# Optimal Structure for Prefix-Substring Queries

**Paweł Gawrychowski**
Faculty of Mathematics and Computer Science, University of Wrocław, Poland

**Florin Manea**
Department of Informatics, University of Göttingen, Germany

**Jonas Richardsen**
Department of Informatics, University of Göttingen, Germany

## Abstract

The prefix-substring matching problem [Gu, Farach, and Beigel, SODA 1994] consists in preprocessing a string $s$ of length $n$ for the following queries: given a triple $(i, j, k) \in \{0, \ldots, |s|\}^3$ with $1 \leq j \leq k$, representing a prefix $s[1 : i]$ and a substring $s[j : k]$ of $s$, find the longest prefix of $s$ that is a suffix of $s[1 : i]s[j : k]$. This is an useful primitive in e.g. dynamic text indexing, compressed pattern matching, and pattern matching on block graphs. The border tree uses some basic periodicity properties to answer such queries in $\mathcal{O}(\log n)$ time after $\mathcal{O}(n)$ time preprocessing of $s$. We design a linear-space structure that answers such queries in constant time after $\mathcal{O}(n)$ time preprocessing of $s$ over a polynomial alphabet, which is worst-case optimal.

**2012 ACM Subject Classification** Theory of computation → Pattern matching

**Keywords and phrases** Border Tree, Prefix-Substring Query, Data Structures

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2026.7

**Funding** Paweł Gawrychowski is partially supported by the Polish National Science Centre grant number 2023/51/B/ST6/01505. The work of Florin Manea and Jonas Richardsen was partly supported by the German Research Foundation (DFG) through the Heisenberg project 466789228 and the research project 562495345.

## 1 Introduction

It is usual for papers in the area of algorithms on strings to start with preprocessing the input string to support some standard queries. Gu, Farach, and Beigel [16] defined the following prefix-substring matching problem that consists in preprocessing a given string $s$ of length $n$ for the following queries: given $(i, j, k) \in \{0, \ldots, |s|\}^3$ with $1 \leq j \leq k$, representing a prefix $s[1 : i]$ and a substring $s[j : k]$ of $s$, find the longest prefix of $s$ that is a suffix of $s[1 : i]s[j : k]$. The original motivation behind such queries was maintaining a dynamic indexing data structure. Using some basic periodicity properties, namely the fact that all borders of a given string of length at most $n$ can be decomposed into $\mathcal{O}(\log n)$ arithmetic progressions, the authors were able to design a data structure that answers prefix-substring queries in $\mathcal{O}(\log n)$ time after $\mathcal{O}(n)$ time preprocessing. Their structure has found quite a few other applications, for example it has been used in to design an efficient algorithm for pattern matching in LZW and LZ compressed texts [3, 10] to obtain algorithms working in $\mathcal{O}(n \log m + m)$ and $\mathcal{O}(n \log^2(N/n) + m)$ time, respectively (where $n$ is the size of the compressed representation, $N$ is the length of the text, and $m$ is the length of the pattern).

For pattern matching in both LZW and LZ compressed texts, Gawrychowski designed faster algorithms working in $\mathcal{O}(n + m)$ and $\mathcal{O}(n \log(N/m) + m)$ time, respectively [14, 15]. The starting point in both results is an improved data structure for the related prefix-suffix matching problem, in which the query consists of $(i, j) \in \{0, \ldots, |s|\}^3$, representing a prefix $s[1 : i]$ and a suffix $s[j : n]$ of $s$, and asks for detecting an occurrence of the whole $s$ in $s[1 : i]s[j : n]$. A simple but useful observation is that such an occurrence corresponds to a


*long* border of either $s[1:i]$ or $s[j:n]$, where the long borders of $t[1:\ell]$ are $b \in \{\lceil \ell/2 \rceil, \ldots, \ell\}$ such that $t[1:b] = t[(\ell-b+1):\ell]$. By basic properties of periodicity, all long borders form a single arithmetic progression, which allows for answering such a query in constant time after $\mathcal{O}(n)$ time preprocessing. However, such a simple approach does not suffice to answer the more difficult prefix-substring queries in the same complexity, and both algorithms need to follow a more indirect route to achieve the improved time complexity: for LZW this requires amortizing the time complexity over the subsequent queries [14], and for LZ batching similar queries after an initial transformation of the input [15]. A later paper by Ganardi and Gawrychowski for pattern matching in SLP compressed texts [13] (which can be seen as a special case of LZ compression) avoids using prefix-substring queries altogether by, informally speaking, over-approximating the longest prefix with constantly many substrings of the pattern.

Pissis [19] observed that any data structure for answering longest common extension queries can be used to answer prefix-suffix queries. By plugging in the best known result, this results in a data structure with constant query time and $\mathcal{O}(n/\log_\sigma n)$ preprocessing time, for a string of length $n$ over an integer alphabet $\{1, \ldots, \sigma\}$. He also observed that such queries are very useful in pattern matching on node-labeled graphs, see e.g. [2, 4, 5, 8]).

Thus, the complexity of prefix-suffix queries is well understood. This is certainly not the case for prefix-substring queries, though, with the best bound being $\mathcal{O}(\log n)$ query after $\mathcal{O}(n)$ preprocessing [16]. While in the applications related to compressed pattern matching it was possible to avoid the extra factor of $\log n$ by exploiting the additional structure of the queries, this has led to certain technical complications that would best be avoided. Further, for the applications related to pattern matching on node-labeled graphs such additional structure of the queries is less clear. It would hence be desirable to design a structure with $\mathcal{O}(1)$ query time after linear time preprocessing. It is a priori not clear if such a structure exists: it may be the case that, say, prefix-substring matching is as difficult as predecessor search, leading to a lower bound of $\Omega(\log\log n)$ for structures of size $\tilde{\mathcal{O}}(n)$.

We show how to answer prefix-substring queries in constant time with a structure of size $\mathcal{O}(n)$ that can be constructed in $\mathcal{O}(n)$ time for a string of length $n$ over a polynomial alphabet (which is a standard assumption in the area; otherwise, the construction time increases to $\mathcal{O}(n \log n)$ due to sorting). We emphasise that, as observed in prior work, this allows for finding the longest prefix of $s$ that is a suffix of $s[i:\ell]s[j:k]$, for $(i, \ell, j, k) \in \{0, \ldots, |s|\}^4$: we simply retrieve first the answer $h$ to the prefix-substring query $(0, i, \ell)$, and then retrieve the answer to the query $(h, j, k)$.

Our paper is structured as follows: we introduce the basic notions and data structures in Section 2, then give an overview of the considered problem and of our approaches in Section 3. In particular, we present a sequence of solutions for the problem, starting with simple ones (to set a baseline) and continuing with more involved and efficient ones (which, on the one hand, highlight potential obstacles one may meet when using certain techniques, and, on the other hand, provide us with deeper insights in the combinatorics of the problem), and conclude with the optimal solution. The technical details are presented in the rest of the paper (and in the appendix, for space reasons).

## 2 Preliminaries

Let $\mathbb{N}$ be the set of strictly positive integer numbers and, for $n \in \mathbb{N}$, let $[n] := \{1, \ldots, n\}$. For an alphabet $\Sigma$, a string $s$ of length $n$ (over $\Sigma$) is a concatenation of characters $s[1]s[2]\cdots s[n]$ with $s[i] \in \Sigma$ for $i \in [n]$. For $i, j \in [n]$, define $s[i:j] := s[i]\cdots s[j]$ for $i \leq j$ and $s[i:j] := \varepsilon$ otherwise (where $\varepsilon$ is the unique string of length 0). We call $s[i:j]$ a substring of $s$ and

define $\mathsf{Substr}(s) := \{s[i:j] \mid i, j \in [|s|]\}$ to be the set of all substrings of $s$. As a shorthand, we write $s[i:]$ and $s[:j]$ instead of $s[i:|s|]$ and $s[1:j]$ respectively, where $|s|$ denotes the length of $s$. Furthermore, we define $s^R := s[|s|]s[|s|-1]\cdots s[1]$ to be the mirror image of $s$ and, for $m \in \mathbb{N}$, let $s^m$ be the string obtained by repeating $s$ exactly $m$ times.

**Borders and periodicity.** For strings $r$ and $s$ with $|r| \leq |s|$ we say that $r$ is a prefix (respectively, suffix) of $s$, if and only if $r = s[:|r|]$ (respectively, $r = s[|s|-|r|+1:]$); $r$ is a border of $s$, if and only if it is both a prefix and a suffix of $s$. We define $\mathsf{Pref}(s) := \{\varepsilon\} \cup \{s[:i] \mid i \in [|s|]\}$, $\mathsf{Suff}(s) := \{\varepsilon\} \cup \{s[i:] \mid i \in [|s|]\}$ and $\mathsf{Bord}(s) := \mathsf{Pref}(s) \cap \mathsf{Suff}(s)$ as the sets of prefixes, suffixes and borders of $s$ respectively. For $y \in [|s|]$, we say that the prefix $s[:y]$ is the prior of the suffix $s[y+1:]$ and that the suffix $s[y+1:]$ is the posterior of the prefix $s[:y]$.

Given some $p < |s|$, we say that $s$ is $p$-periodic, if and only if $s[i] = s[i+p]$ for all $i \in [|s|-p]$. Note that this is equivalent to the condition $s[:|s|-p] = s[1+p:]$, meaning that $s$ is $p$-periodic if and only if $s$ has a border of length $|s|-p$. We say that the period $\mathrm{per}(s)$ of $s$ is the smallest value of $p$ such that $s$ is $p$-periodic or $\infty$ if there is no such value. In the latter case, we also say that $s$ is aperiodic. The following lemma is well-known [18].

▶ **Lemma 1** (Periodicity lemma). *If a string $s$ of length $n$ is both $p$-periodic and $q$-periodic and $p+q \leq n$, then $s$ is also $\gcd(p,q)$-periodic.*

Using this lemma, we can derive the following properties about borders and their periods. The proofs of these three results are given in Appendix A, for completeness.

▶ **Lemma 2.** *Let $s$ be a string of length $n$ with $p = \mathrm{per}(s)$ and let $m \in [n-1]$ such that $s[:m]$ is a border of $s$ with $\mathrm{per}(s[:m]) = q \neq p$. Then $p \geq \frac{1}{2}m$ and $n \geq \frac{3}{2}m$.*

▶ **Lemma 3.** *Let $s$ be a string of length $n$. Then $|\{\mathrm{per}(r) \mid r \in \mathsf{Bord}(s)\}| \in \mathcal{O}(\log n)$.*

Let $\mathsf{Bord}_p(s) := \{r \mid r \in \mathsf{Bord}(s), \mathrm{per}(r) = p\}$ denote the set of borders of $s$ with period $p$.

▶ **Lemma 4.** *For string $s$, with $|s| = n$, and integer $p \in [n-1]$, the elements of $\{|r| \mid r \in \mathsf{Bord}_p(s)\}$ form an arithmetic progression with step $p$. For $p = \infty$, $\mathsf{Bord}_p(s)$ contains at most one non-empty string.*

**Computational model.** We assume the standard unit-cost word RAM with logarithmic word-size $\omega$ (so each memory word consists of $\omega \geq \log n$ bits). Standard arithmetical and bitwise operations (and indirect addressing) on such words are assumed to take $\mathcal{O}(1)$ time [12]. We assume that our input is over a polynomial alphabet, i.e., $|\Sigma| \in n^{\mathcal{O}(1)}$, where $n$ is the length of the input string. We refer to a structure that can be stored in $\mathcal{O}(n)$ memory words as linear-space.

**Longest common prefix.** For string $s$, with $|s| = n$, and $i, j \in [n]$, let $\mathsf{LCP}_s(i,j) := \max\{|r| \mid r \in \mathsf{Pref}(s[i:]) \cap \mathsf{Pref}(s[j:])\}$ be the length of the longest common prefix of $s[i:]$ and $s[j:]$.

▶ **Lemma 5** ([17]). *Given a string $s$ of length $n$, we can construct a linear-space data structure in $\mathcal{O}(n)$ time allowing us to answer $\mathsf{LCP}_s(i,j)$ queries for any $i, j \in [n]$ in constant time.*

Using this data structure, we can also find longest periodic prefixes. Given some $p \in [n-1]$, we can calculate $\ell = \mathsf{LCP}_s(i, i+p)$, if $\ell > 0$ we have $s[i:i+\ell-1] = s[i+p:i+\ell+p-1]$ and $s[i+\ell] \neq s[i+p+\ell]$, meaning that $s[i:i+\ell+p-1]$ is the longest $p$-periodic prefix of $s[i:]$. By constructing the data structure for $s^R$, we can similarly determine longest common suffixes and longest periodic suffixes.

**Suffix tree.** For some string $s$ of length $n$, the suffix tree of $s$ is a compressed trie $T = (V, E)$, i.e., a tree with the following properties:

- Each edge is labeled with some non-empty string over the alphabet $\Sigma \cup \$$ with $\$ \notin \Sigma$ being a special character.
- The labels of any two edges starting at the same node do not share a common prefix.
- Each inner node has at least two children.

Let $v \in V$ be some node and let $s_1, \ldots, s_m$ be the labels of the path from the root to $v$. We now define $\mathsf{Strings}(v) := \{s_1 \cdots s_{m-1} r \mid r \in \mathsf{Pref}(s_m) \setminus \{\varepsilon\}\}$ as the set of strings corresponding to $v$, for the root node $v_r$ it is $\mathsf{Strings}(v_r) = \{\varepsilon\}$. The defining property of the suffix tree is now that $\bigcup_{v \in V} \mathsf{Strings}(v) = \mathsf{Substr}(s\$)$. We use the following results for the suffix tree:

▶ **Theorem 6** ([9]). *The suffix tree of a string $s$, with $|s| = n$, is constructed in $\mathcal{O}(n)$ time.*

For any $i, j \in [n]$, there is exactly one node $v$ in the suffix tree of $s$ such that $s[i:j] \in \mathsf{Strings}(v)$. We write $v = \mathsf{SuffNode}_s(i, j)$ and call $v$ the node corresponding to the substring $s[i:j]$.

▶ **Lemma 7** ([6]). *Given a string $s$ of length $n$, we can construct a linear-space data structure in $\mathcal{O}(n)$ time allowing us to find $\mathsf{SuffNode}_s(i, j)$ for any $i, j \in [n]$ in constant time.*

We also use the following standard observation:

▶ **Observation 8.** *Let $s$ be a string, let $u$ and $v$ be nodes in the suffix tree of $s$ and let $s_u \in \mathsf{Strings}(u)$ and $s_v \in \mathsf{Strings}(v)$ with $|s_u| \leq |s_v|$ (if $u \neq v$ this condition can be omitted). Then $s_u$ is a prefix of $s_v$ if and only if $u$ is an ancestor of $v$.*

In this paper, we take the relations "is ancestor of" and "is descendant of" to be reflexive, i.e., a node is both its own ancestor and its own descendant, unless explicitly noted otherwise.

The following result is shown in Appendix A.

▶ **Lemma 9.** *Let $s$ be a string, let $r \in \mathsf{Suff}(s)$ be a suffix of $s$ and $v$ be the node in the suffix tree of $s$ corresponding to $r$. Then $|r| \geq |r'|$ for all $r' \in \mathsf{Strings}(v) \cap \mathsf{Substr}(s)$ (this excludes all strings ending with \$) and $r$ is the only suffix of $s$ corresponding to $v$.*

**Border tree.** For some string $s$ of length $n$, the border tree of $s$ is defined as follows. The nodes of the tree correspond one-to-one to the prefixes of $s$ and for any node $v$ corresponding to some prefix $r \neq \varepsilon$, the parent of $v$ is the node that corresponds to the longest border of $r$. This is well defined, because $\mathsf{Bord}(r)$ always contains at least $\varepsilon$ and any border of $r$ is a prefix of $r$ and therefore also a prefix of $s$. The node corresponding to $\varepsilon$ has no parent and is the root node of the tree. The border tree can now be compressed by grouping the borders by their periodicities (on each path). From Lemma 4 it follows that we can encode each group of borders as an arithmetic progression using constant space. The compressed border tree then has depth $\mathcal{O}(\log n)$ as seen in Lemma 3.

▶ **Lemma 10** ([16]). *Given a string $s$ of length $n$, we can construct both the uncompressed and compressed border tree of $s$ in $\mathcal{O}(n)$ time.*

From the definition of the border tree, it follows that for some prefix $r \in \mathsf{Pref}(s)$, the ancestors of the node corresponding to $r$ in the border tree correspond exactly to the borders of $r$. For the compressed version of the border tree, this is not true anymore. Consider some section on some path, where all nodes correspond to prefixes with the same periodicity. During compression, this section is replaced with a single node. In this process, all nodes that were children of any node in the section become children of the new compressed node, even

though they have different sets of ancestors. In particular, from the arithmetic progression of prefixes that is represented by their new parent, only the prefixes up to some length are actual borders. We can determine this length, for some node $v$, by taking the smallest prefix $r$ associated with $v$, the longest border of which has to have length $|r| - \mathrm{per}(r)$, we can then trim the arithmetic progression corresponding to the parent of $v$, removing all prefixes longer than this value.

**Induced nodes and partner query.** Given two trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$, we can relate these trees to each other by using a common set of labels. Hereby, each label gets assigned to one node from each tree. We then say that two nodes $v_1 \in V_1$ and $v_2 \in V_2$ are induced, if and only if there are descendants $u_1$ of $v_1$ and $u_2$ of $v_2$ that share the same label. The partner of $v_1$ with respect to $v_2$, denoted $\mathsf{partner}(v_1/v_2)$, is the lowest ancestor $v_2'$ of $v_2$, such that $v_1$ and $v_2'$ are induced.

▶ **Lemma 11** ([1]). *Given trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$, we can construct data structures allowing us to compute $\mathsf{partner}(v_1/v_2)$ for any $(v_1, v_2) \in V_1 \times V_2$ in*

- *$\mathcal{O}(\log^\delta n)$ time for any constant $\delta > 0$, for a linear-space data structure constructed in $\mathcal{O}(n)$ time.*
- *$\mathcal{O}(\log \log n)$ time, for a data structure constructed in $\mathcal{O}(n \log \log n)$ time and space.*

**Micro-macro decomposition.** Given a tree, we can decompose it, as follows. First, we select a subset of the nodes from the tree, including the root node. We call these nodes special. For each special node $u$, we partition its children into non-empty sets. Each set of siblings corresponds to one fragment of the decomposition, containing the siblings, their parent $u$ (which becomes the root node of the fragment), and all their descendants which have $u$ as their lowest special ancestor. Since the root node is special, each node has a special ancestor and each non-special node belongs to exactly one fragment. Therefore, we obtain a set of micro-trees (the fragments) and a macro-tree that consists only of the special nodes, arranged as to preserve the ancestor/descendant relations between each pair of node. It is possible to construct such a decomposition for any desired fragment size efficiently (even though this is well known – see [11], for instance – we give a proof in Appendix A, fitting to our setting, for completeness):

▶ **Lemma 12.** *Given a tree of size $n$ and some $z \in [n]$, it is possible to find a micro-macro decomposition that has $\mathcal{O}(n/z)$ fragments of size at most $z$ in $\mathcal{O}(n)$ time.*

**Other data structures.** We also use the following well known data structures.

▶ **Lemma 13** (RMQ [7]). *For $n \in \mathbb{N}$ and list of numbers $i_1, \ldots, i_n \in \mathbb{N}$, we can construct a linear-space data structure in $\mathcal{O}(n)$ time allowing us to retrieve $\max\{i_k \mid k \in [a, b]\}$ for any $a, b \in [n], a \leq b$ in $\mathcal{O}(1)$ time.*

▶ **Lemma 14** (Fusion tree [12]). *Given $n \in \mathbb{N}$ and sets $S_1, S_2, \ldots, S_k \subset [n]$ such that, for every $i$, $|S_i| \in \mathcal{O}(\log n)$, we can construct in $\mathcal{O}(n + \sum_i |S_i|)$ time and space a data structure allowing us to determine the predecessor and successor of $k$ in any $S_i$, i.e., $\max\{m \in S_i \mid m \leq k\}$ and $\min\{m \in S_i \mid m \geq k\}$, for any $k \in [n]$ in $\mathcal{O}(1)$ time.*

## 3 Problem definition and overview of solution

▶ **Definition 15.** *For string $s$, with $|s| = n$, we call $(i, j, k) \in [n]^3$, with $j \leq k$, a prefix-substring query and denote by*

$$\mathit{PrefSubstrQueries}(s) := \{(i, j, k) \in [n]^3 \mid j \leq k\}$$

*the set of all queries*[1]*. For $(i, j, k) \in \mathit{PrefSubstrQueries}(s)$, we define*

$$\mathit{PrefSubstrMatches}_s(i, j, k) := \mathit{Pref}(s) \cap \mathit{Suff}(s[: i]s[j : k]);$$

*an element $r \in \mathit{PrefSubstrMatches}_s(i, j, k)$ is a match for the query $(i, j, k)$. Furthermore, we write*

$$\mathit{PrefSubstr}_s(i, j, k) := \max\{|r| \mid r \in \mathit{PrefSubstrMatches}_s(i, j, k)\}$$

*to refer to the length of the longest match. This is well-defined, since $\varepsilon$ is always a match.*

The problem PrefixSubstringQueries is now defined as follows: preprocess an input string $s$ such that we can answer prefix-substring queries for $s$. For each given query $(i, j, k)$, we need to return $\mathsf{PrefSubstr}_s(i, j, k)$ (in an online fashion, before reading the next query).

Note that the answer to PrefixSubstringQueries can be used to efficiently reconstruct the set of all matches, using the following property (proven in Appendix B):

▶ **Lemma 16.** *For any string $s$ and any query $(i, j, k) \in \mathit{PrefSubstrQueries}(s)$, the answer $x = \mathit{PrefSubstr}_s(i, j, k)$ satisfies $\mathit{PrefSubstrMatches}_s(i, j, k) = \mathit{Bord}(s[: x])$.*

Consider now some element $s[: x]$ of $\mathsf{PrefSubstrMatches}_s(i, j, k)$. This is a suffix of $s[: i]s[j : k]$. If $|s[: x]| \leq |s[j : k]|$, then $s[: x]$ is a suffix of $s[j : k]$ and consequently, $s[: x] \in \mathsf{PrefSubstrMatches}_s(i, j, k)$ for all $i \in [|s|]$. Otherwise, it "crosses" over into $s[: i]$, so it is not independent of $i$. Let us formalize this notion:

▶ **Definition 17.** *Let $s$ be a string and let $(i, j, k) \in \mathit{PrefSubstrQueries}(s)$ be a query. Then we write*

$$\mathit{PrefSubstrMatches}^{c}_s(i, j, k) := \{u \in \mathit{PrefSubstrMatches}_s(i, j, k) \mid |u| > |s[j : k]|\}$$

*to denote the set of crossing matches and*

$$\mathit{PrefSubstrMatches}^{nc}_s(i, j, k) := \mathit{PrefSubstrMatches}_s(i, j, k) \setminus \mathit{PrefSubstrMatches}^{c}_s(i, j, k)$$

*to denote the set of non-crossing matches.*

*We also define $\mathit{PrefSubstr}^{c}_s(i, j, k)$ and $\mathit{PrefSubstr}^{nc}_s(i, j, k)$ to be the lengths of the longest elements of $\mathit{PrefSubstrMatches}^{c}_s(i, j, k)$ and $\mathit{PrefSubstrMatches}^{nc}_s(i, j, k)$ respectively. If the former is empty, we set $\mathit{PrefSubstr}^{c}_s(i, j, k) = 0$.*

We will solve PrefixSubstringQueries by separately finding the length of the longest crossing and the length of the longest non-crossing match, it is then $\mathsf{PrefSubstr}_s(i, j, k) = \max(\mathsf{PrefSubstr}^{nc}_s(i, j, k), \mathsf{PrefSubstr}^{c}_s(i, j, k))$. This is outlined in the following paragraphs.

[1] To focus on non-trivial prefix-substring queries, we exclude the case $i = 0$ in this definition. This case is discussed at the end of Section 3.

**Non-crossing matches.** Clearly, the non-crossing matches are exactly the borders of $s[:k]$ of length at most $|s[j:k]|$. Thus, determining the length of the longest non-crossing match, $\mathsf{PrefSubstr}_s^{nc}(i,j,k)$, is equivalent to determining the length of the longest such border of $s[:k]$. Each such match $s[:x]$ satisfies that $(s[:x])^R$ is a prefix of $(s[j:k])^R$ and a suffix of $s^R$, so the longest match can then be found in the suffix tree of $s^R$ as the lowest ancestor of the node corresponding to $(s[j:k])^R$, that corresponds to a suffix. Determining this is possible in $\mathcal{O}(1)$ time.

**Crossing matches: simple solutions.** For crossing match $s[:x] \in \mathsf{PrefSubstrMatches}_s^c(i,j,k)$, we can split the match at $y = x - |s[j:k]|$, such that $s[:y]$ is a suffix (and as such, a border) of $s[:i]$ and $s[y+1:]$ has prefix $s[j:k]$. We then say that $s[:y]$ *induces* the match $s[:x]$. Like this, PrefixSubstringQueries can be rephrased as: "Find the longest border $s[:y]$ of $s[:i]$ such that its posterior starts with $s[j:k]$". By checking, for each border $s[:y]$ of $s[:i]$ individually, whether $s[y+1:]$ has prefix $s[j:k]$, we obtain a naive algorithm that answers queries in $\mathcal{O}(n)$ time, after $\mathcal{O}(n)$ time preprocessing. This can be optimized by exploiting the structure of the borders, yielding an algorithm with $\mathcal{O}(\log n)$ query time, after $\mathcal{O}(n)$ preprocessing.

**Crossing matches: reduction to partner query.** Sticking with our approach to find the longest $y \in [n]$, such that $s[:y]$ is a border of $s[:i]$ and $s[y+1:]$ has prefix $s[j:k]$, we observe that this can be expressed as a problem on the border and suffix tree, as follows. Labeling the node $v_y$ corresponding to $s[:y]$ in the border tree and the node $\mathsf{SuffNode}_s(y+1,n)$ in the suffix tree with label $y$ for each $y \in [n]$, we then want to find the largest label $y$ such that the node labeled with $y$ in the border tree is an ancestor of the node corresponding to $s[:i]$ and the node labeled with $y$ in the suffix tree is a descendant of $\mathsf{SuffNode}_s(j,k)$. This is quite similar to solving $\mathsf{partner}(\mathsf{SuffNode}_s(j,k)/v_i))$, only that the result $v_{i'}$ of this query is only guaranteed to have a descendant with a label $y'$ that also occurs in the descendants of $\mathsf{SuffNode}_s(j,k)$, instead of having such a label itself. Still, it turns out that the matches are exactly the borders of $s[:y'+t]$ of length at most $i'+t$, meaning that the longest match can be found similarly as the non-crossing matches. Apart from solving the partner query, this reduction only requires linear preprocessing and constant query time.

**Crossing matches: final algorithm.** To obtain a constant time solution for crossing matches, we split the borders $s[:y]$ of $s[:i]$ into three groups:

- For $y \le t := |s[j:k]|$, the set of such $y$ satisfying that the posterior $s[y+1:]$ has prefix $s[j:k]$ is an arithmetic progression with some step $p$. We can find the longest such $s[:y]$ that induces a match by comparing the lengths of the longest $p$-periodic suffixes of $s[:i]s[j:k]$ and $s[:y+t]$ for all such $y$, using the fact that each $s[:y+t]$ has a different such length.
- For $y > t$ and $\mathrm{per}(s[:y]) < t$, we observe that the borders $s[:y]$ of $s[:i]$ with this condition only have a constant number of different periodicities. For each periodicity $p$, we can in constant time determine the longest induced match, using the same method with which we optimized the initial naive algorithm.
- For $\mathrm{per}(s[:y]) \ge t$, we see that almost all (all, but the longest) prefixes $s[:y]$ with some shared periodicity $p = \mathrm{per}(s[:y]) \ge t$ share a common prefix of length $t$ and therefore, for each such periodicity $p$, we only need to consider two candidates for inducing the longest match and in total, we consider $\mathcal{O}(\log n)$ candidates. If we sort them in the preorder of their respective posteriors in the suffix tree, all candidate prefixes that induce a match

lie in some interval of this list, corresponding to the preorder interval representing the subtree rooted in $\mathsf{SuffNode}_s(j,k)$. We can then find this interval using fusion trees and the longest match in the interval using an $\mathsf{RMQ}$ query.

While this enables us to answer queries in constant time, the last case requires maintaining $\mathcal{O}(\log n)$-sized data structures for each prefix of $s$. To avoid this, we perform a micro-macro decomposition of the border tree with fragment size $\lfloor \log n \rfloor$ and then only build these data structures for the prefixes corresponding to the $\mathcal{O}(n/\log n)$ special nodes. For some query $(i,j,k) \in \mathsf{PrefSubstrQueries}(s)$, we find the fragment containing the node $v_i$ corresponding to $s[:i]$ and its special root $u_i$. We then check all borders corresponding to ancestors of $u_i$, using the method from above. The remaining borders of $s[:i]$ correspond to ancestors of $v_i$ in that fragment. The subset of these borders that induce a match can be expressed as an interval in 3D space, with the dimensions being the preorder and postorder indices of the corresponding nodes in the fragment and the preorder index of the posterior in the suffix tree. Since there are at most $\mathcal{O}(\log n)$ such borders, we can encode sets of borders in a constant number of machine words and use bit operations to operate on the sets. This allows for an $\mathcal{O}(1)$ time solution, using $\mathcal{O}(\log n)$ preprocessing per fragment, i.e., $\mathcal{O}(n)$ preprocessing time in total.

Combining this with Lemma 20, addressing non-crossing matches, we obtain the following.

▶ **Theorem 18.** *For string $s$, with $|s| = n$, we can construct a linear-space data structure in $\mathcal{O}(n)$ time allowing us to determine $\mathsf{PrefSubstr}_s(i,j,k)$ for any $(i,j,k) \in \mathsf{PrefSubstrQueries}(s)$ in $\mathcal{O}(1)$ time.*

Note that we only consider here queries $(i,j,k)$ with $i \geq 1$. In fact, our algorithm addressing non-crossing matches can be canonically used to answer queries $(0,j,k)$, so the statement of Theorem 18 holds also for all queries of the form $(0,j,k)$, with $1 \leq j \leq k \leq n$.

## 4 Non-Crossing Matches

Let us start with the non-crossing matches, which, as we will see, are the easier of the two kinds of matches. As hinted in the overview, it is

$$\begin{aligned}
\mathsf{PrefSubstrMatches}_s^{nc}(i,j,k) &= \mathsf{PrefSubstrMatches}_s(i,j,k) \setminus \mathsf{PrefSubstrMatches}_s^{c}(i,j,k)\\
&= \{u \in \mathsf{PrefSubstrMatches}_s(i,j,k) \mid |u| \leq |s[j:k]|\}\\
&= \{u \in \mathsf{Pref}(s) \cap \mathsf{Suff}(s[:i]s[j:k]) \mid |u| \leq |s[j:k]|\}\\
&= \{u \in \mathsf{Pref}(s) \cap \mathsf{Suff}(s[j:k]) \mid |u| \leq |s[j:k]|\}\\
&= \{u \in \mathsf{Pref}(s[:k]) \cap \mathsf{Suff}(s[:k]) \mid |u| \leq |s[j:k]|\}\\
&= \{u \in \mathsf{Bord}(s[:k]) \mid |u| \leq |s[j:k]|\}
\end{aligned}$$

Therefore, the longest non-crossing match will be the longest border of $s[:k]$ of length at most $|s[j:k]|$. To find this, we use the following lemma, shown in Appendix D:

▶ **Lemma 19.** *Given string $s$, with $|s| = n$, we can construct a linear-space data structure in $\mathcal{O}(n)$ time allowing us to determine $\max\{|r| \mid r \in \mathsf{Bord}(s[:k]), |r| \leq \ell\}$ for any $k, \ell \in [n]$ in $\mathcal{O}(1)$ time.*

**Proof sketch.** We construct the suffix tree of $s^R$. The desired solution $x$ satisfies that $(s[:x])^R$ is the longest suffix of $s^R$ that is also a prefix of $(s[j:k])^R$. In terms of the suffix tree, determining this corresponds to finding the lowest ancestor of $\mathsf{SuffNode}_{s^R}(n-k+1, n-j+1)$ that corresponds to a suffix of $s$; this ancestor can be precomputed for every node. ◀

By setting $\ell = |s[j:k]| = k - j + 1$, we obtain the following lemma.

▶ **Lemma 20.** *For string $s$, with $|s| = n$, we can construct a linear-space data structure in $\mathcal{O}(n)$ time allowing us to determine $\mathsf{PrefSubstr}_s^{nc}(i,j,k)$ for any $(i,j,k) \in \mathsf{PrefSubstrQueries}(s)$ in $\mathcal{O}(1)$ time.*

## 5 Crossing Matches

Now let us turn to crossing matches, and begin with some simple solutions, which set a baseline for this problem. For a crossing match $s[:x] \in \mathsf{PrefSubstrMatches}_s^c(i,j,k)$ we can split the match at $y = x - |s[j:k]|$, such that $s[:y]$ is a suffix (and as such, a border) of $s[:i]$ and $s[y+1:]$ has prefix $s[j:k]$. As already defined in the overview, we then say that $s[:y]$ *induces* the match $s[:x]$ and try to find the longest border $s[:y]$ of $s[:i]$ such that its posterior starts with $s[j:k]$. Checking each border of $s[:y]$ individually yields a naive algorithm, answering queries in $\mathcal{O}(n)$ time after $\mathcal{O}(n)$ preprocessing (see Lemma 32 in Appendix).

We saw in Lemma 3 and Lemma 4 that $\mathsf{Bord}(s[:i])$ can be partitioned into $\mathcal{O}(\log n)$ arithmetic progressions of borders. It turns out that we can check each of these progressions of borders in $\mathcal{O}(1)$ time (see Appendix E for the proof).

▶ **Lemma 21.** *Let $s$ be a string of length $n$. We can construct a linear-space data structure in $\mathcal{O}(n)$ time allowing us to find the largest $y$ satisfying $s[:y] \in \mathsf{Bord}_p(s[:i])$ and $s[j:k] \in \mathsf{Pref}(s[y+1:])$ for any query $(i,j,k) \in \mathsf{PrefSubstrQueries}(s)$ and periodicity $p \in [i-1] \cup \{\infty\}$ in constant time, provided we are given (an $\mathcal{O}(1)$ size representation of) $\mathsf{Bord}_p(s[:i])$ as well.*

Now it suffices to just check all groups of borders to find the solution (see Appendix E):

▶ **Lemma 22.** *For string $s$, with $|s| = n$, we can construct a linear-space data structure in $\mathcal{O}(n)$ time allowing us to retrieve $\mathsf{PrefSubstr}_s^c(i,j,k)$ for $(i,j,k) \in \mathsf{PrefSubstrQueries}(s)$ in $\mathcal{O}(\log n)$ time.*

### 5.1 Reduction to Partner Query

We can now focus on more efficient (yet not optimal) solutions. As derived previously, the problem can be phrased as finding the longest border $s[:y]$ of $s[:i]$, such that the posterior $s[y+1:]$ has $s[j:k]$ as a prefix. We can frame this problem in terms of the uncompressed border and suffix tree as follows: Consider the border and suffix trees for $s$. For any $t \in [n]$, let $v_t$ be the node corresponding to $s[:t]$ and label both $v_t$ in the border tree and $\mathsf{SuffNode}_s(t+1,n)$ in the suffix tree with label $t$. The necessary conditions on $s[:y]$ can now be expressed in terms of the suffix and border tree:

- $s[:y]$ is a suffix/border of $s[:i]$, if and only if the node labeled with $y$ in the border tree is an ancestor of the node corresponding to $v_i$.
- $s[y+1:]$ must have $s[j:k]$ as a prefix, i.e., the node labeled with $y$ in the suffix tree must be a descendant of $\mathsf{SuffNode}_s(j,k)$.

Therefore, the largest such $y$ can be determined by finding the lowest ancestor of $v_i$ in the border tree, labeled with some label $y$, such that there is also a descendant of $\mathsf{SuffNode}_s(j,k)$ in the suffix tree with the same label. This can be reduced to a partner query.

▶ **Lemma 23.** *If we can construct a data structure in time $P$ allowing to answer any partner query on two trees of size $\mathcal{O}(n)$ for some $n \in \mathbb{N}$ in time $Q$, then, given a string $s$ of length $n$, we can create a data structure in time $P + \mathcal{O}(n)$ allowing us to determine $\mathsf{PrefSubstr}_s^c(i,j,k)$ for any query $(i,j,k) \in \mathsf{PrefSubstrQueries}(s)$ in $Q + \mathcal{O}(1)$ time.*

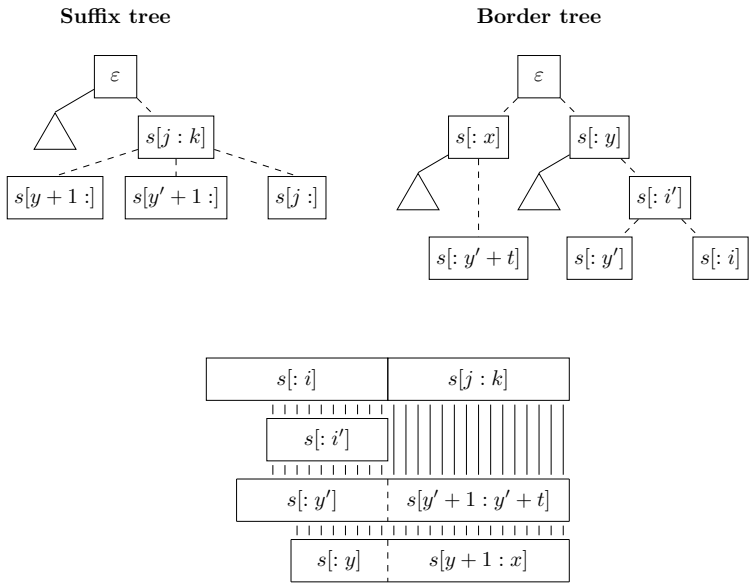


**Figure 1** Sketch for the proof of Lemma 23. In the top part, we have the suffix tree and border tree, reduced to the relevant nodes, showing the ancestor/descendant relations between them (by dashed edges). In the bottom part, we see how these relations translate into prefix/suffix/border relations between the respective substrings, with the vertical lines representing equality of substrings.

**Proof.** See Figure 1 for an illustration of the proof.

During preprocessing, we construct the border and suffix tree, both labeled as described above, and data structures to answer partner queries on the trees and $\mathsf{SuffNode}_s$ queries. We also construct the data structure necessary for Lemma 19.

When given a query $(i,j,k) \in \mathsf{PrefSubstrQueries}(s)$, we initially determine the node $\mathsf{partner}(\mathsf{SuffNode}_s(j,k)/v_i)$ in the border tree. The resulting node corresponds to some prefix $s[:i']$. We know that $u_{j,k} := \mathsf{SuffNode}_s(j,k)$ and $v_{i'}$ are induced, i.e, we can find a label $y'$ such that there are descendants of each $u_{j,k}$ and $v_{i'}$ with this label. More precisely, the node $u_{y'+1} := \mathsf{SuffNode}_s(y'+1,n)$ in the suffix tree and the node $v_{y'}$ in the border tree are descendants of $u_{j,k}$ and $v_{i'}$ respectively.

Consider now the label $y$, representing the unknown solution, and the corresponding nodes $\mathsf{SuffNode}_s(y+1,n)$ and $v_y$ in suffix and border tree, respectively. We know that $\mathsf{SuffNode}_s(y+1,n)$ must be a descendant of $u_{j,k}$, therefore, $u_{j,k}$ and $v_y$ are induced. Because the partner query finds the lowest induced ancestor, $v_y$ cannot be lower than $v_{i'}$ on the path from the root to $u_i$ and it must be $y \le i'$. Similarly, the solution $x = \mathsf{PrefSubstr}^c_s(i,j,k)$ must satisfy $x \le i'+t$ with $t = |s[j:k]|$.

Now, $v_{i'}$ is an ancestor of both $v_i$ and $v_{y'}$, i.e., $s[:i']$ is a suffix of both $s[:i]$ and $s[:y']$. Also, $u_{y'+1}$ is a descendant of $u_{j,k}$, i.e., $s[y'+1:]$ has $s[j:k]$ as prefix. As a consequence of these two facts, $s[:i]s[j:k]$ and $s[:y'+t] = s[:y']s[y'+1:y'+t]$ share a common suffix of length $i'+t$. But then, $s[:x]$ (and, by extension, any match) must be a suffix (and consequently, a border) of $s[:y'+t]$. In particular, $s[:x]$ is the longest border of $s[:y'+t]$ of length at most $i'+t$ and we can determine $x$ using Lemma 19. ◀

Using Lemma 11, we obtain:

▶ **Corollary 24.** *Given a string $s$, with $|s| = n$, we can construct data structures allowing us to retrieve $\mathsf{PrefSubstr}^c_s(i,j,k)$ for $(i,j,k) \in \mathsf{PrefSubstrQueries}(s)$ in*

- *$\mathcal{O}(\log^\delta n)$ time for $\delta > 0$, using a $\mathcal{O}(n)$-sized structure, or*
- *$\mathcal{O}(\log\log n)$ time, using a $\mathcal{O}(n \log\log n)$-sized structure.*

## 5.2 Final algorithm

We will now present an algorithm to answer queries in $\mathcal{O}(1)$ time. We already have Lemma 20 for finding the length of the longest non-crossing match in constant time, so we again focus on crossing matches. In fact, let us subdivide $\mathsf{PrefSubstrMatches}^c_s(i,j,k)$ further:

▶ **Definition 25.** *For string $s$, with $|s| = n$, $(i,j,k) \in \mathit{PrefSubstrQueries}(s)$, and $t := |s[j : k]| = k - j + 1$, we call a crossing match $r \in \mathit{PrefSubstrMatches}^c_s(i,j,k)$ short, if and only if $|r| \leq 2t$ and long, otherwise. We denote by*

$$\mathit{ShortPrefSubstrMatches}^c_s(i,j,k) := \{r \in \mathit{PrefSubstrMatches}^c_s(i,j,k) \mid |r| < 2t\}$$

*and*

$$\mathit{LongPrefSubstrMatches}^c_s(i,j,k) := \{r \in \mathit{PrefSubstrMatches}^c_s(i,j,k) \mid |r| \geq 2t\}$$

*the set of short and long crossing matches respectively.*

**Short crossing matches.** As before, each match $s[: x] \in \mathsf{ShortPrefSubstrMatches}^c_s(i,j,k)$ corresponds to some $s[: y]$ with $y = x - t < t$ that satisfies $s[: y] \in \mathsf{Bord}(s[: i])$ and $s[j : k] \in \mathsf{Pref}(s[y + 1 :])$. Previously, we have been checking, for each border $s[: y]$, whether its posterior has prefix $s[j : k]$, also making use of the structure that we found on borders. Now, we invert this procedure, we instead consider all suffixes $s[z :]$ with $z \leq t$ that have prefix $s[j : k]$ and then we check, for each of them, if their prior is a border of $s[: i]$. As it turns out, as for the borders of $s[: i]$, the candidate suffixes also follow a structure that can be used to efficiently find our desired solution.

▶ **Lemma 26.** *Let $s$ be a string and let $r \in \mathit{Substr}(s)$ be a substring of $s$. Then*

$$\{z \mid r \in \mathit{Pref}(s[z :]), z \leq |r|\}$$

*is an arithmetic progression and $r$ is $p$-periodic with $p$ being the step of the progression (if the progression has more than one element).*

**Proof.** Let $t := |r|$ and let $z_1 < \cdots < z_\ell$ be the elements from the set. If there is only one element, we are done, so we assume $\ell \geq 2$ now. For $k \in [\ell - 1]$, define $q_k := z_{k+1} - z_k \leq t - 1$. Now for any $k \in [\ell - 1]$,

$$r[q_k :] = s[z_k + q_k : z_k + t - 1] = s[z_{k+1} : z_{k+1} - q_k + t - 1] = r[: t - q_k]$$

is a border or $r$, therefore $r$ is $q_k$-periodic. Let $p := \mathrm{per}(r)$ with $p \leq q_{k'}$ for all $k' \in [\ell - 1]$. If $\ell = 2$, we have a progression with step $q_1$ and we are done. Otherwise, we can choose $k' \in [\ell - 1] \setminus \{k\}$, and we have $q_k + p \leq q_k + q_{k'} \leq \sum_{k=1}^{\ell-1} q_k = z_\ell - z_1 < |r|$, by Lemma 1 we see that $r$ is $\gcd(p, q_k)$-periodic which, due to minimality of $p$, implies that $p$ divides $q_k$. But now

$$s[z_k : z_k + q_k + t - 1] = s[z_k : z_k + q - 1]s[z_{k+1} : z_{k+1} + t - 1] = r[: q_k]r = (r[: p])^{\frac{q_k}{p}} r$$

is also $p$-periodic and $s[z_k + p : z_k + p + t - 1] = s[z_k : z_k + t - 1] = r$, i.e, $s[z_k + p :]$ also has prefix $r$ and it must be $q_k = p$. As this holds for all $k \in [\ell - 1]$, we are done. ◀

Now consider some node $v$ in the suffix tree of $s$ and some string $r \in \mathsf{Strings}(v) \cap \mathsf{Substr}(s)$ corresponding to that node. By Observation 8, the sets of suffixes starting with $r$ are exactly the suffixes that correspond to descendants of $v$. Note that this includes a potential suffix

corresponding to $v$ itself, since by Lemma 9, the suffix has length at least $|r|$. As we can see, this set is independent from which $r \in \mathsf{Strings}(v) \cap \mathsf{Substr}(s)$ we choose. We can make use of this fact, combined with the constant size representation that we have seen above, to efficiently precompute the candidate suffixes $s[z:]$.

▶ **Lemma 27.** *For string $s$, with $|s| = n$, we can construct a linear-space data structure in $\mathcal{O}(n)$ time allowing us to determine a constant size representation of $\{z \mid s[j:k] \in \mathsf{Pref}(s[z:]), z \leq |s[j:k]|\}$ for any $j, k \in [n], j \leq k$ in $\mathcal{O}(1)$ time.*

**Proof.** During preprocessing, we first construct a suffix tree for $s$ and some data structure allowing us to answer $\mathsf{SuffNode}_s$ queries in constant time. Now we precompute the arithmetic progression $\{z \mid r_v \in \mathsf{Pref}(s[z:]), z \leq |r_v|\}$ for each node $v$ of the suffix tree, where $r_v$ is the longest element of $\mathsf{Strings}(v) \cap \mathsf{Substr}(s)$.

This precomputation is done bottom-up. Any leaf node of $s$ corresponds to at most one suffix of $s$, so creating the progression is trivial for these nodes. For some inner node $v$ with $t := |r_v|$ and $c$ children, we can compute the set in time $\mathcal{O}(c)$, as follows. First, we copy the progressions of each child node, removing all elements $z > t$ in the process by pruning the length of the progression (note that the cutoff $t$ gets smaller as we move up the tree, so we never have to add any elements back). Then we merge the (disjoint) progressions of the child node. This can be done efficiently, since there will only be at most two non-empty progressions in total, and at most one with more than a single element.

We can see this by considering the resulting progression $\{z, z+p, \ldots, z+(\ell-1)p\}$ with some $z, p, \ell \in [n]$ and $p < t$ (we assume $\ell \geq 2$, otherwise the claim is obviously true). For $k \in \{0\} \cup [\ell - 2]$ we have

$$s[z+kp : z+kp+t] = s[z+kp : z+kp+t-1]s[z+kp+t] = r_v r_v[t-p+1]$$

since $s[z+kp+t]$ is the $t-p+1$-th character of $s[z+(k+1)p:]$. Therefore, all but the last suffix in the progression share a common prefix of length $t+1$ and this prefix belongs to some child of $v$. Consequently, the progression of this child contains all but (at most) one of the relevant suffixes.

Since each child has just a single parent, the total number of children (and similarly the required construction time) is $\mathcal{O}(n)$.

Now, when we receive a query $j, k \in [n], j \leq k$, we determine the node $v = \mathsf{SuffNode}_s(j,k)$ corresponding to $s[j:k]$ and then return the arithmetic progression for $v$ after we removed all elements larger than $|s[j:k]|$. ◀

It remains to show how to use the structure of the suffixes to quickly find the longest match:

▶ **Lemma 28.** *For string $s$, with $|s| = n$, we can construct a linear-space data structure in $\mathcal{O}(n)$ time allowing us to determine the length of the longest element of $\mathsf{ShortPrefSubstrMatches}^c_s(i,j,k)$ for any $(i,j,k) \in \mathsf{PrefSubstrQueries}(s)$ in $\mathcal{O}(1)$ time.*

**Proof.** See Figure 2 for an illustration of the proof.

In the preprocessing, we construct the data structure from Lemma 27, together with data structures for answering $\mathsf{LCP}$ queries on both $s$ and $s^R$ in $\mathcal{O}(1)$ time (see Lemma 5).

When we receive a query $(i,j,k) \in \mathsf{PrefSubstrQueries}(s)$, we use the data structure from the previous lemma to determine $\{z \mid s[j:k] \in \mathsf{Pref}(s[z:]), z \leq t\}$ for $t := |s[j:k]|$. We also remove from this progression all $z > i+1$, as the prior $s[:z-1]$ must be a border of $s[:i]$. Let $Z = \{z_0, z_0+p, \ldots, z_0+(\ell-1)p\}$ be the resulting progression. If $\ell \leq 2$, we just check each element individually. In particular, $s[:z-1]$ is a suffix of $s[:i]$, if and only if

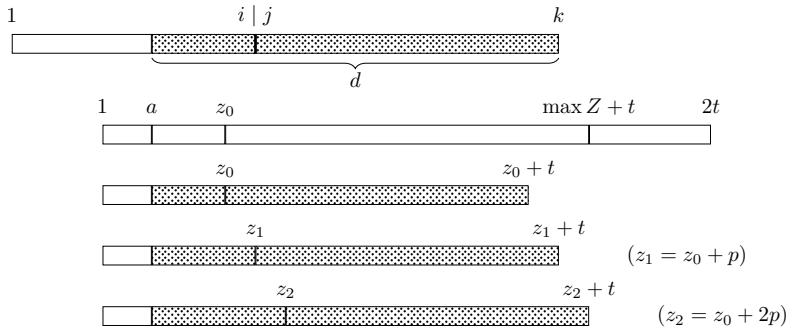


**Figure 2** Sketch of the proof of Lemma 28. In the top part, we have the string $s[:i]s[j:k]$ and its longest $p$-periodic suffix (marked with dots) of length $d$. This is aligned with the longest $p$-periodic suffix $s[a:z+t-1]$ of the $\ell = 3$ candidate matches $s[:z+t-1]$ for $z \in Z$ in the bottom. In this case, the suffix of the middle candidate has length $d$.

$\mathsf{LCP}_s(1, i-z+2) \geq z-1$. Otherwise, we know that $r := s[j:k]$ is $p$-periodic and the same must hold for $s[z_0 : z_0 + (\ell-1)p + t - 1] = (r[:p])^{\ell-1}r$. Similar to Lemma 21, where each posterior started with a different number of repetitions of some string, here we can see that the lengths of the longest $p$-periodic suffix of each potential match $s[:z-1+t]$ for $z \in Z$ are pairwise distinct. This helps us to find the longest match, by comparing these lengths to the length of the longest $p$-periodic suffix of $s[:i]s[j:k]$.

First, we determine, for all $z \in Z$, the longest $p$-periodic suffix of $s[:z-1+t]$. This is done by determining the longest $p$-periodic suffix of $s[:\max Z - 1 + t]$ using an $\mathsf{LCP}$ query on $s^R$, let $s[a:\max Z - 1 + t]$ be that suffix, $a \leq z_0$. Now for any $z \in Z$, the longest $p$-periodic suffix of $s[:z-1+t]$ is $s[a:z-1+t]$: obviously, it is $p$-periodic and $s[a-1:z-1+t]$ cannot be $p$-periodic, otherwise $s[a-1] = s[a+p-1]$ with $a+p-1 < z_0+t$ and $s[a-1:\max Z-1+t]$ would also have to be $p$-periodic, which contradicts our assumption about $a$.

Next, we determine the length $d$ of the longest $p$-periodic suffix of $s[:i]s[j:k]$, it is $d \geq t$. For this, we start by computing the length $d'$ of the longest common suffix of $s[:i]$ and $s[j:j+p-1]$ via an $\mathsf{LCP}$ query on $s^R$. If $d' < p$, then $s[i-d'] \neq s[j+p-d'-1]$ and it is $d = d' + t$. Otherwise, we find the length $d''$ of the longest $p$-periodic suffix of $s[:i]$. Since the last $p$ characters of $s[:i]$ and the first $p$ characters of $s[j:k]$ are the same, we can extend this suffix by $s[j:k]$ and obtain a $p$-periodic suffix of $s[:i][j:k]$. Consequently, $d = d'' + t$.

Comparing these lengths, we have to consider two different cases. First, we consider the case $a = 1$, i.e., for every $z \in Z$, the entire candidate $s[:z-1+t]$ is $p$-periodic. If $z-1+t \leq d$, then $s[:z-1+t]$ is exactly the $(z-1+t)$-length suffix of $s[:i]s[j:k]$, since both are $p$-periodic and share the last $t > p$ characters. If $z-1+t > d$, then it cannot be a suffix of $s[:i]s[j:k]$ by maximality of $d$. Therefore, the largest $z \in Z$ with $z-1+t \leq d$ corresponds to our solution. If $a > 1$, and the longest $p$-periodic suffixes of $s[:z-1+t]$ and $s[:i]s[j:k]$ do not match for some $z \in Z$, then the former cannot be a suffix of the latter. If they were, they would have to share a common $p$-periodic suffix of length $\min(z+t-a, d)+1 \leq z-1+t \leq i+t$, contradicting one of our assumptions about their longest $p$-periodic suffixes. Therefore, we find the $z \in Z$ with $z-1+t = d$ and, if it exists, check if $s[:z-1+t]$ is a suffix of $s[:i]s[j:k]$. ◀

**Long crossing matches.** Now, consider again the suffix tree and border tree. In Section 5.1 we have seen that for some $s[:y]$ to induce a match for the query $(i,j,k) \in \mathsf{PrefSubstrQueries}(s)$, the $\mathsf{SuffNode}_s(y+1, n)$ must be a descendant of $\mathsf{SuffNode}_s(j,k)$ and

the node corresponding to $s[: y]$ must be an ancestor of the node corresponding to $s[: i]$ in the border tree. We now label the nodes in the suffix tree in preorder, let $\ell_y$ be the label assigned to $\mathsf{SuffNode}_s(y+1, n)$. We also label, for each $y \in [n]$, the node corresponding to $s[: y]$ in the border tree with $\ell_y$.

Because we use preorder labels, the labels in any subtree of the suffix tree correspond to intervals. To find the longest match, we now need to determine the intersection of the interval corresponding to the subtree rooted in $\mathsf{SuffNode}_s(j, k)$ and all labels corresponding to ancestors of $s[: i]$ in the suffix tree. But the second set has size $\mathcal{O}(n)$, thus this is not directly possible to do in constant time. In the previous section, we saw how we can find short crossing matches. For longer matches we can make use of the following property:

▶ **Lemma 29.** *Let $s$ be a string of length $n$ and let $p \in [n-1]$ with $|\mathsf{Bord}_p(s)| \geq 2$. Let $\{y_0, y_0+p, \ldots, y_0+(\ell-1)p\} = \{|r| \mid r \in \mathsf{Bord}_p(s)\}$. Then the posteriors $s[y_0+1:], s[y_0+p+1:], \ldots, s[y_0+(\ell-2)p+1:]$ share a common prefix of length $p$.*

**Proof.** Due to Lemma 4, $s[: y_0 + (\ell-1)p]$ is $p$-periodic and

$$s[y_0+1 : y_0+p] = s[y_0+p+1 : y_0+2p] = \cdots = s[y_0+(\ell-2)p+1 : y_0+(\ell-1)]$$

is a common prefix of length $p$. ◀

This means that for $p \geq |s[j : k]|$ the posteriors of all but the longest element of $\mathsf{Bord}_p(s[: i])$ either all have prefix $s[j : k]$, or none of them do. Therefore, since we are only interested in the longest border inducing a match, it suffices to only consider the labels of the longest two borders of $\mathsf{Bord}_p(s[: i])$. This allows us to make the set of labels small enough to achieve a constant query time, albeit using extra time during preprocessing:

▶ **Lemma 30.** *For string $s$, with $|s| = n$, we can construct a data structure in $\mathcal{O}(n \log n)$ time and space allowing us to retrieve $\mathsf{PrefSubstr}^c_s(i, j, k)$ for any $(i, j, k) \in \mathsf{PrefSubstrQueries}(s)$ in $\mathcal{O}(1)$ time.*

**Proof.** During preprocessing, we construct the suffix tree and, since we want to exploit the periodicity of the borders, the compressed version of the border tree. We assign labels to both trees as described above, but we only assign label $\ell_y$ to a node in the border tree, if $s[: y]$ is one of the two largest borders corresponding to that node and maintain a set $Y$ containing all such values of $y$. We also store, for each node of the suffix tree, the boundaries of the interval of labels for the subtree rooted in said node. For each node $v$ in the compressed border tree, we create a fusion tree $L_v$ containing all the labels $\ell_y$ on the ancestors of the node and a fusion tree $P_v$ containing the period of each ancestor (and, for each period value, a reference to the corresponding node). We also create a list $Y_v := \{y \in [n] \mid \ell_y \in L_v\}$, sorted by the values of $\ell_y$ and a data structure for answering RMQ-queries on this list (see Lemma 13). Since each node only has two labels and the compressed border tree has depth $\mathcal{O}(\log n)$, both of these sets have size $\mathcal{O}(\log n)$ as well. Since we need to do this for all $\mathcal{O}(n)$ nodes, we require a total of $\mathcal{O}(n \log n)$ time. We also construct the data structures from Lemmas 21 and 28.

When we receive a query $(i, j, k) \in \mathsf{PrefSubstrQueries}(s)$, we start by determining $\mathsf{SuffNode}_s(j, k)$, the corresponding label interval $[\ell^-, \ell^+]$ and the node $v$ corresponding to $s[: i]$ in the border tree. By finding the successor $\ell_{y^-}$ of $\ell^-$ and the predecessor $\ell_{y^+}$ of $\ell^+$ in $L_v$, we see that $y^-$ and $y^+$ are exactly the boundaries of the interval in $Y_v$ that contains values of $y$ corresponding to a match, namely $\{y \in [n] \mid \ell_y \in L_v \cap [\ell^-, \ell^+]\}$. Using an RMQ-query, we can find the largest such $y$.

By now, we only have checked all $y$ with $\mathrm{per}(s[:y]) \geq p$. To check the rest, we start by finding the predecessor $p$ of $t := |s[j:k]|$ in $P_v$ and the corresponding node $v'$ (that has periodicity $p \leq t$). We then check this node using Lemma 21 in constant time and move up the tree, checking each node we encounter, until we reach a node corresponding to borders of length at most $t$. This requires only a constant number of steps. Due to Lemma 2, the borders corresponding to $v'$ have length at most $2p \leq 2t$ and, in each step, the length of borders decreases by a factor of at least $\frac{3}{2}$. The remaining borders can be checked using Lemma 28. ◀

**Reducing preprocessing time.** While we can now answer queries in constant time, this requires $\mathcal{O}(n \log n)$ preprocessing time. In particular, we have to create, for each of the $\mathcal{O}(n)$ nodes in the border tree, a data structure of size $\mathcal{O}(\log n)$. In order to reduce this it seems futile to try and reduce the size of each data structure to be constant. Instead, we achieve a better preprocessing time by computing the structure only for a subset of nodes. For this, we resort to a micro-macro decomposition of the compressed border tree. By then constructing the structure from the previous section only on the macro-tree, we can answer the query $(i,j,k) \in \mathsf{PrefSubstrQueries}(s)$ if $s[:i]$ corresponds to a special node. For a non-special node in some fragment $T'$, its ancestors can be decomposed into two sets: the ancestors of the root node of $T'$, which we can check in the same way, and the ancestors of the node in $T'$. For the latter set, checking it can be reduced to a range query, which, due to the small size of $T'$, turns out to be possible to answer in constant time.

▶ **Lemma 31.** *For string $s$, with $|s| = n$, we can construct a linear-space data structure in $\mathcal{O}(n)$ time allowing us to retrieve $\mathsf{PrefSubstr}^c_s(i,j,k)$ for any $(i,j,k) \in \mathsf{PrefSubstrQueries}(s)$ in $\mathcal{O}(1)$ time.*

**Proof.** During preprocessing, we construct the data structure from Lemma 30, excluding the data structures specific to each node, i.e., $L_v$, $P_v$ and $Y_v$ (including the structure for answering $\mathsf{RMQ}$-queries on $Y_v$). We then create a micro-macro decomposition of the compressed border tree, consisting of $\mathcal{O}(n/\log n)$ fragments of size at most $\lfloor \log n \rfloor$ (see Lemma 12). For each of the special nodes, we create the $\mathcal{O}(\log n)$-sized data structure from Lemma 30.

For each fragment $F$ of size $m$, we now construct the following data structure. First, we associate each $y$, where $s[:y]$ corresponds to some node $v_y$ of $F$ with a triple $(p_y, \overline{p}_y, \ell_y)$, where $p_y$ and $\overline{p}_y$ are the preorder and postorder numbers of $v_y$ (in $F$, not the entire tree) and $\ell_y$ is the preorder number of $\mathsf{SuffNode}_s(y+1,n)$ in the suffix tree (same as in the proof of the previous lemma). We then create a fusion tree for $Y_F = \{y_1, \ldots, y_{2m}\}$ with $y_1 < \cdots < y_{2m}$ being the lengths of all the relevant prefixes in $F$, i.e., the longest two prefixes corresponding to each node of $F$. We also construct a fusion tree for $L_F = \{\ell_y \mid y \in Y_F\}$. Next, we precompute the following sets:

- $\{y \in Y_F \mid p_y \in [1,r]\}$ for all $r \in [m]$ (in ascending order),
- $\{y \in Y_F \mid \overline{p}_y \in [r,m]\}$ for all $r \in [m]$ (in descending order),
- $\{y \in Y_F \mid \ell_y \in [1,\ell]\}$ for all $\ell \in L_F$ (in ascending order),
- $\{y \in Y_F \mid \ell_y \in [\ell,n]\}$ for all $\ell \in L_F$ (in descending order).

Each of these sets can be encoded using a number $w \in [n^2]$ and thus, stored in two machine words, as follows. The set $Y_F(w) \subset Y_F$ encoded by $w$ contains $y_r$, if and only if the $r$-th least significant bit of $w$ is set. If done in the order given above, each set can be constructed by adding one element to its predecessor, allowing for total $\mathcal{O}(m)$ construction time.

Hence, we construct data structures for each special node and fragment in $\mathcal{O}(\log n)$ time. As there are at most $\mathcal{O}(n/\log n)$ of those, constructing all data structures is done in $\mathcal{O}(n)$ time.

Given a query $(i, j, k) \in \mathsf{PrefSubstrQueries}(s)$, we determine $\mathsf{SuffNode}_s(j, k)$, the corresponding label interval $[\ell^-, \ell^+]$ and the node $v_i$ corresponding to $s[: i]$ in the border tree. We also find the fragment $F$ containing $v_i$ and its root node $u_i$. We can use the data structure from Lemma 30 on $u_i$ to determine the longest border corresponding to an ancestor of $u_i$ that induces a match. The remaining borders of $s[: i]$ all correspond to ancestors of $v_i$ in the fragment, the lengths of the borders inducing a match is exactly

$$Y_i := \{y \in Y_F \mid (p_y, \overline{p}_y, \ell_y) \in [1, p_i] \times [\overline{p}_i, m] \times [\ell^-, \ell^+]\}.$$

The first two dimensions ensure that the node corresponding to $s[: y]$ is an ancestor of $v_i$ and the last dimension ensures that the posterior has prefix $s[j : k]$. To determine $Y_i$, we start by finding the minimum and maximum element $\ell_{\min}$ and $\ell_{\max}$ of $L_F \cap [\ell^-, \ell^+]$, which are the successor of $\ell^-$ and the predecessor of $\ell^+$ in $L_F$ respectively. $Y_i$ is then the intersection of

- $\{y \in Y \mid p_y \in [1, p_i]\}$,
- $\{y \in Y \mid \overline{p}_y \in [\overline{p}_i, m]\}$,
- $\{y \in Y \mid \ell_y \in [1, \ell_{\max}]\}$ and
- $\{y \in Y \mid \ell_y \in [\ell_{\min}, n]\}$.

Using bit operations, the bit representation of this intersection can be found in constant time and the same is true for the largest element in $Y_i$, this corresponds to the most significant bit set in the representation. ◀

As described in the overview, this last lemma completes the proof of Theorem 18:

▶ **Theorem 18.** *For string $s$, with $|s| = n$, we can construct a linear-space data structure in $\mathcal{O}(n)$ time allowing us to determine $\mathsf{PrefSubstr}_s(i, j, k)$ for any $(i, j, k) \in \mathsf{PrefSubstrQueries}(s)$ in $\mathcal{O}(1)$ time.*

## References


1. Paniz Abedin, Sahar Hooshmand, Arnab Ganguly, and Sharma V. Thankachan. The heaviest induced ancestors problem: Better data structures and applications. *Algorithmica*, 84(7):2088–2105, 2022. doi:10.1007/S00453-022-00955-7.
2. Jarno N. Alanko, Davide Cenzato, Nicola Cotumaccio, Sung-Hwan Kim, Giovanni Manzini, and Nicola Prezza. Computing the LCP array of a labeled graph. In *CPM*, volume 296 of *LIPIcs*, pages 1:1–1:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.CPM.2024.1.
3. Amihood Amir, Gary Benson, and Martin Farach. Let sleeping files lie: Pattern matching in z-compressed files. *J. Comput. Syst. Sci.*, 52(2):299–307, 1996. doi:10.1006/JCSS.1996.0023.
4. Rocco Ascone, Giulia Bernardini, Alessio Conte, Massimo Equi, Estéban Gabory, Roberto Grossi, and Nadia Pisanti. A unifying taxonomy of pattern matching in degenerate strings and founder graphs. In *WABI*, volume 312 of *LIPIcs*, pages 14:1–14:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.WABI.2024.14.
5. Jasmijn A. Baaijens, Paola Bonizzoni, Christina Boucher, Gianluca Della Vedova, Yuri Pirola, Raffaella Rizzi, and Jouni Sirén. Computational graph pangenomics: a tutorial on data structures and their applications. *Nat. Comput.*, 21(1):81–108, 2022. doi:10.1007/S11047-022-09882-6.
6. Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In *CPM*, volume 191 of *LIPIcs*, pages 8:1–8:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CPM.2021.8.
7. Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.

8 Massimo Equi, Roberto Grossi, Veli Mäkinen, and Alexandru I. Tomescu. On the complexity of string matching for graphs. In *ICALP*, volume 132 of *LIPIcs*, pages 55:1–55:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ICALP.2019.55.

9 Martin Farach. Optimal suffix tree construction with large alphabets. In *FOCS*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.

10 Martin Farach and Mikkel Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998. doi:10.1007/PL00009202.

11 Greg N. Frederickson. A data structure for dynamically maintaining rooted trees. *J. Algorithms*, 24(1):37–65, 1997. doi:10.1006/JAGM.1996.0835.

12 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

13 Moses Ganardi and Pawel Gawrychowski. Pattern matching on grammar-compressed strings in linear time. In *SODA*, pages 2833–2846. SIAM, 2022. doi:10.1137/1.9781611977073.110.

14 Pawel Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: Fast, simple, and deterministic. In *ESA*, volume 6942 of *Lecture Notes in Computer Science*, pages 421–432. Springer, 2011. doi:10.1007/978-3-642-23719-5_36.

15 Pawel Gawrychowski. Optimal pattern matching in LZW compressed strings. *ACM Trans. Algorithms*, 9(3):25:1–25:17, 2013. doi:10.1145/2483699.2483705.

16 Ming Gu, Martin Farach, and Richard Beigel. An efficient algorithm for dynamic text indexing. In *SODA*, pages 697–704. ACM/SIAM, 1994. URL: http://dl.acm.org/citation.cfm?id=314464.314675.

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

18 M. Lothaire. *Combinatorics on words, Second Edition.* Cambridge mathematical library. Cambridge University Press, 1997.

19 Solon P. Pissis. Optimal prefix-suffix queries with applications. In *SOSA*, pages 166–171. SIAM, 2025. doi:10.1137/1.9781611978315.13.


## A Proofs from Section 2

▶ **Lemma 2.** *Let $s$ be a string of length $n$ with $p = \mathrm{per}(s)$ and let $m \in [n-1]$ such that $s[:m]$ is a border of $s$ with $\mathrm{per}(s[:m]) = q \neq p$. Then $p \geq \frac{1}{2}m$ and $n \geq \frac{3}{2}m$.*

**Proof.** We start by showing that $p \geq \frac{m}{2}$. If $p \geq m$ we are done. Otherwise, $s[:m]$ as a substring of $s$ must also be $p$-periodic and $q < p$. Now if $p + q < m$, then by Lemma 1, $s[:m]$ would be $\gcd(p, q)$ periodic and $q$ would have to divide $p$. But then $s[:p]$ (which is $q$-periodic itself), can be written as a repetition of $s[:q]$ and $s$ would be $q$-periodic as well. Therefore, we have $m \leq p + q < 2p$.

The longest border of $s$ (that is not $s$ itself) has length $n - p$, i.e., $n \geq m + p \geq \frac{3}{2}m$. ◀

▶ **Lemma 3.** *Let $s$ be a string of length $n$. Then $|\{\mathrm{per}(r) \mid r \in \mathit{Bord}(s)\}| \in \mathcal{O}(\log n)$.*

**Proof.** Let $r_1, \ldots, r_k \in \mathsf{Bord}(s)$ be borders with $0 < |r_i| < |r_j|$ and $\mathrm{per}(r_i) \neq \mathrm{per}(r_j)$ for all $i, j \in [k], i < j$. Then by the previous lemma, we have $n \geq |r_k| \geq \frac{3}{2}|r_{k-1}| \geq \cdots \geq (\frac{3}{2})^{k-1}|r_1|$, i.e., $k \in \mathcal{O}(\log n)$. ◀

▶ **Lemma 4.** *For string $s$, with $|s| = n$, and integer $p \in [n-1]$, the elements of $\{|r| \mid r \in \mathit{Bord}_p(s)\}$ form an arithmetic progression with step $p$. For $p = \infty$, $\mathit{Bord}_p(s)$ contains at most one non-empty string.*

**Proof.** Note that a substring $s'$ of a periodic string $s$ has $\mathrm{per}(s') \leq \mathrm{per}(s)$. In the case of the borders, this means that if we sort the borders by periodicity, the periodicity values will also be sorted. Consequently, the group of borders with the same periodicity $q$ occur in a sequence and, since the largest border (that is not itself) of some $q$-periodic string is exactly $q$ shorter, the lengths of the group in the border form an arithmetic progression with step $q$.

For $p = \infty$, if there were two aperiodic borders $r, r' \in \mathsf{Bord}_\infty(s)$ with $0 < |r| < |r'|$, then $r$ would also be a border of $r'$ and $r'$ would have period $q = |r'| - |r|$ with $1 \leq q < |r'|$, which is a contradiction. ◀

▶ **Lemma 9.** *Let $s$ be a string, let $r \in \mathit{Suff}(s)$ be a suffix of $s$ and $v$ be the node in the suffix tree of $s$ corresponding to $r$. Then $|r| \geq |r'|$ for all $r' \in \mathit{Strings}(v) \cap \mathit{Substr}(s)$ (this excludes all strings ending with \$) and $r$ is the only suffix of $s$ corresponding to $v$.*

**Proof.** Assume we had $r' \in \mathsf{Strings}(v) \cap \mathsf{Substr}(s)$ with $|r'| > |r|$. By applying Observation 8 twice, we see that $r\$$ must correspond to a descendant of $v$, and, since $|r'| \geq |r\$|$, $r\$$ must be a prefix of $r'$, which is a contradiction. The second part follows since the suffixes of $s$ have pairwise different lengths. ◀

▶ **Lemma 12.** *Given a tree of size $n$ and some $z \in [n]$, it is possible to find a micro-macro decomposition that has $\mathcal{O}(n/z)$ fragments of size at most $z$ in $\mathcal{O}(n)$ time.*

**Proof.** We compute the fragments bottom-up the tree. For each node $u$, we maintain the number $F_u$ of its descendants (including itself) that are not already part of some finished fragment and make sure that $F_u$ does not grow larger than $z$. Initially, set $F_u = 1$ for all leaf nodes $u$.

For some inner node $u$ let $V_u$ be the set of its children. Partition this into singleton sets, i.e., $\mathcal{V} := \{\{v\} \mid v \in V_u\}$. Maintain a set of all small partitions $V \in \mathcal{V}$ with $\sum_{v \in V} F_v < \lfloor z/2 \rfloor$. Merge the partitions by repeatedly combining two small partitions until there is at most one small partition left. We now have a set of partitions each smaller than $z$. If there is only one partition $V$, we set $F_u = 1 + \sum_{v \in V} F_v$. If there is more than one partition, we make $u$ a special node and create a fragment for each of the partitions. Since other fragments will not contain any descendant of $u$, we set $F_u = 0$. Note that we created $k \geq 2$ fragments and at least $k-1$ of those contain at least $\lfloor z/2 \rfloor$ non-special nodes. Therefore, our fragments contain on average $\Omega(z)$ non-special nodes and we cannot construct more than $\mathcal{O}(n/z)$ fragments. ◀

## B Proofs from Section 3

▶ **Lemma 16.** *For any string $s$ and any query $(i, j, k) \in \mathit{PrefSubstrQueries}(s)$, the answer $x = \mathit{PrefSubstr}_s(i, j, k)$ satisfies $\mathit{PrefSubstrMatches}_s(i, j, k) = \mathit{Bord}(s[:x])$.*

**Proof.** For any element $s[:y] \in \mathsf{PrefSubstrMatches}_s(i, j, k)$ we have $|s[:y]| \leq |s[:x]|$ and $s[:y], s[:x] \in \mathsf{Pref}(s), \mathsf{Suff}(s[:i][j:k])$. It follows that $s[:y]$ must both be a prefix and a suffix of $s[:x]$, i.e., a border.

Any border $b$ of $s[:x]$ is both a prefix and a suffix of $s[:x]$, therefore, by transitivity, both a prefix of $s$ and a suffix of $s[:i][j:k]$, i.e., $b \in \mathsf{PrefSubstrMatches}_s(i, j, k)$. ◀

## C Crossing matches: Naive algorithm

▶ **Lemma 32.** *Given a string $s$ of length $n$, we can construct a linear-space data structure in $\mathcal{O}(n)$ time allowing us to determine $\mathit{PrefSubstr}^c_s(i, j, k)$ for any query $(i, j, k) \in \mathit{PrefSubstrQueries}(s)$ in $\mathcal{O}(n)$ time.*

**Proof.** During preprocessing, we construct a border tree for $s$ and a data structure allowing us to answer $\mathsf{LCP}$ queries on $s$ in constant time (see Lemma 5).

To answer a query $(i, j, k)$, we obtain all the borders of $s[: i]$ from the border tree (these correspond to ancestors of the node corresponding to $s[: i]$) and then check, for each border $s[: y]$, whether its posterior has $s[j : k]$ as a prefix. This is the case, if and only if $\mathsf{LCP}_s(y + 1, j) \geq |s[j : k]|$. Since there are up to $i \in \mathcal{O}(n)$ borders of $s[: i]$, processing queries still requires linear time. ◀

## D Proofs from Section 4

▶ **Lemma 19.** *Given string $s$, with $|s| = n$, we can construct a linear-space data structure in $\mathcal{O}(n)$ time allowing us to determine $\max\{|r| \mid r \in \mathsf{Bord}(s[: k]), |r| \leq \ell\}$ for any $k, \ell \in [n]$ in $\mathcal{O}(1)$ time.*

**Proof.** During preprocessing, we construct a suffix tree for $s^R$. For each suffix of $s^R$, we visit the corresponding node and mark it as a suffix node; note that each suffix corresponds to a different node as seen in Lemma 9.

We then traverse the suffix tree in preorder, maintaining a stack of suffix nodes on the path from the root to the current node we visit. Using this, we can, for each node, store a reference to its lowest ancestor that is a suffix node. We also construct the data structure that allows to answer $\mathsf{SuffNode}_{s^R}$ queries (see Lemma 7).

When given a query $(k, \ell) \in [n]^2$, we determine $v = \mathsf{SuffNode}_{s^R}(n - k + 1, n - k + \ell)$, i.e., the node corresponding to $(s[k - \ell + 1 : k])^R$ in the suffix tree. Any $s[: x] \in \mathsf{Bord}(s[: k])$ is a suffix of $s[: k]$, therefore, $(s[: x])^R$ is a prefix of $(s[k - \ell + 1 : k])^R$ and a suffix of $s^R$, and the corresponding node $u$ must be an ancestor of $v$ (by Observation 8) and a suffix node. Since we want to find the maximal value for $x$, we take the lowest such ancestor $u$ and the corresponding suffix $s[: x]$ is our answer (if $u = v$ and the corresponding suffix is longer than $\ell$, we use the lowest suffix node ancestor of the parent of $v$ instead). If there exists no such suffix node, we return zero instead. ◀

## E Proofs from Section 5

▶ **Lemma 21.** *Let $s$ be a string of length $n$. We can construct a linear-space data structure in $\mathcal{O}(n)$ time allowing us to find the largest $y$ satisfying $s[: y] \in \mathsf{Bord}_p(s[: i])$ and $s[j : k] \in \mathsf{Pref}(s[y + 1 :])$ for any query $(i, j, k) \in \mathsf{PrefSubstrQueries}(s)$ and periodicity $p \in [i - 1] \cup \{\infty\}$ in constant time, provided we are given (an $\mathcal{O}(1)$ size representation of) $\mathsf{Bord}_p(s[: i])$ as well.*

**Proof.** During preprocessing, we construct a data structure allowing us to answer $\mathsf{LCP}$ queries on $s$ in constant time (see Lemma 5).

If $p = \infty$, then there is only a constant number of borders, which we can check individually as seen in the previous section. Therefore, we focus on the case $p < \infty$.

The lengths of borders form an arithmetic progression, i.e., there are $y_0, \ell \in \mathbb{N}$ such that $\mathsf{Bord}(s[: i]) = \{s[: y_0 + p], s[: y_0 + 2p], \ldots, s[: y_0 + \ell p]\}$. Due to periodicity, it is

$$r := s[y_0 + 1 : y_0 + p] = s[y_0 + p + 1 : y_0 + 2p] = \cdots = s[y_0 + (\ell - 1)p + 1 : y_0 + \ell p]$$

and, for some $b \in [\ell]$, we can write $s[: y_0 + bp] = s[: y_0]r^b$ and $s[y_0 + bp + 1 :] = r^{\ell - b}s[y_0 + \ell p + 1 :]$

Obviously, each of the posteriors starts with some repetition of $r$ and, for each of them, $r$ is repeated a different number of times. If we consider both $s[y_0 + 1 :]$ and $s[y_0 + \ell p + 1 :]$, we can see that the former must start with more repetitions of $r$ than the latter. Therefore,

by computing $\mathsf{LCP}_s(y_0+1, y_0+\ell p+1)$ we find the first mismatch between $s[y_0+1:]$ and $s[y_0+\ell p+1:]$. This tells us how many repetitions of $r$ are at the start of $s[y_0+\ell p+1:]$ and, consequently, how many there are at the start of each of the posteriors. By comparing this number with the number of repetitions of $r$ at the start of $s[j:k]$, we can quickly decide which borders might induce a match. This is explained in detail now.

We can determine the number of repetitions at the start of $s[j:k]$ as follows. Let us say that, as determined previously, $s[y_0+1:] = r^{\ell}s[y_0+\ell p+1]$ starts with $\ell'$ repetitions of $r$. We now calculate $t = \mathsf{LCP}_s(y_0+1, j)$.

If $|s[j:k]| \leq t$ and $|s[j:k]| \leq \ell' p$, then $s[j:k]$ consists only of repetitions of $r$. Therefore, any of the borders whose posterior starts with at least $\lceil |s[j:k]|/p \rceil$ repetitions of $r$ induces a match and, if there is a posterior starting with $\lfloor |s[j:k]|/p \rfloor$ repetitions, the corresponding prior might induce a match (this can quickly be verified with another $\mathsf{LCP}$ query).

If $\ell' p \leq t$ (and $\ell' p \leq |s[j:k]|$), then we know that $s[j:k]$ starts with at least $\ell'$ repetitions of $r$. Since all the posteriors start with fewer repetitions of $r$, there is no match in this case.

Otherwise, i.e., if $t < \ell' p$ and $t < |s[j:k]|$, we now that $s[j:k]$ starts with exactly $\lfloor t/p \rfloor$ repetitions of $r$. The only candidate for inducing a match is the border, whose posterior starts with exactly that amount of repetitions of $r$ (verified via $\mathsf{LCP}$). ◀

▶ **Lemma 22.** *For string $s$, with $|s| = n$, we can construct a linear-space data structure in $\mathcal{O}(n)$ time allowing us to retrieve $\mathit{PrefSubstr}^c_s(i,j,k)$ for $(i,j,k) \in \mathit{PrefSubstrQueries}(s)$ in $\mathcal{O}(\log n)$ time.*

**Proof.** During preprocessing, we construct a compressed border tree for $s$ and a data structure allowing us to answer $\mathsf{LCP}$ queries on $s$ in constant time (see Lemma 5).

Given a query $(i,j,k) \in \mathsf{PrefSubstrQueries}(s)$, we start by locating the node corresponding to $s[..i]$ in the border tree. By moving up the tree, we can obtain representations of $\mathsf{Bord}_p(s[:i])$ for all relevant values of $p$ and check each of them using Lemma 21 in constant time. As soon as we find some border inducing a match, we return its length. ◀