Optimal Data Structure for Internal Pattern Matching Queries in a Text and Applications

Tomasz Kociumaka∗1, Jakub Radoszewski†1, Wojciech Rytter‡1,2, and Tomasz Walen§1

1Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Poland
[kociumaka, jrad, rytter, walen]@mimuw.edu.pl
2Faculty of Mathematics and Computer Science, Copernicus University, Toruń, Poland

Abstract

We present a linear-space data structure which enables very fast (usually constant time) answers to several types of internal queries — questions about factors (also called substrings) of a text. A factor-in-factor occurrence query asks for a representation of the set of all occurrences of one factor in another factor of the same text of length . It assumes that , in this case the representation consists of a constant number of arithmetic progressions. This problem can be viewed as an internal version of the well-studied pattern matching problem. Our data structure is optimal: it has linear size and the query time is constant, also the construction time is linear. Using the solution to the factor-in-factor problem, we obtain very efficient data structures answering queries about: primitivity of factors, periods of factors, general substring compression, and cyclic equivalence of two factors. All these results improve upon the best previously known counterparts. Using our data structure for the period queries, we also provide the best known solutions for the recently introduced factor suffix selection queries and for finding in a text (a more general version of maximal repetitions, also called runs). With the latter improvement we obtain the first linear time algorithm finding for a fixed , which matches the linear time complexity of the algorithm computing runs. We benefit here from the linear construction time of our data structure.

The model of internal queries in texts is connected to the well-studied problem of text indexing. Both models have their origins in the introduction of suffix trees. However, there is an important difference: in our model the size of the representation of a query is constant and therefore enables faster query time. Our results can be viewed as efficient solutions to “internal” equivalents of several basic problems of regular pattern matching and make an improvement in a majority of related already published results. We introduce several novel techniques extending the method of pattern matching by sampling. We apply probabilistic tools, related to range minima in random permutations. The construction algorithms of our data structures are randomized but the queries are deterministic.

∗Supported by Polish budget funds for science in 2013-2017 as a research project under the ‘Diamond Grant’ program.
†The author receives financial support of Foundation for Polish Science.
‡Supported by grant no. N206 566740 of the National Science Centre.
§Supported by Iuventus Plus grant (IP2011 058671) of the Polish Ministry of Science and Higher Education.
1 Introduction

There are many algorithmic problems concerning factors (substrings) in a word. In these problems we need to construct a data structure which answers efficiently queries specified by factors of a given word. This constitutes a growing field in the area of text processing. Its origins start with the invention of suffix trees that can be used to answer the most basic types of internal queries: equality of factors and longest common prefix queries, with constant query time and linear space.

One of the first studies in this area, on a family of problems of compressibility of factors, was given by Cormode and Muthukrishnan (SODA’05) [6]; some of these results were later improved in [21]. Other typical problems include: range longest common prefix queries (range LCP) [1, 32], periodicity [22, 10], minimal/maximal suffixes [3], etc.

A related model is text indexing, in which one desires to preprocess a given word for future queries specified by (usually shorter) patterns. In this setting the query time is $\Omega(m)$, where $m$ is the size of the query pattern. Our model better fits the scenario when a number of data texts are stored and we only query for factors of these texts. Indeed, each query can now be specified in constant space and therefore $o(m)$ time algorithms answering queries are possible.

A routine approach to factor-related queries is based on applications of orthogonal range searching, see [26]. For the current state of knowledge this implies $\Omega(\log \log n)$ query time and, in most cases, super-linear space. Moreover, the construction time is $\Omega(n\sqrt{\log n})$, most often $\Omega(n \log n)$. We design tools based on text processing that are better-tailored for factor-related queries and allow to obtain constant query time with linear space and (expected) linear construction time. We benefit from the fast construction time when we apply our techniques to problems in a static setting.

We identify one of the basic problems in this new area which we call factor-in-factor occurrence queries and show its usefulness. This problem can be viewed as a direct analogue of the well-studied pattern matching problem. We also consider a number of problems related to periods of factors. Computation of different types of periodicities is one of the central parts of algorithmics on words. A similar type of queries (for tiling periodicity) was studied in [20]. A natural extension of testing equality of words is the problem of cyclic equivalence, also called conjugacy [27]. We introduce this problem in the context of factors and give an optimal solution. As applications of our results we obtain faster solutions to a few recently studied problems of pattern matching and text compression.

We consider linearly sortable alphabets, that is, we assume that $\Sigma$, the set of letters of the given word $v$, can be sorted in linear time (e.g. $\Sigma \subseteq \{0, 1, \ldots, |v|^{O(1)}\}$). A factor of $v$ is a word of the form $v[i] \ldots v[j]$. The factors in each query are represented by a start- and end-position of their occurrence. The results are for word-RAM model with word size $w = \Omega(\log n)$, with $n$ being the length of $v$, and the algorithms are deterministic, unless otherwise stated.

1.1 Previous Work

We say that a positive integer $p$ is a period of $v$ if there exists a word $u$ of length $p$ and an integer $k$ such that $v$ is a prefix of $u^k$. The word is called periodic if it has a period at most half of its length. The following three types of queries were already studied.

<table>
<thead>
<tr>
<th>Period Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given a factor $x$ of $v$, report all periods of $x$ (represented by disjoint arithmetic progressions).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2-Period Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given a factor $x$ of $v$, decide whether $x$ is periodic and, if so, compute its shortest period.</td>
</tr>
</tbody>
</table>
Known efficient algorithms for these types of queries apply orthogonal range searching queries in 2-dimensional rank space (i.e., coordinates of points are in the range [1, n], where n is the number of points, query rectangles are orthogonal). Three types of such queries were used to answer aforementioned queries: the range-emptiness queries (rempt) which ask if a query rectangle contains any of the given points, range successor queries (rsucc) which ask for the smallest y-coordinate of a point in the rectangle, and range searching for minimum queries (rmin) where points are additionally equipped with weights and we ask for the minimum-weight point in the rectangle. Note that each of these problems generalizes the previous ones.

The Period Queries problem was introduced and first studied in [22]. The solutions have \( O(\log n) \) query time with \( O(n \log n) \) space, and \( O(Q_{rsucc} \log n) \) query time with \( O(n + S_{rsucc}) \) space. Currently the best trade-offs for the range successor queries are: \( Q_{rsucc} = O(\log^2 n) \) for \( S_{rsucc} = O(n) \) [31], \( Q_{rsucc} = O(\log \log n) \) for \( S_{rsucc} = O(n \log \log n) \) [33], and \( Q_{rsucc} = O(1) \) for \( S_{rsucc} = O(n^{1+\varepsilon}) \) [11]. Period Queries, in spite of their very recent introduction, have already found applications. In [2] the authors use them to design factor suffix selection queries. In [23] they are used to compute all subrepetitions in a word, a notion extending the notion of a run in a word.

The 2-Period Queries problem is a special case of the Period Queries problem briefly introduced in [10], with a solution in \( O(Q_{rmin}) \) query time and \( O(n + S_{rmin}) \) space. Here \( Q_{rmin} = O(\log n) \) for \( S_{rmin} = O(n) \) [28, 21], \( Q_{rmin} = O(\log \log n) \) for \( S_{rmin} = O(n \log^\varepsilon n) \) [4], and \( Q_{rmin} = O(1) \) for \( S_{rmin} = O(n^{1+\varepsilon}) \) [11]. However, the solution to general Period Queries presented in [22] implies slightly better trade-offs: \( O(Q_{rsucc}) \) query time with \( O(n + S_{rsucc}) \) space, and \( O(1) \) query time with \( O(n \log n) \) space.

Bounded Longest Common Prefix Queries were introduced in [21] as a tool for the following Generalized Substring Compression Queries: given two factors \( x \) and \( y \) of \( v \), compute the part of the LZ77 [34] compression \( LX(y \mid x) \) that corresponds to \( x \), where \( \$ \notin \Sigma \). This problem was introduced in [6] and was also referred to as substring compression with an additional context substring. In [21] a solution to Bounded Longest Common Prefix Queries with \( O(Q_{rmin} + Q_{rempt} \log |p|) \) query time and \( O(n + S_{rempt} + S_{rmin}) \) space implied an \( O(C(Q_{rmin} + Q_{rempt} \log \frac{|p|}{C})) \) time algorithm for Generalized Substring Compression Queries. The following trade-offs are currently known for range-emptiness queries: \( Q_{rempt} = O(\log^2 n) \) for \( S_{rempt} = O(n) \) and \( Q_{rempt} = O(\log \log n) \) for \( S_{rempt} = O(n \log \log n) \) [4], and \( Q_{rempt} = O(1) \) for \( S_{rempt} = O(n^{1+\varepsilon}) \) [11].

As a by-product of [21], a solution to the following decision version of internal pattern matching problem is obtained: a data structure of size \( O(n + S_{rempt}) \) that given factors \( x, y \) checks whether \( x \) occurs in \( y \) in \( O(Q_{rempt}) \) time, provided that \( x \) is given by its locus in the suffix tree. All occurrences of \( x \) in \( y \) can be reported in additional time proportional to the number of these occurrences.

1.2 Our Results

We introduce queries that find all occurrences of one factor of \( v \) in another factor.

Factor-in-Factor Occurrence Queries

Given factors \( x \) and \( y \) of \( v \) with \( |y| \leq 2|x| \), report all occurrences of \( x \) in \( y \) (represented as an arithmetic progression).

Our main result is the following theorem together with its corollaries.
Theorem 1. Factor-in-Factor Occurrence Queries can be answered in \( O(1) \) time by a data structure of size \( O(n) \), which can be constructed in \( O(n) \) expected time.

Remark 1. For Factor-in-Factor Occurrence Queries, the requirement \(|y| \leq 2|x|\) can be dropped at the cost of increasing the query time to \( O(|y|/|x|) \) and allowing several arithmetic progressions on the output.

A number of applications of Factor-in-Factor Occurrence Queries are presented. We consider 2-Period Queries and Period Queries defined above as well as new types of queries.

<table>
<thead>
<tr>
<th>Prefix-Suffix Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given factors ( x ) and ( y ) of ( v ) and a positive integer ( d ), report all prefixes of ( x ) of length between ( d ) and ( 2d ) that are also suffixes of ( y ) (represented as an arithmetic progression of their lengths).</td>
</tr>
</tbody>
</table>

A word \( y \) is called a cyclic rotation of a word \( x \) if \( y = x[i+1] \ldots x[n]x[1] \ldots x[i] \) for some \( i \).

<table>
<thead>
<tr>
<th>Cyclic Equivalence Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given factors ( x ) and ( y ) of ( v ), decide whether ( x ) is a cyclic rotation of ( y ) and, if so, report all corresponding cyclic shift values (represented as an arithmetic progression).</td>
</tr>
</tbody>
</table>

Corollary 1. Using a data structure of \( O(n) \) size, which can be constructed in \( O(n) \) expected time, one can answer:

- Prefix-Suffix Queries in \( O(1) \) time,
- 2-Period Queries in \( O(1) \) time,
- Period Queries in \( O(\log |x|) \) time,
- Cyclic Equivalence Queries in \( O(1) \) time.

As we already mentioned, [2] and [23] rely on Period Queries. Corollary 1 lets us improve these results as follows. For the factor suffix selection defined in [2], the query time with \( O(n) \) space improves from \( O(\log^{2+\varepsilon} n) \) to \( O(\log^2 n) \).

A \( \delta \)-subrepetition is a generalization of the notion of a run for exponent at least \( 1+\delta \), with \( \delta \leq 1 \). Thus runs are 1-subrepetitions. The best previously known algorithm for finding \( \delta \)-subrepetitions [23] worked in \( O(n \log n + \frac{n}{\delta^2} \log \frac{1}{\delta}) \) time. With our data structure it improves to \( O(n + \frac{n}{\delta^2} \log \frac{1}{\delta}) \). In particular, we obtain the first linear time algorithm finding \( \delta \)-subrepetitions for a fixed \( \delta \), which matches an \( O(n) \) time algorithm for computing runs.

Another application of Factor-in-Factor Occurrence Queries is the following.

Corollary 2. Using a data structure of \( O(n + S_{\text{rempt}} + S_{\text{rsucc}}) \) size one can answer Bounded Longest Common Prefix Queries in \( O(Q_{\text{rsucc}} + Q_{\text{rempt}} \log \log |p|) \) time.

The data structure of Corollary 2 yields a solution to Generalized Substring Compression Queries with \( O(C(Q_{\text{rsucc}} + Q_{\text{rempt}} \log \log \frac{|p|}{C})) \) query time, as compared to \( O(C(Q_{\text{rmin}} + Q_{\text{rempt}} \log \frac{|x|}{C})) \) time queries of [21]. Here \( C \) is the number of phrases reported.

1.3 Our Techniques

We use in a completely novel way a classic approach to pattern matching by sampling. To search for occurrences of \( x \) in \( y \), both of which are factors of \( v \), we assign to \( x \) a sample which is also a factor
of $x$. Then we find occurrences of this sample in $y$ and check which of them extend to occurrences of $x$ in $y$.

The sample has length $2^{\lceil \log |x| - 1 \rceil}$ (it is a so-called basic factor). It may be either non-periodic, in which case $y$ contains only a constant number of occurrences of this sample, or periodic, in which case all its occurrences can be located en masse with the aid of a unique maximal repetition (a run) induced by the sample in $x$.

In the non-periodic case we use a probabilistic argument to make sure that the expected number of different samples of factors of $v$ is $O(n)$. We identify such samples using a space-efficient variant of the Dictionary of Basic Factors data structure (which is normally of size $\Theta(n \log n)$). In the periodic case we precompute the structure of all runs in the word to be able to efficiently find all runs in $y$ that are consistent with the run of the sample. This consistency-check is based on combinatorial properties of Lyndon words. The space bound of our data structure relies on a known fact that the sum of exponents of runs in a word is linear.

### 1.4 Organization of the Paper

We start with recalling basic notions of combinatorics on words in Section 2. Afterwards in Section 3 we present the main ideas of our data structure and in Section 4 we develop the necessary probabilistic tools. The main parts of the data structure for FACTOR-IN-FACTOR OCCURRENCE QUERIES are described in Sections 5, 6 and 7. In Section 5 we cover non-periodic case although to simplify presentation we only describe the case of a square-free text. Then in Section 6 we show the solution of the periodic case. Section 7 presents a general solution that combines the techniques from the two previous sections (actually the results of Section 6 are used as a black-box while for the techniques of Section 5 we apply a few minor modifications). Full descriptions of the data structures for periodic and general case are given in Sections 8 and 9. Later, in Section 10, we describe a linear construction algorithm of the whole data structure. We conclude the paper with a detailed presentation of applications of FACTOR-IN-FACTOR OCCURRENCE QUERIES (Section 11).

### 2 Preliminaries

Consider a word $v = v[1]v[2] \ldots v[n]$ of length $|v| = n$, where $v[i] \in \Sigma$. For $1 \leq i \leq j \leq n$, a word $u = v[i] \ldots v[j]$ is called a factor of $v$. By $v[i, j]$ we denote an occurrence of $u$ at position $i$ called a fragment of $v$. Throughout the paper by $[i, j]$ we denote an integer interval $\{i, \ldots, j\}$.

The following fact specifies a known efficient data structure for comparison of factors of a word. It consists of the suffix table with its inverse, LCP table and a data structure for range minimum queries on the LCP table, see [7, 14].

**Fact 1** (Equality Testing). Let $v$ be a word of length $n$. After $O(n)$ preprocessing, one can test in $O(1)$ time whether two given fragments are occurrences of the same factor.

A fragment of $v$ of the form $BF_k(i) = v[i, i+2^k-1]$ is called a $k$-basic fragment. By $n_k = n - 2^k + 1$ we denote the number of $k$-basic fragments of $v$. A factor that occurs as a $k$-basic fragment is called a $k$-basic factor. By $m_k$ we denote the number of different $k$-basic factors of $v$ ($m_k \leq n_k$).

The dictionary of basic factors (DBF in short), see [7, 14], consists of $\lceil \log n \rceil$ layers. The $k$-th layer is a table $DBF_k$ such that $DBF_k[i]$ is an identifier of $BF_k(i)$. The identifiers are consecutive positive integers that satisfy $DBF_k[i] \leq DBF_k[i']$ if and only if $BF_k(i) \leq BF_k(i')$. 

| 4 |
We say that a positive integer \( p \) is a period of \( v \) if \( v[i] = v[i + p] \) holds for all \( i \in [1, n - p] \). The shortest period is denoted as \( \text{per}(v) \). We call \( v \) periodic if \( 2\text{per}(v) \leq |v| \) and primitive if \( \text{per}(v) \) is not a proper divisor of \( |v| \).

A run (a maximal repetition) in \( v \) is a periodic fragment \( \alpha = v[i, j] \) which cannot be extended neither to the left nor to the right without increasing the shortest period \( p = \text{per}(\alpha) \), that is, \( v[i - 1] \neq v[i + p - 1] \) and \( v[j - p + 1] \neq v[j + 1] \), provided that the respective letters exist. We define the exponent of a run as \( \exp(\alpha) = \frac{|\alpha|}{\text{per}(\alpha)} \). In our algorithms runs are represented together with their periods.

**Example 1.** The word \( v = \text{baababaababb} \) contains three runs with period 1: \( \text{aa} \) twice, as \( v[2, 3] \) and as \( v[7, 8] \), and \( v[11, 12] = \text{bb} \); two runs with period 2: \( v[3, 7] = \text{ababa} \) and \( v[8, 11] = \text{bab} \); one run with period 3: \( v[5, 10] = \text{abaaba} \); and one run with period 5: \( v[1, 11] = \text{baababaabab} \).

The structure of runs in a word can be used to represent all repetitions in a word in a compact way. The following fact gathers deep combinatorial and algorithmic results concerning runs useful throughout the paper.

**Fact 2** ([24, 25, ?, 12, 8]). In a word of length \( n \) both the number of runs and the sum of their exponents are \( O(n) \). Moreover, all the runs in a word can be computed in \( O(n) \) time.

The following fact states, in particular, that the result of Factor-in-Factor Occurrence Queries is well-defined. Its proof is given in Section 8.1.

**Fact 3.** Let \( x, y \) be words with \( |y| \leq 2|x| \). Then the set of positions where \( x \) occurs in \( y \) forms a single arithmetic progression. Moreover, if there are at least 3 occurrences, the difference of this progression is \( \text{per}(x) \).

### 3 Our Approach

We sketch our approach for answering Factor-in-Factor Occurrence Queries focusing on the non-periodic case. For simplicity we assume in this section that \( v \) is square-free. Recall that a word \( u \) is called a square if \( u = vwv \) for a word \( w \), and a word is called square-free if it does not contain any squares as factors, see also [27].

We call a set \( X \subseteq \mathbb{N} \) a \( \Delta \)-sparse set if for any distinct elements \( a, b \in X \), \( |a - b| > \Delta \).

**Observation 1** (Sparsity of Occurrences). Let \( u \) and \( v \) be words and assume \( v \) is square-free. Then the set of positions where \( u \) occurs in \( v \) is \( |u| \)-sparse.

In particular, Factor-in-Factor Occurrence Queries for a square-free word \( v \) return at most one occurrence.

The first approach to solve Factor-in-Factor Occurrence Queries in the square-free case using the idea of samples could be as follows. For each of the basic factors of \( v \) we store a sorted list of all its occurrences. The sample \( x' \) for a query pattern \( x \) is its prefix being a \( \lfloor \log |x| \rfloor \)-basic factor. Using the precomputed lists we find all occurrences of \( x' \) in \( y \), by Observation 1 there are at most 3 such occurrences. Afterwards standard techniques (Fact 1) let us verify which of them extend to occurrences of \( x \). This approach requires \( \Theta(n \log n) \) space due to the number of possible samples. To obtain \( O(n) \) space, we will perform a more careful selection of samples so that their total number is linear. In the following definition we slightly change the approach and compute samples only for basic fragments of \( v \).
Definition 1. Let \( v \) be a square-free word. We call \( \text{repr}_k \) a \( k \)-representative assignment for \( v \), if the following conditions are satisfied:

1. \( \text{repr}_k : [1, n_{k+1}] \rightarrow [1, n_k] \), i.e. \( \text{repr}_k \) assigns to each \((k+1)\)-basic fragment a \( k \)-basic fragment,
2. \( \text{repr}_k(i) \in [i, i + 2^k] \) for each \( i \), i.e. each fragment is assigned one of its subfragments,
3. \( \text{repr}_k(i) - i = \text{repr}_k(i') - i' \) if \( \text{BF}_{k+1}(i) = \text{BF}_{k+1}(i') \), i.e. if two basic fragments are occurrences of the same factor, their corresponding subfragments are assigned.

The values of \( k \)-representative assignment are called \( k \)-representative positions or representative occurrences of the corresponding \( k \)-basic factors. The set of \( k \)-representative positions is denoted as \( \text{Repr}_k \). We say that a \( k \)-basic fragment is \( \text{representative} \) if it starts at a \( k \)-representative position.

In the data structure we store only the representative occurrences of all \( k \)-basic factors. Note that for a fixed basic factor this set might be empty or contain some occurrences, but not necessarily all of them. Thus property (3) is crucial for the correctness of our approach to queries.

Let \( S_m \) be the set of all permutations of \( \{1, \ldots, m\} \). For a permutation \( \pi_k \in S_m \) we set

\[
\text{repr}_k(i) = \arg\min\{ID_k[j] : j \in [i, i + 2^k]\}
\]

(1)

where \( ID_k[j] = \pi_k(\text{DBF}_k[j]) \). It turns out that this definition satisfies the conditions for a representative assignment and, moreover, one can choose \( \pi_k \) so that the total size of \( \text{Repr}_k \) is \( O(n) \). The formal proof, which uses probabilistic results of Section 4, is postponed until Section 5.

Let us conclude with a sketch of our approach to constructing \( \text{repr}_k \). We later show that given any superset of \( \text{Repr}_k \) it is easy to construct \( \text{repr}_k \). We construct a candidate set \( C_k \), which is a small superset of \( \text{Repr}_k \), in two steps. First, we find \( A_k = \{j \in [1, n_k] : ID_k[j] \leq \ell_k\} \) for an appropriate parameter \( \ell_k \). Then we extend it, setting \( C_k = \text{FillGaps}(A_k, 2^k, [1, n_k]) \), where \( \text{FillGaps} \) is defined as follows:

\[
\text{FillGaps}(A, \Delta, I) = A \cup \bigcup \{[i, i + \Delta] : [i, i + \Delta] \subseteq I \setminus A\}.
\]

![Diagram](image1.png)

Figure 1: \( C = \text{FillGaps}(A, 4, [1, 34]) \). Here \( A = \{7, 9, 13, 20, 21, 26, 32, 34\} \) and \( C \) is obtained from \( A \) by inserting all maximal subintervals of the domain that are disjoint with \( A \) and contain more than 4 integers (in this example, 17 elements are inserted).

As we shall see in Section 4 in a more abstract setting, the set \( C_k \) generated in this way is always a superset of \( \text{Repr}_k \) and its expected size is \( O\left(\frac{nk}{2^k}\right) \). Details of the construction algorithm, already in the general case, are presented in Section 10.

4 Probabilistic Tools

Let \( a \) be a sequence of length \( n \) over \([1, m]\) and let \( \pi \in S_m \). We say that the sequence \( a \) is \( \Delta \)-diverse if for each element \( \sigma \in [1, m] \) the set \( \{i : a_i = \sigma\} \) is \( \Delta \)-sparse. Fix a positive integer \( \Delta \) and for \( i \in [1, n - \Delta] \) define

\[
f_\pi(i) = \arg\min\{\pi(a_j) : j \in [i, i + \Delta]\}.
\]
In case of ties, which are possible if \( a \) is not \( \Delta \)-diverse, we take the leftmost index. We say that the values of \( f_\pi \) are local \( \pi \)-minima.

Consider a function \( g \) defined on an interval \([\ell, r]\). We define a piecewise constant representation of \( g \) as a collection of triples \((\ell_i, r_i, v_i)\) such that \( g(x) = v_i \) for \( x \in [\ell_i, r_i] \) and \{[\ell_i, r_i] : i\} is a partition of \([\ell, r]\). The size of the representation is the number of triples.

**Example 2.** An illustration of \( f_\pi \) for \( m = 4 \), \( \Delta = 4 \), \( \pi = (3, 2, 1, 4) \) and an example sequence \( a \). Shades of gray represent intervals in a piecewise constant representation of \( f_\pi \): \((1, 2, 2), (3, 4, 4), (5, 6, 7), (7, 9, 11)\). Note that \( a \) is 2-diverse.

![Illustration of Example 2](image)

<table>
<thead>
<tr>
<th>( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_i )</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>( \pi(a_i) )</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>( f_\pi(i) )</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Lemma 1.** Assume that \( a \) is a \( \frac{\Delta}{2} \)-diverse sequence of length \( n \) over \([1, m]\) and let \( \pi \) be a permutation of \([1, m]\) drawn uniformly at random. Let \( A = \{i : \pi(a_i) \leq \ell\} \) for a parameter \( \ell \), and \( C = \text{FillGaps}(A, \Delta, [1, n]) \). Then

(a) \( C \) contains all local \( \pi \)-minima,
(b) if \( \ell = \left\lfloor \frac{2m \log \frac{\Delta}{\Delta}}{\Delta} \right\rfloor \), then \( E[|C|] = O\left(\frac{n \log \Delta}{\Delta}\right) \).

Here we prove (a) only, the proof of (b) is presented in Appendix A.

**Proof of (a).** Assume \( j = f_\pi(i) \) is a local \( \pi \)-minimum. If \( \pi(a_j) \leq \ell \), then \( j \in A \), so in particular \( j \in C \). Otherwise, not only \( \pi(a_j) > \ell \), but for any \( i' \in [i, i + \Delta] \) we have \( \pi(a_{i'}) > \ell \). Thus \([i, i + \Delta] \subseteq [1, n] \setminus A\), i.e. the FillGaps operation adds this interval, and in particular \( j \), to \( C \). \( \square \)

**Corollary 3.** Let \( a \) be a \( \frac{\Delta}{2} \)-diverse sequence and let \( \pi \) be a permutation of \([1, m]\) drawn uniformly at random. Then the local \( \pi \)-minima function \( f_\pi \) admits a piecewise constant representation of expected size bounded by \( O\left(\frac{n \log \Delta}{\Delta}\right) \).

**Proof.** By Lemma 1 the expected number of local \( \pi \)-minima is \( O\left(\frac{n \log \Delta}{\Delta}\right) \). To obtain the same bound for the size of a piecewise constant representation, it suffices to prove that \( f_\pi \) is non-decreasing. For a proof by contradiction assume \( j' = f_\pi(i') < f_\pi(i) = j \) for \( i < i' \). Note that \( i < i' \leq j' < j \leq i + \Delta < i' + \Delta \). Consequently \( j' \in [i, i + \Delta] \) and \( j \in [i', i' + \Delta] \). The former implies \( \pi(a_{j'}) < \pi(a_j) \) by definition of \( f_\pi(i) \) while the latter implies \( \pi(a_{j'}) \leq \pi(a_j) \) by definition of \( f_\pi(i') \), a contradiction. \( \square \)

## 5 Square-Free Case

In Section 3 we have already presented a rough description of our data structure for the square-free case. In this section we fill in the missing details of the data structure and the query algorithm in this case. We start with a justification of our selection of a representative assignment (1).

**Lemma 2.** Let \( v \) be a square-free word. There exist permutations \( \pi_k \in S_m \) such that \( \text{repr}_k \) given by (1) form a family of representative assignments for \( v \) and admit piecewise constant representations of total size \( O(n) \).
In the evaluator we additionally store the identifiers for the total size of piecewise constant representations is \(O(x)\). Our naive algorithm (compare Fact 1) to detect which extend to an actual occurrence

\[ BF_k(i) = BF_{k+1}(i') \]

implies that for any \(\delta \in [0, 2^k]\) we have \(BF_k(i + \delta) = BF_k(i' + \delta)\), therefore the minimum in (1) for \(repr_k(i)\) and \(repr_k(i')\) will represent the corresponding \(k\)-basic fragments.

We apply the probabilistic method to show that one can choose \(\pi_k\) so that \(repr_k\) admits a piecewise constant representation of size \(O \left( \frac{nk}{2^k} \right)\). Note that Observation 1 implies that \(DBF_k\) is a \(2^k\)-diverse sequence. Moreover, we have defined \(repr_k\) as a function assigning local \(\pi_k\)-minima for the sequence \(DBF_k\) with \(\Delta = 2^k\). Thus, by Corollary 3, if \(\pi_k\) is drawn uniformly at random, the expected size of the piecewise constant representation of \(repr_k\) is \(O \left( \frac{nk}{2^k} \right)\). In particular, one can choose \(\pi_k\) so that this quantity is actually \(O \left( \frac{nk}{2^k} \right)\). Summing up over all \(k\)’s we obtain the desired \(O(n)\) bound.

The solution of for the square-free case requires two auxiliary abstract data structures. Their implementation is described in Appendix B.

**Evaluator**

**Input:** A function \(g : [1, n] \rightarrow U\) that admits a piecewise constant representation of size \(m\) (the elements of \(U\) fit in \(O(1)\) words).

**Queries:** Given \(i\) compute \(g(i)\).

**Lemma 3.** For \(g\) specified in a piecewise constant representation of size \(m\), there exists an evaluator \(E(g)\) of size \(O(m + \frac{n}{\log n})\) that answers queries in \(O(1)\) time and can be constructed in \(O(m + \frac{n}{\log n})\) time.

**Locator**

**Input:** An indexed family \(A = (A_i)\) of \(d\)-sparse subsets of \([1, n]\).

**Queries:** Given an index \(i\) and a range \(P\) of length \(O(d)\) return \(A_i \cap P\).

**Lemma 4.** For a family \(A = (A_i)\) there exists a locator \(L(A)\) of size \(O(\sum_i |A_i|)\) that can answer queries in \(O(1)\) time. It can be constructed in \(O(\sum_i |A_i|)\) expected time given \(\{(i, j) : j \in A_i\}\).

Now we are ready to provide a rigorous description of our data structure for the square-free case.

**Theorem 2.** For a square-free word \(v\) of length \(n\) there exists a data structure of \(O(n)\) size that can answer FACTOR-IN-FACTOR OCCURRENCE QUERIES in \(O(1)\) time.

**Proof.** The data structure consists of \([\log n]\) layers. For each layer \(k\) we store an evaluator of the representative assignment and a locator of representative positions, that is \(E(repr_k)\) and \(L(A^k)\), where \(A^k_{id}\) is the set of representative occurrences of the \(k\)-basic factor whose identifier \(ID_k\) is \(id\).

In the evaluator we additionally store the identifiers \(ID_k\) of the representative positions. We also maintain the global data structure specified in Fact 1. If \(repr_k\) is chosen so that it satisfies Lemma 2, the total size of piecewise constant representations is \(O(n)\), hence \(|Repr_k| = O(n)\) and the total size of \(A^k\) is \(O(n)\). This concludes that the total size of evaluators and locators is \(O(n)\).

The query algorithm for \(x = v[\ell, r]\) and \(y = v[\ell', r']\) works as follows. If \(|x| = 1\), we use a naive algorithm (compare \(x\) with each letter of \(y\)). Otherwise we use the data structures for the \(k\)-th layer, for \(k = |\log |x| - 1|\): we use \(E(repr_k)\) to obtain \(j = repr_k(\ell)\) and \(id = ID_k[j]\). We set \(\delta = j - \ell\) and use \(L(A^k)\) to compute \(A^k_{id} \cap [\ell' + \delta, r' + 1 - |x|]\), i.e. the representative occurrences of the \(k\)-basic factor \(BF_k(j)\) which might be induced by an occurrence of \(x\) in \(y\) (see Figure 2). We obtain a constant number of them and use Fact 1 to detect which extend to an actual occurrence of \(x\).
Figure 2: The query algorithm finds \( j = \text{repr}_k(\ell) \) and the identifier \( id \) of \( BF_k(j) \), here depicted with a gray rectangle. Then, it finds all representative occurrences of \( id \), which might be induced by occurrences of \( x \) in \( y \). Here these occurrences lie at positions \( occ_1 \) and \( occ_2 \). Potential occurrences of \( x \) are marked with dashed rectangles.

6 Overview of Periodic Case

In the periodic case of FACTOR-IN-FACTOR OCCURRENCE QUERIES we assume that \( x \) contains a periodic \( k \)-basic factor. In the following subsection we introduce a notion of a \( k \)-run, a central notion in the solution of the periodic case. Afterwards we show how to answer queries in this case.

6.1 Repetitive Structure of Words

We say that a run \( \alpha \) extends a fragment \( u \) if \( u \) is a subfragment of \( \alpha \) and \( \text{per}(u) = \text{per}(\alpha) \). For any periodic fragment \( u \), there is a unique run \( \alpha \) extending \( u \), we denote it as \( \text{run}(u) \) (see Figure 3). If \( u \) is not periodic, we set \( \text{run}(u) = \bot \). The following lemma, proved in Appendix C, might be of independent interest.

![Figure 3: \( \text{run}(u) = \alpha \). If \( u \) is a \( k \)-basic factor then \( \alpha \) is a \( k \)-run.](image)

**Lemma 5.** There exists a data structure of \( O(n) \) size, which given a fragment \( u \) returns \( \text{run}(u) \) in constant time. Such a data structure can be constructed in \( O(n) \) time.

A run \( \alpha \) is called a \( k \)-run if \( |\alpha| \geq 2^k \) and \( \text{per}(\alpha) < 2^k \). Alternatively, a run is a \( k \)-run if it extends a periodic fragment of length between \( 2^k \) and \( 2^{k+1} - 1 \). Note that the definition implies that if \( \text{run}(u) = \alpha \neq \bot \), then \( \alpha \) is a \( k \)-run for \( k = \lfloor \log |u| \rfloor \).

![Figure 4: A sample run \( \alpha \) with \( |\alpha| = 19 \), \( \text{per}(\alpha) = 3 \) and \( \exp(\alpha) = 6\frac{1}{7} \). It is a \( k \)-run for \( k = 2, 3, 4 \).](image)

By \( R(v) \) we denote the set of all runs in \( v \), and by \( R_k(v) \) the set of \( k \)-runs in \( v \). Note that \( \bigcup R_k(v) = R(v) \), but the sum is not necessarily disjoint. A fixed run \( \alpha \) can be a \( k \)-run for at most
exp(α) values of k (see Figure 4), which by Fact 2 implies that \(\sum_k |R_k(v)| = \mathcal{O}(n)\). In Appendix C we prove a stronger property of k-runs, from which we derive the \(\mathcal{O}(n)\) space bound for several components of our data structure.

A word that is both primitive and lexicographically minimal in the class of its cyclic rotations is called a Lyndon word. Let \(u\) be a word with period \(p\). The Lyndon root \(\lambda\) of \(u\) is the Lyndon word that is a cyclic rotation of \(u[1,p]\). Then \(u\) can be represented as \(\lambda'\lambda^k\lambda''\) where \(\lambda'\) is a proper suffix of \(\lambda\), and \(\lambda''\) is a proper prefix of \(\lambda\). The Lyndon representation of \(u\) is defined as \((|\lambda'|, k, |\lambda''|)\). As shown in [10], Lyndon representations of all runs can be computed in \(\mathcal{O}(n)\) time. We say that two runs are compatible if they have the same Lyndon root.

6.2 Queries

Let \(x \cap y\) denote the common subfragment of overlapping fragments \(x, y\). The following observation shows why k-runs are useful in the solution of the periodic case.

**Observation 2.** Assume that \(x = v[\ell, r]\) and \(x' = v[\ell', r']\) are occurrences of the same factor and that \(x\) has a periodic k-basic fragment \(z\). Let \(\alpha = \text{run}(z)\). Then there exists a k-run \(\alpha'\) that is compatible with \(\alpha\), such that \(x \cap \alpha\) and \(x' \cap \alpha'\) are the corresponding subfragments of \(x\) and \(x'\).

\[\alpha \geq \max(2^k, 2\text{per}(\alpha))\]

![Figure 5: Synchronization of runs on two occurrences of the same factor (Observation 2). The runs may have different lengths but their intersections with the occurrences of the factor start and end at the same positions relative to these occurrences.](image)

Observation 2 lets us take the following approach for finding \(x\) in \(y\). First, we locate \(z\), a periodic k-basic fragment of \(x\), and compute the k-run \(\alpha = \text{run}(z)\) using Lemma 5. Then, we find all k-runs \(\alpha'\) which are compatible with \(\alpha\) and intersect \(y\). Now we consider two cases.

If \(\alpha\) does not cover \(x\), i.e. \(x \cap \alpha \neq x\), then the corresponding k-run \(\alpha'\) also does not cover \(x'\). Since \(x \cap \alpha\) and \(x' \cap \alpha'\) must be corresponding subfragments of \(x\) and \(x'\) respectively, given \(\alpha'\) we have a single possible position of \(x'\). Thus, for a fixed \(\alpha'\) it suffices to apply Fact 1 for at most one candidate (some might be already out of consideration as exceeding \(y\)). On the other hand, if \(\alpha\) covers \(x\), then any occurrence \(x'\) of \(x\) is guaranteed to be covered by the corresponding k-run \(\alpha'\), in particular \(x' \subseteq y \cap \alpha'\). If \(|y \cap \alpha'| < |x|\), then clearly there is no such \(x'\). Otherwise, we find the occurrences of \(x\) in \(y \cap \alpha'\) (already represented as an arithmetic progression) using the Lyndon representations of both fragments, see Section 8 for details.

We get at most one arithmetic progression for each k-run \(\alpha'\). By Fact 3, these progressions can be combined to a single arithmetic progression.
7 Overview of General Case

The central notion of Section 5 was that of a representative assignment. If we generalized Definition 1 to arbitrary words with periodicities directly, the representative assignment would require $\Omega(n \log n)$ space, e.g. for the word $v = a^n$. However, if the query pattern $x = v[\ell, r]$ contains a periodic $k$-basic factor with $k = \lceil \log |x| - 1 \rceil$, we can apply the data structure for periodic case to answer this query. Thus, if $BF_{k+1}(\ell)$ contains a periodic $k$-basic fragment, we leave $repr_k(\ell)$ undefined (denoted $repr_k(\ell) = \perp$) and modify the query algorithm, so that it launches the data structure for periodic case whenever it gets an undefined representative.

Observation 1 does not hold for arbitrary word $v$. If we assume that $u$ is non-periodic, we obtain a slightly weaker result.

Observation 3 (General Sparsity of Occurrences). Let $u$ and $v$ be words and assume $u$ is not periodic. Then the set of positions where $u$ occurs in $v$ is $\frac{|u|}{2}$-sparse.

Definition 2. Let $v$ be an arbitrary word. We call $repr_k$ a $k$-representative partial assignment for $v$, if the following conditions are satisfied:

1. $repr_k : [1, n_{k+1}] \rightarrow [1, n_k] \cup \{\perp\}$, i.e. if defined, $repr_k$ assigns to each $(k+1)$-basic fragment a $k$-basic fragment,
2. $repr_k(i) \in [i, i + 2^k] \cup \{\perp\}$ for each $i$, i.e. each fragment is assigned one of its subfragments or $\perp$,
3. $repr_k(i) - i = repr_k(i') - i'$ or $repr_k(i) = repr_k(i') = \perp$ if $BF_{k+1}(i) = BF_{k+1}(i')$, i.e. if two basic factors are equal, either the corresponding subfragments are assigned or both representatives are undefined.
4. $repr_k(i) = \perp$ if and only if $BF_{k+1}(i)$ contains a periodic $k$-basic factor.

The notions of representative positions $Repr_k$ and representative occurrences carry on. Generalizing the approach presented in Section 3, we choose permutations $\pi_k \in S_{m_k}$, define $ID_k[j] = \pi_k(DBF_k[j])$ and set

$$repr_k(i) = \begin{cases} \perp & \text{if } BF_k(j) \text{ is periodic for some } j \in [i, i + 2^k], \\ \arg\min\{ID_k[j] : j \in [i, i + 2^k]\} & \text{otherwise}. \end{cases}$$

Like in the square-free case, we can choose $\pi_k$ so that $repr_k$ admit piecewise constant representations of total size $O(n)$, see Section 9 for details.

The main idea of the construction algorithm remains unchanged: we are still looking for a candidate set $C_k$ that is a small superset of $Repr_k$. As previously we start with $A_k = \{j : ID_k[j] \leq \ell_k\}$ but we need to use FillGaps much more carefully: instead of $U = [1, n_k]$, we apply it separately for each maximal block of positions where non-periodic $k$-basic fragments start. In particular, we need to use some techniques from the periodic case to find these blocks and to identify positions $i$ where $repr_k(i)$ is defined. The construction algorithm is presented in detail in Section 10.

8 Full Description of Periodic Case

In Section 8.1 we introduce a number of combinatorial and algorithmic tools, among which the most important is the so-called $k$-RUN LOCATOR. We also give a proof of Fact 3 which states that
$O(1)$-sized answers to FACTOR-IN-FACTOR OCCURRENCE QUERIES always exist. Then we describe the complete data structure for the periodic case in Section 8.2.

### 8.1 Toolbox

We start by recalling two classic lemmas. We use them to prove Fact 3 and to introduce an additional combinatorial tool for the periodic case (Fact 4).

**Lemma 6** (Periodicity Lemma [15, 29]). Let $v$ be a word with periods $p$ and $q$. If $p + q \leq |v|$, then $\gcd(p,q)$ is also a period of $v$.

**Lemma 7** (Synchronization of Primitive Words [7]). Let $\lambda$ be a primitive non-empty word. Then $\lambda$ has exactly two occurrences in $\lambda \ell$.

**Fact 3.** Let $x, y$ be words with $|y| \leq 2|x|$. Then the set of positions where $x$ occurs in $y$ forms a single arithmetic progression. Moreover, if there are at least 3 occurrences, the difference of this progression is $\text{per}(x)$.

**Proof.** Assume that $x$ occurs in $y$ at positions $i_1 < i_2 < \ldots < i_m$. If $m \leq 2$, the conclusion of the fact is trivially satisfied, so assume that $m \geq 3$. Let $x = q^k q'$, where $|q| = \text{per}(x)$ and $q'$ is a proper prefix of $q$. Note that if $x$ occurs in $y$ at positions $i, i'$ with $i < i' < i + |x|$, then $i' - i$ is a period of $x$. Moreover, $\sum_{j=1}^{m-1} (i_{j+1} - i_j) = i_m - i_1 \leq |y| - |x| \leq |x|$. Therefore for each $j \in [1, m-1]$ the value $i_{j+1} - i_j$ is a period of $x$ and by the Periodicity Lemma (Lemma 6) it is a multiplicity of $\text{per}(x)$. It suffices to show that it is actually equal to $\text{per}(x)$. Let us fix $j \in [1, m-1]$ and let $\ell = \frac{i_{j+1} - i_j}{\text{per}(x)}$, note that $\ell$ is an integer in $[1, k]$. Consider a word $z = y[i_j, i_{j+1} + |x|]$. Observe that $z = q^\ell x = q^{k+\ell} q'$. Clearly, $x$ occurs in $z$ at position $1 + \text{per}(x)$. Consequently $x$ occurs in $y$ at position $i_j + \text{per}(x)$, which implies that $i_{j+1} - i_j = \text{per}(x)$. 

**Fact 4.** Let $x$ and $y$ be periodic with common Lyndon root. Then the set of positions where $x$ occurs in $y$ is an arithmetic progression that can be computed in $O(1)$ time given the Lyndon representations of $x$ and $y$.

**Proof.** Let $\lambda$ be the common Lyndon root of $x$ and $y$ and let their Lyndon representations be $(p, k, s)$ and $(p', k', s')$ respectively. Synchronization Property (Lemma 7) implies that $\lambda$ occurs in $y$ only at positions $i$ such that $i \equiv p' + 1 \pmod{\lambda}$. Consequently, $x$ occurs in $y$ only at positions $i$ such that $i \equiv p' - p + 1 \pmod{\lambda}$. Clearly $x$ occurs in $y$ at all such positions $i$ in $[1, |y| - |x| + 1]$. Therefore it is a matter of simple arithmetics to compute the arithmetic progression of these positions.

Apart from the data structure computing $\text{run}(u)$, the run extending $u$ for arbitrary periodic factor $u$, in the periodic case we also need a couple of additional data structures. Their implementation is provided in Appendix C.

<table>
<thead>
<tr>
<th>k-Run Locator</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A word $v$ of length $n$.</td>
</tr>
<tr>
<td><strong>Queries:</strong> Given an integer $p$ and a range $P \subseteq [1, n]$ with $</td>
</tr>
</tbody>
</table>

**Lemma 8.** There exist $k$-run locators $K_k(v)$ that answer queries in $O(1)$ time, take $O(n)$ space in total, and can be constructed in $O(n)$ expected total time.
Let \( R \) be a finite set of integers. We call \( B \) a block of \( R \) if \( B \) is an inclusion-wise maximal interval contained in \( R \). A block representation of \( R \) is a sorted list of all its blocks.

**Lemma 9.** For \( k \in [0, \lfloor \log n \rfloor] \) let \( P_k = \{ i \in [1, n_k] : BF_k(i) \text{ is periodic} \} \). In \( O(n) \) time we can compute all sets \( P_k \), each of them represented both in the block representation and as a bit vector.

We also use the classic successor queries. The following version with efficient construction algorithm is described in Appendix B.

**Lemma 10.** For an arbitrary set \( R \subseteq [1, n] \) there exists a data structure \( S(R) \) of size \( O\left( \frac{n}{\log n} \right) \), which answers successor queries on \( R \) (\( \text{succ}_R(i) = \min(R \cap [i, n]) \)) in \( O(1) \) time. Moreover \( S(R) \) can be constructed in \( O\left( \frac{n}{\log n} \right) \) time if \( R \) is given as a bit vector.

### 8.2 Data Structure

The data structure consists of a global part, consisting of the set \( \mathcal{R}(v) \) of all runs together their Lyndon representations, as well as the data structure for computing \( \text{run}(u) \) (see Lemma 5) and the data structure of Fact 1 to check occurrences. It also contains \( \lfloor \log n \rfloor \) layers, the \( k \)-th one consists of the data structure for successor queries for \( P_k = \{ i \in [1, n_k] : BF_k(i) \text{ is periodic} \} \) (see Lemma 10) and the \( k \)-run locator \( K_k(v) \) (see Lemma 8).

The query algorithm for \( x \) and \( y \) has already been sketched in Section 6. Here we provide the missing implementation details. Successor queries on \( P_k \) let us find the smallest \( j > i \) such that \( BF_k(j) \) is periodic. In particular they allow to find \( z \), the leftmost \( k \)-periodic basic subfragment of \( x \). Once we get \( z \), Lemma 5 is used to compute \( \alpha = \text{run}(z) \). Then we use \( K_k(v) \) to compute the \( k \)-runs \( \alpha' \) which intersect \( y \) and have \( \text{per}(\alpha') = \text{per}(\alpha) \). This is done in constant time, which in particular implies that the number of such \( k \)-runs \( \alpha' \) is constant. For each \( \alpha' \) we start with verifying if \( \alpha \) and \( \alpha' \) compatible, i.e. whether their Lyndon roots are equal. We use the precomputed Lyndon representations to localize the Lyndon roots as fragments of \( v \) and then simply check if they are occurrences of the same factor using Fact 1. For \( k \)-runs \( \alpha' \) which pass this test we proceed as described in Section 6, either checking a single candidate position (if \( x \cap \alpha \neq x \)) or Lyndon representations otherwise. In the latter case if \( \lfloor y \cap \alpha \rfloor \geq |x| \) we derive the Lyndon representations of \( x \cap \alpha \) and \( y \cap \alpha' \) from the representations of \( \alpha \) and \( \alpha' \) and apply Fact 4. Finally we merge several arithmetic progressions into a single one, which is simple since Fact 3 guarantees that their union indeed forms a single arithmetic progression.

The construction algorithm works as follows. The set \( \mathcal{R}(v) \) is computed using the algorithm of Kolpakov & Kucherov [24] using the results of Crochemore & Ilie [8] so that it does not require constant-sized alphabet to run in \( O(n) \) time. Then we use the algorithm of Crochemore et al. [10] to compute Lyndon representations of runs. We also launch Lemma 9 to compute \( P_k \) represented as bit vectors and pass them to the algorithm constructing \( S(P_k) \) (Lemma 10). The construction of remaining components comes down to running the algorithms provided by the appropriate lemmas. Consequently, we obtain the main result of this section.

**Theorem 3.** There exists a data structure of \( O(n) \) size which can be constructed in \( O(n) \) expected time that answers Factor-in-Factor Occurrence Queries in \( O(1) \) time provided that the pattern \( x \) contains a periodic \( k \)-basic factor for \( k = |\log |x| - 1| \).
9 Full Description of General Case

In this section we complete the description of the data structure for the general case of FACTOR-IN-FACTOR OCCURRENCE QUERIES. We prove Lemma 11, which is a general version of Lemma 2 that worked only in the square-free case. The construction algorithm of our data structure is shown in the next section.

Lemma 11. Let \( v \) be an arbitrary word. There exist permutations \( \pi_k \in S_{nk} \) such that \( \text{repr}_k \) given by (2) form a family of representative partial assignments for \( v \) and admit piecewise constant representations of total size \( O(n) \).

Proof. It is easy to see that \( \text{repr}_k \) given as (2) satisfies the conditions required for a representative partial assignment (see Definition 2). Again, we will use the probabilistic method to show that if \( \pi_k \) is drawn uniformly at random, then the expected size of a piecewise constant representation of \( \text{repr}_k \) is \( O\left(\frac{nk}{2^k}\right) \). This will imply that for some choice of \( \pi_k \) the actual size is \( O\left(\frac{nk}{2^k}\right) \), which sums up to \( O(n) \) for all values of \( k \).

Here, unlike in the proof of Lemma 2, we cannot simply use Corollary 3 to bound the expected size of representations. First, let us prove that the number of blocks of consecutive of \( \bot \)'s in \( \text{repr}_k \) is \( O\left(\frac{n}{2^k}\right) \). Observe that if \( BF_k(j) \) is periodic, then \( \text{repr}_k(i) = \bot \) for all \( i \in [j - 2^k, j] \cap [1, nk] \). Thus each block of \( \bot \)'s, possibly except for the first and the last one, contains at least \( 2^k + 1 \) positions. Consequently, the number of such blocks is at most \( 2 + \frac{nk}{2^k + 1} = O\left(\frac{n}{2^k}\right) \).

Now, for each maximal block \( B = [i, i'] \) of positions where \( \text{repr}_k \) is defined, we consider an extension \( B' = [i, i' + 2^k] \) and the sequence of \( DBF_k[j] \) for \( j \in B' \). Since \( \text{repr}_k \) is defined for all positions in \( B \), all \( k \)-basic factors whose identifiers occur in this sequence are non-periodic. Consequently, this sequence is \( \frac{n}{2^k} \)-diverse by Observation 3. Moreover, \( \text{repr}_k \) restricted to \( B \) simply assigns local \( \pi_k \)-minima for that sequence, so we can use Corollary 3 to deduce that the expected size of piecewise constant representation of \( \text{repr}_k \) restricted to \( B \) is \( O\left(\frac{|B'|k}{2^k}\right) = O\left(k + \frac{|B|}{2^k}\right) \). Now consider all such blocks \( B \). Their number is \( O\left(\frac{n}{2^k}\right) \) (proportional to the number of blocks of \( \bot \)'s) and their total size is obviously \( O(n) \). Thus by linearity of expectation we can bound the expected size of a piecewise constant representation of the whole \( \text{repr}_k \) by \( O\left(\frac{n}{2^k}\right) \).

Theorem 4. For any word \( v \) of length \( n \) there exists a data structure of size \( O(n) \) that can answer FACTOR-IN-FACTOR OCCURRENCE QUERIES in \( O(1) \) time.

Proof. The proof is analogous to that of Theorem 2. Apart from the data structure of Theorem 3 for the periodic case, we use the same components: the data structure from Fact 1, evaluators \( E(\text{repr}_k) \) and locators \( L(A_k) \) (constructed for partial representative assignments \( \text{repr}_k \)). The locators \( L(A_k) \) are now defined for \( d = 2^{k-1} \), with \( 2^{k-1} \)-sparsity of \( A^{kd}_i \) being a consequence of Observation 3.

Relative to the square-free case, there are just two differences in the query algorithm for \( x = v[\ell, r] \) and \( y = v[\ell', r'] \). First, if \( \text{repr}_k(\ell) \) (for \( k = \lfloor \log |x| - 1 \rfloor \)) turns out to be undefined, we find out that we are in the periodic case, so we launch the component responsible for that case. The second modification concerns the very last step, returning the output. In Section 5 we were guaranteed that there is at most one occurrence, so returning it as an arithmetic progression was trivial. Here, we either pass the arithmetic progression obtained from the periodic case, or obtain a constant number of occurrences, which, by Fact 3, also form an arithmetic progression.
10 Construction of the Data Structure

We start with introducing additional algorithmic tools used solely in the construction algorithm. Their detailed implementation is provided in Appendix D. Afterwards we proceed with a description of the construction algorithm for the general case of Factor-in-Factor Occurrence Queries.

10.1 Space-efficient DBF

The standard implementation of the Dictionary of Basic Factors uses $\Theta(n \log n)$ space [7, 14]. We introduce its compact version which provides the same operations as regular DBF but with linear space and construction time. Its main component is a data structure of Gawrychowski [17] which efficiently locates basic factors in the suffix tree.

**COMPACTDBF**

**Input:** a word $v$ of length $n$

**Queries:** for an integer $k$:

1. given a position $i$ return $DBF_k[i]$,
2. given an identifier $j$ return $\{i : DBF_k[i] = j\}$,
3. return $m_k$, the number of distinct identifiers in $DBF_k$.

**Lemma 12.** For a word $v$ of length $n$ there exists $\text{COMPACTDBF}D(v)$ which takes $O(n)$ space, can be constructed in $O(n)$ time and can answer (1) and (3) queries in $O(1)$ time, and (2) queries in with $O(1)$ time delay per item reported.

Recall that in order to define $\text{repr}_k$ we have used identifiers $ID_k[j] = \pi_k(DBF_k[j])$, where $\pi_k$ is a permutation of $S_{m_k}$. $\text{RANDOMIZEDDBF}$ is a modification of $\text{COMPACTDBF}$, which instead of $DBF_k[i]$ operates on $ID_k[i] = \pi_k(DBF_k[i])$ as identifiers for queries (1) and (2), where for each level $k$, $\pi_k$ is drawn uniformly at random from $S_{m_k}$.

**Lemma 13.** For a word $v$ of length $n$ there exists a $\text{RANDOMIZEDDBF} D^*(v)$ which takes $O(n)$ space, can be constructed in $O(n)$ expected time and can answer (1) and (3) queries in $O(1)$ time, and (2) queries in with $O(1)$ time delay per item reported.

$\text{RANDOMIZEDDBF}$ is the source of randomization in our construction algorithm. Note that for different values of $k$, permutations $\pi_k$ are not independent. Actually, we could not draw them independently, as this would require $\Omega(n \log^2 n)$ bits of randomness as opposed to $O(n \log n)$ we can get during the $O(n)$ time construction.

10.2 Computing Candidates

As indicated in Sections 3 and 7, the crucial step of the construction algorithm is building candidate sets $C_k$, supersets of $\text{Repr}_k$ whose total size is $O(n)$.

**Lemma 14.** Let $v$ be an arbitrary word of length $n$. There exists an algorithm which returns sets $C_k \subseteq [1, n_k]$ together with identifiers $ID_k[j]$ for $j \in C_k$ such that:

- there exist permutations $\pi_k \in S_{m_k}$ such that $ID_k[j] = \pi_k(DBF_k[j])$ and, for the representative partial assignment $\text{repr}_k$ defined using (2) with these identifiers, $C_k \supseteq \text{Repr}_k$.
- $\mathbb{E}[\sum_k |C_k|] = O(n)$.
The expected running time of the algorithm is \( \mathcal{O}(n + \sum_k |C_k|) = \mathcal{O}(n) \).

**Proof.** The algorithm is based on an instance of \textsc{RandomizedDBF} \( \mathcal{D}^*(v) \), which in particular makes a random choice of the underlying permutations \( \pi_k \). We start with presenting our construction of sets \( C_k \), then prove its correctness using the results of Section 4 and conclude with providing an efficient implementation of our algorithm.

Recall that in Section 8 we have defined \( P_k \) as the set of positions \( j \) such that \( BF_k(j) \) is periodic. By \( N_k \) we denote its complement \([1,n_k] \setminus P_k\). The candidate sets \( C_k \) are constructed in two steps. First we set \( A_k = \{ j \in N_k : \mid ID_k[j] \mid \leq \ell_k \} \) for \( \ell_k = \left\lfloor \frac{km_k}{2^k+1} \right\rfloor \). Then for each block \( B \) of \( N_k \) we compute \( \text{FillGaps}(A_k \cap B, 2^k, B) \) and return the union of these sets as \( C_k \).

Consider a block \( B' \) of the set of positions where \( \text{repr}_k \) is defined. Note that if we extend \( B \) by \( 2^k \) positions to the right, we obtain a block \( B' \) of \( N_k \). Moreover, \( \text{repr}_k \) for positions in \( B' \) assigns local \( \pi_k \)-minima for \( DBF_k \) restricted to \( B \). By Observation 3, \( DBF_k \) restricted to \( B' \) is \( 2^k-1 \)-sparse. Consequently we can apply Lemma 1, which gives \( \mathbb{E}[|B \cap C_k|] = \mathcal{O} \left( \frac{|B|k}{2^k} \right) \). By linearity of expectation we conclude that \( \mathbb{E}[|C_k|] = \mathcal{O} \left( \frac{kn_k}{2^k} \right) \) and \( \mathbb{E}[\sum_k |C_k|] = \mathcal{O}(n) \). This proves the correctness of our algorithm, it remains to provide an efficient implementation.

Lemma 9 lets us efficiently compute sets \( P_k \), both in a block representation and as bit vectors. The former can be easily transformed to a block representation of \( N_k \) and the latter can be used to test in \( \mathcal{O}(1) \) time for \( j \in [1,n_k] \) whether \( BF_k(j) \) is periodic. We construct \( A_k \) separately for every \( k \). We use a type (3) query on \( \mathcal{D}^*(v) \) to determine \( m_k \), which is necessary to compute \( \ell_k \). Then for each identifier \( \leq \ell_k \) we use a type (2) query on \( \mathcal{D}^*(v) \) to get one occurrence of the corresponding \( k \)-basic factor. We use the bit-vector representation of \( P_k \) to test if this \( k \)-basic factor is periodic. For non-periodic factors we proceed with the execution of the type (2) query adding to \( A_k \) all the positions where the factor occurs. This way \( A_k \) is constructed in \( \mathcal{O}(1 + |A_k| + \frac{km_k}{2^k}) = \mathcal{O}(|A_k| + \frac{kn_k}{2^k}) \) time. Then we simultaneously sort all \( A_k \)'s, which increases the cost by a single \( \mathcal{O}(n) \) term.

Once \( A_k \) are sorted we apply the \( \text{FillGaps} \) operations, again independently for each \( k \). We simultaneously traverse \( A_k \) and the blocks of \( N_k \). This lets us determine all blocks of \( N_k \setminus A_k \), and add to \( C_k \) all elements of those blocks of size at least \( 2^k + 1 \). This is equivalent to running \( \text{FillGaps}(A_k \cap B, 2^k, B) \) separately for every block \( B \) of \( N_k \). Apart from \( \mathcal{O}(|C_k|) \) time to traverse \( A_k \) and actually fill the gaps, this procedure requires additional time proportional to the number of blocks of \( N_k \). However, since these representations for all sets \( N_k \) were constructed in \( \mathcal{O}(n) \) total time, this extra cost sums up to \( \mathcal{O}(n) \). Finally, we equip each \( j \in C_k \) with \( ID_k[j] \) using type (1) query on \( \mathcal{D}^*(v) \).

Unfortunately, the bound on \( \sum_k |C_k| \) provided by Lemma 14 holds only in expectation, and we are to construct a data structure with a guaranteed \( \mathcal{O}(n) \) size bound. Nevertheless it easy to modify this algorithm so that \( \sum_k |C_k| \) is guaranteed to be \( \mathcal{O}(n) \).

**Lemma 15.** The algorithm of Lemma 14 can be modified so that \( \sum_k |C_k| \) is guaranteed to be \( \mathcal{O}(n) \), with the running time still \( \mathcal{O}(n) \) in expectation.

**Proof.** We run the algorithm of Lemma 14 and repeat until the actual value of \( \sum_k |C_k| \) does not exceed twice its expectation. If the random bits used by subsequent iterations are independent, by Markov inequality each iteration succeeds with probability at least \( \frac{1}{2} \). Consequently the probability that the \( i \)-th iteration is performed is at most \( \frac{1}{2^i} \). The expected running time of a single iteration is \( \mathcal{O}(n) \), so the total expected running time is \( \mathcal{O}(n) \).
10.3 Construction of the Representatives

The most involved part of the construction algorithm is building a small piecewise constant representation of a representative partial assignment \( \text{repr} \). The choice of an appropriate assignment is already made by the algorithm of Lemma 15, which gives sets \( C_k \) equipped with identifiers \( ID_k[j] \) for \( j \in C_k \). The subsequent step involves the following abstract function, whose implementation is described in Appendix D.

Function \textbf{Slider}

\begin{itemize}
    \item \textbf{Input}: Positive integers \( d \leq m \) and a set \( A \) of pairs \( (q, p) \) with \( q \in \mathbb{Z} \) and \( p \in [1, m] \).
    \item \textbf{Output}: A piecewise constant representation of \( G : [1, m - d] \to A \) defined as follows: \( G(i) \) is the lexicographically smallest pair \( (q, p) \in A \) among pairs with \( p \in [i, i + d] \), \( \perp \) if no such pair exists.
\end{itemize}

\textbf{Lemma 16.} \textbf{Slider} can be implemented in \( O(|A|) \) time, provided that pairs in \( A \) are sorted by \( p \) in the input.

We run \textbf{Slider} for \( \{(ID_k[j], j) : j \in C_k\} \) and \( d = 2^k \), which gives a piecewise constant representation of a function

\[ G(i) = \begin{cases} \perp & \text{if } C_k \cap [i, i + 2^k] = \emptyset, \\ \min \{ (ID_k[j], j) \in C_k : j \in [i, i + 2^k] \} & \text{otherwise}. \end{cases} \]

Since \( C_k \) is guaranteed to contain all representative positions, whenever \( \text{repr}_k(i) = j \neq \perp \) we have \( G(i) = (ID_k[j], j) \). Thus, in order to construct a piecewise constant representation of \( \text{repr}_k \), it suffices to find all the maximum intervals where \( \text{repr}_k \) is defined, use a representation of \( G \) in that intervals and set \( \perp \) elsewhere.

Recall that we have defined \( N_k = \{ j : BF_k(j) \text{ is non-periodic} \} \) and block representations of all sets \( N_k \) can be obtained in \( O(n) \) time (see Lemma 9). Also, note that \( \text{repr}_k(i) \neq \perp \) if and only if \( [i, i + 2^k] \subseteq N_k \). Therefore it suffices to take all intervals in the representation of \( N_k \), remove those of length at most \( 2^k \) and trim the remaining by \( 2^k \) positions from the right. Consequently, a piecewise constant representation of \( \text{repr}_k \) can be constructed in time proportional to \( |C_k| \) and the size of the block representation of \( N_k \). Both terms sum up to \( O(n) \) for all values of \( k \).

Once we have \( \text{repr}_k \), we can run the construction algorithm of evaluator \( E(\text{repr}_k) \). We also prepare the set \( \{(ID_k[j], j)\} \), which is passed to the construction algorithm of a locator \( L(A^k) \), where \( A^k \) is an indexed family of sets \( A_{id}^k \), with \( A_{id}^k \) defined as the set of all representative occurrences of the \( k \)-basic factor whose identifier \( ID_k \) is \( id \).

Finally, we construct the global components, i.e. the data structure of Theorem 3 for the periodic case and the component of Fact 1. This way we obtain the result mentioned already in Section 1.2 that concludes the whole description of the data structure for FACTOR-IN-FACTOR OCCURRENCE QUERIES.

**Theorem 1.** FACTOR-IN-FACTOR OCCURRENCE QUERIES can be answered in \( O(1) \) time by a data structure of size \( O(n) \), which can be constructed in \( O(n) \) expected time.

11 Applications

In this section we show how FACTOR-IN-FACTOR OCCURRENCE QUERIES can be used in answering other types of internal queries considered in this paper.
11.1 Prefix-Suffix Queries & Period Queries

**Prefix-Suffix Queries**

Given factors $x$ and $y$ of $v$ and a positive integer $d$, report all prefixes of $x$ of length between $d$ and $2d$ that are also suffixes of $y$ (represented as an arithmetic progression of their lengths).

**Definition 3.** If a word $z$ is simultaneously a prefix of a word $x$ and a suffix of a word $y$, we call it a prefix-suffix of a pair $(x, y)$.

First, observe that we can assume that $|x| = |y|$. Indeed, if $|x| \neq |y|$ we can shorten the longer of the factors (removing the suffix in case of $x$ and the prefix in case of $y$). Let $d'$ be the common length of $x$ and $y$. If $d' > 2d$, we can further shorten both $x$ and $y$ in the same manner as before, so that $d' = 2d$. On the other hand, if $d' < d$, clearly an empty set needs to be reported. Thus we may assume that $d' \in [d, 2d]$, which lets us apply the following lemma, see also Figure 6:

**Lemma 17** ([22]). Let $d$ be a positive integer and let $x$, $y$ be words such that $d \leq |x| = |y| \leq 2d$. Let $x'$ be the prefix of $x$ of length $d$ and let $y'$ be the suffix of $y$ of length $d$. The following conditions are equivalent for an integer $\ell \geq d$:

- (a) $(x, y)$ has a prefix-suffix of length $\ell$,
- (b) $y'$ occurs in $x$ at position $\ell - d + 1$ and $x'$ occurs in $y$ at position $|y| - \ell + 1$.

![Figure 6](image-url)

Figure 6: A pair $(x, y)$ has a prefix-suffix $z$ of length $\ell$ if and only if $y'$ and $x'$ occur at certain positions in $x$ and $y$, respectively.

Denote the set of all positions where $x'$ occurs in $y$ by $Occ(x', y)$ and the set of all occurrences of $y'$ in $x$ by $Occ(y', x)$. Lemma 17 implies that we need to compute the following set of lengths

$$\{o + d - 1 : o \in Occ(y', x)\} \cap \{|y| - o + 1 : o \in Occ(x', y)\}.$$  

Since common elements of two arithmetic progressions form an arithmetic progression, we can already see that our result indeed forms an arithmetic progression.

We can use **Factor-in-Factor Occurrence Queries** to find both $Occ(x', y)$ and $Occ(y', x)$, both in constant time (see Figure 7). Then, computing the result is a matter of shifting, reversing and intersecting arithmetic progressions. The former two operations can easily be implemented in constant time, but the latter requires more attention. If the length of one of the sequences that we intersect is constant, we can simply verify which elements belong to the other. Also, if the sequences have the same difference, intersecting them in $O(1)$ time is trivial. It has been proved in [22] that if $|Occ(x', y)| \geq 3$ and $|Occ(y', x)| \geq 3$, then both of these sets form arithmetic progressions with the same difference. Thus, the two aforementioned special cases of intersection suffice for our needs.
Consequently, the data structure of Theorem 1 can answer **Prefix-Suffix Queries** in $O(1)$ time, which gives the following result.

**Theorem 5.** Using a data structure of $O(n)$ size, which can be constructed in $O(n)$ expected time, one can answer **Prefix-Suffix Queries** in $O(1)$ time.

**Corollary 4.** Using a data structure of $O(n)$ size, which can be constructed in $O(n)$ expected time, one can answer **Period Queries** in $O(\log |x|)$ time.

**Proof.** **Period Queries** can be answered using the data structure for **Prefix-Suffix Queries**. To compute all periods of $x$ we use **Prefix-Suffix Queries** to find all borders of $x$ (words which are simultaneously prefixes and suffixes of $x$) of length between $2^k - 1$ and $2(2^k - 1)$ for each $k \in [0, \lceil \log(|x| + 1) \rceil]$. Lengths of borders can be easily transformed to periods, since any word $x$ has period $p$ if and only if it has a border of length $|x| - p$.

**2-Period Queries**

Given a factor $x$ of $v$, decide whether $x$ is periodic and, if so, compute its shortest period.

While 2-Period Queries can be trivially reduced to **Prefix-Suffix Queries** (asking for borders of $x$ of length at least $\frac{|x|}{2}$), our techniques give a much simpler solution, with an additional merit in the form of deterministic construction algorithm.

**Corollary 5.** Using a data structure of $O(n)$ size, which can be constructed in $O(n)$ time, one can answer **2-Period Queries** in $O(1)$ time.

**Proof.** Recall that for any periodic factor $x$ we have defined $\text{run}(x)$ as the run extending $x$, and as $\bot$ if $x$ is not periodic. Lemma 5 gives a data structure computing $\text{run}(x)$ in $O(1)$ time, that can be constructed in $O(n)$ deterministic time. Moreover, if $x$ is periodic then $\text{per}(x) = \text{per(}\text{run}(x))$.

**11.2 Cyclic Equivalence Queries**

We define $\text{Rot}(u) = u[n]u[1] \ldots u[n-1]$. Additionally, for an integer $r$ we write $\text{Rot}(u, r)$ to denote $\text{Rot}$ applied $r$ times on $u$. Note that $\text{Rot}(u, r) = \text{Rot}(u, r')$ if $r \equiv r' \pmod{n}$. We say that $w$ is a **cyclic rotation** of $u$ if there exists an integer $r$ such that $w = \text{Rot}(u, r)$.

**Cyclic Equivalence Queries**

Given factors $x$ and $y$ of $v$, decide whether $x$ is a cyclic rotation of $y$ and, if so, report all corresponding cyclic shift values (represented as an arithmetic progression).
Before we proceed with a solution, let us state a stronger version of Fact 1. For words \(x, y\) denote the length of the longest common prefix of \(x\) and \(y\) by \(\text{lcp}(x, y)\).

**Fact 5.** Let \(v\) be a word of length \(n\). After \(O(n)\) preprocessing the following queries can be answered in \(O(1)\) time for any words \(x, y\) represented as concatenations of \(O(1)\) factors of \(v\):

(a) compute \(\text{lcp}(x, y)\),
(b) decide if \(x = y\),
(c) for an integer \(p \leq |x|\) find the longest prefix of \(x\) which has period \(p\).

**Proof.** We use the same data structure as for Fact 1, i.e. the suffix array with its inverse, and the LCP array equipped with the data structure for range minimum queries. A classic application of this toolbox is computing \(\text{lcp}\) for a pair of suffixes \([7, 14]\). A straightforward generalization of this algorithm lets us work with concatenations of a constant number of factors, which gives (a). Equality queries (b) are an immediate consequence of \(\text{lcp}\) queries. For (c) it suffices to observe that the answer is equal to \(p + \text{lcp}(x, x[p + 1, |x|])\).

Now, we are ready to present an algorithm for **Cyclic Equivalence Queries**. We can clearly assume that \(|x| = |y|\). Let us denote the common length of \(x\) any \(y\) by \(d\), and the desired set of cyclic shifts \(\{r \in [0, d - 1] : y = \text{Rot}(x, r)\}\) by \(R(x, y)\). The following observation not only is useful for computing \(R(x, y)\), but combined with Fact 3 also proves that this set indeed forms an arithmetic progression.

**Observation 4.** Let \(x, y\) be words of common length. Then \(R(x, y)\) is equal to the set of positions among \(\{0, \ldots, |y| - 1\}\) where \(x\) occurs in \(yy\).

Below, we give an algorithm which computes \(R(x, y) \cap [0,\lfloor \frac{d}{2}\rfloor]\), i.e. cyclic shifts not exceeding \(\frac{d}{2}\). Note that \(y = \text{Rot}(x, r)\) if and only if \(x = \text{Rot}(y, d - r)\), so running this algorithm both for \((x, y)\) and \((y, x)\) lets us easily retrieve \(R(x, y)\).

Let \(x'\) be the prefix of \(x\) of length \(\lceil \frac{d}{2}\rceil\). Note that any occurrence of \(x\) in \(yy\) at position \(\leq \frac{d}{2}\) induces an occurrence of \(x'\) in \(y\). We use **Factor-in-Factor Occurrence Queries** to find all positions where \(x'\) occurs in \(y\), each of them is a candidate shift value. If the number of occurrences is constant (at most 2), we can verify each candidate, using Fact 5(b) to test whether \(x\) actually occurs in \(yy\) at the appropriate position.

![Cyclic Equivalence Queries](Figure 8: Cyclic Equivalence Queries)

Otherwise, Fact 3 guarantees that the occurrences lie at positions \(j, j + p, \ldots, j + kp\) where \(p = \text{per}(x')\). We need to find out at which of these positions \(x\) actually occurs in \(yy\). We apply Fact 5(c) to find two values: \(\ell\), the length of the longest prefix of \(x\) which admits period \(p\), and \(m\), the length of the longest prefix of \(y[j, |y|]y\) which admits period \(p\) (see Figure 8). Observe that for any \(i \in [0, k]\) the longest prefix of \(y[j + pi, |y|]y\) which admits period \(p\) has length \(m - pi\). Consequently, we have two cases:
If \( \ell = |x| \) (\( p \) is a period of \( x \)), then \( x \) occurs in \( yy \) at all positions \( j + pi \) such that \( |x| \geq m - pi \).

Otherwise, among the candidates considered, \( x \) may occur in \( yy \) only at position \( j + pi \) with \( \ell = m - pi \). We check this candidate using Fact 5(b).

Thus, the data structure for Factor-in-Factor Occurrence Queries accompanied with the one of Fact 5 can answer Cyclic Equivalence Queries. This implies the following result.

**Theorem 6.** Using a data structure of \( O(n) \) size, which can be constructed in \( O(n) \) expected time, one can answer Cyclic Equivalence Queries in \( O(1) \) time.

### 11.3 Generalized Substring Compression

In this section we improve the results of [21] for Generalized Substring Compression Queries.

<table>
<thead>
<tr>
<th>Generalized Substring Compression Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given two factors ( x ) and ( y ) of ( v ), compute ( LZ(x</td>
</tr>
</tbody>
</table>

We actually provide a more efficient algorithm for the following auxiliary problem, introduced in [21].

<table>
<thead>
<tr>
<th>Bounded Longest Common Prefix Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given two factors ( x ) and ( y ) of ( v ), find the longest prefix ( p ) of ( x ) which is a factor of ( y ).</td>
</tr>
</tbody>
</table>

Before we proceed with a solution, we recall a number of tools related to suffix trees and mention several results developed in [21] for the original solution for Bounded Longest Common Prefix Queries.

#### 11.3.1 Tools

The suffix trie of \( v \) is the trie of all suffixes of \( v \). Each factor \( x \) of \( v \) corresponds to a unique node in the suffix trie, called the *locus* of \( x \), such that \( x \) is *spelled* by the letters on the path from the root to that node.

The suffix tree of \( v \) \([7, 14, 18]\), denoted \( T(v) \), is the compacted suffix trie of \( v \), i.e. nodes that are not branching (with 2 or more children) nor terminal (loci of suffixes of \( v \)) are dissolved. The dissolved nodes are called *implicit*, the remaining nodes are called *explicit*. An implicit node \( x \) can be represented as a pair \((u, d)\), where \( u \) is the lowest explicit descendant of \( x \), and \( d \) is the distance (the number of letters) from \( x \) to \( u \). The pair \((u, d)\) is called the *locus* of the factor corresponding to \( x \), and \( u \) is called its *explicit locus*.

The suffix tree of \( v \) takes linear space and can be constructed in linear time provided that the letters of \( v \) can be sorted in linear time \([7]\). The following result is due to Gawrychowski \([17]\).

**Lemma 18.** The suffix tree \( T(v) \) can be preprocessed in \( O(|v|) \) time so that given integers \( i, k \) the locus of the basic factor \( BF_k(i) \) can be determined in \( O(1) \) time.

We also use as a black-box several results developed in [21] for the original solution for Bounded Longest Common Prefix Queries. The result of Lemma 20 we have already referred to as a decision version of internal pattern matching queries.

<table>
<thead>
<tr>
<th>Interval Longest Common Prefix Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given an interval ([\ell, r]) and a factor ( x ) of ( v ), find the longest prefix ( p ) of ( x ) which occurs at some position ( t \in [\ell, r] ) in ( v ).</td>
</tr>
</tbody>
</table>
Lemma 19 ([21]). Using a data structure of $O(n + S_{rsucc})$ size, one can answer INTERVAL LONGEST COMMON PREFIX QUERIES in $O(Q_{rsucc})$ time, provided that $x$ in given by its locus in $T(v)$.

Lemma 20 ([21]). For a word $v$ of length $n$ there exists a data structure of size $O(n + S_{rempt})$, such that given factors $x, y$ one can decide whether $x$ occurs in $y$ in $O(Q_{rempt})$ time, provided that $x$ is given by its locus in $T(v)$.

11.3.2 Query Algorithm

Assume $x = v[\ell', r']$ and $y = v[\ell, r]$. First, we search for the largest $k$ such that the prefix of $x$ of length $2^k$ ($BF_k(\ell')$) occurs in $y$. We use a variant of binary search involving exponential search (also called galloping search), which requires $O(\log K)$ steps where $K$ is the optimal value of $k$. At each step for a fixed $k$ we need to decide if $BF_k(\ell')$ occurs in $y$. This can be done in $O(Q_{rempt})$ time: we find the locus of $BF_k(\ell')$ using Lemma 18 and then apply Lemma 20.

At this point we have an integer $K$ such that the optimal prefix $p$ has length $2^K \leq |p| < 2^{K+1}$. So far the complexity was $O(Q_{rempt} \log K) = O(Q_{rempt} \log \log |p|)$ time.

Let $p'$ be the prefix obtained from INTERVAL LONGEST COMMON PREFIX QUERY for $x$ and $[\ell, r-2^{K+1}]$. Note that $BF_{K+1}[\ell']$ does not occur in $x$, so $|p'| < 2^{K+1}$ and therefore the occurrence of $p'$ starting within $[\ell, r-2^{K+1}]$ lies within $y$. Thus $|p'| \leq |p|$; moreover, if $p$ occurs at a position within $[\ell, r-2^{K+1}]$, then clearly $p = p'$.

The other possibility is that $p$ occurs in $y$ only near its end, i.e. within the suffix of $y$ of length $2^{K+1}$, which we denote as $y'$. Let $x'$ be the prefix of $x$ of length $2^K$. Note that $x'$ is a prefix of $p$, so any occurrence of $p$ in $y'$ induces an occurrence of $x'$ in $y'$. We use FACTOR-IN-FACTOR OCCURRENCE QUERIES to locate all positions where $x'$ occurs in $y'$, these are the only possible positions where $p$ might occur in $y'$. If the number of these positions is constant (at most 2), we can verify each in constant time: it suffices to ask an $lcp$ query (see Fact 5(a)) for $x$ and the appropriate suffix of $y'$.

Otherwise we know that $x'$ is periodic and we know its period $q$, which by Fact 3 is equal to the difference in the arithmetic progression of occurrences. Let $y_r$ be the suffix of $y'$ starting with the $r$-th leftmost occurrence of $x'$ in $y'$. We compute two values using Fact 5(c): $d$, the length of the longest prefix of $x$ which admits period $q$, and $d_1$, the length of the longest prefix of $y_1$ which admits period $q$ (see Figure 9). Note that for $y_r$ the corresponding value is $d_r = d_1 - (r-1)q$.

![Figure 9: Bounded Longest Common Prefix Queries](image)

**Observation 5.** Let $u, u'$ be words such that lcp$(u, u') \geq q$. Let $d$ ($d'$) be the length of the longest prefix of $u$ (resp. $u'$) which has period $q$. If $d \neq d'$, then lcp$(u, u') = \min(d, d')$. Otherwise, lcp$(u, u') \geq d$. 

22
Observation 5 lets us restrict our attention to \( y_1 \) (which maximizes \( \min(d, d_r) \)) and \( y_r \) such that \( d = d_r \) (if any). Thus, even if there are more occurrences of \( x' \) in \( y' \), we need to consider only 2 of them.

Consequently, we can always choose the final solution as the best among three candidates: one obtained from the \textsc{Interval Longest Common Prefix Query}, and two corresponding to the occurrences of \( x' \) in \( y' \), with the actual lengths obtained using \textsc{lcp} queries.

Thus, the data structure for \textsc{Factor-in-Factor Occurrence Queries}, accompanied with the suffix tree \( T(v) \) and the data structure of Lemma 18, as well as the data structures of Lemmas 19, 20 and Fact 5, can answer \textsc{Bounded Longest Common Prefix Queries} in \( O(Q_{rsucc} + Q_{rempt} \log \log |p|) \) time.

\textbf{Lemma 21.} Using a data structure of \( O(n + S_{rempt} + S_{rsucc}) \) size, one can answer \textsc{Bounded Longest Common Prefix Queries} in \( O(Q_{rsucc} + Q_{rempt} \log \log |p|) \) time.

\textbf{Theorem 7.} Using a data structure of \( O(n + S_{rempt} + S_{rsucc}) \) size, one can answer \textsc{Generalized Substring Compression Queries} in \( O(C(Q_{rsucc} + Q_{rempt} \log \log |p|)) \) time, where \( C \) is the number of phrases reported.

\textit{Proof.} The algorithm for \textsc{Generalized Substring Compression Queries} is identical to the one presented in [21], it just uses our solution for \textsc{Bounded Longest Common Prefix Queries} instead of the original one. Thus, if the output phrases are of length \( p_1, \ldots, p_C \), it runs in \( O(\sum_{i=1}^{C}(Q_{rsucc} + Q_{rempt} \log \log |p_i|)) \) time, which using Jensen’s inequality for the concave function \( \log \log \) gives the desired time bound. \hfill \Box

\textbf{Acknowledgement}

The authors wish to thank Moshe Lewenstein for drawing their attention to \textsc{Generalized Substring Compression Queries}.

\textbf{References}


A Proof of Lemma 1(b)

Lemma 1. Assume that \( \pi \) is a \( \frac{\Delta}{2} \)-diverse sequence of length \( n \) over \([1,m]\) and let \( \pi \) be a permutation of \([1,m]\) drawn uniformly at random. Let \( A = \{ i : \pi(a_i) \leq \ell \} \) for a parameter \( \ell \), and \( C = \text{FillGaps}(A, \Delta, [1, n]) \). Then

(a) \( C \) contains all local \( \pi \)-minima,

(b) if \( \ell = \left\lfloor \frac{2m \log \Delta}{\Delta} \right\rfloor \), then \( \mathbb{E}(|C|) = \mathcal{O}(\frac{n \log \Delta}{\Delta}). \)

Before we give the proof, let us recall a standard fact and apply it in an auxiliary claim.

Fact 6. Let \( U \) be a set, \( T \) be its subset of size \( t \), and let \( S \) be drawn uniformly at random from the family of subsets of \( U \) of size \( s \). Then \( \mathbb{P}[S \cap T = \emptyset] \leq \left(1 - \frac{t}{|T|}\right)^s \leq \exp \left(-\frac{ts}{|T|}\right). \)

Claim 1. Let \( P \subseteq [1,n] \) be an interval of size \( \left\lceil \frac{\Delta}{2} \right\rceil \). Then \( \mathbb{P}[P \cap A = \emptyset] \leq \frac{2}{\Delta}. \)

Proof. Let \( V_P = \{ a_i : i \in P \} \). Observe that \( \frac{\Delta}{2} \)-diversity implies that \( |V_P| = |P| \leq m \). Note that \( P \cap A = \emptyset \) if and only if \( \pi(V_P) \cap [1, \ell] = \emptyset \), or equivalently \( V_P \cap \pi^{-1}([1, \ell]) = \emptyset \). Observe that \( L = \pi^{-1}([1, \ell]) \) is a subset of \([1,m]\) of size \( \ell \) drawn from the uniform distribution. Thus by Fact 6

\[
\mathbb{P}[P \cap A = \emptyset] = \mathbb{P}[V_P \cap L = \emptyset] \leq \exp \left( -\frac{\ell |P|}{m} \right) \leq \exp \left( -\frac{2m \log \Delta}{\Delta} - 1 \right) \frac{|P|}{m} = \exp \left( -\frac{2 |P| \log \Delta}{\Delta} + \frac{|P|}{m} \right) \leq \exp (-\log \Delta + 1) \leq \frac{2}{\Delta}. \]

Proof of Lemma 1(b). First, let us bound the expected size of \( A \).

\[
\mathbb{E}(|A|) = \sum_{j=1}^{n} \mathbb{P}[\pi(a_j) \leq \ell] = \sum_{j=1}^{n} \frac{\ell}{m} = \frac{n \ell}{m} = \frac{n}{m} \left\lfloor \frac{2m \log \Delta}{\Delta} \right\rfloor \leq \frac{2n \log \Delta}{\Delta}.
\]

Let us consider a position \( j \in C \setminus A \). By definition of FillGaps there must be an integer interval \( R \) such that \( j \in R \subseteq [1,n] \setminus A \) and \( |R| > \Delta \). Let us define \( R_{\leq j} = [j - \left\lfloor \frac{\Delta}{2} \right\rfloor + 1, j] \) and \( R_{\geq j} = [j, j + \left\lceil \frac{\Delta}{2} \right\rceil - 1] \). Note that \( R_{\leq j} \subseteq R \) or \( R_{\geq j} \subseteq R \). Moreover \( R_{\leq j} \subseteq R \) implies \( R_{\leq j} \subseteq [1,n] \) and \( R_{\leq j} \cap A = \emptyset \), by Claim 1 this holds with probability at most \( \frac{2}{\Delta} \). A similar reasoning holds for \( R_{\geq j} \). Therefore:

\[
\mathbb{E}(|C \setminus A|) = \sum_{j=1}^{n} \mathbb{P}[j \in C \setminus A] = \sum_{j=1}^{n} \frac{1}{\Delta} = \mathcal{O}\left(\frac{n}{\Delta}\right).
\]

Consequently \( \mathbb{E}(|C|) = \mathbb{E}(|A|) + \mathbb{E}(|C \setminus A|) = \mathcal{O}\left(\frac{n \log \Delta}{\Delta} + \frac{n}{\Delta}\right) = \mathcal{O}\left(\frac{n \log \Delta}{\Delta}\right). \)

B Abstract Data Structures

In this section with provide an efficient implementation of the data structure for successor queries, as well as two auxiliary data structures used as building blocks of our main construction: \textsc{Evaluator} and \textsc{Locator}. Thus we prove Lemmas 10, 3 and 4. We start with a description of auxiliary data structures used in the first two lemmas. To prove the third lemma, we use perfect hashing [16].
B.1 Rank & Select

For a bit vector $B$ we define $\text{rank}_B(i)$ as the number of positions $j \leq i$ such that $B[j] = 1$, and $\text{select}_B(i)$ as the position of the $i$-th 1-bit in $B$, i.e. minimum $j$ such that $\text{rank}_B(j) = i$, $\bot$ if no such position exists.

While there are many data structures computing $\text{rank}$ and $\text{select}$ (the classic ones can be found in [19, 5, 30]), most of them focus on space and query time bounds and do not provide an efficient construction algorithm. The word-RAM model allows to give the input bit vector $B$ of length $n$ in $\mathcal{O}(\frac{n}{\log n})$ machine words, thus a desired construction time is $\mathcal{O}(\frac{n}{\log n})$. We are not aware of any paper or book providing a construction algorithm running in that time, and thus for the sake of completeness we describe such an algorithm below.

Lemma 22. For a bit vector $B$ of length $n$ one can construct in $\mathcal{O}(\frac{n}{\log n})$ time a data structure of size $\mathcal{O}(\frac{n}{\log n})$ that can answer $\text{rank}_B$ and $\text{select}_B$ queries in constant time.

Proof. First, note that answers to all possible $\text{rank}$ and $\text{select}$ queries for bit vectors of size $\ell = \lceil \frac{n}{2} \log n \rceil$ can be memoized in $\mathcal{O}(n^{1/2+\varepsilon}) = o\left(\frac{n}{\log n}\right)$ time and space.

Let us divide $B$ into blocks $B_1, \ldots, B_m$ of length $\ell$ (if $B_m$ is shorter, we append $B_m$ with zeros). We can arrange $B$ so that each block corresponds to a single machine word.

For $\text{rank}_B$ queries, we simply memoize the answers for positions divisible by $\ell$. For this, we use the precomputed results of $\text{rank}$ queries to count ones in each block and then compute prefix sums of such a sequence. This clearly takes $\mathcal{O}(\frac{n}{\log n})$ time and space. To answer a query for a position $j$, we add up the result for the largest position divisible by $\ell$ not exceeding $j$ and the memoized result of a corresponding query in the block containing the $j$-th position.

The data structure for $\text{select}$ queries is more involved. For each $i \in [1, m]$ we store the index $j$ of the $i$-th non-zero block (if it exists). We define a bit vector $D_B$, such that $D_B[i] = 1$ if the $i$-th 1-bit in $B$ is the first 1-bit in its block. Such a bit vector can be computed in $\mathcal{O}(\frac{n}{\log n} + \frac{n}{\ell})$ time: it suffices to start with a null vector and flip all bits of index $\text{rank}_B(k\ell) + 1$ for $k \in [0, \lceil \frac{n}{\ell} \rceil]$.

Observe that $\text{rank}_{D_B}(i) = j$ implies that the $i$-th 1-bit of $B$ lies in $j$-th non-zero block of $B$, whose index among all blocks has been precomputed. Once we know this index $k$ and $\text{rank}_B((k-1)/\ell)$, a $\text{select}_B$ query reduces to a $\text{select}$ query within the block, for which we have a memoized result.

In total the components of the data structure for $\text{select}$ queries clearly take $\mathcal{O}\left(\frac{n}{\log n}\right)$ space and time to construct.

B.2 Successor Queries

Lemma 10. For an arbitrary set $R \subseteq [1, n]$ there exists a data structure $S(R)$ of size $\mathcal{O}(\frac{n}{\log n})$, which answers successor queries on $R$ ($\text{succ}_R(i) = \min(R \cap [i, n])$) in $\mathcal{O}(1)$ time. Moreover $S(R)$ can be constructed in $\mathcal{O}(\frac{n}{\log n})$ time if $R$ is given as a bit vector.

Proof. Let $B$ be the bit vector representing $R$. Observe that $\text{succ}_R(i) = \text{select}_B(\text{rank}_B(i-1) + 1)$ (with $\text{rank}_B(0)$ defined as 0, and $\text{select}_B(j)$ defined as $\infty$ for $j$ greater than the number of 1-bits in $B$). Thus, it suffices to use the data structure for $\text{rank}_B$ and $\text{select}_B$ queries provided in Lemma 22.

\hfill \square
Lemma 3. For \( g \) specified in a piecewise constant representation of size \( m \), there exists an evaluator \( \mathcal{E}(g) \) of size \( \mathcal{O}(m + \frac{n}{\log n}) \) that answers queries in \( \mathcal{O}(1) \) time and can be constructed in \( \mathcal{O}(m + \frac{n}{\log n}) \) time.

Proof. Let \( R = \{(\ell_1, r_1, v_1)\} \) be the piecewise constant representation of \( g \). We store the values \( v_i \) in an array indexed with \( i \). Let us define a bit vector \( B_R \) with \( B_R[j] = 1 \) if and only if \( j = \ell_i \) for some \( i \). \( B_R \) will be represented in \( \mathcal{O}(\frac{n}{\log n}) \) words. Clearly \( B_R \) can be constructed in \( \mathcal{O}(\frac{n}{\log n} + m) \) time starting from a null vector and setting \( B_R[\ell_i] = 1 \) for all \( i \). Observe that \( g(j) = v_i \) where \( i = \text{rank}_{B_R}(j) \).

We build the data structure for \( \text{rank} \) queries in \( B_R \), applying Lemma 22. In total we obtain the desired \( \mathcal{O}(\frac{n}{\log n} + m) \) bounds on the space and construction time, and constant-time queries.

B.4 Locator

Lemma 4. For a family \( A = (A_i) \) of \( d \)-sparse subsets of \([1, n]\). There exists a locator \( \mathcal{L}(A) \) of size \( \mathcal{O}(\sum_i |A_i|) \) that can answer queries in \( \mathcal{O}(1) \) time. It can be constructed in \( \mathcal{O}(\sum_i |A_i|) \) time given \( \{(i, j) : j \in A_i\} \).

Proof. We divide the universe \([1, n]\) into blocks \( B_\ell = [\ell d, (\ell + 1)d - 1] \) for \( \ell \in \mathbb{Z} \). The data structure is based on perfect hashing, see [16]. For each \( j \in A_i \) we store an item with key \( \langle \frac{j}{d}, i \rangle \) and value \( (j, i) \). For a query we extend \( P \) to \( P' \) such that \( P' \) is composed of full blocks. Note that \( |P'| \leq |P| + 2d = \mathcal{O}(d) \). Let \( P' = [\ell d, \ell' d - 1] \). For each \( m \in [\ell, \ell' - 1] \) (note that there are \( \mathcal{O}(1) \) such values) we retrieve all items with the key \( (m, i) \). Clearly this gives \( \{(j, i) : j \in P' \cap A_i\} \). The size of this set is constant by the sparsity condition. Now it suffices to filter out pairs with \( j \in P' \setminus P \) and return \( j \) for the remaining ones.

C Algorithmic Tools for Periodic Case

In this section we provide proofs of three lemmas of algorithmic nature that we used in the solution of the periodic case of FACTOR-IN-FACTOR OCCURRENCE QUERIES. Before that we present several more facts related to periodicities. The first one is a classic result.

Lemma 23 (Three Squares Lemma [13, 9]). Let \( v_1, v_2, v_3 \) be words such that \( v_1^2 \) is a prefix of \( v_2^2 \), \( v_2^2 \) is a prefix of \( v_3^2 \) and \( v_1 \) is primitive. Then \( |v_1| + |v_2| \leq |v_3| \).

Fact 7. \( \sum_k \sum_{v \in \mathcal{R}_k(v)} \frac{|A|}{2^k} = \mathcal{O}(n) \); \( \sum_k |\mathcal{R}_k(v)| = \mathcal{O}(n) \).
Proof. Let us fix a run $\alpha$. We have

$$
\sum_{k: \alpha \in \mathcal{R}_k(v)} \frac{|\alpha|}{2^k} = \sum_{k: \text{per}(\alpha)<2^k} \frac{|\alpha|}{2^k} = \sum_{k=\lceil \log \text{per}(\alpha) \rceil+1}^{\infty} \frac{|\alpha|}{2^k} = \frac{|\alpha|}{2^{\lceil \log \text{per}(\alpha) \rceil}} \leq \frac{|\alpha|}{2^{\text{per}(\alpha)}} = \exp(\alpha).
$$

Summing up over all runs and applying the bound for the sum of exponents of runs (Fact 2), we get (a). As for part (b), if $\alpha$ is a $k$-run, then $|\alpha| \geq 2^k$, so $|\alpha| \geq 1$. Thus (b) is a consequence of (a):

$$
\sum_k |\mathcal{R}_k(v)| \leq \sum_k \sum_{\alpha \in \mathcal{R}_k(v)} \frac{|\alpha|}{2^k} = \mathcal{O}(n).
$$

\[\square\]

Observation 6. Let $\alpha \neq \alpha'$ be runs with period $p$. Then $|\alpha \cap \alpha'| < p$.

Definition 4. We say that a word $u$ is $k$-periodic, if $u$ is periodic, $|u| \geq 2^k$ and $\text{per}(u) < 2^k$.

Fact 8. Let $u$ be a $k$-periodic fragment of $v$. Then $\text{run}(u)$ is the unique run $\alpha$ such that $\text{per}(\alpha) \leq |u| - \text{per}(u)$ and $u \cap \alpha = u$. Moreover $\text{run}(u)$ is a $k$-run and $\text{per}(\text{run}(u)) = \text{per}(u) \leq \frac{|u|}{2}.$

Proof. The latter statement is an immediate consequence of the definitions. Assume that $\beta$ is a different run satisfying the aforementioned conditions. Note that both $\text{per}(\alpha)$ and $\text{per}(\beta)$ are periods of $u$ and $|\alpha| + |\beta| \leq |u|$. Periodicity Lemma (Lemma 6) implies that $\text{per}(\alpha) = \text{per}(\beta)$. However, $u$ is a subfragment of $\alpha \cap \beta$, which leads to a contradiction by Observation 6. \[\square\]

Lemma 24. Let $u_1, u_2, u_3$ be $k$-periodic fragments of $v$, all starting at the same position $i$. Then $\text{run}(u_1), \text{run}(u_2), \text{run}(u_3)$ cannot be all distinct.

Proof. For a proof by contradiction assume that runs $\alpha_i = \text{run}(u_i)$ are pairwise distinct. Observation 6 implies that these runs must have pairwise distinct periods $p_i$. Without loss of generality assume $p_1 < p_2 < p_3$, i.e. $u_1, u_2, u_3$ are three periodic factors of length at least $2^k$ with different shortest periods $p_1 < p_2 < p_3 < 2^k$ starting at the position $i$ in $v$. By the Three Squares Lemma (Lemma 23) we conclude that $p_1 + p_2 \leq p_3 < 2^k$. Now consider the $k$-basic fragment $u$ starting at position $i$. We have $2p_1 < 2^k$, therefore $u$ is a $k$-periodic fragment. Observe that both $\alpha_1$ and $\alpha_2$ satisfy the statement of Fact 8, which implies that $\alpha_1 = \text{run}(u) = \alpha_2$, a contradiction. \[\square\]

Fact 9. Any position lies within at most 2 runs of period $p$.

Proof. Consider any three distinct runs $\alpha_1 = v[i_1, j_1], \alpha_2 = v[i_2, j_2]$ and $\alpha_3 = v[i_3, j_3]$ with period $p$. Assume that $i_1 \leq i_2 \leq i_3$. From Observation 6 we get

$$
i_3 > j_2 - p + 1 = j_2 - i_2 + 1 - p + i_2 = |\alpha_2| - p + i_2 \geq p + i_2 > p + j_1 - p + 1 > j_1.
$$

Thus $i_3 > j_1$ which means that $\alpha_1$ and $\alpha_3$ do not intersect. \[\square\]

C.1 Proofs of Algorithmic Lemmas for Periodic Case

Lemma 5. There exists a data structure of $\mathcal{O}(n)$ size, which given a fragment $u$ returns $\text{run}(u)$ in constant time. Moreover, the data structure can be constructed in $\mathcal{O}(n)$ time.
Proof. Consider a function \( R_k : [1, n_k] \rightarrow 2^{\mathcal{R}_k(v)} \) which assigns to a position \( i \) the set of \( k \)-runs inducing a \( k \)-periodic fragment starting at position \( i \) (see Definition 4). Lemma 24 implies that \( |R_k(i)| \leq 2 \) for each \( i \). Note that for \( \alpha = v[i, j] \) we have \( \alpha \in R_k(i') \) if and only if \( i' \in [i, \min(j - \per(\alpha) - 2, j - 2^k) + 1] \).

For a \( k \)-run \( \alpha = v[i, j] \) define \( \text{beg}_k(\alpha) = i \) and \( \text{end}_k(\alpha) = \min(j - 2\per(\alpha), j - 2^k) \). Observe that \( R_k(i) \neq R_k(i - 1) \) implies \( i = \text{beg}_k(\alpha) \) or \( i = \text{end}_k(\alpha) \) for some \( k \)-run \( \alpha \). Thus \( R_k \) admits a piecewise constant representation of size at most \( 2|\mathcal{R}_k(v)| \).

Summing up over all \( k \) and applying Fact 7 this implies that the total size of representation of \( R_k \) is \( \mathcal{O}(n) \), which means that evaluators \( E(R_k) \) take \( \mathcal{O}(n) \) space in total. The piecewise constant representation of \( R_k \) can easily be constructed by an algorithm which traverses the word from left to right maintaining a set \( A \) of (at most 2) \( k \)-runs. At each position \( i \) it removes from \( A \) all \( k \)-runs \( \alpha \) with \( \text{end}_k(\alpha) = i \) and adds those with \( \text{beg}_k(\alpha) = i \). Such events can be prepared and sorted in \( \mathcal{O}(n) \) time simultaneously for all \( k \).

We answer queries as follows. If \( u = v[i, j] \) is periodic, then it is \( k \)-periodic for \( k = \lfloor \log |u| \rfloor \), so the run inducing \( v[i, j] \) belongs to \( R_k(i) \). We use \( E(R_k) \) to find all such \( k \)-runs in constant time. Finally, we check if \( \alpha = \text{run}(u) \) using the characterization of Fact 8, i.e. verifying that \( \alpha \) covers \( u \) and \( \per(\alpha) \leq \frac{|u|}{2^k} \).

---

**k-Run Locator**

**Input:** A word \( v \) of length \( n \).

**Queries:** Given an integer \( p \) and a range \( P \subseteq [1, n] \) with \( |P| = \mathcal{O}(2^k) \), compute all \( \alpha \in \mathcal{R}_k(v) \) for which \( \per(\alpha) = p \) and \( \alpha \cap P \neq \emptyset \).

---

**Lemma 8.** There exist \( k \)-run locators \( K_k(v) \) that answer queries in \( \mathcal{O}(1) \) time, take \( \mathcal{O}(n) \) space in total, and can be constructed in \( \mathcal{O}(n) \) expected total time.

**Proof.** The data structure is similar to **Locator**, see Lemma 4. Let us divide \([1, n]\) into blocks \( B_1, \ldots, B_m \) of size \( 2^k \) (the last one possibly shorter). Note that \( m = \mathcal{O}(\frac{n}{2^k}) \). We build a hash table, for each \( k \)-run \( \alpha \) and each index \( i \) such that \( B_i \) overlaps \( \alpha \) we store an item with key \((i, \per(\alpha))\) and value \( \alpha \). Note that the number of items is bounded by \( \sum_{\alpha \in \mathcal{R}_k(v)} \left( \frac{|\alpha|}{2^k} + 2 \right) \), which when summed over all \( k \) is \( \mathcal{O}(n) \) by Fact 7. Moreover, by Fact 9 there are at most 4 values for a fixed key. Indeed, any \( k \)-run intersecting a fixed block \( B_i \) must contain the first or the last position in that block, and each of these positions might be contained in at most two \( k \)-runs of period \( p \).

For a query range \( P \) we extend \( P \) to \( P' \) so that \( P' = B_i \cup \ldots \cup B_j \) (with \( j - i = \mathcal{O}(1) \)) and find all \( k \)-runs of period \( \alpha \) overlapping \( P' \). As we have just shown, there are \( \mathcal{O}(1) \) such \( k \)-runs, so we can easily check which of them overlap with \( P \), also duplicates can be removed in constant time.

---

**Lemma 9.** For \( k \in [0, \lfloor \log n \rfloor] \) let \( P_k = \{ i \in [1, n_k] : \text{BF}_k(i) \text{ is periodic} \} \). In \( \mathcal{O}(n) \) time we can compute all sets \( P_k \), each of them represented both in the block representation and as a bit vector.

**Proof.** Each periodic \( k \)-basic fragment is induced by a unique \( k \)-run. Moreover, for a fixed \( k \)-run \( \alpha \) the set of positions where \( k \)-basic fragments induced by \( \alpha \) start, forms an interval. If \( \alpha = v[i, j] \), the interval is \([i, j - 2^k + 1]\) if \( \per(\alpha) \leq 2^{k-1} \) or \( \emptyset \) otherwise.

In order to compute the block representations of \( P_k \), it suffices to sort these intervals and join some of them so that we get blocks of \( P_k \). This takes \( \mathcal{O}(n + \sum_k |\mathcal{R}_k(v)|) = \mathcal{O}(n) \) time in total by Fact 7. The bit vector of \( P_k \) can be obtained from a block representation in time proportional to the
the implementation of CompactDBF that linked list of terminal nodes to visit them with $O(u)$ in the subtree rooted in explicit locus $u$. The number of events is linear, so in $O(Bv)$ from $Bv$ suffix the data structure for rank $u$ explicit locus, forms a range. We can construct such ranges for each explicit node. For a range $u$ and then determine its identifier using a rank query. In total we obtain $O(n)$ construction time over all $k$'s.

D Algorithmic Tools for Construction Algorithm

In this section we present two of algorithmic tools which we use for the construction algorithm, therefore proving Lemmas 12, 13 and 16.

D.1 Compact DBF and Randomized DBF

Recall the definitions of the suffix tree and loci as well as Lemma 18, all given in Section 11.3.1. We say that an explicit node $u$ is a $k$-basic node if $u$ is an explicit locus of a $k$-basic factor. Observe that there is a natural bijection between identifiers in DBF$_k$ and the $k$-basic nodes. While storing it explicitly for all $k$ takes too much space, we shall devise an alternative way to evaluate it.

There are up to $2n$ explicit nodes, let us assign them pre-order identifiers $id$. Note that such identifiers also preserve lexicographic order of the corresponding factors. We have the following observation.

**Observation 7.** If the explicit locus of BF$_k(i)$ is $u$, then DBF$_k[i]$ is the number of $k$-basic nodes with identifiers not exceeding $id(u)$.

Due to the observation, it suffices to store a bit vector $B_k$ such that $B_k[i] = 1$ if and only if the explicit node $u$ with $id(u) = i$ is $k$-basic. Then rank queries on $B_k$ (see Appendix B) can be used to determine the identifier of $u$ in DBF$_k$. Similarly, select queries on $B_k$ let us find the $id$ of the node which corresponds to a given identifier in DBF$_k$. In order to be able to report the occurrences of the $k$-basic factor with $O(1)$-time delay, for each explicit node we maintain pointers to the leftmost and rightmost terminal node in the corresponding subtree. Additionally, we maintain a linked list of terminal nodes in lexicographic order of their labels.

We use Lemma 22 to efficiently construct the data structures for rank and select queries on $B_k$, but first we need to determine these bit vectors. Note that a single node can be $k$-basic for many values of $k$, but these values form a range, since the set of lengths of factors, for which $u$ is an explicit locus, forms a range. We can construct such ranges for each explicit node. For a range $[k_1, k_2]$ of a node $u$, $id(u) = i$, we construct events $(k_1, i)$ and $(k_2 + 1, i)$. Then $B_k$ can be computed from $B_{k-1}$ by flipping all bits $i$ for which $(k, i)$ is an event, with $B_{-1}$ defined as a null vector. The number of events is linear, so in $O(n)$ time we can construct all vectors $B_k$ and equip them with the data structure for rank and select queries.

Now, answering queries is simple: for (1) we find the explicit locus of BF$_k(i)$ using Lemma 18, and then determine its identifier using a rank$_B_k$ query. For (2) we use a select$_B_k$ query to get an explicit locus $u$. Note that the corresponding basic factor occurs at position $i$ if and only if the suffix $v[i, n]$ has its locus in the subtree rooted in $u$. Thus, it suffices to visit all terminal nodes in the subtree rooted in $u$. We use the pointers to leftmost and rightmost terminal node and the linked list of terminal nodes to visit them with $O(1)$-time delay. Finally for (3) it suffices to note that $m_k$ is the number of $k$-basic nodes, which is the total number of 1-bits in $B_k$. This concludes the implementation of COMPACTDBF and gives the announced result.
Lemma 12. For a word $v$ of length $n$ there exists COMPACTDBF $D(v)$ which takes $O(n)$ space, can be constructed in $O(n)$ time and can answer (1) and (3) queries in $O(1)$ time, and (2) queries with $O(1)$ time delay per item reported.

D.1.1 RandomizedDBF

Lemma 13. For a word $v$ of length $n$ there exists a RANDOMIZEDDBF $D^\star(v)$ which takes $O(n)$ space, can be constructed in $O(n)$ expected time and can answer (1) and (3) queries in $O(1)$ time, and (2) queries in with $O(1)$ time delay per item reported.

Proof. To obtain random identifiers, it suffices to randomly shuffle identifiers $id$ in the previous construction (in particular the bit vectors $B_k$ are indexed using these random identifiers). Then for each $k$ the identifiers of $k$-basic nodes also form a random order, since a (uniformly) random order of a set induces a uniformly random order of any subset. \qed

D.2 Slider Function

Function $Slider$

Input: Positive integers $d \leq m$ and a set $A$ of pairs $(q, p)$ with $q \in \mathbb{Z}$ and $p \in [1, m]$.

Output: A piecewise constant representation of $G : [1, m - d] \rightarrow A$ defined as follows: $G(i)$ is the lexicographically smallest pair $(q, p) \in A$ among pairs with $p \in [i, i + d]$, $\perp$ if no such pair exists.

Lemma 16. $Slider$ can be implemented in $O(|A|)$ time, provided that pairs in $A$ are sorted by $p$ in the input.

Before we proceed with the proof let us state a folklore result.

Fact 10. A simple queue can be augmented so that it can return its minimal element (settling ties arbitrarily) and all operations $enqueue$, $dequeue$ and $find-min$ on the queue work in $O(1)$ amortized time.

Proof of Lemma 16. We traverse all values $i \in [1, m - d]$ in the increasing order maintaining $Q_i = \{(q, p) \in A : p \in [i, i + d]\}$ stored in the augmented queue of Fact 10, with minima computed on pairs lexicographically.

Note that $G(i) = \min Q_i$, moreover $Q_i$ can be obtained from $Q_{i-1}$ by $enqueueing$ all pairs with their second coordinate equal to $i + d$ and $dequeueing$ all pairs with their second coordinate equal to $i - 1$. We can store these operations as events and sort the events (merging the lists of events of both kinds), so that we perform any work only for $i$ with some events associated. For every such $i$ we evaluate $G(i)$ using the $find-min$ query, and if the value is different than previously, we start a new interval in the piecewise constant representation of $G$. \qed