, we assume that all considered strings are given in packed representation. When it is clear from the context, we may omit the term *packed*.

3 Computing a Shortest Unique Substring

We decompose the problem into four cases based on the length ℓ and the period p of a SUS:

1. **Short case**: $\ell \leq \frac{1}{5} \log_\sigma n$ (see [10]);
2. **Medium aperiodic case**: $\ell \in (\frac{1}{5} \log_\sigma n, 2^{\sqrt{\log n}})$ and $p > \frac{1}{45} \log_\sigma n$ (see [10]);
3. **Long aperiodic case**: $\ell \geq \log^4 n$ and $p > \frac{1}{9} \log^4 n$ (see Section 3.2);
4. **Periodic case**: $\ell \geq 3\tau$ and $p \leq \frac{1}{3}\tau$, for $\tau = \lfloor \frac{1}{15} \log_\sigma n \rfloor$ or $\tau = \lfloor \frac{1}{3} \log^4 n \rfloor$ (see Section 3.3). Note that when $\tau = 0$, this case is obsolete because the period of any string is positive.

We defer the proofs for the *short* (see Lemma 15) and *medium aperiodic* (see Corollary 16) cases to the full version [10]: the former case is standard, and for the latter one we draw heavily from previous work [9]. For those cases, we assume familiarity with *suffix trees* and *wavelet trees*; a formal exposition of these is also given in [10].

▶ **Lemma 15** ([10]). *Given an instance of Shortest Unique Substring, in $\mathcal{O}(n/\log_\sigma n)$ time, we can either compute a SUS of S of length at most $\ell := \lfloor \frac{1}{5} \log_\sigma n \rfloor$ if one exists, or conclude that no SUS of length at most ℓ exists.*

▶ **Corollary 16** ([10]). *Given an instance of Shortest Unique Substring, we can compute a SUS P of S in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time if it has length $|P| \in [\frac{1}{5} \log_\sigma n, 2^{\sqrt{\log n}}]$ and period $p > \frac{1}{45} \log_\sigma n$.*

3.1 SUS as a Skyline Problem

In this section, we introduce and solve a simple geometric problem that serves as a subroutine for computing a SUS in both the long aperiodic and the periodic cases.

▶ **Definition 17** (Domination in $\mathbb{Z}^2_{\geq 0}$). *Let $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ be two points in $\mathbb{Z}^2_{\geq 0}$. We say that p_1 is* dominated by *p_2 if $x_1 \leq x_2$ and $y_1 \leq y_2$.*

▶ **Definition 18** (Shadow). *Let P be a multiset of points in $\mathbb{Z}^2_{\geq 0}$. The* shadow *of a point $p \in P$ is the set of points in $\mathbb{Z}^2_{\geq 0}$ that are dominated by p but not dominated by any $p' \in P \setminus \{p\}$.*

▶ **Definition 19** (Skyline). *Let P be a multiset of points in $\mathbb{Z}^2_{\geq 0}$. The* primary skyline *of P is the union of the shadows of all points in P.*

▶ **Example 20.** The primary skyline of $\{(7,3),(7,3)\}$ is the empty set.

In the following, we formalize the Minimum Skyline Point problem, explain its relevance to finding a SUS of S, and present a linear-time solution; see Figure 2 for an example.

Minimum Skyline Point
Input: A multiset P of points in $\mathbb{Z}^2_{\geq 0}$.
Output: A point $(x, y) \in \mathbb{Z}^2_{\geq 0}$ that lies in the primary skyline of P and minimizes $x + y$, if one exists.

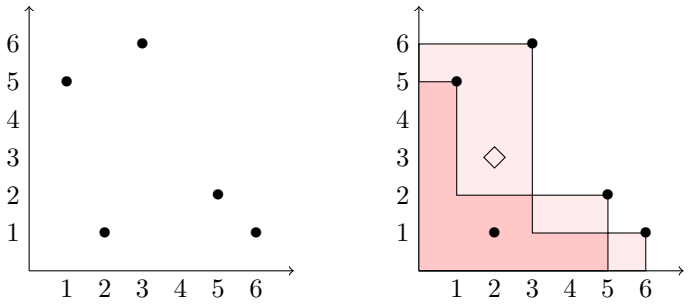


Figure 2 A set P (left) and its primary skyline (right). The union of the shadows of the points in P consists of all points within the shaded polygon but outside its heavily shaded part – the points in the heavily shaded part are dominated by multiple points from P. For this instance of Minimum Skyline Point, the output would be point $(2,3)$, shown with a rhombus.

▶ **Lemma 21.** *Any instance of Minimum Skyline Point can be solved in $\mathcal{O}(|P|)$ time if the points in P are given as a list sorted with respect to one of the two coordinates.*

Proof. We first establish two claims.

▷ Claim 22. Suppose that the primary skyline is nonempty and let $(x^\star, y^\star)$ be a point in the primary skyline with minimal sum of coordinates. Then, either $x^\star = 0$ or there exists a point $(x, y) \in P$ with $x = x^\star - 1$.

Proof. Assume, toward a contradiction, that the claim is false; that is, that we have $x^\star > 0$ and there is no point $(x, y) \in P$ with $x = x^\star - 1$. Let $p^\star = (x^\star, y^\star)$. By the definition of the primary skyline, $p^\star$ must be dominated by exactly one point in P. Now consider the point $p_{\text{left}} = (x^\star - 1, y^\star)$, which, due to the minimality of the coordinate-wise sum of $p^\star$, must be dominated by at least two points in P. Therefore, there exists a point $p' \in P$ that dominates p_{left} and does not dominate $p^\star$. Let $p' = (x', y')$. Since p' does not dominate $p^\star$, we must have either $x' < x^\star$ or $y' < y^\star$. We obtain a contradiction in each case:

- If $x' < x^\star$, then by our assumption that no point $(x, y) \in P$ has $x = x^\star - 1$, we must have $x' \leq x^\star - 2$. Hence, $x' < x^\star - 1$, and thus p' does not dominate p_{left}, a contradiction.
- If $y' < y^\star$, then p' clearly does not dominate p_{left}, a contradiction. ◁

▷ Claim 23. Suppose that the primary skyline is nonempty and let $(x^\star, y^\star)$ be a point in the primary skyline with minimal sum of coordinates. Let $P' = \{(x, y) \in P \mid x \geq x^\star\}$ and let p_{unique} be the unique point of P' that dominates $(x^\star, y^\star)$. If $|P'| = 1$, we have $y^\star = 0$. Else, we have $y^\star = y' + 1$, where y' is the maximum y-coordinate of a point in $P' \setminus \{p_{\text{unique}}\}$.

Proof. By definition, $p^\star := (x^\star, y^\star)$ is dominated by exactly one point, namely p_{unique}. Since points are only dominated by points weakly to their right, we have that p_{unique} must be in P'. We distinguish between two cases:

Case 1: $|P'| = 1$. We have $P' = \{p_{\text{unique}}\}$. If $y^\star > 0$, then the point $p_{\text{down}} = (x^\star, y^\star - 1)$ is in $\mathbb{Z}^2_{\geq 0}$ and the only point of P that dominates it is p_{unique}. Thus, p_{down} lies in the primary skyline, but its coordinate sum $x^\star + y^\star - 1$ is smaller than that of $p^\star$, contradicting the minimality of $p^\star$. Hence, $y^\star = 0$.

Case 2: $|P'| \geq 2$. Let $P_{\text{other}} = P' \setminus \{p_{\text{unique}}\}$, and let y' be the maximum y-coordinate of a point in P_{other}. Since $p^\star$ is dominated uniquely by p_{unique}, every point $(x'', y'') \in P_{\text{other}}$ satisfies $y'' < y^\star$. Hence, $y' < y^\star$. Conversely, since p_{unique} dominates $p^\star$, its y-coordinate is at least $y^\star$. Therefore, the point $(x^\star, y^\star)$ is dominated by exactly one point in P' if and only if $y' < y^\star$. By the minimality of $p^\star$, we have $y^\star = y' + 1$. ◁

Algorithm. We assume, without loss of generality, that the points in P are sorted by x-coordinate in non-decreasing order. We scan P from right to left, while maintaining $y_{\max}$ and y', the two largest y-coordinates encountered among processed points (where $y_{\max} \geq y'$). We initialize $y_{\max}$ and y' to -1, indicating that no processed point has contributed a valid y-coordinate yet.

We iterate through P by grouping points with the same x-coordinate. Let x_i denote the current x-coordinate of the group currently being processed, and let $P_{\text{curr}} = \{(x, y) \in P \mid x = x_i\}$ be the set of points at this coordinate. Before processing the points in P_{curr}, we evaluate the candidate x-coordinate $x^\star = x_i + 1$. Because we have scanned from right to left, all points $(x, y) \in P$ such that $x \geq x^\star$ have already been processed. Let P' denote this set of previously processed points. We determine the corresponding $y^\star$ following Claim 23:

- If $|P'| = 1$, we set $y^\star = 0$.
- If $|P'| \geq 2$ and $y_{\max} > y'$, we set $y^\star = y' + 1$.
- Otherwise (if $|P'| = 0$ or $y_{\max} = y'$), the primary skyline does not intersect the set $\{(x^\star, y) \mid y \in \mathbb{Z}_{\geq 0}\}$, so we skip this candidate.

After evaluating $y^\star$, we update $y_{\max}$ and y' using the y-values in P_{curr} so that they remain the two largest y-coordinates among all processed points.

Finally, after all points in P have been processed, we perform a last check for the candidate $x^\star = 0$ using the final values of $y_{\max}$ and y'. The algorithm maintains the candidate $(x^\star, y^\star)$ that minimizes $x^\star + y^\star$ and returns it as a witness. Each point of P is processed exactly once using $\mathcal{O}(1)$ simple operations, thus the algorithm runs in $\mathcal{O}(|P|)$ time. ◀

Intuition for application to SUS. In the long aperiodic case (Section 3.2) and in the periodic case (Section 3.3), we show that a SUS of S can be found by solving several instances of the Minimum Skyline Point problem. The core idea is to transform the SUS problem into a geometric one. Let us consider one such instance. We first identify a set $\mathcal{S}$ of carefully-selected substrings of S, each anchored around a *common fragment* $F = S[i \,..\, j]$. Each substring $U \in \mathcal{S}$ is then represented by an integer point (x, y), where x and y are the lengths of the extensions of U to the left and the right, respectively, relative to the common fragment F.

▶ **Example 24.** Let $S = \texttt{abracadabra}$, $F = S[4\,..\,6] = \texttt{cad}$, and $\mathcal{S} = \{\texttt{bracada}, \texttt{cadabra}\}$. The substrings are represented as points by the lengths of their extensions relative to F: $\texttt{bracada} \mapsto (3, 1)$ (since $\texttt{bra}$ is of length 3 and $\texttt{a}$ is of length 1), and $\texttt{cadabra} \mapsto (0, 4)$ (since the left extension is empty and $\texttt{abra}$ is of length 4).

Since strings in $\mathcal{S}$ are substrings of S anchored around a common fragment, each point maps to one substring of S. Then, a string $U \in \mathcal{S}$ is a substring of string $V \in \mathcal{S}$ if and only if the point representing U is dominated by the point representing V. To find a candidate SUS in $\mathcal{S}$, we look for a minimal extension anchored around F that is contained within exactly one substring from $\mathcal{S}$. This is equivalent to solving a Minimum Skyline Point instance: finding an integer point $(x^\star, y^\star)$ that lies in the primary skyline of the points representing $\mathcal{S}$ that minimizes $x^\star + y^\star$. The total length of the resulting SUS is then $|F| + x^\star + y^\star$.

3.2 Long Aperiodic Case

This section addresses the computation of a SUS in the case when $\ell \geq \log^4 n$ and $p > \frac{1}{9}\log^4 n$. Let us start with a high-level overview of our solution for this case. We construct a string synchronizing set consisting of *anchor* positions in S that enable the identification of identical sufficiently long aperiodic patterns. These anchors are then used to construct two tries whose root-to-leaf paths represent occurrences of these patterns. SUSs are found by locating the shortest paths that occur only once. We achieve this by formulating several instances of the Minimum Skyline Point problem based on these tries.

We apply Theorem 14 on S with $\tau := \lfloor \frac{1}{3}\log^4 n \rfloor$ and denote the resulting τ-synchronizing set by **Sync**. The following proposition is crucial: if a sufficiently long, aperiodic pattern occurs twice in the string, then every anchor within one occurrence must have a corresponding anchor at the same relative position in the other occurrence. Therefore, to find a unique pattern, we must locate an anchor whose surrounding fragments – those immediately preceding and succeeding it – do not occur around any other anchor and have minimal total length.

▶ **Proposition 25** (follows by Definition 13)**.** *Suppose that a substring X of S with length $|X| = m \geq 3\tau$ and period $\mathsf{per}(X) > \frac{1}{3}\tau$ occurs at distinct positions i and j in S. Then for all $q \in [0, m - 2\tau]$, we have $i + q \in \mathbf{Sync}$ if and only if $j + q \in \mathbf{Sync}$.*

We construct two tries T and T^R. For every anchor position $i \in \mathbf{Sync}$, we insert $S[i\,..\,n)$ into T and $S[0\,..\,i)^R$ into T^R. In both tries, the leaves are labeled with their corresponding anchor positions. For efficiency, we implement them as *compacted* tries, where nodes with a single child are suppressed and are called *implicit*; all other nodes (i.e., the root, branching nodes, and leaves) are maintained and are called *explicit*. For any explicit node v, let $L(v)$ denote the set of leaf labels in the subtree rooted at v, let $\mathsf{str}(v)$ denote the concatenation of the edge labels from the root to v, and let $\mathsf{sd}(v)$ denote the string depth of v. Note that these definitions extend naturally to implicit nodes. If an occurrence of a pattern P contains an anchor $i \in \mathbf{Sync}$, then there exist nodes u in T^R and v in T such that $P = \mathsf{str}(u)^R \cdot \mathsf{str}(v)$ and $i \in L(u) \cap L(v)$. Moreover, the size $|L(u) \cap L(v)|$ equals the number of occurrences of P, provided that P is sufficiently long and aperiodic. We formalize our task as follows; for convenience, for the remainder of this section, we assume that the parent of the root node is itself.

Two Trees SUS
Input: Two rooted trees T_1 and T_2, each with size at most N and with leaves uniquely labeled from $[0, N)$, a weight function w with range $\mathbb{Z}_{\geq 0}$ where $w(u) > w(v)$ for every strict ancestor v of every node u, and an integer k.
Output: A pair (u, v) of nodes u in T_1 and v in T_2 minimizing $w(\mathsf{parent}(u)) + \max(w(\mathsf{parent}(v)), k-1)$, subject to $|L(u) \cap L(v)| = 1$ and $w(v) \geq k$ (if they exist).

▶ **Lemma 26.** *Consider a Shortest Unique Substring instance and let* $\tau := \lfloor \frac{1}{3} \log^4 n \rfloor$. *If* S *has a SUS of length* $\ell \geq 3\tau$ *and period* $p > \frac{1}{3}\tau$, *we can reduce its computation in* $\mathcal{O}(n/\log_\sigma n)$ *time to an instance of Two Trees SUS with* $N = \mathcal{O}(n/\tau)$ *and* $k = 2\tau$.

Proof. We begin by constructing a τ-synchronizing set **Sync** of S. Using this set, we construct two compacted tries T and T^R. For every anchor $i \in \mathbf{Sync}$, we insert the suffix $S[i\,..\,n)$ into T and the reversed prefix $S[0\,..\,i)^R$ into T^R, labeling the corresponding leaves with the anchor i in both tries. We make all implicit nodes in T at string depth $k-1$ explicit. Hence, we have $N = \mathcal{O}(|\mathbf{Sync}|) = \mathcal{O}(n/\tau) = \mathcal{O}(n/\log^4 n)$.

Any fragment P in S that contains at least one anchor $i \in \mathbf{Sync}$ can be decomposed into $P = \mathsf{str}(u)^R \cdot \mathsf{str}(v)$, where u is a node in T^R and v is a node in T, both of which can be implicit. If P has length at least 3τ and period greater than $\frac{1}{3}\tau$, then by the definition of τ-synchronizing sets, all occurrences of P have anchors at the same relative positions. Specifically, any occurrence of P starting at position j must have $j + q \in \mathbf{Sync}$ if and only if the original occurrence at position i had an anchor at the same offset (Proposition 25). Thus, if the node pair (u, v) represents the fragment P, the number of occurrences of P is exactly $|L(u) \cap L(v)|$. To find a SUS, we must find a pair (u, v) of (possibly implicit) nodes such that $|L(u) \cap L(v)| = 1$ (uniqueness), $\mathsf{sd}(v) \geq 2\tau$ (to satisfy the anchor offset property), and $\mathsf{sd}(u) + \mathsf{sd}(v)$ is minimized.

To do so, we set the weights of nodes in the two trees as follows. In both tries, we set the weight of the root node to 0 and assign to each other node u a weight $w(u) := \mathsf{sd}(u)$. We thus create an instance of Two Trees SUS with $k = 2\tau$. If the optimal node pair (u, v) satisfies $w(\mathsf{parent}(u)) + w(\mathsf{parent}(v)) + 2 < 3\tau$, we conclude that S does not have a SUS of length $\ell \geq 3\tau$ and period $p > \frac{1}{3}\tau$. (Note that since we have made all nodes with string depth $k-1$ explicit, for any node v with $w(v) \geq k$, v's parent has weight at least $k-1$, and hence $\max(w(\mathsf{parent}(v)), k-1) = w(\mathsf{parent}(v))$.) Otherwise, we return as a SUS the string $\mathsf{str}(u)^R[0\,..\,w(\mathsf{parent}(u))] \cdot \mathsf{str}(v)[0\,..\,w(\mathsf{parent}(v))]$ – the indices of the occurrence of this string in S can be inferred from the common leaf label in the subtrees of u and v.

The time complexity depends on the construction of the τ-synchronizing set, which requires $\mathcal{O}(n/\log_\sigma n)$ time using Theorem 14, and the construction of the tries. For the tries, we first construct an LCE data structure over S in $\mathcal{O}(n/\log_\sigma n)$ time [25] supporting $\mathcal{O}(1)$-time LCP queries. We sort the N suffixes of S in $\mathcal{O}(N \log N)$ time using merge sort, performing each comparison with an LCP query and a letter comparison. Given the sorted list of suffixes and the LCE data structure, the tries can then be constructed in $\mathcal{O}(N)$ time [24]. The total time for constructing the tries is $\mathcal{O}(n/\log_\sigma n) + \mathcal{O}(N \log N) \subseteq \mathcal{O}(n/\log_\sigma n)$. ◀

3.2.1 Solving the Two Trees SUS Problem

We solve Two Trees SUS as follows. We first decompose the trees T_1 and T_2 into *heavy paths*. For each pair of heavy paths, one from T_1 and one from T_2, we then construct a Minimum Skyline Point instance, which we solve in linear time using Lemma 21.

Heavy-light decomposition. We first recall the widely-used heavy-light decomposition [35].

▶ **Definition 27** (Heavy-Light Decomposition [35]). *Consider a rooted tree $\mathcal{T}$. We obtain a* heavy-light decomposition *of $\mathcal{T}$ by marking each edge as either heavy or light as follows. For every internal node of $\mathcal{T}$, the outgoing edge leading to the child with the largest number of descendants is marked as heavy, while all other outgoing edges are marked as light; ties are resolved arbitrarily. A maximal path of heavy edges is a* heavy path.

A heavy-light decomposition can be constructed in linear time [35]. The following fact holds for any heavy-light decomposition:

▶ **Fact 28** (Lemma 1 [35]). *In a tree with N leaves, any root-to-leaf path intersects at most $\log N$ heavy paths in the decomposition.*

Furthermore, we can construct a *heavy-path tree* that encodes the ancestral relations among all heavy paths and leaves. This is done by contracting every heavy edge, such that all remaining edges are light. In this auxiliary structure, each node corresponds to a contracted heavy path or a leaf. By Fact 28, the heavy-path tree has height at most $\log N$, which allows for efficient traversal across the original tree structure.

A reduction using pairs of heavy paths. We first note that any internal node of a tree belongs to exactly one heavy path in its heavy-light decomposition. We compute the heavy-light decompositions for T_1 and T_2 and consider every pair of heavy paths h_1 from T_1 and h_2 from T_2, provided that they share at least one leaf label.

For a heavy path h, let $L(h)$ denote the set of all leaves descending from the root of h. We define the function $d_h : L(h) \to \mathbb{Z}_{\geq 0}$, where for some leaf $\ell \in L(h)$, $d_h(\ell)$ denotes the weight of ℓ's lowest ancestor within h. Namely, $d_h(\ell)$ denotes the maximum weight of a node on h that is an ancestor of leaf ℓ. With these definitions, we have the following observation:

▶ **Observation 29.** *Given a heavy path h, a node v on h, and a leaf ℓ descending from the root of h, we have that $\ell \in L(v)$ if and only if $w(v) \leq d_h(\ell)$.*

To solve Two Trees SUS, we show that it suffices to solve a Minimum Skyline Point instance for every pair of heavy paths (h_1, h_2) sharing at least one leaf label. By Fact 28, each leaf belongs to at most $\log N$ sets $L(h)$ per tree. We can thus construct each subset $L(h_1) \cap L(h_2)$ by enumerating, for each leaf label ℓ, all $\log^2 N$ pairs of heavy paths above it.

▶ **Lemma 30.** *Any instance of Two Trees SUS can be solved in $\mathcal{O}(N \log^3 N)$ time.*

Proof. We construct the heavy-light decompositions of T_1 and T_2 in $\mathcal{O}(N)$ time [35]. This process partitions each tree into a set of disjoint heavy paths.

For every leaf label $\ell \in [0, N)$ present in both T_1 and T_2, we identify all pairs of heavy paths (h_1, h_2) such that h_1 lies on the root-to-leaf path in the first tree and h_2 lies on the root-to-leaf path in the second tree. Since any root-to-leaf path intersects at most $\mathcal{O}(\log N)$ heavy paths (Fact 28), there are $\mathcal{O}(\log^2 N)$ such pairs for each of the N labels. For each pair, we generate a tuple $(h_1, h_2, d_{h_1}(\ell), d_{h_2}(\ell))$, where d_{h_1} and d_{h_2} are retrieved from the weights of the light-edge endpoints. All tuples are generated in $\mathcal{O}(N \log^2 N)$ time in total and are stored in one list. We sort the list, using the heavy path identifiers (h_1, h_2) as primary keys and the value $d_{h_1}(\ell)$ as the secondary key. This ensures that all tuples corresponding to the same pair (h_1, h_2) appear consecutively, and are then ordered by $d_{h_1}(\ell)$. Using merge sort, this step takes $\mathcal{O}(N \log^2 N \cdot \log(N \log^2 N)) = \mathcal{O}(N \log^3 N)$ time. Let us denote the sublist corresponding to the pair (h_1, h_2) of heavy paths by $\mathcal{H}(h_1, h_2)$.

▷ Claim 31. Consider an instance of Two Trees SUS, in which the output nodes are restricted to a given pair of heavy paths h_1 and h_2 from T_1 and T_2. Given the list $\mathcal{H}(h_1, h_2)$, we can reduce this instance in $\mathcal{O}(g)$ time to an instance of the Minimum Skyline Point problem over a multiset of points of size $\mathcal{O}(g)$, where $g = |\mathcal{H}(h_1, h_2)|$.

Proof. Recall that in the definition of Two Trees SUS, the integer k is given as a minimum weight on one of the returned nodes. Let $\mathcal{L} = L(h_1) \cap L(h_2)$. Assume that $\mathcal{L} \neq \emptyset$; otherwise, the instance has no solution. We wish to find nodes $u \in h_1$ and $v \in h_2$ minimizing $w(\mathsf{parent}(u)) + \max(w(\mathsf{parent}(v)), k-1)$ such that $|L(u) \cap L(v)| = 1$ and $w(v) \geq k$. By Observation 29, this reduces to finding the minimum $(x_1, x_2) \in \mathbb{Z}^2_{\geq 0}$ such that exactly one leaf $\ell \in \mathcal{L}$ satisfies $d_{h_1}(\ell) \geq x_1$ and $d_{h_2}(\ell) \geq x_2$. This is equivalent to solving a Minimum Skyline Point on the multiset $P' = \{(d_{h_1}(\ell), d_{h_2}(\ell)) \mid \ell \in \mathcal{L}\} \sqcup \{(\infty, k-1), (\infty, k-1)\}$, thus finding a solution $(x^\star, y^\star)$ with minimal $x^\star + y^\star$ – the extra point that we insert in the multiset twice ensures that if the primary skyline is not empty, then $y^\star \geq k$. Given $\mathcal{H}(h_1, h_2)$, we can construct this instance in $\mathcal{O}(g)$ time. Finally, to obtain a solution to Two Trees SUS, we select the nodes u in h_1 and v in h_2 satisfying $w(\mathsf{parent}(u)) + 1 = x^\star$ and either $w(\mathsf{parent}(v)) + 1 = y^\star$ or $w(\mathsf{parent}(v)) + 1 \leq k = y^\star \leq w(v)$. ◁

We apply Claim 31 to each pair of heavy paths that share at least one leaf label, and solve each instance of the Minimum Skyline Point problem using Lemma 21 in time linear in the number of points. Since the total number of tuples across all pairs of heavy paths is $\mathcal{O}(N \log^2 N)$, the total time required for applications of Claim 31 and Lemma 21 is $\mathcal{O}(N \log^2 N)$. We maintain the global minimum value of $w(\mathsf{parent}(u)) + \max(w(\mathsf{parent}(v)), k-1)$ found across all pairs of heavy paths and return a witness pair of nodes as the final solution. ◀

3.2.2 Wrapping Up

The final complexity of the long aperiodic case is determined by the combination of the reduction to the Two Trees SUS problem and the subsequent application of the heavy path based Minimum Skyline Point algorithm.

▶ **Lemma 32.** *Given an instance of Shortest Unique Substring, we can compute a SUS of S in $\mathcal{O}(n/\log_\sigma n)$ time if it has length $\ell \geq \log^4 n$ and period $p > \frac{1}{9}\log^4 n$.*

Proof. Using Lemma 26, we reduce the long aperiodic case to an instance of the Two Trees SUS problem. The number of leaf labels (anchors) is $N = \mathcal{O}(|\mathbf{Sync}|) = \mathcal{O}(n/\log^4 n)$. As established in the reduction, constructing the synchronizing set, the LCE data structure, and the two compacted tries T and T^R takes $\mathcal{O}(n/\log_\sigma n)$ time. We then solve the resulting Two Trees SUS instance using the algorithm from Lemma 30. The time required is $\mathcal{O}(N \log^3 N)$ and as N is in $\mathcal{O}(n/\log^4 n)$, we obtain a running time of $\mathcal{O}(n/\log n)$. Therefore, both the reduction and the solver fit within the target time bound of $\mathcal{O}(n/\log_\sigma n)$. The algorithm identifies a pair (u, v) minimizing $w(\mathsf{parent}(u)) + \max(w(\mathsf{parent}(v)), k-1)$ subject to the uniqueness and length constraints. Since the result of the Two Trees SUS problem provides the minimal unique extension for any pattern overlapping an anchor, the resulting substring is a valid SUS for the long aperiodic case. ◀

3.3 Periodic Case

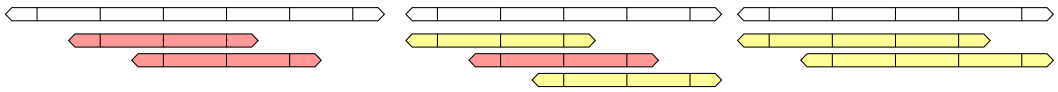
In this section, we address the SUS computation in highly periodic substrings of S. Specifically, we consider substrings of S of medium length with period at most $\frac{1}{45}\log_\sigma n$, as well as long substrings of S with period at most $\frac{1}{9}\log^4 n$. These two cases are handled using τ-runs with $\tau = \lfloor \frac{1}{15}\log_\sigma n \rfloor$ and $\tau = \lfloor \frac{1}{3}\log^4 n \rfloor$, respectively.

For the first group of τ-runs (where $\tau = \lfloor \frac{1}{15} \log_\sigma n \rfloor$), we employ their standard Lyndon representation (Definition 7). These runs can be efficiently computed and grouped by means of Lemma 10. Unfortunately, Lemma 10 is not applicable when $\tau = \lfloor \frac{1}{3} \log^4 n \rfloor$. To deal with this, we utilize the recently introduced *sparse-Lyndon representation* [11] that allows for efficiently grouping the τ-runs for $\tau = \lfloor \frac{1}{3} \log^4 n \rfloor$ according to their Lyndon roots, but representing via their sparse-Lyndon root. We formalize this discussion in Lemma 33.

▶ **Lemma 33** (Proposition 36 [11]). *For any string S of length n over an integer alphabet $[0, \sigma)$, with $\sigma = n^{\mathcal{O}(1)}$, all runs in S can be computed and grouped by equal Lyndon roots in $\mathcal{O}(n/\log_\sigma n)$ time. For runs with period at most $2\lfloor \frac{1}{18} \log_\sigma n \rfloor$, we compute their standard Lyndon representations, while for runs with larger periods we compute their sparse-Lyndon representations.*

To streamline the subsequent analysis, we hereafter slightly *abuse terminology*: the terms τ-run, Lyndon root, and Lyndon representation will be used uniformly for the two classes of runs considered in this section. In the context of the second class ($\tau = \lfloor \frac{1}{3} \log^4 n \rfloor$), these terms implicitly refer to their sparse-Lyndon counterparts.

This unified terminology is justified by the fact that both representations share analogous properties and can be handled identically within our algorithmic framework. We group all τ-runs by Lyndon root in $\mathcal{O}(n/\log_\sigma n)$ time using Lemma 33; the only difference is that for some groups, we compute the Lyndon representation of runs, while for others, we compute the sparse-Lyndon representation. For further details and a thorough discussion of sparse-Lyndon representations, we refer the reader to [11].



(a) $e < e' - 2$ (Case 1). **(b)** $e = e' - 2$ (Case 2). **(c)** $e = e' - 1$ (Case 3).

Figure 3 Cases 1–3 of Lemma 34, illustrating the potential alignment of a run with exponent e within a run with exponent e'. Red alignments are guaranteed to be contained within the longer run; yellow alignments are contained only if the extensions of the runs satisfy specific inequalities.

▶ **Lemma 34.** *Let F and F' be two runs in S with the same Lyndon root λ, where $\mathsf{Lrepr}(F) = (\lambda, e, \alpha, \beta)$ and $\mathsf{Lrepr}(F') = (\lambda, e', \alpha', \beta')$. We can determine whether F is a* unique substring *of F' as follows:*

1. *If $e < e' - 2$, then F is a substring of F', but it is not unique.*
2. *If $e = e' - 2$, then F is a substring of F'. It is unique if and only if $\alpha > \alpha'$ and $\beta > \beta'$.*
3. *If $e = e' - 1$, then F is a unique substring of F' if and only if* either *$\alpha > \alpha'$* or *$\beta > \beta'$, but not both. (Note: if both $\alpha > \alpha'$* and *$\beta > \beta'$ hold, F is not a substring of F'.)*
4. *If $e = e'$, then F is a unique substring of F' if and only if $\alpha \leq \alpha'$ and $\beta \leq \beta'$.*
5. *If $e > e'$, then F is not a substring of F'.*

Proof. We analyze each case separately by aligning the occurrences of the Lyndon roots in F and F' in all possible ways and comparing their extensions; see Figure 3 for illustrative examples. This allows us to determine if no alignment of F is contained within F' (i.e., F is not a substring of F'), exactly one alignment is contained (unique substring), or multiple alignments are contained (not unique).

1. If $e < e' - 2$, we can align the first occurrence of the Lyndon root of F with either the second or third occurrence of the Lyndon root of F'. In both cases, both extensions of F coincide with complete occurrences of the Lyndon root of F', meaning that F is contained in both alignments and is therefore not a unique substring.
2. If $e = e' - 2$, there are three possible alignments of the Lyndon roots. In the second alignment, both extensions of F coincide with complete occurrences of the Lyndon root of F', so F is a substring of F'. In the first alignment, the left extensions of the runs are aligned, meaning that F is contained in this alignment unless $\alpha > \alpha'$. Similarly, in the third alignment, the same reasoning applies to the right extensions. Therefore, F is a unique substring of F' if and only if $\alpha > \alpha'$ and $\beta > \beta'$.
3. If $e = e' - 1$, there are two possible alignments. In the first alignment, the left extensions of the runs are aligned while the right extension of F coincides with a complete occurrence of the Lyndon root of F'; thus, F is contained in this alignment if $\alpha \leq \alpha'$. The same reasoning applies symmetrically for the second alignment. Therefore, F is a unique substring of F' if and only if either $\alpha > \alpha'$ or $\beta > \beta'$, but not both.
4. If $e = e'$, there is only one possible alignment of the Lyndon roots. For F to be a substring of F', both extensions of F must be shorter than or equal to those of F'; that is, $\alpha \leq \alpha'$ and $\beta \leq \beta'$.
5. If $e > e'$, any alignment will result in an extension of F' being aligned with a Lyndon root of F; therefore, F cannot possibly be a substring of F'. ◀

For the following definitions, let $\mathcal{R}_\lambda$ be the set of τ-runs in S sharing the same Lyndon root λ. Let $r := |\lambda|$, and let $e_\lambda^{\max}$ denote the maximum exponent among all τ-runs in $\mathcal{R}_\lambda$.

▶ **Definition 35** (Mapping $\mathfrak{h}$). *Let $R \in \mathcal{R}_\lambda$ be a run with $\mathsf{Lrepr}(R) = (\lambda, e, \alpha, \beta)$. We define the function $\mathfrak{h}$, which maps runs to sets of up to three points as follows:*

- *If $e < e_\lambda^{\max} - 2$, then $\mathfrak{h}(R) = \emptyset$;*
- *If $e = e_\lambda^{\max} - 2$, then $\mathfrak{h}(R) = \{(\alpha, \beta)\}$;*
- *If $e = e_\lambda^{\max} - 1$, then $\mathfrak{h}(R) = \{(r + \alpha, \beta), (\alpha, r + \beta)\}$;*
- *If $e = e_\lambda^{\max}$, then $\mathfrak{h}(R) = \{(2r - 1, \beta), (\alpha, 2r - 1), (r + \alpha, r + \beta)\}$.*

▶ **Definition 36** (Mapping $\mathfrak{g}$). *Let $p = (x, y)$ be an integer point in $[0, 2r - 1]^2$. We define the function $\mathfrak{g}$ as a mapping from such points to candidate runs as follows:*

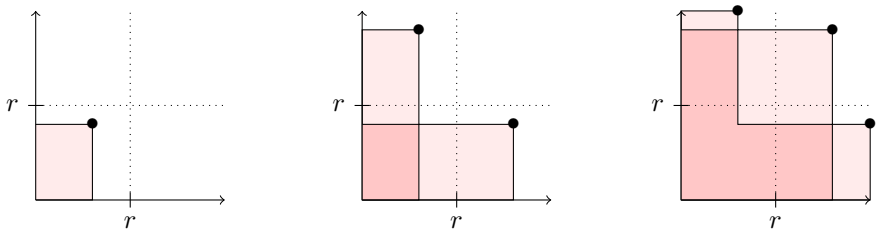
- *$\mathfrak{g}(p)$ has Lyndon root λ;*
- *The left and right extensions of $\mathfrak{g}(p)$ are $\alpha' = x \bmod r$ and $\beta' = y \bmod r$, respectively;*
- *The exponent e' of $\mathfrak{g}(p)$ is determined by the quadrant: $e' = e_\lambda^{\max}$, if ($x \geq r$ and $y \geq r$); $e' = e_\lambda^{\max} - 1$, if ($x \geq r$ and $y < r$) or ($x < r$ and $y \geq r$); $e' = e_\lambda^{\max} - 2$, if ($x < r$ and $y < r$).*

▶ **Example 37.** Let $\mathcal{R}_\lambda$ be a set of runs sharing the Lyndon root $\lambda = \mathtt{ab}$ with period $r = |\lambda| = 2$. Assume that $\mathcal{R}_\lambda$ consists of three runs, with a maximum exponent of $e_\lambda^{\max} = 6$:

- $\mathsf{Lrepr}(R_1) = (\mathtt{ab}, 6, 1, 1) \implies \mathfrak{h}(R_1) = \{(3, 1), (1, 3), (3, 3)\}$;
- $\mathsf{Lrepr}(R_2) = (\mathtt{ab}, 5, 0, 1) \implies \mathfrak{h}(R_2) = \{(2, 1), (0, 3)\}$;
- $\mathsf{Lrepr}(R_3) = (\mathtt{ab}, 4, 1, 0) \implies \mathfrak{h}(R_3) = \{(1, 0)\}$.

The geometric domain is $[0, 3]^2$, as $2r - 1 = 3$. Applying the mapping function $\mathfrak{h}$ to all runs in $\mathcal{R}_\lambda$ generates the point set $P = \{(3, 1), (1, 3), (3, 3), (2, 1), (0, 3), (1, 0)\}$.

Consider the point $(2, 2)$. Since $x \geq r$ and $y \geq r$ (specifically, $2 \geq 2$), the mapping $\mathfrak{g}(2, 2)$ assigns the exponent $e' = 6$ and extensions $\alpha' = 2 \bmod 2 = 0$ and $\beta' = 2 \bmod 2 = 0$. The mapping $\mathfrak{g}$ thus yields the run: $\mathfrak{g}(2, 2) = (\lambda, 6, 0, 0)$. This representation corresponds to the string $\lambda^6 = (\mathtt{ab})^6 = \mathtt{abababababab}$. Note that the point $(2, 2)$ lies on the primary



(a) $e = e_\lambda^{\max} - 2$. **(b)** $e = e_\lambda^{\max} - 1$. **(c)** $e = e_\lambda^{\max}$.

Figure 4 The mapping $\mathfrak{h}$ from runs to sets of points in $[0, 2r-1]^2$. The light shaded part represents the primary skyline, while dark shaded regions indicate areas dominated by multiple points, where no unique substring can exist.

skyline of P and is minimal in terms of $x + y$. This geometric property ensures that the corresponding run $\mathfrak{g}(2, 2)$ belongs to exactly one periodic alignment among the runs in $\mathcal{R}_\lambda$ (specifically within R_1), making `abababababab` a valid SUS candidate.

▶ **Lemma 38.** *Let $R \in \mathcal{R}_\lambda$ be a run with $\mathsf{Lrepr}(R) = (\lambda, e, \alpha, \beta)$, and let $p = (x, y)$ be a point in the domain $[0, 2r-1]^2$. Then p is in the primary skyline of $\mathfrak{h}(R)$ if and only if $\mathfrak{g}(p)$ is a unique substring of R.*

Proof. Let $\mathfrak{g}(p) = R'$, where $\mathsf{Lrepr}(R') = (\lambda, e', \alpha', \beta')$. By the definition of $\mathfrak{g}$, we have $\alpha' = x \bmod r$ and $\beta' = y \bmod r$. The quadrant containing p determines the candidate exponent e', while the set of points $\mathfrak{h}(R)$ (and thus its primary skyline) is determined by the run exponent e; see Figure 4. We verify the correspondence with the cases in Lemma 34 by distinguishing how e compares to e'.

- **Case $e' = e$:** The point p can be dominated by exactly one point from $\mathfrak{h}(R)$ with relative coordinates (α, β). Hence, p belongs to the primary skyline if and only if it is dominated by this single point, which occurs when $\alpha' \le \alpha$ and $\beta' \le \beta$. This corresponds exactly to the condition for $\mathfrak{g}(p)$ to be a unique substring of R (Lemma 34: Case 4).
- **Case $e' = e - 1$:** In this case, p can be dominated by at most two points from $\mathfrak{h}(R)$. Point p is dominated by the first point if $\alpha' \le \alpha$ and by the second if $\beta' \le \beta$. Thus, p belongs to the primary skyline if it is dominated by exactly one of these points. This requires either $\alpha' \le \alpha$ or $\beta' \le \beta$ but not both. Again, this matches the condition for $\mathfrak{g}(p)$ to be a unique substring of R (Lemma 34: Case 3).
- **Case $e' = e - 2$:** This occurs only when $e = e_\lambda^{\max}$ and p is in the lower-left quadrant $(x, y < r)$. Here, p is always dominated by the point $(r + \alpha, r + \beta)$ of $\mathfrak{h}(R)$. For p to be in the primary skyline, it must *not* be dominated by the other two points $(2r - 1, \beta)$ and $(\alpha, 2r - 1)$. This lack of dominance occurs if and only if $\alpha' > \alpha$ and $\beta' > \beta$. This happens exactly when $\mathfrak{g}(p)$ is a unique substring of R (Lemma 34: Case 2). ◀

▶ **Lemma 39.** *Given a set $\mathcal{R}_\lambda$ of runs sharing the same Lyndon root λ, a SUS among these runs can be found by solving a Minimum Skyline Point instance containing at most $3|\mathcal{R}_\lambda|$ points. The corresponding set of (unsorted) points can be constructed in $\mathcal{O}(|\mathcal{R}_\lambda|)$ time.*

Proof. Let $e_\lambda^{\max}$ denote the maximum exponent among all runs in $\mathcal{R}_\lambda$. By Lemma 34, we know that any SUS within these runs must have an exponent equal to $e_\lambda^{\max} - 2$, $e_\lambda^{\max} - 1$ or $e_\lambda^{\max}$. Such a substring thus corresponds to $\mathfrak{g}(p)$ for some point $p \in [0, 2r-1]^2$, with $r = |\lambda|$.

By Lemma 38, p must lie on the primary skyline of $\mathfrak{h}(R)$ for some $R \in \mathcal{R}_\lambda$. Moreover, p cannot belong to the primary skylines of multiple runs; otherwise, $\mathfrak{g}(p)$ would not be unique overall. Hence, p must be dominated by exactly one point among all sets $\mathfrak{h}(R)$, which can be identified by solving the Minimum Skyline Point instance on the multiset $\bigsqcup_{R\in\mathcal{R}_\lambda} \mathfrak{h}(R)$. Since $|\mathfrak{h}(R)| \leq 3$ for every R, this instance has size $\mathcal{O}(|\mathcal{R}_\lambda|)$. Finally, if the resulting point from this instance is (x, y), it corresponds to a substring of length $r \cdot (e_\lambda^{\max} - 2) + x + y$. ◀

▶ **Lemma 40.** *Given an instance of Shortest Unique Substring and an integer τ, such that $\tau = \lfloor \frac{1}{15} \log_\sigma n \rfloor$ or $\tau = \lfloor \frac{1}{3} \log^4 n \rfloor$, we can compute a SUS of S in $\mathcal{O}(n/\log_\sigma n)$ time if it has length $\ell \geq 3\tau$ and period $p \leq \frac{1}{3}\tau$.*

Proof. By Lemma 10 and Lemma 33, all runs in S can be computed and grouped by their Lyndon roots in $\mathcal{O}(n/\log_\sigma n)$ time. Each τ-run is encoded in $\mathcal{O}(1)$ space. Next, using Lemma 39, we construct Minimum Skyline Point instances for all groups. Since each τ-run generates a constant number of points, the total number of points across all instances is $\mathcal{O}(n/\tau)$. These points are computed in $\mathcal{O}(n/\tau)$ time as per Lemma 10. To solve all Minimum Skyline Point instances in linear time (Lemma 21), the points for each $\mathcal{R}_\lambda$ instance must be sorted along one axis. This is achieved by globally sorting all generated points using bucket sort. Because the coordinates x and y for any point in an instance for root λ are bounded by $2|\lambda| - 1 < 2\tau$, the bucket sort can be performed in $\mathcal{O}(n/\tau + \tau)$ time. After sorting, the points are regrouped per their Lyndon roots. For either chosen value of τ, we have $\mathcal{O}(n/\tau + \tau) = \mathcal{O}(n/\log_\sigma n)$. This yields a total running time of $\mathcal{O}(n/\log_\sigma n)$. ◀

3.4 Putting It All Together

▶ **Theorem 1** (SUS). *Given a packed string S of length n over an integer alphabet $[0, \sigma)$, a shortest unique substring of S can be computed in $\mathcal{O}(n \log \sigma/\sqrt{\log n})$ time.*

Proof. We have described algorithms to find SUSs with differing lengths and periods: Lemma 15 handles any length up to $\frac{1}{5}\log_\sigma n$ with any period; Corollary 16 handles lengths in the range $[\frac{1}{5}\log_\sigma n, 2^{\sqrt{\log n}}]$ with period greater than $\frac{1}{45}\log_\sigma n$; and Lemma 32 handles lengths above $\log^4 n$ with period greater than $\frac{1}{9}\log^4 n$. Finally, with parameter values $\tau := \lfloor \frac{1}{15}\log_\sigma n \rfloor$ and $\tau := \lfloor \frac{1}{3}\log^4 n \rfloor$, Lemma 40 handles the larger length ranges with smaller periods. We run each of the algorithms and return the globally shortest substring output across all instances. The total running time is $\mathcal{O}(n \log \sigma/\sqrt{\log n})$ which is asymptotically dominated by the complexity of the medium-length aperiodic case (Corollary 16). ◀

4 Computing a Shortest Absent Substring

We begin by defining an auxiliary problem closely related to Shortest Unique Substring.

Shortest Exclusive Substring
Input: Two strings S_1 and S_2 with $n = \|S_1\| + \|S_2\|$ over an integer alphabet $\Sigma = [0, \sigma)$.
Output: A shortest substring of S_1 that does not occur in S_2 (if one exists).

Similar to the solution for Shortest Unique Substring, we solve the Shortest Exclusive Substring problem by decomposing it into four cases based on the length ℓ and the period p of the sought substring of S_1, obtaining the following result.

▶ **Theorem 41** ([10]). *Any instance of Shortest Exclusive Substring can be solved in $\mathcal{O}(n \log \sigma/\sqrt{\log n})$ time when S_1 and S_2 are given in packed representation.*

The proof of Theorem 41 is deferred to [10]. We next present an efficient construction of de Bruijn sequences in the packed setting, which may be of independent interest.

▶ **Definition 42** (De Bruijn Sequence [14]). *A* de Bruijn sequence *of order k over an alphabet Σ of size σ is a string of length $\sigma^k + k - 1$ in which every string from Σ^k occurs exactly once.*

▶ **Lemma 43.** *A packed de Bruijn sequence of order k over the integer alphabet $[0, \sigma)$ for $\sigma \geq 2$ can be constructed in $\mathcal{O}(\frac{\sigma^k}{k}) = \mathcal{O}(\frac{n}{\log_\sigma n})$ time, where n is the length of the sequence. In particular, we can construct its prefix of length ℓ in $\mathcal{O}(\frac{\ell}{\log_\sigma n} + 1)$ time.*

Proof. A *Lyndon word* is a string that is lexicographically strictly smaller than all of its proper suffixes. As noted by Fredricksen and Maiorana [19], the concatenation of all Lyndon words whose length divides k, listed in lexicographical order, forms a de Bruijn sequence. Duval [15] provided an algorithm to generate all Lyndon words of length at most k in lexicographical order, generating each word by modifying the previous one.

We achieve the stated running time in the packed setting, by observing that $k \leq \lfloor \log_\sigma n \rfloor$, and using $\mathcal{O}(1)$ machine words to iterate over all Lyndon words in $[0, \sigma)^{\leq k}$.

In what follows, we describe Duval's algorithm [15] without proving its correctness; we only explain how it can be performed efficiently in the packed setting. We will return a string S, initialized as 0. We then maintain a length-k string w initialized as 0^k and a binary string w' whose ith bit is set if and only if $w[i] = \sigma - 1$ (we update w' together with w). We repeatedly apply the following steps of the generation loop while $w[0] \neq \sigma - 1$:

1. Locate $j = \max\{i : w[i] \neq \sigma - 1\}$. This is done by finding the rightmost 0 in w' in $\mathcal{O}(1)$ time using standard bitwise operations.
2. Replace $w[j]$ with $w[j] + 1$ in $\mathcal{O}(1)$ time.
3. The prefix $w[0\,..\,j]$ is now a Lyndon word. If $(j + 1)$ divides k, we append $w[0\,..\,j]$ to S. This takes $\mathcal{O}(1)$ time as the appended string fits into $\mathcal{O}(1)$ machine words.
4. Replace w with the length-k prefix of $(w[0\,..\,j])^\infty$. This can be naively performed using $\mathcal{O}(k/(j+1))$ operations. Consider a potential function $\pi(w) = |\{i : w[i] = \sigma - 1\}|$, noting that Step 2 increases $\pi(w)$ by at most 1. Conversely, in Step 4, values $w[i] = \sigma - 1$ for $i > j$ are replaced with $w[i \bmod (j+1)]$. Since $w[0] < \sigma - 1$ (the algorithm terminates when $w[0]$ reaches $\sigma - 1$), $\Omega(k/(j + 1))$ copies of $\sigma - 1$ are overwritten by setting $w = (w[0\,..\,j])^\infty$. As $\pi(w) \geq 0$ at all times, the total number of operations we perform in Step 4 is asymptotically upper bounded by the number of times Step 2 is executed, and hence the amortized running time of Step 4 is $\mathcal{O}(1)$.

After the last iteration, we complete the sequence by appending 0^{k-1}. Observe that, for each integer ℓ there are at most $\frac{\sigma^\ell}{\ell}$ Lyndon words of length ℓ, and hence at most $\sum_{\ell \leq k} \frac{\sigma^\ell}{\ell} \leq 3\frac{\sigma^k}{k}$ Lyndon words of length at most k. This proves that there are $\mathcal{O}(\frac{\sigma^k}{k})$ iterations of the loop, and the algorithm takes $\mathcal{O}(\frac{n}{\log_\sigma n})$ time (where $n = \sigma^k + k - 1$ and $k \leq \lfloor \log_\sigma n \rfloor$) as claimed. To output a length-ℓ prefix of this sequence we terminate the algorithm once the length of the output sequence reaches ℓ, potentially removing up to $k - 1$ excessive letters from the final Lyndon word. Since $k/\log_\sigma n = \mathcal{O}(1)$, the total time complexity is

$$\mathcal{O}\left(\frac{\ell + k}{\log_\sigma n} + 1\right) = \mathcal{O}\left(\frac{\ell}{\log_\sigma n} + 1\right). \qquad \blacktriangleleft$$

▶ **Theorem 2** (SAS). *Given a packed string S of length n over an integer alphabet $[0, \sigma)$, a shortest absent substring of S can be computed in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time.*

Proof. Let $k := \lfloor \log_\sigma n \rfloor + 1$. Since $\sigma^k > n$, the length of any SAS is at most k. We first check for a SAS of length at most $k - 1$ by applying Theorem 41 to a packed de Bruijn sequence S_1 of order $k - 1$ (Lemma 43) and $S_2 := S$. If this yields a substring of length

at most $k-1$, we are done. Otherwise, if $\sigma^k = \mathcal{O}(n)$, we create a full packed de Bruijn sequence of order k and repeat the process to find a SAS of length k. If $\sigma^k = \omega(n)$, we instead generate a prefix S_1' of a de Bruijn sequence of order k of length $n+1$ in $\mathcal{O}(\frac{n}{\log_\sigma n})$ time. S_1' contains $n-k+2$ distinct substrings of length k. Since S_2 has at most $n-k+1$ such substrings, at least one substring of S_1' must be absent from S_2. We find this witness of length k using Theorem 41 for S_1' and S_2. The calls to Theorem 41 dominate the total running time, which is $\mathcal{O}\left(\frac{n\log\sigma}{\sqrt{\log n}}\right)$. ◀

References

1 Ricardo A. Baeza-Yates. Improved string searching. *Softw. Pract. Exp.*, 19(3):257–271, 1989. doi:10.1002/SPE.4380190305.

2 Hideo Bannai and Jonas Ellert. Lyndon arrays in sublinear time. In *31st Annual European Symposium on Algorithms, ESA 2023*, volume 274 of *LIPIcs*, pages 14:1–14:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ESA.2023.14.

3 Djamal Belazzougui. Worst case efficient single and multiple string matching in the RAM model. In *Combinatorial Algorithms - 21st International Workshop, IWOCA 2010*, volume 6460 of *Lecture Notes in Computer Science*, pages 90–102. Springer, 2010. doi:10.1007/978-3-642-19222-7_10.

4 Oren Ben-Kiki, Philip Bille, Dany Breslauer, Leszek Gasieniec, Roberto Grossi, and Oren Weimann. Towards optimal packed string matching. *Theoretical Computer Science*, 525:111–129, 2014. doi:10.1016/j.tcs.2013.06.013.

5 Philip Bille. Fast searching in packed strings. *Journal of Discrete Algorithms*, 9(1):49–56, 2011. doi:10.1016/j.jda.2010.09.003.

6 Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen. Deterministic indexing for packed strings. In *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78 of *LIPIcs*, pages 6:1–6:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.6.

7 Dany Breslauer, Leszek Gasieniec, and Roberto Grossi. Constant-time word-size string matching. In *Combinatorial Pattern Matching - 23rd Annual Symposium, CPM 2012, Proceedings*, volume 7354 of *Lecture Notes in Computer Science*, pages 83–96. Springer, 2012. doi:10.1007/978-3-642-31265-6_7.

8 Panagiotis Charalampopoulos, Maxime Crochemore, Gabriele Fici, Robert Mercas, and Solon P. Pissis. Alignment-free sequence comparison using absent words. *Inf. Comput.*, 262:57–68, 2018. doi:10.1016/J.IC.2018.06.002.

9 Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, and Solon P. Pissis. Faster algorithms for longest common substring. *ACM Trans. Algorithms*, 2025. doi:10.1145/3774754.

10 Panagiotis Charalampopoulos, Manal Mohamed, Solon P. Pissis, Hilde Verbeek, and Wiktor Zuba. Faster algorithms for unique or absent substrings. *CoRR*, 2026. doi:10.48550/arXiv.2605.04826.

11 Panagiotis Charalampopoulos, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Counting distinct square substrings in sublinear time. In *50th International Symposium on Mathematical Foundations of Computer Science, MFCS 2025*, volume 345 of *LIPIcs*, pages 36:1–36:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.MFCS.2025.36.

12 Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest palindromic substring in sublinear time. In *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022*, volume 223 of *LIPIcs*, pages 20:1–20:9. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CPM.2022.20.

13 Maxime Crochemore, Filippo Mignosi, Antonio Restivo, and Sergio Salemi. Data compression using antidictionaries. *Proc. IEEE*, 88(11):1756–1768, 2000. doi:10.1109/5.892711.

14 Nicolaas Govert de Bruijn. A combinatorial problem. *Proc. Koninklijke Nederlandse Akademie V. Wetenschappen*, 49:758–764, 1946. URL: http://resolver.tudelft.nl/uuid:f9a56551-402a-4428-98e6-1469e38e64c1.

15 Jean-Pierre Duval. Génération d'une section des classes de conjugaison et arbre des mots de lyndon de longueur bornée. *Theor. Comput. Sci.*, 60:255–283, 1988. doi:10.1016/0304-3975(88)90113-2.

16 Jonas Ellert. Sublinear time Lempel-Ziv (LZ77) factorization. In *String Processing and Information Retrieval - 30th International Symposium, SPIRE 2023, Proceedings*, volume 14240 of *Lecture Notes in Computer Science*, pages 171–187. Springer, 2023. doi:10.1007/978-3-031-43980-3_14.

17 Jonas Ellert and Tomasz Kociumaka. Time-optimal construction of string synchronizing sets. In *43rd International Symposium on Theoretical Aspects of Computer Science, STACS 2026*, LIPIcs, pages 36:1–36:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2026. doi:10.4230/LIPIcs.STACS.2026.36.

18 Johannes Fischer and Paweł Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Proceedings*, volume 9133 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 2015. doi:10.1007/978-3-319-19929-0_14.

19 Harold Fredricksen and James Maiorana. Necklaces of beads in k colors and k-ary de bruijn sequences. *Discret. Math.*, 23(3):207–210, 1978. doi:10.1016/0012-365X(78)90002-X.

20 Kimmo Fredriksson. Shift-or string matching with super-alphabets. *Information Processing Letters*, 87(4):201–204, 2003. doi:10.1016/S0020-0190(03)00296-5.

21 Szymon Grabowski and Kimmo Fredriksson. Bit-parallel string matching under hamming distance in o(n[m/w]) worst case time. *Inf. Process. Lett.*, 105(5):182–187, 2008. doi:10.1016/J.IPL.2007.08.021.

22 Bernhard Haubold, Nora Pierstorff, Friedrich Möller, and Thomas Wiehe. Genome comparison without alignment using shortest unique substrings. *BMC Bioinform.*, 6:123, 2005. doi:10.1186/1471-2105-6-123.

23 Costas S. Iliopoulos, Dennis W. G. Moore, and William F. Smyth. A characterization of the squares in a fibonacci string. *Theor. Comput. Sci.*, 172(1-2):281–291, 1997. doi:10.1016/S0304-3975(96)00141-7.

24 Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2001. doi:10.1007/3-540-48194-X_17.

25 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 756–767. ACM, 2019. doi:10.1145/3313276.3316368.

26 Dominik Kempa and Tomasz Kociumaka. Breaking the $O(n)$-barrier in the construction of compressed suffix arrays and suffix trees. In *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023*, pages 5122–5202. SIAM, 2023. doi:10.1137/1.9781611977554.CH187.

27 Dominik Kempa and Tomasz Kociumaka. Lempel-Ziv (LZ77) factorization in sublinear time. In *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024*, pages 2045–2055. IEEE, 2024. doi:10.1109/FOCS61266.2024.00122.

28 Dominik Kempa and Tomasz Kociumaka. On the hardness hierarchy for the o($n\sqrt{\log n}$) complexity in the word RAM. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing, STOC 2025*, pages 290–300. ACM, 2025. doi:10.1145/3717823.3718291.

29 J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. Fast construction of wavelet trees. *Theor. Comput. Sci.*, 638:91–97, 2016. doi:10.1016/j.tcs.2015.11.011.

30 Gonzalo Navarro and Mathieu Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Combinatorial Pattern Matching, 9th Annual Symposium, CPM 98, Proceedings*, volume 1448 of *Lecture Notes in Computer Science*, pages 14–33. Springer, 1998. doi:10.1007/BFB0030778.

31 Jian Pei, Wush Chi-Hsuan Wu, and Mi-Yen Yeh. On shortest unique substring queries. In *29th IEEE International Conference on Data Engineering, ICDE 2013*, pages 937–948. IEEE Computer Society, 2013. doi:10.1109/ICDE.2013.6544887.

32 Diogo Pratas and Jorge Miguel Silva. Persistent minimal sequences of SARS-CoV-2. *Bioinform.*, 36(21):5129–5132, 2021. doi:10.1093/BIOINFORMATICS/BTAA686.

33 Jakub Radoszewski and Wiktor Zuba. Computing string covers in sublinear time. In *String Processing and Information Retrieval - 31st International Symposium, SPIRE 2024, Proceedings*, volume 14899 of *Lecture Notes in Computer Science*, pages 272–288. Springer, 2024. doi:10.1007/978-3-031-72200-4_21.

34 Raquel M. Silva, Diogo Pratas, Luísa Castro, Armando J. Pinho, and Paulo Jorge S. G. Ferreira. Three minimal sequences found in ebola virus genomes and absent from human DNA. *Bioinform.*, 31(15):2421–2425, 2015. doi:10.1093/BIOINFORMATICS/BTV189.

35 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

36 Takuya Takagi, Shunsuke Inenaga, Kunihiko Sadakane, and Hiroki Arimura. Packed compact tries: A fast and efficient data structure for online string processing. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 100-A(9):1785–1793, 2017. doi:10.1587/TRANSFUN.E100.A.1785.

37 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.