Locally Consistent Parsing for Text Indexing in Small Space

Or Birenzwige
Bar Ilan University
birenzo@cs.biu.ac.il

Shay Golan
Bar Ilan University
golansh1@cs.biu.ac.il

Ely Porat
Bar Ilan University
porately@cs.biu.ac.il

Abstract

We consider two closely related problems of text indexing in a sub-linear working space. The first problem is the Sparse Suffix Tree (SST) construction, where a text $S$ is given in a read-only memory, along with a set of suffixes $B$, and the goal is to construct the compressed trie of all these suffixes ordered lexicographically, using only $O(|B|)$ words of space. The second problem is the Longest Common Extension (LCE) problem, where again a text $S$ of length $n$ is given in a read-only memory with some parameter $1 \leq \tau \leq n$, and the goal is to construct a data structure that uses $O(n^{\tau})$ words of space and can compute for any pair of suffixes their longest common prefix length. We show how to use ideas based on the Locally Consistent Parsing technique, that was introduced by Sahinalp and Vishkin [35], in some non-trivial ways in order to improve the known results for the above problems. We introduce new Las-Vegas and deterministic algorithms for both problems.

For the randomized algorithms, we introduce the first Las-Vegas SST construction algorithm that takes $O(n)$ time. This is an improvement over the last result of Gawrychowski and Kociumaka [19] who obtained $O(n)$ time for Monte-Carlo algorithm, and $O(n\sqrt{\log |B|})$ time for Las-Vegas algorithm. In addition, we introduce a randomized Las-Vegas construction for a data structure that uses $O(n^{\tau})$ words of space, can be constructed in linear time and answers LCE queries in $O(\tau)$ time.

For the deterministic algorithms, we introduce an SST construction algorithm that takes $O(n(\log \frac{n}{|B|} + \log* n))$ time (for $|B| = \Omega(\log n \log* n)$). This is the first almost linear time, $O(n \cdot \text{polylog} n)$, deterministic SST construction algorithm, where all previous algorithms take at least $\Omega\left(\min\{n|B|, \frac{n^2}{|B|}\}\right)$ time. For the LCE problem, we introduce a data structure that uses $O(n^{\tau})$ words of space and answers LCE queries in $O(\tau \sqrt{\log* n})$ time, with $O(n(\log \tau + \log* n))$ construction time (for $\tau = O(\frac{n}{\log n \log* n})$). This data structure improves both query time and construction time upon the results of Tanimura et al. [37].
1 Introduction

Text indexing is one of the most fundamental problems in the area of string algorithms and information retrieval (e.g. see [40, 24, 39, 30, 13, 3, 15, 16, 6, 17, 1, 26, 23, 8, 34]). In this paper we consider two closely related problems of text indexing in sub-linear working space. In both problems, we are given a string $S$ of length $n$ to process, in a read-only memory. The first problem is the Sparse Suffix Tree (SST) construction which is a generalization of the classical Suffix Tree data structure. The Suffix Tree index of $S$ is a compact trie of all $S$’s suffixes. Since the 70’s, several algorithms were introduced for the suffix tree construction which takes $O(n)$ time [40, 30, 39]. In the spare suffix tree, only a subset of suffixes, $B \subseteq [n]$, need to be indexed. This generalization was introduced in 1996 by Kärkkäinen and Ukkonen [24] who introduced a construction algorithm for the special case where all the suffixes are in the positions which are integer multiples of some $k \in \mathbb{N}$. Their algorithm takes $O(n)$ time and uses linear working space in the number of suffixes. For the general case of arbitrary set of suffixes, one can construct an SST by pruning of the full Suffix Tree, which requires $\Theta(n)$ words of working space. However, when the space usage is limited to be linear in the number of desired suffixes, another approach is required. Recently, Gawrychowski and Kociumaka [19] introduced an optimal randomized Monte-Carlo algorithm which builds an SST in linear time, using only $O(|B|)$ words of space.

The second problem we consider in this paper is the Longest Common Extension (LCE) problem. In this problem we are given a text $S$ of length $n$ in a read-only memory, and the goal is to construct a data structure that can compute for any pair of suffixes their longest common prefix length, or formally $\text{LCE}(i,j) = \min\{k \geq 0 \mid S[i+k] \neq S[j+k]\}$. It is well known that one can solve this problem using $O(n)$ words of space with $O(n)$ construction time and $O(1)$ query time. We consider the trade-off version, where there is a parameter $1 \leq \tau \leq n$ and the goal is to construct a data structure using only $O(\frac{n}{\tau})$ words of space, while achieving fast construction and query time. The trade-off LCE problem is closely tied to the SST problem, and in several papers the LCE was used as the main tool for constructing the SST efficiently [19, 3].

For the LCE problem, there is a lower bound in the cell-probe model, by Kosolobov [28], who proved that any data structure for this problem that uses $S(n) = \Omega(n)$ bits of space and $T(n)$ query time, must have $S(n) \cdot T(n) = \Omega(n \log n)$ for large enough alphabet. Kosolobov also conjectured that for data structure who used $S(n) < o(n)$ bits of space a similar lower bound of $T(n) \cdot S(n) = \Omega(n \log S(n))$ holds. A trade-off of $O(\tau)$ query time and $O(\frac{n}{\tau})$ words of space is already known, both in randomized and deterministic construction algorithms [4], and therefore the research efforts are invested in reducing the construction time. Tanimura et al. [37] introduced a deterministic data structure with $O(n\tau)$ construction time algorithm, and with a slightly increasing of the query time to $O(\tau \min\{\log \tau, \log \frac{n}{\tau}\})$. Gawrychowski et al. [19] have shown a linear time Monte-Carlo construction of an LCE data structure with $O(\tau)$ query time. In a close model of encoding data structures, Tanimura et al. [38] introduce a data structure that encode the string, and then can delete the original string. The complexity of the data structure is dependent on the Lempel-Ziv 77 factorization of the string, and in some cases it can beat the lower bound of Kosolobov [28].

1.1 Our Contribution

**LCE data structures.** We provide new LCE data structures that answer queries deterministically, and uses only $O(\frac{n}{\tau})$ words of space. The construction algorithm are either deterministic or randomized Las-Vegas algorithms. Our results are stated in the following theorems, see full comparison in Appendix A.
Theorem 1.1 (Las-Vegas Algorithm). For any \(1 \leq \tau \leq O\left(\frac{n}{\log n}\right)\) there is a data structure of size \(O(\frac{n}{\tau})\) words that can be constructed in \(O(n)\) time with high probability and answers LCE queries in \(O(\tau)\) time. For \(\tau = \omega\left(\frac{n}{\log n}\right)\) the same data structure can be constructed in expected \(O(n)\) time.

For the deterministic trade-off, we begin with a data structures with the following complexities.

Lemma 1.2 (Deterministic Algorithm). For any \(1 \leq \tau \leq O\left(\frac{n}{\log n \log \frac{n}{\tau}}\right)\) there is a data structure of size \(O(\frac{n}{\tau})\) words which can be constructed deterministically in \(O(n(\log \tau + \log^* n))\) time and answers LCE queries in \(O(\tau \log^* n)\) time.

Then, we improve the results as stated in the following theorem.

Theorem 1.3 (Deterministic Algorithm). For any \(1 \leq \tau \leq O\left(\frac{n}{\log n \log \frac{n}{\tau}}\right)\) there is a data structure of size \(O(\frac{n}{\tau})\) words which can be constructed deterministically in \(O(n(\log \tau + \log^* n))\) time and answers LCE queries in \(O(\tau \sqrt{\log^* n})\) time.

**Sparse Suffix Tree Construction.** For the SST construction problem we introduce an optimal randomized Las-Vegas algorithm, which improves upon the previous algorithm of Gawrychowski and Kociumaka [19] which developed an optimal Monte-Carlo algorithm and a Las-Vegas algorithm with additional \(\sqrt{\log |B|}\) factor. In addition, we introduce the first almost-linear deterministic SST construction algorithm. Our results are stated in the following theorems.

Theorem 1.4. Given a set \(B \subseteq [n]\) of size \(\Omega(\log^2 n) \leq |B| \leq n\), there exists a randomized Las-Vegas algorithm which constructs the Sparse Suffix Tree of \(B\) in \(O(n)\) time with high probability using only \(O(|B|)\) words of space. For \(|B| = o(\log^2 n)\) there exists a randomized Las-Vegas algorithm with the same complexities except that the running time is just expected to be \(O(n)\).

Theorem 1.5. Given a set \(B \subseteq [n]\) of size \(\Omega(\log n \cdot \log^* n) \leq |B| \leq n\), there exists a deterministic algorithm which constructs the Sparse Suffix Tree of \(B\) in \(O(n(\log \frac{n}{|B|} + \log^* n))\) time using only \(O(|B|)\) words of space.

### 1.2 Algorithmic Overview

We first give an overview on the LCE data structures, and then describe how to use them (with additional techniques) to build an SST using only a small amount of space. For both problems we use a parameter \(1 \leq \tau \leq n\), such that the working space of the data structures and the construction algorithms are limited to \(O(\frac{n}{\tau})\) words of space.

Since the naïve way to compute \(\text{LCE}(i,j)\) takes \(O(\text{LCE}(i,j))\) time, by just comparing pairs of characters until a mismatch is found, the case where \(\text{LCE}(i,j) = O(\tau)\) is solvable in \(O(\tau)\) time without any preprocessing. Hence, we invest our efforts on speeding up the computation of queries where \(\text{LCE}(i,j) = \ell > \omega(\tau)\). In this case, we use a special partitioning of \(S\) into blocks. The partitioning has the property that if a string appears as a substring of \(S\) multiple times, then all these substrings are composed of the same blocks exactly, except for possibly small margins of the substrings. More detailed, if \(S[i..i + \ell - 1] = S[j..j + \ell - 1]\) is long enough, then both \(S[i..i + \ell - 1]\) and \(S[j..j + \ell - 1]\) can be considered as composed of three parts, a prefix, a suffix and the middle part between them with the following properties. Both the prefix and suffix are short strings (e.g. of length \(O(\tau)\) for the randomized algorithm), or they have some other special properties, and the middle part is composed of full blocks, which are exactly the same in both substrings, regardless the length of this part. Thus, using this partitioning, the computation of \(\text{LCE}(i,j)\) is done in three phases: we first verify that \(\text{LCE}(i,j)\) is long enough and if so - we find the beginning of the common
middle part. Then, we compute the length of the middle part (which might be very long) by a
-designated data structure that performs this computation in constant time. At the end we just
compute the length of the short suffix, right after the end of the last block of the middle.

The key concepts of this technique are similar to the Locally Consistent Parsing of Sahinalp
and Vishikin [36, 10] and also to the techniques of Kociumaka et al. [27]. However, we design a new
way to combine Karp-Rabin fingerprints together with approximately min-wise hash functions,
in order to use this method using only small amount of space. In previous results for the LCE
problem [3, 4, 19, 37], a set of positions of size $O(\frac{n}{\tau})$ was also considered by the data structures. In
contrast to these algorithms, where the selected positions were dependent only on the length of the
text, we introduce the first algorithm that exploit the actual text using local properties in order to
decide which positions to select.

In order to achieve the desired special partitioning of the string into blocks, our algorithms pick
positions in $S$, which will be the endpoints of the blocks. Our goal is to pick $O(\frac{n}{\tau})$ positions, such
that the distance between any two consecutive positions is $O(\tau)$ and, for the locality, the decision
whether a position $p \in \{1, 2, \ldots, n\} = [n]$ is picked, depends only on the neighborhood of $p$ up to
$O(\tau)$ characters away, which is the substring $S[p - \delta..p + \delta]$ for some $\delta \in O(\tau)$.

The above discussion captures the key concepts of the algorithms, however, in certain cases it
is impossible to pick such a set of positions. For example, if the string is $S = \text{aaa...a}$, it is obvious
that any local selection, must pick all the positions (except possibly for the margins) or none of
them. Thus, it is impossible to fulfill both the requirement of $O(\frac{n}{\tau})$ positions and that the distance
between any two consecutive positions will be $O(\tau)$. Generally, this problem occurs when the string
$S$ contains long substrings whose period (see Section 2) length is $O(\tau)$. Thus, our algorithms use
period-based techniques to treat positions in such substrings. So, the partitioning of $S$ will contain
two types of blocks: periodic blocks, which are substrings with period length at most $\tau$ and regular
blocks, that contains no repetition. For the sake of clarity, all the details about periodic blocks are
omitted in this overview.

It remains to describe how the algorithm pick the special positions, and how it uses this positions
to answer LCE queries on these positions fast. We introduce two methods, and therefore two
algorithms, for the position selection. The former is a randomized Las-Vegas method and the later
is a deterministic method, with a slightly worse construction and query time.

**The randomized method.** For the randomized data structure, the positions are picked in the
following way. The algorithm uses a hashing technique, that maps each string of length $\tau$ into an
integer value in the range $[1, n^c]$ for some constant $c$. The mapping function is almost min-wise
function, which means that given $k$ different strings the probability of each one of them to be
mapped into the minimum value among all the strings is at most $O(\frac{1}{k})$. For any position $p \in [n]$
the algorithm computes the fingerprint of $S[p..p+\tau-1]$ and considered it as the ID of position $p$.
Then, the algorithm scans the positions with a sliding window of length $\tau$, and for any consecutive
$\tau$ positions, the algorithm picks the position with the minimum ID. Notice that it is possible, and
even advisable, that a single position will be picked due to multiple overlapping shifts of the sliding
window. It is straightforward that in any sequence of $\tau$ positions this method will pick at least
one position and that the selection of each position depends only on a substring of $S$ of length
$3\tau$. The only possible problem with fulfilling the desired properties is the possibility of choosing
too many positions ($\omega(\frac{n}{\tau})$). With a simple probabilistic analysis of this method it is easy to prove
that under random selection of the hash function, the expected number of picked positions is
$O(\frac{1}{\tau})$, assuming that in each offset of the sliding window, each substring of length $\tau$ appears only
once, i.e. the non-periodic case. The details of this method, including the treatment of periodic
substring, appears in Section 4. Later, in Section 8 we show how to improve this method to $O(\frac{n}{\tau})$ picked positions, with linear construction time with high probability. The main idea of this improvement is to evaluate whether the selected min-wise hash function provides a small enough set of positions by sampling a few substrings, and if the set of positions is too large, the algorithm chooses another function.

The deterministic method. In the deterministic method, we mimic the randomised selection, but with a technique of deterministic coin-tossing, going back to Cole and Vishkin [9] and appeared in several other papers [10, 20, 31, 36]. In the high level, this technique transforms a string over an alphabet $\Sigma$ into a same length string over a constant alphabet in $O(n \log^* n)$ time. The transformed string has the special property that for any pair of consecutive positions, if the original characters before the transformations were distinct, then the characters on these positions after the transformation are distinct as well. In addition, the transformation of each position depends only on the neighbourhood of the position of $O(\log^* n)$ positions, assuming that $S$ does not contain any kind of periodic substring. Using this transformation, the algorithm splits $S$ into blocks of size at least 2 and at most $O(1)$, by selecting all the positions which their transformed value is a local minimum. By a repeatedly applying this transformation, while after the first transformation we consider each block as a character over a larger alphabet, the algorithm creates $O(\frac{n}{\tau})$ blocks of size $O(\tau)$ characters. Each of these blocks depends only on a substring of $S$ of length $O(\tau \log^* n)$, which adds only a factor of $\log^* n$ over the complexities of algorithms that use the randomized selection. The total computation time is $O(n(\log \tau + \log^* n))$ and with careful scheduling it can be implemented using only $O(\frac{n}{\tau} + \log n \cdot \log^* n)$ words of space. During the transformations of the string into blocks the algorithm recognizes all the long substrings of $S$ with period at most $\tau$ for the special treatment. The technique is similar to previous algorithms as [10], however, this is the first implementation where the recursive construction do not use any randomness and also does not require $\Omega(n)$ working space.

Compute LCE query using the selected positions. Given the partitioning of $S$ induced by the selected positions, we define for any two indices $i', j'$ which are beginning of some blocks, the longest common block-prefix as the largest $\ell'$ such that $S[i'..i' + \ell' - 1] = S[j'..j' + \ell' - 1]$ and all the blocks that compose the two substrings are exactly the same. In order to compute this length, we create a new string $S'$ by converting each block into a unique letter. To detect repeated occurrences of the same block, and give them the same letter, we sort the blocks in linear time using a technique of [19]. The result string is of length $O(\frac{n}{\tau})$ and this is also the alphabet size, so we build a suffix tree of this string in $O(\frac{n}{\tau})$ time and space. Using this suffix tree, it is easy to compute the value of $\ell'$ in constant time.

As described above, given the partitioning of $S$ induced by the selected positions, for any two indices $i, j \in [n]$ with $\text{LCE}(i, j) = \ell > \Omega(\tau)$ the substrings $S[i..i + \ell - 1] = S[j..j + \ell - 1]$ are composed of a short prefix, a short suffix, and a middle part which composed of full blocks, exactly the same in both substrings. Thus, the computation of $\text{LCE}(i, j)$ follows this decomposition and computes the LCE in three phases. In the first phase the algorithm compares $O(\tau)$ pairs of characters (or $O(\tau \log^* n)$ for the deterministic selection). If there is an exact match, the algorithm finds the beginning of the middle part and computes its end using the suffix tree of the blocks. Then, the algorithm just compares additional $O(\tau)$ pairs of characters (or $O(\tau \log^* n)$ for the deterministic selection) after the end of the middle part, and it is guaranteed that the mismatch will be found.
Sparse Suffix Tree construction. It is well-known that the SST of a set \( B \subseteq [n] \) of suffixes can be built in \( O(|B|) \) time and space given the Sparse Suffix array (SSA) and the LCP array \( [26] \) of \( B \), which are the lexicographic order of the suffixes, and the longest common prefix of each adjacent suffixes due to this order, respectively. Hence, our focus is on sorting the suffixes and on computing their LCP. To do so we divide the problem to two steps. First the algorithm selects a partitioning-set \( P \) and builds the SSA of all suffixes starting at positions of \( P \). Then, the algorithm uses the SSA of \( P \) to construct the SSA of \( B \).

We use a property that our partitioning-set selection algorithms induce on \( P \). This property ensures that if two suffixes of positions in \( P \) share a long prefix, then both prefixes are composed of the same blocks in the partitioning, except for possibly small portion at the end of the common prefix. Intuitively, the algorithm sorts the blocks and create a new string \( S'_{P} \) of length \( |P| \), such that the \( i \)'th character in \( S'_{P} \) is the rank of the \( i \)'th block. Then, we prove that the lexicographic order of any two suffixes of \( S \) starting at positions from \( P \) is exactly the same as the lexicographic order of the corresponding suffixes of \( S'_{P} \). The actual construction is a little bit more complicated since it is required to extend the blocks we sort in order to achieve the aforementioned result.

On the second stage for creating the SSA of \( B \), we use methods which are similar to the methods we used to compute the SSA of \( P \). The algorithm creates for any suffix \( i \in B \) a short representative string, which basically, ignoring some details, is a prefix of the \( i \)'th suffix of a proper length. Then the algorithm sorts all the representative strings of the suffixes. Ranking of the representative strings is sufficient to determine the order of two suffixes with different corresponding representative strings. To determine the order of two suffixes which have the same representative string we use the SSA of \( P \) from the first stage. The representative string length is chosen such that the fact that two suffixes correspond to the same representative string guarantees that there exist two selected positions in \( P \) such that the order of the suffixes from \( B \) could be determined by the order of the suffixes from \( P \), which already been computed. Thus, by combining the representative strings ranking with the SSA of \( P \), the algorithm can sort all suffixes of \( B \) in \( O(n) \) time using only \( O(n) \) words of space.

Even better Deterministic LCE data structure. Using the SST construction, we show how to combine the deterministic selection with the difference covers \([29, 8]\) technique, to get almost optimal trade-off for the LCE data structure of \( O(\frac{n}{\tau}) \) space and \( O(\tau \sqrt{\log^* n}) \) query time with \( O(n(\log \tau + \log^* n)) \) deterministic construction time. We mention that although the technique of difference covers already been used for the LCE problem (see \([34, 4, 19]\)), the usage in our algorithm is different. While in all previous work the considered positions were all the positions in a given range, in our algorithm the positions are subset of the locally selected positions. Thus, some properties such as the ability to compute in constant time the common difference for any two indices are impossible to exploit. Even tough we need to spend time on finding the synchronized selected positions, our selection gives a \( O(\tau \sqrt{\log^* n}) \) bound on the amount of comparison steps needed to take until the synchronized positions are found.

Organization. In section 2 we give the basic definitions and tools we use in the paper. Then in Section 3 we give the formal definition for the set of selected positions, and show how to use it to create an LCE data structure. We complete the data structures with the randomized Las-Vegas position selection in Section 4 and with the deterministic selection in Section 5. In Section 6 we show how to use these selections in order to achieve an SST construction algorithms. Sections 7 and 8 show how to improve the basic results, where Section 7 discuss on the trade-off improvement for deterministic LCE data structure, and Section 8 discuss on how to ensure the Las-Vegas selection will run in linear time with high probability. In these sections we omit the discussion about the
case of periodic block, which we postpone to the appendices, so in Section 9 we overview the main technique that we use for this case.

2 Preliminaries

For $1 \leq i < j \leq n$ denote the interval $[i..j] = \{i, i + 1, \ldots, j\}$, and $[n] = \{1, 2, \ldots, n\}$.

We consider in this paper strings over an integer alphabet $\Sigma = \{1, 2, \ldots, nO(1)\}$. A string $W$ of length $|W| = \ell$ is a sequence of characters $W[1]W[2] \ldots W[\ell]$ over $\Sigma$. A substring of $W$ is denoted by $W[x..y] = W[x]W[x + 1] \ldots W[y]$ for $1 \leq x \leq y \leq \ell$. If $x = 1$ the substring is called a prefix of $W$, and if $y = \ell$, the substring is called a suffix of $W$.

A prefix of $W$ of length $y \geq 1$ is called a period of $W$ if and only if $W[i] = W[i + y]$ for all $1 \leq i \leq \ell - y$. The shortest period of $W$ is called the principal period of $W$, and its length is denoted by $\rho_W$. If $\rho_W \leq \frac{|W|}{2}$ we say that $W$ is periodic.

Sparse suffix trees. Given a set of strings, the compact trie [32] of these strings is the tree induced by dissolving all the nodes with exactly one child in the trie [11, 18] of the strings. Each edge in the compact trie has a label which is a substring of part of the given strings. Each node in the tree is associated with a string, which is the concatenation of all the labels of the edges of the simple path from the root to the node. The suffix tree of a string $S$ is the compact trie of all the suffixes of $S$. The sparse suffix tree of a string $S$ of length $n$, with a set $B \subseteq [n]$ is the compact trie of all the suffixes $\{S[i..n] \mid i \in B\}$.

Fingerprints. For a natural number $n$ we denote $[n] = \{1, 2, \ldots, n\}$. For the following let $u, v \in \cup_{i=0}^n \Sigma^i$ be two strings of size at most $n$. Porat and Porat [33] and Breslauer and Galil [7] proved that for every constant $c > 1$ there exists a fingerprint function $\phi : \cup_{i=0}^n \Sigma^i \to [n^c]$, which is closely related to the hash function of Karp and Rabin [25], such that:

1. If $|u| = |v|$ and $u \neq v$ then $\phi(u) \neq \phi(v)$ with high probability (at least $1 - \frac{1}{n}$).

2. The sliding property: Let $w = uv$ be the concatenation of $u$ and $v$. If $|w| \leq n$ then given the length and the fingerprints of any two strings from $u, v$ and $w$, one can compute the fingerprint of the third string in constant time.

Approximately Min-Wise Hash Function We say that $\mathcal{F}$ is a $(\epsilon, k)$-min-wise independent hash family if for any $X \subseteq [n], |X| < k$ and $x \in [n] - X$ when picking a function $h \in \mathcal{F}$ uniformly we have: $\Pr[h(x) < \min_{y \in X} \{h(y)\}] = \frac{1}{|X| + 1} \cdot (1 \pm \epsilon)$

Indyks [22] have proved that any $O(\log \frac{1}{\epsilon})$-independent hash family is also a $(\epsilon, \frac{n}{\epsilon})$-min-wise independent hash family, for some constant $c > 1$. Furthermore, since the family of polynomials of degree $l - 1$ over $GF(n)$, assuming $n$ is a prime, is an $l$-independent hash family, then for $l = O(\log \frac{1}{\epsilon})$ the polynomials’ family is $(\epsilon, \frac{n}{\epsilon})$-min-wise independent hash family. For any polynomial from the family, evaluating its value for a given input can be done in $O(\log \frac{1}{\epsilon})$ time.

Other tools. Other useful lemmas and theorems are presented in Appendix F.

3 LCE with Selected Locations

Following the discussion above in Section 1.2, we introduce here the formal definition of the selected positions that defines the partitioning of $S$ into blocks.
The partitioning of $S$ is induced by a set $P$ of indices from $[n]$, such that each block is a substring beginning in some index of $P \cup \{1\}$ and the concatenation of all the blocks is exactly $S$. The following definition introduce the essential properties of the set $P$, required for our data structures.

**Definition 3.1.** A set of positions $P \subseteq [n]$ is called a $(\tau, \delta)$-partitioning set of $S$, for some parameters $1 \leq \tau \leq \delta \leq n$, if and only if it has the following properties:

1. **Local Consistency** - For any two indices $i, j \in [1 + \delta..n]$ such that $S[i - \delta..i + \delta] = S[j - \delta..j + \delta]$ we have $i \in P \iff j \in P$.

2. **Compactness** - Let $p_i < p_{i+1}$ be two consecutive positions in $P \cup \{1, n+1\}$ then we have one of the following.

   (a) (Regular block) $p_{i+1} - p_i \leq \tau$.

   (b) (Periodic block) $p_{i+1} - p_i > \tau$ such that $u = S[p_i..p_{i+1} - 1]$ is a periodic string with period length $\rho_u \leq \tau$.

Here we describe how to use a $(\tau, \delta)$-partitioning set of size $O(\frac{n}{\tau})$ in order to construct an LCE data structure in $O(n)$ time using $O(\frac{n}{\tau})$ words of space. The data structure uses $O(\frac{n}{\tau})$ words of working space and answers queries in $O(\delta)$ time. In Section 4, we introduce a randomized Las-Vegas algorithm that finds a $(2\tau, 2\tau)$-partitioning set of size $O(\frac{n}{\tau})$ in $O(n)$ expected time (which we later improve to be with high probability in Section 8), using only $O(\frac{n}{\tau})$ words of space. In Section 5, we introduce a deterministic algorithm that finds a $(O(\tau), O(\tau \log \log n))$-partitioning set of size $O(\frac{n}{\tau})$ in $O(n(\log \tau + \log \log n))$ time using only $O(\frac{n}{\tau} + \log n \log \log n)$ words of space.

**Remark about edge cases.** When using a $(\tau, \delta)$-partitioning set we assume that all our queries are of indices in the range $[1 + \delta..n - \delta]$. This assumption is covered by the algorithm can consider a string $S'$ which is the concatenation of $S$ with $\delta$ occurrences of a special character # in each side of $S$. Notice that if $S$ is given in a read only, the algorithm can simulate the string $S'$ with additional constant space. In addition, The size of $S'$ is $n + 2\delta = O(n)$.

**The partitioning string.** Given a $(\tau, \delta)$-partitioning set $P$ of size $O(\frac{n}{\tau})$ we consider the partitioning of $S$ induced by $P$. Let $p_1 < \cdots < p_h$ be all the indices in $P \cup \{1, n+1\}$, then the $i$th block of $S$ (for $1 \leq i < h$) is $S_i = S[p_i..p_{i+1} - 1]$. Since there are at most $O(\frac{n}{\tau})$ blocks, we associate with each block a unique symbol from $\Sigma' = \{1, \ldots, O(\frac{n}{\tau})\}$, such that if two blocks are equal (correspond to the same string) they have the same symbol. The unique symbols are determined by sorting the blocks in $O(n)$ time due to Theorem F.5, where the only exception are periodic blocks whose length is larger than $\tau$. For those blocks, we take only the first $2\tau$ characters into the sorting and add at the end a new character which is the length of the block (we assume that the alphabet of $S$ contains $\{1, 2, \ldots, n\}$, otherwise we extend the alphabet). The value of each block is its rank among all the (distinct) blocks. It is straightforward that the value of each block is a unique identifier which equals among all copies of the block. Using this values, we consider the partitioning string $S_P$ over $\Sigma'$, defined such that $S_P[i]$ is the value of the $i$th block $S[p_i..p_{i+1} - 1]$. The string $S_P$ is constructed in $O(n)$ time from the partitioning set $P$ and its length is $O(\frac{n}{\tau})$.

**The data structure.** The data structure builds a suffix tree $T_P$ on the string $S_P$ using the algorithm of Farach-Colton [12] in $O(\frac{n}{\tau})$ time. Each node in $T_P$ maintains the length of the substring of $S$ (over $\Sigma$) corresponds to the substring of $S_P$ from the root of $T_P$ to the node, as its
Lemma 3.3. Given a $(\tau, \delta)$-partitioning set of size $O(\frac{n}{\tau})$, there exists a deterministic construction algorithm of an LCE data structure the takes $O(n)$ time using $O(\frac{n}{\tau})$ words of space. The space usage of the data structure is $O(\frac{n}{\tau})$ and its query time is $O(\delta)$.

Thus, combining Lemma 3.3 with the randomized selection of Sections 4 and 8 (Lemma 8.6) we have proved Theorem 1.1. Similarly, combining Lemma 3.3 with the deterministic selection of Section 5 (Corollary D.7) we have proved Lemma 1.2.

4 Randomized Selection

Using fingerprint hash functions and approximately min-wise independent hash functions, we prove that a $(2\tau, 2\tau)$-partitioning set consisting $O(\frac{n}{\tau})$ positions can be constructed in linear time with high probability. In this section we show a method for constructing such a partitioning set of size
\(O\left(\frac{n}{\tau}\right)\) using linear time in expectation. Later, in Section 8 we improve the method to ensure linear construction time with high probability. All the algorithms use \(O\left(\frac{n}{\tau}\right)\) words of working space.

We define the ID function for all positions \(j \in [1..n-\tau+1]\) such that ID\((j)\) is dependent only on \(S[j..j+\tau-1]\), hence if \(S[j..j+\tau-1] = S[k..k+\tau-1]\) for some indices \(j\) and \(k\) then ID\((j) = \text{ID}(k)\).

Using the ID function, we create a partitioning of \(S\), denoted as \(P\). For each interval \([i..i+\tau-1]\) the positions within the interval with the smallest ID value are picked for \(P\). In other words,

\[
P = \left\{ j \in [1..n-\tau] | \exists \ell \in [j+1..j+\tau-1] : \text{ID}(j) = \min_{k \in [\ell..\ell+\tau-1]} \{\text{ID}(k)\} \right\}
\]

From the locality properties of the ID function, the following lemma follows immediately.

**Lemma 4.1.** Given an ID function as described above, the set \(P\) induced by ID is a \((2\tau, 2\tau)\)-partitioning set.

**Proof.** We will prove that the two properties of Definition 3.1 holds.

**Property 1 - Local Consistency** Let \(i,j \in [1 + 2\tau..n-2\tau]\) such that \(S[i-2\tau..i+2\tau] = S[j-2\tau..j+2\tau]\). Since ID\((a)\) is dependent only on \(S[a..a+\tau-1]\), we have that ID\((a) = \text{ID}(a-i+j)\) for any \(a \in [i-2\tau..i+\tau+1]\).

Assume \(i \in P\) and let \(\ell \in [i-\tau+1..i]\) be an index such that ID\((i) = \min_{k \in [\ell..\ell+\tau-1]} \{\text{ID}(k)\}\). We have that \([\ell..\ell+\tau-1] \subset [i-\tau+1..i+\tau-1] \subset [i-2\tau..i+\tau+1]\). Hence, for any \(k \in [\ell..\ell+\tau-1]\) we have ID\((k) = \text{ID}(k-i+j)\). In particular,

\[
\text{ID}(j) = \text{ID}(i) = \min_{k \in [\ell..\ell+\tau-1]} \{\text{ID}(k)\} = \min_{k \in [\ell-i+j..\ell-i+j+\tau-1]} \{\text{ID}(k)\}.
\]

Thus, due to the definition of \(P\), \(j \in P\). Hence, we prove that \(i \in P \Rightarrow j \in P\), the other direction is symmetric.

**Property 2** Let \(1 = p_0 < p_1 < \cdots < p_k < p_{k+1} = n+1\) be all the positions in the partition \(P \cup \{1, n+1\}\). First notice that if the first selected position is not \(p_0 = 1\), then \(p_1\) is selected from the interval \([1..\tau]\), and therefore \(p_1-p_0 < \tau\). Secondly, the rightmost position in \(P\), \(p_k\), is the position picked on the interval \([n-2\tau+1..n-\tau]\). Thus, \(p_k-1 < 2\tau\). Lastly, for the general case where both \(p_i, p_{i+1} \in P\) assume by way of contradiction that \(p_{i+1} - p_i > \tau\). Therefore no position is selected in the interval \([p_j..p_j+\tau] \subseteq [p_j..p_{j+1}-1]\), and by definition of \(P\) this cannot happen.

**Remark about the periodic case** Notice that if \(S\) contains a long periodic substring with a small period, then the same IDs will appear in every period (except perhaps in the last \(\tau\) positions within the periodic substring). In particular, many positions will have the minimum ID value, hence \(P\) will be a very large partitioning set, no matter which ID function we use. This means that long periodic substring with small periods should be taken with a special care. We say a substring \(S' = S[i..j]\) is a \((d, \rho)\)-run if \(S'\) is a periodic substring of length at least \(d\) characters and has a principal period of at most \(\rho\) characters.

**Overview of the solution** We show next the solution for the non-periodic case: We prove that when \(S\) contains no \((\tau, \frac{\tau}{6})\)-runs, it is possible to construct a partitioning set of \(O\left(\frac{n}{\tau}\right)\) positions in expected linear time. In Appendix C.2, we show that all \((\tau, \frac{\tau}{6})\)-runs can be found in linear time and constant space using fingerprints. We then combine the two methods to partition the string to blocks —find all \((\tau, \frac{\tau}{6})\)-runs and mark them as blocks, and then use the non-periodic algorithm to partition the substring between every two runs in a total of expected linear time and \(O\left(\frac{n}{\tau}\right)\) space.
The ID function Our goal is to use an ID function that induces a partitioning set of size $O(n)$ that we will be able to compute in $O(n)$ time in expectation. To achieve this, we fix a fingerprint function $\phi$ and an approximately min-wise hash function $h \in \mathcal{F}$ where $\mathcal{F}$ is a $(\frac{1}{2}, \tau)$-min-wise independent hash family (see Section 2). We denote $\phi_i = \phi(S[i..i+\tau-1])$ and the algorithm uses the function $ID_{\phi,h}(i) = h(\phi_i)$. The fingerprint function is used for the locality of $ID_{\phi,h}$, and the hash function is used for diversifying the fingerprints that will have minimal value.

As mentioned before, we assume $S$ contains no $(\tau, \frac{\tau}{2})$-runs. We show that when using $ID_{\phi,h}$ where $\phi$ and $h$ are picked uniformly, the size of the partitioning set $P$ is $O(n)$ in expectation.

Lemma 4.2. If $\phi$ and $h$ are picked uniformly then the partitioning set $P$ defined by $ID_{\phi,h}$, has expected size of $E_{\phi,h}[|P|] = O(\frac{n}{\tau})$ words of space.

Proof Sketch; full proof appears in Appendix C.1. We assume that there are no fingerprints collisions, since the contribution to the expectation of the other case is constant due to the small probability of fingerprints collision. We use the linearity of expectation, which means that $E_{\phi,h}[|P|] = \sum_{i=1}^{n} Pr_{\phi,h}(A_i)$ where $A_i$ is the event that $i \in P$. We bound $Pr_{\phi,h}(A_i)$ by noticing that for any $i$, we have $i \in P$ only if $h(\phi_i)$ is the minimum ID among all the IDs of $[i-i-\tau/2.. i]$ or the minimum among all the IDs of $[i..i+\tau/2]$. Due to the almost min-wise function $h$, the probability of each one of these events is $O(\frac{1}{\tau})$ and therefore $Pr_{\phi,h}(A_i) \leq O(\frac{1}{\tau})$. Hence, $E_{\phi,h}[|P|] = \sum_{i=1}^{n} Pr_{\phi,h}(A_i) = O(\frac{n}{\tau})$.

Lemma 4.3. The partitioning-set $P$ induced by the function $ID_{\phi,h}$ on a substring containing no $(\tau, \frac{\tau}{2})$-runs can be constructed in expected $O(n)$ time using expected $O(\frac{n}{\tau})$ words of space.

Proof. For every interval of size $\tau$ the algorithm will find the positions with minimum IDs and add them to $P$, using only $O(1)$ additional space. When sliding the interval by one step, using the sliding property of the fingerprints, the next ID is calculated in $O(1)$ time. Let $p_j \in P$ be the rightmost position picked for $P$ for some interval $[k..k+\tau-1] \subset [1..n-\tau]$. Since $p_j$ has the minimum ID value in that interval, then if $p_j \in [k+1.. k+\tau]$ the algorithm can find out in $O(1)$ if $k+\tau \in P$ by calculating its ID and comparing it with $ID(p_j)$.

Otherwise, if $p_j = k$, then the information about the minimum ID is “lost” for the next interval. To regain the information on the position with minimum ID, the algorithm recalculates all the IDs on that interval. This recalculation will take $O(\tau)$ time due to the sliding property of the fingerprint, and it happens only if the new interval doesn’t contain the currently rightmost position in $P$. Therefore we can charge the $O(\tau)$ time on that position, and it is charged at most once. Immediately after we charge it, a new rightmost position in $P$ is picked.

Therefore, the algorithm runs in $O(n + \tau \cdot |P|)$ using $O(|P|)$ space, which by Lemma 4.2 is expected to be $O(n)$ time using expected $O(\frac{n}{\tau})$ words of space.

Corollary 4.4. A $(2\tau, 2\tau)$-partitioning set of $S$ can be constructed using $O(\frac{n}{\tau})$ words of space with expected $O(n)$ construction time.

Proof. Due to Lemma 4.2, when $ID_{\phi,h}$ is picked randomly the expected size of the $(2\tau, 2\tau)$-partitioning set induced by this ID function is $\frac{cn}{\tau}$ for some constant $c > 1$. By Markov’s inequality this means that $Pr_{\phi,h}(|P| > \frac{3cn}{\tau}) < \frac{1}{3}$. Running the algorithm described in the proof of Lemma 4.3 until either $P$ is found or the algorithm fails by determining that $|P| > \frac{3cn}{\tau}$ takes $O(n)$ time, and uses at most $O(\frac{n}{\tau})$. Since the probability of the algorithm to fail is less than $\frac{1}{3}$, then the number of different ID function tries is a geometric random variable with a constant number of tries in expectation. Therefore, the expected overall running time is still $O(n)$. 


5 Deterministic Selection

In this section we introduce a deterministic algorithm that finds a \( \mathcal{O}(\tau, \mathcal{O}(\tau \log* n)) \)-partitioning set of size \( \mathcal{O}(\frac{n}{\tau}) \). The key concepts we use are based on the techniques of Cormode and Muthukrishnan [10], which in turn are similar to previous papers [20, 31, 9, 35]. Here we focus on the definition of the positions selected by the algorithm. Later, in Appendix D.2 we describe how to select these positions using only \( \mathcal{O}(\frac{n}{\tau} + \log n \log* n) \) words of space in \( \mathcal{O}(n(\log \tau + \log* n)) \) time. The complete correctness proof appears in Appendix D.1.

The algorithm creates a hierarchical decomposition of \( S \), such that the top-level has \( \mathcal{O}(n/\tau) \) blocks. At the bottom level (level 0) each letter in \( S \) is considered as a block, so the set of picked positions is all \( [n] \). We create levels in bottom-up fashion, preserving the following properties. At each level \( \mu \) there are at most \( \mathcal{O}\left(\frac{n}{(3/2)^{\mu}}\right) \) blocks. Moreover, the length of each block is \( \mathcal{O}\left((3/2)^{\mu}\right) \), except for blocks which are substrings of \( S \) that has period length of at most \( (3/2)^{\mu} \), which can be larger. In addition, the locality of the partitioning is obtained by limiting the dependency of each block in level \( \mu \) to at most \( \mathcal{O}(\log* n) \) blocks in level \( \mu - 1 \). These properties are a generalization of Definition 3.1 (partitioning set) of size \( \mathcal{O}(\frac{n}{\tau}) \), so by repetitively applying the transformation from level \( \mu = 0 \) up to level \( \mu = \log_{3/2} \tau \) we obtain a \( \mathcal{O}(\tau, \mathcal{O}(\tau \log* n)) \)-partitioning set.

Now we explain how the decomposition of level \( \mu \) is defined based on the decomposition of level \( \mu - 1 \) sub-blocks. Given the \( (\mu - 1) \)th level of the hierarchy, we consider parsing of \( S \) into four types of maximal non-overlapping substrings.

1. Sub-blocks which their length is at least \( (3/2)^{\mu} \).
2. Maximal contiguous sequence of sub-blocks that consist of a repeated substring and are not of type 1.
3. Sequences of at least \( c \log* n \) sub-blocks, that are not of type 1 or 2(where \( c \) is a constant to be determined).
4. Sequences of less than \( c \log* n \) sub-blocks, that are not of type 1 or 2.

We identify each block with the index of its leftmost endpoint, which we called the representative position of the block. As a consequence, the position 1 is a representative position at any level of the hierarchy. Now we describe how to move from one level of the hierarchy to the level above. The description treats each type of such maximal substring separately. For the sake of simplicity, we describe it now as an algorithm which has no space constraints, which we overcome in Appendix D.2.

The first type contains sub-blocks of length at least \( (3/2)^{\mu} \) and the algorithm just maintains them as blocks in level \( \mu \). The second type is sequences of (at least two) identical sub-blocks. In this case, the algorithm merges all the sub-blocks in each sequence into one large block. To identify such sub-blocks, the algorithm compare each sub-block with its successor, and if they are the same, they merged in level \( \mu \).

The alphabet reduction. For the third type, we use the following method. For the sake of intuition, we begin with the special case where \( \mu = 1 \), which is the transformation from level 0 (where each letter of \( S \) has a unique block) to the next level. We follow the technique of [10] and reduce each letter into a number from the set \{0, 1, 2, 3, 4, 5\} using a local method. The alphabet reduction is done in \( \mathcal{O}(\log* n) \) iterations, which we call inner-iterations (to distinguish them from the outer-iterations of \( \mu \) from 0 to \( \log_{3/2} \tau \)). We assume that the alphabet of \( S \) is \( \Sigma = \{1, 2, \ldots, |\Sigma|\} \).
of size $|\Sigma| = \mathcal{O}(n^{O(1)})$. Hence, before the reduction begins each letter can be considered as a bit string of length $\mathcal{O}(\log n)$. At each inner iteration, the algorithm reduces the length of the binary string in a log factor. We denote by $\text{label}(i, 0) = S[i]$ the letter of $S$ in position $i$. For any two consecutive letters of the $(j − 1)$th inner iteration, $\text{label}(i, j − 1)$ and $\text{label}(i + 1, j − 1)$, we define $\text{label}(i, j)$ as follows. Let $\psi$ be the index of the least significant bit in which $\text{label}(i, j − 1)$ and $\text{label}(i + 1, j − 1)$ are different, and let $\text{bit}(\text{label}(i, j − 1), \psi)$ be the value of the $\psi$’th bit of $\text{label}(i, j − 1)$. We define $\text{label}(i, j) = 2\psi + \text{bit}(\text{label}(i, j − 1), \psi)$, i.e. the concatenation of the index $\psi$ with the value of the bit in this index.

**Lemma 5.1** ([10, Lemma 1]). For any $i$, if $S[i] \neq S[i + 1]$ then $\text{label}(i, j) \neq \text{label}(i + 1, j)$ for any $j \leq 0$.

Each inner iteration of this reduction, reduces an alphabet of size $\sigma$ to an alphabet of size $2 \lfloor \log \sigma \rfloor$. We run such inner-iterations until the size of the alphabet is constant. Hence, after $\mathcal{O}(\log^* n)$ inner-iterations the size of the alphabet becomes constant, we denote by $c$ the appropriate constant for the $\mathcal{O}(\log^* n)$ inner-iterations. By the following lemma, the size of the final alphabet is exactly 6.

**Lemma 5.2** ([10, Lemma 2]). After the final inner-iteration of alphabet reduction, the alphabet size is 6.

The main challenge with using this technique is how to repeat this process with larger and larger blocks. In [10] there are two methods: a randomized algorithm that uses fingerprints to reduce the size of each block to constant number of words ($\mathcal{O}(\log n)$ bits), or an algorithm with expensive $\mathcal{O}(n)$ additional working space that uses bucket sort. Since we want a deterministic algorithm, and since we have just $\mathcal{O}(\frac{c}{2})$ words of working space we use a different method, which cost us $\log \tau$ factor in the running time and uses the following observation. We observe that each block at any outer-iteration is a string of size at most $\mathcal{O}(n)$, thus we consider each block as a number between one and $\mathcal{O}(|\Sigma|^n) = \mathcal{O}(n^{O(n)})$. Formally, we assume without loss of generality that $0 \notin \Sigma$, the alphabet of $S$, and we define $\Pi = \{0, 1, \ldots, 2^{\lceil \log_2(|\Sigma|) + 1 \rceil}n\}$ as the blocks’ alphabet. Each block $u = S[x \ldots y]$ at any iteration is a string of length at most $n$ over $\Sigma$, and is represented by one symbol from $\Pi$ by concatenating the binary representation of $u$’s the characters, with padding of zeros to the left, i.e. the symbol of $u$ is $\sum_{k=0}^{\lfloor \log_2(|\Sigma|+1) \rfloor} u[k] \cdot 2^{\lfloor \log_2(|\Sigma|+1) \rfloor} k \in \Pi$. The alphabet reduction for any iteration $\mu > 1$ is done exactly as described for $\mu = 1$ except that instead of converting letters from $\Sigma$ the algorithm considers the symbols of $\Pi$ induced by the blocks of level $\mu − 1$.

Notice that $\Pi = \mathcal{O}(|\Sigma|^n) = \mathcal{O}(n^{O(n)})$ and therefore, the alphabet reduction for any outer-iteration takes $\mathcal{O}(\log^*(n^{O(n)})) = \mathcal{O}(\log^* n)$ inner-iterations. For any outer-iteration the algorithm has to execute $\mathcal{O}(\log^* n)$ inner iterations. Each first inner-iteration takes $\mathcal{O}(n)$ time since it has to take in account all the blocks which their length summed to $\Theta(n)$. However, after one inner-iteration each block reduced from $\mathcal{O}(\log n^{O(n)}) = \mathcal{O}(n \log n)$ bits to $\mathcal{O}(\log n)$ bits, and therefore can be represented in constant number of words. Thus, after the first inner-iteration each block comparison takes constant time. Since the number of blocks reduces exponentially and it is $\mathcal{O}\left(\frac{n}{(3/2)^\mu}\right)$, the total time for all the inner iterations, except the firsts of each outer-iteration is $\mathcal{O}(n \log^* n)$. Together with the first inner-iterations the total time required for all the transformations is $\mathcal{O}(n(\log \tau + \log^* n))$.

Remark that at the end of the alphabet reduction, the blocks’ symbols are all in $\{0, 1, 2, 3, 4, 5\}$. At this point the algorithm scan all blocks and compare their symbol’s value with those of their right and left neighbours. The algorithm picks all positions that have a local minimum symbol value as the selected positions in level $\mu$. The only exceptions are regarding the margins of the sequence. For the last $c \log^* n$ characters, the alphabet reduction cannot reach the last level, and
therefore they do not have a corresponding number from \{0, \ldots, 5\}, thus, the algorithm treats them as sequences of type 4. In addition, the leftmost position in the sequence is always selected by the algorithm. Moreover, if the second block in the sequence (from left) receives a symbol which is local minimum, the algorithm ignores it in order to avoid the case that the first sub-block will become a block in level \(\mu\). So, by the merging due to minimal symbol each block of level \(\mu\) is composed of at least 2 sub-blocks of level \(\mu - 1\) and at most 11 sub-blocks (10 as the maximal distance between minimal positions over the symbols \{0, \ldots, 5\} and additional one block for the case of the leftmost block).

**Margin sub-blocks (type 4).** Each sequence of type 4 contains at most \(c \log^* n\) sub-blocks. If the sequence contains exactly one block, this block is kept the same in level \(\mu\) of the hierarchy. If the sequence contains two or three sub-blocks, the algorithm creates one block in level \(\mu\) which is the union of all these three sub-blocks. As for the last case, if the sequence contains at least four sub-blocks, the algorithm creates one block in the \(\mu\)th level which is the union of the two rightmost blocks, and repeats the process from right to left until there are no sub-blocks of level \(i - 1\) in the sequence which are not part of any block in level \(\mu\). Notice that each block created in such sequence contains at least two sub-blocks from the lower level and at most three of them, except for the case where the sequence contains exactly one sub-block.

**Conclusion.** The set of positions induced by level \(\mu = \log_{3/2} \tau\) of the hierarchical decomposition, is a \((O(\tau), O(\tau \log^* n))\)-partitioning set of size \(O(\frac{n}{\tau})\). The correctness proof appears in Appendix D.1. Moreover, the algorithm finds this set using only \(O(\frac{n}{\tau} + \log \tau \cdot \log^* n)\) words of space in \(O(n(\log \tau + \log^* n))\) time, as described in Appendix D.2.

### 6 Sparse Suffix Tree Construction

In this section we introduce a method to build the Sparse Suffix Tree of a set \(B \subseteq [n]\) suffixes of \(S\). Since the output size is \(\Theta(|B|)\) words of space, we are able to use \(O(|B|)\) words of working space without affecting the space usage of the algorithm. Thus, we will use the LCE data structure with \(\tau = \frac{n}{|B|}\) that uses \(O(\frac{n}{|B|}) = O(|B|)\) words of space. We focus on computing the Sparse Suffix Array (SSA) of the suffixes, which is an array that stores \(B\)'s suffixes due to their lexicographic order. Then, one can compute the longest common prefix (LCP) of any pair of consecutive suffixes (due to this order) using the LCE data structure from Section 3. Using the SSA and LCP information one can build the desired SST in \(O(|B|) = O(\frac{n}{\tau})\) time, using the algorithm of Kasai et al. [26]. Our construction algorithms will use the same \((\tau, \delta)\)-partitioning sets that were introduced in Section 4 and Section 5. We will prove that these sets have a special property that enables us to compute efficiently the SSA of the set \(P\) that was selected by the algorithm. Then, we will show how to use this SSA to compute the desired SSA of the set \(B\).

Recall that Lemma 3.2, which was induced by the definition of \((\tau, \delta)\)-partitioning set, said that for any two indices with large common extension both common substrings are composed of the same blocks except for possibly the right and left margins of length \(\delta\). For the SST construction, we require additional property. We want that for every pair of selected positions, i.e. positions from \(P\), with large LCE, the two suffixes also begins with composition into exactly the same blocks, and the only difference in the composition of the common extensions is at the right margin. Formally, we say that a partitioning set is *forward synchronized* if it has the following property.
Definition 6.1 (Forward synchronized partitioning set). A \((\tau, \delta)-\)partitioning set \(P\) is called forward synchronized if and only if for any \(p_i, p_j \in P\), which their consecutive positions in \(P\) are \(p_{i+1}\) and \(p_{j+1}\), respectively, if \(\text{LCE}(p_i, p_j) > p_{i+1} - p_i + \delta\) then \(p_{i+1} - p_i = p_{j+1} - p_j\).

We will show how to build the suffix tree of a forward synchronized \((\tau, \delta)-\)partitioning set of size \(\mathcal{O}(\frac{n}{\tau})\) in \(\mathcal{O}(n \cdot \frac{\tau}{n})\) time, using \(\mathcal{O}(\frac{n}{\tau})\) words of space. Fortunately, our two constructions of partitioning sets, from Section 4 and Section 5 are forward synchronized, as claimed in the following lemma. The proof is given separately to the randomized algorithm (Lemma C.7) and to the deterministic algorithm (Lemma D.6).

Lemma 6.2. Any \((\tau, \delta)-\)partitioning set that is a result of the algorithms from Section 4 or Section 5 is forward synchronized.

6.1 SSA of Forward Synchronized \((\tau, \delta)-\)Partitioning Set

Let \(P\) be a forward synchronized \((\tau, \delta)-\)partitioning set, we show here how to compute the SSA of \(P\) efficiently. Let us consider the decomposition of \(S\) into blocks, induced by \(P\). According to this decomposition, all the suffixes of \(S\) beginning in positions from \(P\) are composed only from complete blocks. Moreover, since \(P\) is forward synchronized, it is guaranteed that whenever comparing two suffixes with large LCE, the two suffixes must begin with identical blocks. As with the definition of \(S_P\) in Section 3, we aim to create a partitioning string \(S'_P\) of length \(|P|\) such that each block of \(S\) will be transformed into one symbol in this string, and any two suffixes begins in positions from \(P\), will have the same order as the corresponding suffixes of the partitioning string. Recall that when we defined \(S_P\) we sorted the substrings of the blocks and transformed each block into its rank. This method seemed to work also for our aim, since when comparing two suffixes, we hope that the first non-identical blocks will determine the order of the suffixes properly, due to the sorting. However, this method has one problem, which is the ordering of blocks when one block is a prefix of another block. In such a case, the order of the suffixes is not determined by these blocks, but by pair of characters which at least one of them is not in these blocks.

We use the following method to create a \(S'_P\), which overcome the difficulty described above. For each regular block (of type 2a in Definition 3.1) beginning at position \(p_i \in P\) we consider the representative string of the block as \(s_i = S[p_i..p_i + 3\delta - 1]\), notice that this string may contain also characters outside the block of \(p_i\). More detailed, since \(\delta \geq \tau\), and due to property 2 of Definition 3.1, the only case where \(s_i\) do not contain characters outside the block is when the block \([p_i, p_{i+1} - 1]\) is a large block with period length of at most \(\tau\) characters. When comparing two suffixes which their first non-identical block is of type 2a, the sorting of the representative strings guarantee that the order of these blocks is equal to the order of the suffixes. Thus, we just need to explain how to treat periodic blocks.

Periodic blocks When comparing two periodic blocks which their prefixes of length \(3\delta > 2\tau\) are equal, it must be the case that all the corresponding characters in the two blocks match (see Lemma F.3), and therefore the shorter block is a prefix of the larger block. Of course, if the two blocks has exactly the same length then the corresponding substrings of \(S\) are identical, and comparison between the suffixes should be done due to the consecutive blocks.

For the sake of intuition let consider first the case that all the periodic blocks are maximal. In this case for a periodic block \([p_i, p_{i+1} - 1]\) if \(S[p_i, p_{i+1} - 1]\) has period of length \(\rho\), then \(S[p_{i+1}] \neq S[p_{i+1} - \rho]\). In such a case, when comparing two suffixes that has the same period, the lexicographic order of the suffixes is defined by the character right after the end of the short block and the corresponding character in the long block. We want the representative string of each
block to be composed of a long enough prefix of the block, say of length $3\delta$, with some additional characters to deal with the situation of one block that is a prefix of another block. Let us consider two suffixes $p_i, p_j \in P$ that begins with periodic blocks that one is the prefix of the other. Assume without loss of generality that the shorter block appears at position $p_i$, and that its length is $\alpha$. In this case the lexicographic order of the suffixes is determined by $S[p_i + \alpha]$ and the corresponding character in the second suffix, $S[p_j + \alpha]$. The suffix $S[p_i..n]$ should appear before $S[p_j..n]$ in the lexicographic order if and only if $S[p_i + \alpha]$ is larger than $S[p_j + \alpha]$. Both the block of $p_i$ and the block of $p_j$ has the same period, hence, let $\rho$ be the length of this period. Thus, for any position $x$ in the block we have $S[x] = S[x - \rho]$ except for possibly the first $\rho \leq \tau$ characters. In particular, $S[p_j + \alpha] = S[p_j + \alpha - \rho]$. On the other hand, since the short block is a prefix of the long block we have $S[P_j + x - \rho] = S[p_j + x - \rho]$ and so $S[p_j + \alpha] = S[p_j + \alpha - \rho]$. Hence, instead of comparing $S[p_i + \alpha]$ with $S[p_j + \alpha]$ it is sufficient to compare $S[p_i + \alpha]$ with $S[p_i + \alpha - \rho]$. Thus, for a periodic block $[p_i, p_{i+1} - 1]$ we focus on comparing the character $S[p_i + \alpha]$, which we call the violation character with the characters $S[p_i + \alpha - \rho]$, which we call the period character. We distinguish between two cases: either the violation character is larger than the period character, or vice versa. If the violation character is larger than the period character, then this block should be after longer blocks with the same period while sorting the suffixes. Otherwise, if the violation character is smaller than the period character, then this block should appear before longer blocks while sorting the suffixes. When comparing two blocks that their violation characters are larger than the period characters, the longer blocks should be first, and when comparing two blocks which their violation characters are smaller than the period characters, the longer block should be last. Thus, for periodic blocks of length at least $3\delta$, instead of just defining the representative string $s_i = S[p_i..p_i + 3\delta - 1]$ we concatenate to this string two symbols. If the violation character is smaller than the period character, the first symbol added is $-1$ and the second is the negative value of the block’s length $(\cdot(-1))$.

In the previous paragraph we assumed that the periodic blocks are maximal, and the characters right after the blocks violate the periodicity. However, this is not necessarily the case in a forward-synchronized partitioning set. It is possible that a periodic block will be terminated before the end of the periodic substring. Moreover, in the constructions that we introduce in this paper the periodic blocks indeed end before the right end of the periodic substring. The exact position of the right endpoint could depend on the $\delta$ characters after the end of the periodic substring. Now, the character right after the end of the block could be part of the periodic substring. Thus, we define for every periodic block the right-violation of the block as the first position after the end of the periodic substring.

**Definition 6.3.** Let $[p_i, p_{i+1} - 1]$ be a periodic block such that the principal period of $S[p_i, p_{i+1} - 1]$ is $\rho$, then right-violation$(p_i) = \min\{j \mid j > p_i + \rho, S[j] \neq S[j - \rho]\}$.

We remark that both in the randomized construction of the partitioning set and in the deterministic construction, during the construction the algorithm computes the period length of any periodic block. Therefore in $O(n)$ time the algorithm computes the right-violation$(p_i)$ for any $p_i \in P$ which is an endpoint of a periodic block, without affecting the asymptotic run-time.

Following the discussion above, now the violation character of the periodic block $p_i$ is $S[\text{right-violation}(p_i)]$ instead of $S[p_{i+1}]$. Moreover, instead of inserting the length of the block $p_i$ to $s_i$, we insert the distance right-violation$(p_i) - p_i$. There is only one case that this construction do not cover. In the case that two blocks has the same period, and the same distance to the end of the period - they will have exactly the same representative string. Notice that if the two blocks have the same length, then it is fine that they have the same representative string since the comparison between the suffixes will be due to the following blocks. However, since the right endpoint of the
block depends on the \( \delta \) characters after the periodic substring. Thus, the blocks can be of different lengths only if the \( \delta \) consecutive characters are different. Thus, to cover also this case, we add to the representative string of the periodic block \( p_i \) after the sign and number also the substring \( S[p_i..p_i + 3\delta - 1] \).

To conclude for a periodic block \( [p_i, p_i+1 - 1] \) we define the representative string \( s_i \) as to be the concatenation of the following. The substring (1) \( S[p_i..p_i + 3\delta - 1] \), (2) if the violation character is smaller than the period character, the first character is \(-1\) and the second character is the length of the periodic substring beginning at \( p_i \) (that is \( \text{right-violation}(p_i) - p_i \)). Otherwise, the first character is \(+1\) and the second character is the negative value of the length of the periodic substring beginning at \( p_i \) (that is \( \text{right-violation}(p_i) - p_i \) \cdot \((-1)) \). At the end we add (3) \( S[p_i..p_i + 2\delta - 1] \).

**Conclusion.** To conclude, for any position \( p_i \in P \) we have a representative string of length at most \( 3\delta + 2 + 2\delta = O(\delta) \) over an alphabet of size \( O(n) \). The algorithm sorts all these substrings, in \( O(n \cdot \log n) \) time due to Lemma F.5 or Lemma F.4 (recall that \( \delta = \Theta(\tau) \) for the randomized selection and \( \delta = \Theta(\tau \log^* n) \) for the deterministic selection). After the sorting, we pass over the sorted array, and compute for each \( s_i \) its rank, which is the number of distinct \( s_j \)'s smaller than \( s_i \). We define the string \( S'_{P} \) as the string of length \(|P|\) where the \( i \)'th character is the rank of \( s_i \). There is a natural bijective mapping between suffixes of \( S'_{P} \) and the suffixes of \( S \) beginning at positions of \( P \). Moreover, in the following lemma we prove that for any \( p_i, p_j \in P \) the suffix \( S[p_i..n] < S[p_j..n] \) if and only if \( S'_{P}[i..|P|] < S'_{P}[j..|P|] \) (the \(<\) relation denotes here the lexicographic less than relation).

**Lemma 6.4.** Let \( P \) be a forward synchronized \((\tau, \delta)\)-partitioning set and let \( p_i, p_j \in P \) be two distinct positions, then \( S[p_i..n] < S[p_j..n] \) if and only if \( S'_{P}[i..|P|] < S'_{P}[j..|P|] \).

**Proof.** We will prove \( S[p_i..n] < S[p_j..n] \Rightarrow S'_{P}[i..|P|] < S'_{P}[j..|P|] \), the proof that \( S[p_i..n] > S[p_j..n] \Rightarrow S'_{P}[i..|P|] > S'_{P}[j..|P|] \), is symmetric.

If the representative strings \( s_i \) and \( s_j \) are different, then due to the discussion above, their relative rank determines the order of the corresponding suffixes, and we have \( S'_{P}[i..|P|] < S'_{P}[j..|P|] \).

Otherwise, if \( s_i = s_j \) we will prove that \( S[p_i..p_i+1 - 1] = S[p_j..p_j+1 - 1] \). When proven, it follows that \( S[p_i..n] < S[p_j..n] \) if and only if \( S[p_i+1..n] < S[p_j+1..n] \). Using an induction argument on the length of the sequence of equal pairs \( s_i+k, s_j+k \), we have that if \( S[p_i..n] < S[p_j..n] \) then \( S'_{P}[i..|P|] < S'_{P}[j..|P|] \).

To prove that when \( s_i = s_j \) then \( S[p_i..p_i+1 - 1] = S[p_j..p_j+1 - 1] \), let us distinguish between the two cases of \( s_i \) construction:

- If \( [p_i..p_i+1 - 1] \) is block of type 2a, then \( p_{i+1} - p_i \leq \tau \leq \delta \) and \( s_i = S[p_i..p_i + 3\delta - 1] \).
  Since \( s_i = s_j \), we have \( \text{LCE}(p_i, p_j) \geq 3\delta \geq p_{i+1} - p_i + \delta \). Thus, due to the fact that \( P \) is forward synchronized, we have that \( p_{i+1} - p_i = p_{j+1} - p_j \). It follows that \( S[p_i..p_i+1 - 1] \) and \( S[p_j..p_j+1 - 1] \) are of the same length, and they are both prefixes of \( s_i = s_j \), therefore they must be equal.

- Otherwise, if \( [p_i..p_i+1 - 1] \) is block of type 2b, then it must be that \( [p_j..p_j+1 - 1] \) is also a block of type 2b since otherwise we can use the proof of the previous case to prove that \( [p_i..p_i+1 - 1] \) is not of type 2b. Since \( S[p_i..p_i + 3\delta - 1] \) and \( S[p_j..p_j + 3\delta - 1] \) are prefixes of \( s_i = s_j \) respectively, then \( S[p_i..p_i + 3\delta - 1] = S[p_j..p_j + 3\delta - 1] \). Due to the definition \( s_i \) and \( s_j \) we have that \( s_i[3\delta + 2] \) is right-violation\( (p_i) - p_i \) in absolute value, and the same goes for \( p_j \). Thus, it follows that right-violation\( (p_i) - p_i \) = right-violation\( (p_j) - p_j \). Remark that \( S[p_i..\text{right-violation}(p_i) - 1] \) and \( S[p_j..\text{right-violation}(p_j) - 1] \) are two strings of the same length, and they both share
periodic substrings with period lengths at most $\tau \leq \delta$. Hence, due to Lemma F.3, we have that $S[p_i..right-violation(p_i) - 1] = S[p_j..right-violation(p_j) - 1]$. In addition, due to the definition of $s_i$ and $s_j$ we have that $S[right-violation(p_i)..right-violation(p_i) + 2\delta - 1] = S[right-violation(p_j)..right-violation(p_j) + 2\delta - 1]$. Therefore, it must be that $\text{LC}(p_i, p_j) \geq 2\delta + \text{right-violation}(p_i) - p_i > \delta + p_{i+1} - p_i$. Since $P$ is forward synchronized it follows that $p_{i+1} - p_i = p_{j+1} - p_j$ and thus $S[p_i..p_{i+1} - 1] = S[p_j..p_{j+1} - 1]$.

\[ \square \]

Hence, to compute the SSA of $S$ with the suffixes of $P$, we simply compute the suffix array of $S'_P$ using the algorithm of Kärkkäinen and Sanders \[23\] in $O(|S'_P|) = O(2^k)$ time. To retrieve the SSA of $P$’s suffixes on the string $S$ we just need to replace every index $i$ in the SA with the index $p_i$. So, in total, the construction of the SSA of $P$ takes $O(n \cdot \frac{\delta}{\tau})$ time.

### 6.2 Construct The SST of $B$ Using The SSA of $P$

Assume we have the SSA of $P$’s suffixes, by first compute the SSA of $B$, then compute the LCP array of these suffixes and finally build the SST based on the SSA and LCP array. To compute the SSA of $B$, we use the same technique we already used for the construction of $P$’s SSA. For any $i \in B$, let $v_i = S[i..i + 3\delta - 1]$ be the string of length $3\delta$ from position $i$. In addition, similar to the construction of the SSA of $P$, if the last $2\delta$ characters in $S[i..i + 3\delta - 1]$ are from one large periodic block we add two characters due to the periodicity violation and the length of this block and $3\delta$ characters from the right violation. The algorithm sorts all the $v_i$s for $i \in B$ in $O(n \cdot \frac{\delta}{\tau})$ time and computes the rank of each $v_i$, denoted as $r_i$. In addition, the algorithm computes for each $i \in B$ the index $\text{succ}_P(i + \delta)$ which is the first selected position after index $i + \delta$. Let us denote by $x_i$ the rank of the suffix $S'[\text{succ}_P(i + \delta)..n]$ among all the suffixes of $P$ (which can be retrieved from the SSA of $P$). For each $i \in B$ we associate the pair $(r_i, x_i)$. The SSA of $B$ is obtained by sorting the list of pairs, due to the following lemma, which its proof is omitted since it is almost the same as the proof of Lemma 6.4.

**Lemma 6.5.** Let $i, j \in B$ be two given suffixes, then $S[i..n] < S[j..n]$ if and only if $(r_i, x_i) < (r_j, x_j)$ (where $<$ is the lexicographic smaller than relation).

The sorting of all the pairs is done in $O(n)$ time using radix sort if $\tau \leq \sqrt{n}$ or by regular sort in $O(\frac{n}{\tau} \log \frac{n}{\tau}) = O(n)$ time if $\tau > \sqrt{n}$. After computing the SSA of $B$, to compute the LCP array, we use the LCE data structure from Section 3 with the same partitioning set $P$. Since each LCE query takes $O(\delta)$ time, and we compute $|P| - 1 = O(\frac{\tau}{\tau})$ LCE queries, the total time for this phase is $O(n \cdot \frac{\delta}{\tau})$. With the SSA and the corresponding LCP array, we compute the SST in $O(|P|) = O(2^k)$ time using the algorithm of Kasai et al. \[26\].

Concluding, given a set of suffixes $B$, one can build the SST of $B$ using only $O(|B|)$ working space, by setting $\tau = \frac{n}{|B|}$. Using the randomized selection, this method yields a randomized Las-Vegas algorithm with $O(n)$ running time with high probability for $|B| \geq \Omega(\log^2 n)$, and $O(n)$ expected running time for $|B| < o(\log^2 n)$. Using the deterministic selection this method yields a deterministic algorithm with $O(n(\log \tau + \log^* n))$ running time, for any $|B| \geq \Omega(\log n \cdot \log^* n)$.

### 7 Even Better Deterministic LCE

In this section we show how to use the deterministic selection technique from Section 5, combining with concepts from the so-called difference covers \[29\] technique and the SST construction.
from Section 6 (Theorem 1.5), to improve the LCE query time from $O(\log^* n)$ to $O(\sqrt{\log^* n})$ without any asymptotic penalty in the space usage or construction time. At the very high level, we consider a partitioning set for $\tau' = \frac{\tau}{\sqrt{\log^* n}}$. By applying the algorithm of section 5, one can find a $(O(\tau'), O(\tau' \log^* n)) = \left(O \left( \frac{\tau}{\sqrt{\log^* n}} \right), O(\tau \cdot \log^* n) \right)$-partitioning set $P$ of size $O(\frac{n}{\tau'}) = O \left( \frac{n}{\tau} \cdot \sqrt{\log^* n} \right)$. Using this partitioning set, one can build a data structure that answers LCE queries fast enough.

Combining SST to simplify the LCE computation. Recall that the computation of $\text{LCE}(i, j) = \ell$ query, in the data structure of Section 3 was composed of three phases. In the first phase the algorithm reads $O(\delta)$ characters, then the algorithm finds a length $k$ such both $i + k$ and $j + k$ are positions in the partitioning-set. The second phase treats a large common substring from $i + k$ and $j + k$ until at most $\delta$ characters prior to $i + \ell$ and $j + \ell$ which are the end of the common substring. Notice that, using the SST construction from Section 6 on the positions of the partitioning set, with LCA support, after finding the common $k$ such that $i + k, j + k \in P$ and $S[i..i + k] = S[j..j + k]$, one can compute $\text{LCE}(i, j)$ by adding $k$ to the common extension of $i + k$ and $j + k$. Therefore, we can merge the second and third phases to be done in constant time. In our construction we will use this idea, and focus on finding an offset $k$ such that $S[i..i + k] = S[j..j + k]$ and both $i + k$ and $j + k$ are in the set of positions.

The filtered positions. To reduce the number of positions from the set $P$ we use the difference covers technique. The following lemma states the properties that we will use.

**Lemma 7.1 ([29]).** Let $m$ and $t$ be integers such that $t < m$. There exists a set $DC \subseteq [m]$ of size $|DC| = O \left( \frac{m}{\sqrt{t}} \right)$ such that, for every $i, j \in [m - t]$ there exists $k \leq t$ such that both $i + k \in DC$ and $j + k \in DC$.

We will use this technique to reduce the size of $P$, with $m$ as the size $|P|$ and $t = \log^* n$. We begin with the set $P$, which is a $(\tau', O(\tau' \log^* n))$-partitioning set of size $O(\frac{n}{\tau'})$. Let $p_1, p_2, \ldots, p_h$ be the positions in $P$ sorted in ascending order. We define the set $Q$ as the set that contains all the position $p_i$ such that $i \mod \sqrt{\log^* n} = 0$ or $r \mod \log^* n < \sqrt{\log^* n}$. It is straightforward that $|Q| = \left( \frac{|P|}{\sqrt{\log^* n}} \right)$. Since $|P| = O(\frac{n}{\tau} \sqrt{\log^* n})$ we have that $|Q| = O(\frac{n}{\tau'})$. Our data structure will build the SST of the set $Q$, using the deterministic algorithm from Section 6. Notice that, one can decide whether a position from $P$ is also in $Q$ without actually store $P$, since the decision is dependent only on the rank of the position in $P$. Hence, the algorithm computes the set $Q$ with only $O(\frac{n}{\tau'})$ words of working space (see Appendix D.2).
The query computation. To compute $\text{LCE}(i, j)$ query the algorithm simply reads the suffixes $S[i..n]$ and $S[j..n]$ simultaneously where for each offset $k$, when comparing $S[i+k]$ and $S[j+k]$ the algorithm checks whether both $i+k \in Q$ and $j+k \in Q$. This process is stopped when one of the following occurs. If a mismatch is reached in offset $k$ ($S[i+k] \neq S[j+k]$), then $k$ is reported as $\text{LCE}(i, j)$. Otherwise, if the algorithm reaches a common offset in $Q$ ($i+k, j+k \in Q$), then $k + \text{LCE}(i+k, j+k)$ is reported as $\text{LCE}(i, j)$. Notice that after spending $O(\tau)$ time, the algorithm can perform in additional constant time whether $i+k \in Q$, or $j+k \in Q$, using the linked list of positions and the auxiliary array, as described for the successor computation. In the following lemma we prove that this process will never take more than $O(\tau \cdot \sqrt{\log^* n})$ time.

Lemma 7.2. There exists a constant $c$ such that for any $i, j \in [n]$ with $\text{LCE}(i, j) = \ell > c\tau \sqrt{\log^* n}$ there exists $k < c\tau \sqrt{\log^* n}$ such that both $i+k \in Q$ and $j+k \in Q$.

Proof. Let $\ell' > 1$ be a constant such that $P$ is a $(\tau', \ell' \log^* n)$-partitioning set (of size $O(\frac{n}{\tau})$), in the terms of Section 3 we have $\delta = \ell' \log^* n = c\tau \sqrt{\log^* n}$. We will prove the lemma holds for $c = 4\ell'$. Let $i, j \in [n]$ be positions such that $\text{LCE}(i, j) > c\tau \sqrt{\log^* n}$. By Lemma 3.2, we have:

$$\{p - i \mid p \in (P \cap [i + \delta.i + \ell - \delta - 1])\} = \{p - j \mid p \in (P \cap [j + \delta.j + \ell - \delta - 1])\}.$$

In particular, since $\ell > 4\delta$ we have that:

$$\Delta = \{p - i \mid p \in (P \cap [i + \delta.i + 3\delta - 1])\} = \{p - j \mid p \in (P \cap [j + \delta.j + 3\delta - 1])\}.$$

Notice that $|[i + \delta.i + 3\delta - 1]| = |[j + \delta.j + 3\delta - 1]| = 2\delta = 2\ell' \log^* n$. Hence, since we have assumed that the length of each block induced by $P$ is at most $\tau'$, we have that the number of elements (offsets) in $\Delta$ is at least $\frac{2\delta}{\tau'} = 2\ell' \log^* n > 2\log^* n$. For every element $k \in \Delta$ we have $i+k \in P$ and $j+k \in P$. Let $p_0, p_{\alpha+1}, \ldots, p_{\alpha+\log^* n-1}$ be the first $2\log^* n$ elements in $\{p - i \mid p \in (P \cap [i + \delta.i + \ell - \delta - 1])\}$. There must be an index $\alpha \leq \alpha' \leq \alpha + \log^* n$ such that $\alpha' \mod \log^* n = 0$. Therefore, all the positions $p_{\alpha'}, p_{\alpha'+1}, \ldots, p_{\alpha'+\log^* n-1}$ are positions in $Q$. Let $\beta'$ be the index corresponding to $\alpha'$ such that $p_{\alpha'} - i = p_{\beta'} - j$. We consider the positions $p_{\beta'}, p_{\beta'+1}, \ldots, p_{\beta'+\log^* n-1}$. Since these are $\sqrt{\log^* n}$ consecutive elements in $P$, it must be that at least one of them is taken into $Q$ (since its index in $P$ is an integer multiple of $\sqrt{\log^* n}$). Since for any $0 \leq x < \sqrt{\log^* n}$ we have $p_{\alpha'+x} - i = p_{\beta'+x} - j$ it must be that there exists at least one $x$ such that both $p_{\alpha'+x} \in Q$ and $p_{\beta'+x} \in Q$ and therefore for $k = p_{\alpha'+x} - i$ we have $i+k \in Q$ and $j+k \in Q$, and $k = p_{\alpha'+x} - i < 3\delta \leq c\tau \sqrt{\log^* n}$.\hfill \Box

Complexities. The size of the set $Q$ is $O(\frac{n}{\tau})$, and using the SST - the algorithm can answer LCE queries in $O(\tau \sqrt{\log^* n})$. The construction time of the data structure is $O(n(\log \tau + \log^* n))$ for finding $P$ and $Q$, and for the construction of $Q$’s SST due to the construction of Section 6. Thus, we have proved Theorem 1.3.

8 Ensuring Randomized Selection in Linear Time With High Probability

In Section 4, we achieved linear time in expectation for selecting a small partitioning set. We used a fingerprint hash function $\phi$ and an approximately-min-wise hash function $h$ to compute an ID for each position, and selected positions with minimum values for the partitioning set. Remark that only the partitioning set’s size was in question —we know it is $O(\frac{n}{\tau})$ in expectation, but can’t
guarantee it. If we ensure with high probability that a pair of hash functions \( \phi \) and \( h \) can be found such that ID\(_{\phi,h} \) will induce small partitioning-set, then we could build it in linear time.

In this section, like in Section 4, we assume that \( S \) doesn’t contain long periodic substrings with short periods. With very small modifications, as described in Appendix C.2, the methods that appear next could work in the general case by "skipping" all long periodic substrings while running the algorithms.

We argue that a good selection of \( \phi, h \) can be determined in \( O(n) \) time with high probability while not exceeding \( O(\frac{n}{\tau}) \) words of working space. More precisely, since \( \phi \) may render wrong results only when it has too many collisions on \( S \), and this is not the case with high probability, our main interest is a fast method to pick \( h \in F \) that will make the average amount of partitioning positions per interval of size \( \tau \) to be constant with high probability.

We divide the problem to two: when \( \tau < \log^2 n \) and when \( \tau \geq \log^2 n \). In the first case since \( \tau \) is small we can use sampling of not too many intervals and count the amount of selected positions from \( P \) in each interval, and use that to approximate the total size of \( P \) with high probability. In the second case we run the algorithm on \( \log n \) different hash functions, which will ensure high probability for one of those hash functions to induce a small enough partitioning set. We pick the partitioning set as a subset of a larger \( (\log n, \log n) \)-partitioning set, instead of picking candidates from every position, to allow the parallel process to run in a total of \( O(n) \) time.

8.1 Case 1: \( \tau < \log^2 n \)

Fix a fingerprint function \( \phi \) with no collisions on \( S \). We will use sampling methods in order to evaluate the amount of partitioning positions that a given approximate min-wise independent hash function \( h \) will render when using ID\(_{\phi,h} \) as the ID function.

For any \( 0 \leq i \leq \frac{n-2}{\tau} \) denote \( P_i = P \cap [i\tau + 1..(i+1)\tau] \) and denote \( C_i = |P_i| \). By using the algorithm described in the proof of Lemma 4.3 on the substring \( S[i\tau + 1..(i+1)\tau] \), \( C_i \) can be calculated in \( O(\tau + |P_i|\tau) \) time using \( O(1) \) words of space (by maintaining only the rightmost \( \tau \) uniformly and independently of the other chosen intervals. Denote \( \overline{C} = \sum_{k=1}^{m} C_{i_k} \).

Observation 8.1. With high probability, \( \overline{C} \cdot \frac{\tau}{\tau} - \frac{\tau}{\tau} < |P| < \overline{C} \cdot \frac{\tau}{\tau} + \frac{\tau}{\tau} \).

Proof. Remark that \( E[C_{i_k}] = |P| \cdot \frac{\tau}{n} \) and therefore also \( E[\overline{C}] = |P| \cdot \frac{\tau}{n} \). By Hoeffding’s inequality, since \( 0 \leq C_{i_k} \leq \tau \) and all \( C_{i_k} \) are independent then:

\[
\Pr \left[ |\overline{C} - E[\overline{C}]| \geq 1 \right] \leq 2 \exp \left( -\frac{2m^2}{m\tau^2} \right) \leq 2 \exp \left( -\frac{2\log^5 n}{\log^4 n} \right) = O \left( \frac{1}{n^2} \right)
\]

Thus, by sampling \( m \) intervals we can calculate \( \overline{C} \) and then evaluate with high probability that \( \overline{C} \cdot \frac{\tau}{\tau} - \frac{\tau}{\tau} < |P| < \overline{C} \cdot \frac{\tau}{\tau} + \frac{\tau}{\tau} \). \( \square \)

Observation 8.2. There exists a constant \( c > 0 \), such that \( \Pr \_{\phi,h} (\overline{C} \leq c) > \frac{2}{3} \).

Proof. By Lemma 4.2, we know that \( E_{\phi,h}[|P|] = O(\frac{n}{\tau}) \) and thus by Markov’s inequality for some \( c' > 0 \): \( \Pr \_{\phi,h} (|P| \leq \frac{c' n}{\tau}) > \frac{2}{3} \). But \( |P| \leq \frac{c' n}{\tau} \Rightarrow \overline{C} \leq c' + 1 \) with high probability. Therefore, \( \Pr \_{\phi,h} (\overline{C} \leq c' + 1) > \Pr \_{\phi,h} (|P| \leq \frac{c' n}{\tau}) > \frac{2}{3} \). \( \square \)
Calculating $\mathcal{C}$ takes at most $O(m\tau^2) = O(\log^9 n)$ since for each sampled interval there could be as much as $\tau$ selected partitioning positions, and it may cost up to $O(\tau)$ time to select each of them in the worst case. With high probability, after at most $2\log n$ repetitions of the algorithm for different approximately min-wise hash functions, $\mathcal{C}$ will be small enough, due to Observation 8.2. The partitioning set induced by the selected functions must be calculated and verified as small enough. This will happen with high probability as well due to Observation 8.2. Thus, with high probability after $O(\text{polylog}(n) + n)$ time a small partitioning set will be found. If the run will fail, it will be detected within $O(n)$ time. Therefore, the total running time is expected to be $O(n)$ time with high probability.

### 8.2 Case 2: $\tau \geq \log^2 n$

The method for this case will consist of two steps: The first step is finding $\phi, h_0$ such that the $(\log n, \log n)$-partitioning-set induced by $\text{ID}_{\phi,h_0}$, denoted $P_0$, is smaller than $O\left(\frac{n}{\log n}\right)$. The second step is to draw $\log n$ approximately min-wise independent hash functions $h_i \in \mathcal{F}$ in random, and to calculate for each $h_i$ a $(2\tau, 2\tau)$-partitioning-set using $\text{ID}_{\phi,h_i}$ that will be a subset of $P_0$.

For each $h_i$ and in each interval $[i..i + \tau - 1]$ we pick the positions with the smallest ID value from $P_0 \cap [i..i + \tau - 1]$. Hence,

$$P_i = \left\{ j \in P_0 \mid \text{ID}_{\phi,h_i}(j) = \min_{k \in [i..i+\tau-1] \cap P_0} \{\text{ID}_{\phi,h_i}(k)\} \right\}$$

We prove that $P_i$ is a $(2\tau, 2\tau)$-partitioning set and that its expected size is $O\left(\frac{n}{\tau}\right)$.

**Lemma 8.3.** $P_i$ is a $(2\tau, 2\tau)$-partitioning set.

**Proof.** We prove that $P_i$ holds the properties of Definition 3.1:

**Property 1**

Let $j, k \in [n - 2\tau]$ such that $S[j - 2\tau..j + 2\tau] = S[k - 2\tau..k + 2\tau]$. Since $\tau > \log^2 n \geq \log n$ then for all $\ell \in [-\tau, \tau]$ it follows that $S[j + \ell - \log n..j + \ell + \log n] = S[s + \ell - \log n..s + \ell + \log n]$. Remark that since $P_0$ is a $(\log n, \log n)$-partitioning set then $j + \ell \in P_0$ if and only if $k + \ell \in P_0$. Notice that because $S[j - 2\tau..j + 2\tau] = S[k - 2\tau..k + 2\tau]$ then it also holds that for all $\ell \in [-\tau, \tau]$ : $\text{ID}_{\phi,h_i}(j + \ell) = \text{ID}_{\phi,h_i}(k + \ell)$. Since $j \in P_i$ if and only if $j \in P_0$ and it has the minimal ID in some interval, and since the same holds for $k$ then from all the above it follows that $j \in P_i$ if and only if $k \in P_i$.

**Property 2**

Remark that since $\tau > \log^2 n \geq \log n$ and since $P_0$ is a $(\log n, \log n)$-partitioning set then in any interval of size $\tau$ there is a position from $P_0$. Therefore, the minimal ID on positions from $P_0$ within an interval of size $\tau$ is well-defined. Hence, the proof of Property 2 from Lemma 4.1 holds the same for $P_i$.

**Lemma 8.4.** $\mathbb{E}[|P_i|] = O\left(\frac{n}{\tau}\right)$

**Proof.** The proof is based on the proof of Lemma 4.2, with one change - positions in $P_i$ are picked from $P_0$ which is a $(\log n, \log n)$-partitioning-set, and not all positions in the interval are candidates. Each interval of size $\frac{\tau}{2}$ will contain at least $2\log n$ candidates to $P_i$ from $P_0$. As in the proof of Lemma 4.2, this implies that $\Pr(j \in P_i) = O\left(\frac{\log n}{\tau}\right)$. Hence, by linearity of expectation over all $O\left(\frac{n}{\log n}\right)$ positions in $P_0$ : $\mathbb{E}[|P_i|] = O\left(\frac{n}{\tau}\right)$. □
From the above Lemma and by Markov's inequality it follows that there exists some constant \( c' > 0 \) such that \( \Pr(|P_i| < \frac{c'n}{\tau}) > \frac{2}{3} \). Therefore, with high probability at least one of the approximate min-wise independent hash functions in \( \{h_i \}_{i=1}^{\log n} \) renders a partitioning set \( P_i \) such that \( |P_i| = \mathcal{O}(\frac{n}{\tau}) \).

In order to pick a hash function such that \( |P_i| < \frac{c'n}{\tau} \) we calculate \( |P_i| \) for all \( h_i \), and with high probability the minimal value will be below \( \frac{c'n}{\tau} \). A naive approach would be to use a modification of the algorithm from the proof of Lemma 4.3, but on all the functions \( h_i \) simultaneously. When moving forward simultaneously, this will cost:

1. \( \mathcal{O}(n) \) time in total for calculating positions of \( P_0 \) throughout the run.
2. \( \mathcal{O}(n) \) time for the fingerprint calculation.
3. \( \mathcal{O}(\frac{n}{\log n}) \) hash function evaluations on the fingerprints of positions in \( P_0 \), for each hash function summing up to \( \mathcal{O}(n) \) time, and an extra \( \mathcal{O}(\log n) \) words of space for the positions counter and the rightmost picked position of all \( P_i \).
4. \( \mathcal{O}(\tau) \) for each step-back of the rolling interval algorithm.

All those operations sums up to \( \mathcal{O}(n) \) total time and \( \mathcal{O}(\frac{n}{\tau} + \log n) \) words of space, plus the amount of time consumed on step-backs for all \( h_i \). In the algorithm described in the proof of Lemma 4.3, after each selected position a step-back may occur. This sums up to \( \mathcal{O}(n\log n) \) time consumed for step-backs in total, and thus bounds the total work time asymptotically.

In Appendix C.3, we present a method to calculate \( |P_i| \) or determining that it is bigger than \( \frac{c'n}{\tau} \) for all \( h_i \) using \( \mathcal{O}(n) \) time and \( \mathcal{O}(\frac{n}{\tau} + \log n) \) words of space. The main change is the way we handle step-backs. This improvement uses \( \mathcal{O}(\frac{n}{\tau \log n}) \) words of space for each hash function \( h_i \) to maintain "future" \( P_i \) candidates. By doing so, we reduce the total amount of cases where we have to step-back and recalculate all the IDs to at most \( \mathcal{O}(\log n) \) step-backs per hash function in the worst case. Thus, the amount of time spent in step-backs will sum up to \( \mathcal{O}(\tau \log^2 n) \) time, which is \( \mathcal{O}(n) \) for \( \tau \leq \frac{n}{\log^2 n} \).

Therefore, the following lemmas will apply immediately:

**Lemma 8.5.** Calculating \( |P_i| \) or determining that \( |P_i| > \frac{c'n}{\log n} \) for all \( h_i \) takes \( \mathcal{O}(n + \tau \log^2 n) \) time using \( \mathcal{O}(\frac{n}{\tau} + \log n) \) words of space.

**Lemma 8.6.** For \( \log^2 n < \tau < \mathcal{O}(\frac{n}{\log^2 n}) \), a partitioning set of size \( \mathcal{O}(\frac{n}{\tau}) \) can be constructed in \( \mathcal{O}(n) \) time with high probability using \( \mathcal{O}(\frac{n}{\tau}) \) words of space.

**Proof.** After calculating \( |P_i| \) or determining that it is too big for all \( h_i \) in linear time, with high probability there is at least one hash function \( h_k \) such that \( |P_k| < \frac{c'n}{\tau^2} \). We calculate \( P_k \) explicitly in \( \mathcal{O}(n) \) time and declare it as the partitioning set. \( \square \)

### 8.3 Summary

We have proved that for all \( 1 < \tau < \mathcal{O}(\frac{n}{\log^2 n}) \) there is a method that returns a \((2\tau, 2\tau)\)-partitioning-set using \( \mathcal{O}(\frac{n}{\tau}) \) space that runs in linear time with high probability.

### 9 Overview on Methods for Long Periodic Blocks

In this section we survey the main idea that is used to treat periodic blocks, which are blocks of type 2b due to Definition 3.1. The complete details regarding each section appear in the corresponding appendix.
We clarify the difficulty caused by long periodic blocks in the partitioning of $S$. We will focus now on the LCE computation in such a case, however the difficulty is similar on the deterministic construction of Section 7. When comparing two suffixes, $S[i..n]$ and $S[j..n]$ that share a large common prefix, of length larger than $2\delta$, we are guaranteed that their partitioning into blocks is identical except for at most $\delta$ positions at the margins of the common substrings. This guarantee is based on the locality property of the partitioning set (Property 1 in Definition 3.1), and is formulated in Lemma 3.2. When we consider the case of only short blocks, we exploit this property to find selected positions with the same distance from $i$ and $j$ efficiently. Since the length of each block is at most $\tau$ in this case, such positions must exists in an offset of at most $\delta + \tau$ positions following $i$ and $j$. This corresponding positions are very useful in order to skip a large common substring, which composed of full blocks and should end at most $\delta + \tau$ characters prior the first mismatch.

However, in the general case with long periodic blocks, it is possible that the first selected positions with the same offset from $i$ and $j$ will be in $\omega(\delta)$ positions after $i$ and $j$. In such a case we cannot compare all the characters until the synchronized positions, since it will take too much time. Thus, we utilize the fact that long blocks are corresponding to periodic substrings of $S$. Given two blocks of type 2b that share a common substring of length at least $2\tau$, we do not need to compare the rest of the characters in order to verify that the corresponding characters match each other. This is true because the common substrings guarantee that both blocks have the same period, and therefore all the corresponding characters match each other. The claim is formalized in Lemma F.3.

Hence, after the algorithm verifies that two substrings of length $\delta + 2\tau$ are the same, while this substrings are fully contained in long blocks the algorithm skips all the pairs of corresponding characters in the two blocks. If the two blocks end in exactly the same offset from $i$ and $j$ then the beginning of the following blocks are correlated positions which we use for skipping over the large middle part of the common substrings. Otherwise, if the two blocks end in different offsets from $i$ and $j$, then due to the locality of the partitioning set, it must be that the first mismatch between the suffixes is at most $\delta$ positions after the last synchronized offset inside the long blocks. Then, in such a case we can compare at most additional $\delta$ characters until we find the first mismatch.
Appendix

A Comparison between LCE algorithms

The following table introduces the previous most efficient data structures for the LCE problem together with our new results. The integer ℓ denoted the LCE query result.

| Data Structure | Space | Query time | Preprocessing time | Correct | Trade-off range | Reference |
|----------------|-------|------------|--------------------|---------|-----------------|-----------|
| Monte Carlo    | nτ   | τ          | n                  | w.h.p.  | 1 ≤ τ ≤ n       | [19, Theorem 3.3] |
| Las Vegas      | nτ log (log n) | τ          | log n/τ            | w.h.p.  | 1 ≤ τ ≤ n       | [5]       |
| Deterministic  | 1    | n          | 1                  | always  | -               | naïve algorithm |
|                | nτ   | n + ε      | nτ                 | always  | -               | Suffix tree+LCA |
|                | nτ log (log n) | τ          | log* n            | nτ      | 1 ≤ τ ≤ n       | [4, Section 4] |
|                | nτ   | n          | min{log τ, log nτ} | always  | 1 ≤ τ ≤ n       | [37, Corollary 12] |
|                | nτ   | n          | log + log* nτ     | always  | 1 ≤ τ ≤ n       | new       |

*The LCE data structure is not introduced explicitly, but it can be constructed with the techniques of [4] and [19].

Table 1: LCE data structures

B Missing Details for the LCE construction

In this appendix we complete the missing details from Section 3 about the construction of the LCE data structure, based on a (τ, δ)-partitioning set of size O(1/δ). Given such a set we construct an LCE data structure in O(n) time, using O(2n/τ) words of space, such that any LCE query can be answered in O(δ) time.

B.1 Missing proof

Proof of Lemma 3.2. Let α ∈ {p − i | p ∈ (P∩[i + δ..i + ℓ − δ − 1])}. By definition, α = p − i such that i + α = p ∈ P and δ ≤ α ≤ ℓ − δ − 1. Since LCE(i, j) = ℓ, we have that S[i..i+ℓ−1] = S[j..j+ℓ−1] and in particular S[i+α−δ..i+α+δ] = S[j+α−δ..j+α+δ]. Therefore, by property 1 of Definition 3.1, we have that p = (i + α) ∈ P ⇒ (j + α) ∈ P. Thus, (j + α) − j = α ∈ {p − j | p ∈ (P∩[j+δ..j+ℓ−δ−1])}. The opposite direction is symmetric.

B.2 LCE with periodic blocks

We will follow the discussion above in Section 9. Recall that the computation of LCE(i, j) = ℓ query in the non-periodic case is composed of three phases. The first phase compares 3δ characters from i and j, the last phase reads at most δ + τ characters prior to positions i + ℓ and j + ℓ and the second phase is responsible to all the characters in between. In the general case, each one of
this phases might be changed due to an occurrence of a periodic block at the end of its area. In such a case, we cannot read the whole block, since its length might be \( \omega(\delta) \), however, we can use properties of periodic strings to learn about the substrings without actually read them in the query time. We will use Lemma F.3, and deduce that for our algorithm that after reading \( 2\tau \) characters of two periodic blocks, the content of the remaining suffix of the blocks can be deduced from the substring we already read. Hence, after reading \( 2\tau \) characters without any mismatch, we can skip the rest of the block until the end of the first block.

For the first phase, if we read \( 3\delta \) characters from position \( i \) and \( j \) without any mismatch, we find the first selected position preceding \( i + \delta \) and the first position preceding \( j + \delta \) which are \( \text{succ}(i + \delta) \) and \( \text{succ}(j + \delta) \). Let us denote \( \alpha_i = \text{succ}(i + \delta) - i \) as the offset of the block’s endpoint from \( i \) and similarly we denote \( \alpha_j = \text{succ}(j + \delta) - j \). If \( \alpha_i = \alpha_j \) i.e. if the positions are \( \text{succ}(i + \delta) = i + k \) and \( \text{succ}(j + \delta) = j + k \) for some \( k \in \mathbb{N} \), we can move to the second phase of the algorithm using the blocks beginning in positions \( \text{succ}(i + \delta) \) and \( \text{succ}(j + \delta) \). Otherwise, let us also define \( \alpha = \min\{\alpha_i, \alpha_j\} \) be the minimal of these offsets. It is not hard to prove that \( \text{LCE}(i, j) \leq \alpha + \delta \) as stated in the following lemma.

**Lemma B.1.** If \( S[i..i + 3\delta] = S[j..j + 3\delta] \) and \( \alpha_i \neq \alpha_j \) then \( \text{LCE}(i, j) \leq \alpha + \delta \).

**Proof.** Assume by contradiction that \( \text{LCE}(i, j) > \alpha + \delta \). Since \( 3\delta \leq \alpha < \text{LCE}(i, j) - \delta \), we have that \( S[i + \alpha - \delta..i + \alpha + \delta] = S[j + \alpha - \delta..j + \alpha + \delta] \). Thus, due to property 1 of Definition 3.1 we have \( i + \alpha \in P \iff j + \alpha \in P \), but in this case \( \alpha_i = \alpha_j \) which contradicts lemma assumption. \( \square \)

Hence, in this case we proceed directly to compare the characters from \( S[i + \alpha] \) and \( S[j + \alpha] \). In total the computation time for this case is \( O(\delta) \).

At the transition from the second phase to the third phase, we have assumed in the non-periodic case that the end of the common block is at most \( \tau + \delta \) characters from the end of the common substring. However, this assumption is not guaranteed any more, since it is possible that position \( i + \ell - \delta \) and \( j + \ell - \delta \) are now in a large periodic block. Thus, if the last common block ended at positions \( i + k \) and \( j + k \), then after comparing \( \delta + \tau \) characters if the algorithm does not find any mismatch, it must be that the blocks beginning at positions \( i + k \) and \( j + k \) are of type 2b. Then, we use the same technique as in the first phase. Let us define \( \beta_i = \text{succ}(i + k + 1) - i \) and \( \beta_j = \text{succ}(j + k + 1) - j \), and let \( \beta = \min\{\beta_i, \beta_j\} \). Then, due to Lemma F.3 that \( S[i..i + \beta] = S[j..j + \beta] \). Hence, it must be the case that \( \beta_i \neq \beta_j \) since otherwise the second phase of the algorithm were return \( k \geq \beta \). Thus, similar to Lemma B.1, it must be that \( \text{LCE}(i, j) \leq \beta + \delta \). Thus, the algorithm compares at most \( \delta \) pairs of characters, beginning with \( S[i + \beta] \) and \( S[j + \beta] \) until it finds a mismatch.

## C Missing Details for the Randomized Selection

In this appendix we complete the algorithm for the randomized selection of \( (2\tau, 2\tau) \)-partitioning set. We begin with the missing proofs from Section 4 in Appendix C.1. Then, in Appendix C.2 we complete the missing details for the general case where there are large periodic substrings in \( S \) with small period. Together with the techniques of Section 8, these appendices complete the randomized algorithm for a small \( (2\tau, 2\tau) \)-partitioning set selection in \( O(n) \) time with high probability.

### C.1 Missing proofs for Section 4

**Proof of Lemma 4.2.** Since \( |P| \leq n \) and the probability that any collision occurred when using a random \( \phi \) can be bounded by \( \frac{1}{n^2} \) then by denoting \( B_\phi \) as the event that \( S \) has a collision using \( \phi \)
we can bound:

$$
\mathbb{E}_{\phi,h}[|P|] = \mathbb{E}_{\phi,h}[|P| | B_{\phi}] \cdot \Pr[\overline{B_{\phi}}] + \mathbb{E}_{\phi,h}[|P| | B_{\phi}] \cdot \Pr[B_{\phi}] \leq \mathbb{E}_{\phi,h}[|P| | B_{\phi}] + 1
$$

Therefore, in order to prove the Lemma it is sufficient to bound the expectation by $O(\frac{n}{\tau})$ when assuming that there are no collisions.

Assume that $\phi$ has no collisions over substrings of $S$. For any $i \in [n]$, let $A_i$ be the indicator random variable of the event that $i \in P$. Thus, $|P| = \sum_{i=1}^{n} A_i$. We will show that $\Pr_{\phi,h}[A_i = 1] = O\left(\frac{1}{\tau}\right)$ for all $i \in [n]$, and by linearity of expectations the lemma follows.

By definition, if $i \in P$ then there exist an index $j$ such that $i \in [j..j+\tau-1]$ and also $h(\phi_i) = \min_{k \in [j..j+\tau-1]} \{h(\phi_k)\}$. Because $i \in [j..j+\tau-1]$ then either $|[j..i]| \geq \frac{\tau}{2}$ or $|[i..j+\tau-1]| \geq \frac{\tau}{2}$. This fact together with the fact that $h(\phi_i)$ is minimal in some interval of size $\tau$, implies that if $i \in P$ then:

$$
\left( h(\phi_i) = \min_{k \in [i+1-\tau/2,i]} \{h(\phi_k)\} \right) \text{ or } \left( h(\phi_i) = \min_{k \in [i..i-1+\tau/2]} \{h(\phi_k)\} \right)
$$

Since $S$ contains no $(\tau, \frac{\tau}{6})$-runs, then any two overlapping occurrences of a substring of length $\tau$ are at least $\frac{\tau}{6}$ apart. Together with the fact that $\phi$ has no collisions it follows that there are at least $\frac{\tau}{6}n$ distinct fingerprints in any interval of size more than $\frac{\tau}{6}$ positions. Therefore, $|\{\phi_j\}_{j \in [i+1-\tau/2,i]}| \geq \frac{\tau}{6}$ and the same holds for the interval $[i..i-1+\tau/2]$. Since $\mathcal{F}$ is $\left(\frac{1}{2}, \tau\right)$-min-wise independent, then:

$$
\Pr_{\phi,h}\left[h(\phi_i) = \min_{k \in [i+1-\tau/2,i]} \{h(\phi_k)\} \right] \leq \frac{6}{\tau}(1 + \frac{1}{2}) = \frac{9}{\tau}
$$

and the same holds for the interval $[i..i-1+\tau/2]$. Therefore, $\Pr_{\phi,h}[A_i = 1] \leq \frac{18}{\tau}$, and thus the lemma holds.

\[\square\]

C.2 Details for Periodic Blocks

Recall that a $(\delta, \rho)$-run is a substring of $S$ of length at least $\delta$ and with period length of at most $\rho$. We have seen in Sections 4 and 8 that when $S$ contains no $(\tau, \frac{\tau}{6})$-runs one can build a small partitioning set in a linear time. Now, we will show how to efficiently partition $S$ into two types of substrings, those with short periods and those which do not contain any large enough substring with short period. Later, we show that together with the techniques of Section 4 a small $(2\tau, 2\tau)$-partitioning set could be built easily. The same technique will also work for the combination with Section 8, to receive a small $(2\tau, 2\tau)$-partitioning set in $O(n)$ time with high probability. Moreover, the technique we use will produce a forward synchronized partitioning-set to allow fast SST construction. Notice that the choice of positions for the partitioning set in the non-periodic case is forward synchronized. This is a sub-result of Lemma C.7.

**Defining candidate intervals** We claim that for every $(\tau, \frac{\tau}{6})$-run there is an interval of size $\frac{\tau}{2}$ that is fully contained in the run, and that this interval contains no other run. We use this claim to efficiently find all $(\tau, \frac{\tau}{6})$-runs.

Let $\alpha$ be an integer such that $0 \leq \alpha \leq \frac{2n}{\tau}$. Define $I_{\alpha} = [\frac{\alpha \tau}{2} + 1..(\alpha+1)\tau]$. We look on a fingerprint function $\psi$ that has no collisions on substrings of length $\frac{\tau}{6}$ from $S$. For all $i \in [n - \frac{\tau}{6} + 1]$ denote $\psi_i = \psi(S[i..i+\frac{\tau}{6}])$. We claim that if $S' = S[i..j]$ is a $(\tau, \frac{\tau}{6})$-run, then it could be detected easily using $\psi$. First notice that if $S'$ is a $(\tau, \frac{\tau}{6})$-run, then its length is at least $\tau$. Therefore, the next observation follows immediately.
Observation C.1. If $S' = S[i..j]$ is a $(\tau, \frac{\tau}{6})$-run, then there is an integer $0 \leq \alpha \leq \frac{2n}{\tau}$ such that $I_\alpha = \left[ \frac{\alpha \tau}{2} + 1..\frac{(\alpha+1)\tau}{2} \right] \subseteq [i..j]$.

We next claim that using $\psi$ we can detect the principal period of any $(\tau, \frac{\tau}{6})$-run.

Claim C.2. Let $\alpha$ be an integer such that $I_\alpha$ is fully contained in $S' = S[i..j]$ which is a $(\tau, \frac{\tau}{6})$-run. Then the size of the set

$$\left\{ \ell \in \left[ \frac{\alpha \tau}{2} + 1..\frac{\alpha \tau}{2} + \frac{\tau}{3} \right] \mid \psi_\ell = \min_{k \in \left[ \frac{\alpha \tau}{2} + 1..\frac{\alpha \tau}{2} + \frac{\tau}{3} \right]} \psi_k \right\}$$

is at least 2, and the minimum distance between two indices in that set is exactly the length of the principal period of $S'$.

Proof. Since the principal period length of $S'$, $\rho_{S'} \leq \frac{\tau}{6}$ then for all $k \in \left[ \frac{\alpha \tau}{2} + 1..\frac{\alpha \tau}{2} + \frac{\tau}{3} - \rho_{S'} \right]$ it follows that $S[k..k + \frac{\tau}{6} - 1] = S[k + \rho_{S'}..k + \rho_{S'} + \frac{\tau}{6} - 1]$ and therefore $\psi_k = \psi_{k + \rho_{S'}}$ for all $k \in \left[ \frac{\alpha \tau}{2} + 1..\frac{\alpha \tau}{2} + \frac{\tau}{3} - \rho_{S'} \right]$. This means that the first $\frac{\tau}{6}$ fingerprints of $S'[\frac{\alpha \tau}{2} + 1..\frac{(\alpha+1)\tau}{2}]$ have exactly $\rho_{S'}$ different values. Since $\rho_{S'} \leq \frac{\tau}{6}$ then it means that all those fingerprint values will appear at least twice, including the fingerprint with minimum value.

Moreover, let $\ell \in \left[ \frac{\alpha \tau}{2} + 1..\frac{\alpha \tau}{2} + \frac{\tau}{6} - \rho_{S'} \right]$ be a position such that the fingerprint $\psi_\ell$ has the minimum value. Since $\psi$ has no collisions, there is no position in $[\ell+1..\ell + \rho_{S'} - 1]$ that also have the minimum fingerprint value. By way of contradiction, assume that for some $\ell' \in [\ell+1..\ell + \rho_{S'} - 1]$ $\psi_{\ell'} = \psi_\ell$. Since $\psi$ has no collisions, this means that $S[\ell..\ell + \frac{\tau}{6} - 1] = S[\ell'..\ell' + \frac{\tau}{6} - 1]$. Since $\ell' - \ell < \rho_{S'} \leq \frac{\tau}{6}$ then the two substrings overlap and thus the substring $S[\ell..\ell + \frac{\tau}{6} - 1]$ has a period of length $\ell' - \ell$. But it is also a substring of $S'$ and thus it has a period of $\rho_{S'}$. From Theorem F.1, since the length of $S[\ell..\ell + \frac{\tau}{6} - 1]$ is $\ell' - \ell + \frac{\tau}{6} \geq \ell' - \ell + \rho_{S'}$ then it has a period of $\gcd(\ell' - \ell, \rho_{S'})$. It follows from Lemma F.2 that since $S[\ell..\ell + \frac{\tau}{6} - 1]$ is a substring of $S'$ and its length is more than $\rho_{S'}$ then $S'$ also has a period of length $\gcd(\ell' - \ell, \rho_{S'})$. But since $\gcd(\ell' - \ell, \rho_{S'}) \leq \min\{\ell' - \ell, \rho_{S'}\} = \ell' - \ell < \rho_{S'}$ then $S'$ has a period of length less than $\rho_{S'}$, which come with contradiction to the principal period’s length minimality.

Finding and Verifying Candidate Intervals We try to find intervals as described above that are a part of a $(\tau, \frac{\tau}{6})$-run. For every interval $I_\alpha = \left[ \frac{\alpha \tau}{2} + 1..\frac{(\alpha+1)\tau}{2} \right]$ we say it is a candidate if the minimum value fingerprint in the first $\frac{2\tau}{9}$ positions of the interval repeats at least twice. Each candidate is potentially a part of a $(\tau, \frac{\tau}{6})$-run. Checking if an interval is a candidate is done by using the fingerprint function $\psi$ and calculating it by rolling window for the first $\frac{2\tau}{9}$ positions of the interval and maintaining the value and position of the minimum fingerprint value. The check will stop if the minimum fingerprint value repeats at least twice, or if none was found. This check takes in the worst case $O(\tau)$ time per interval. For every candidate interval, the difference between the two positions with the minimum fingerprint value is maintained. This difference is the suspected principal period of a run this interval is contained in.

We say a candidate interval is verified if it is fully contained in a $(\tau, \frac{\tau}{6})$-run. From Observation C.1, it follows that each $(\tau, \frac{\tau}{6})$-run has at least one verified candidate. The fact weather a candidate interval is contained in a $(\tau, \frac{\tau}{6})$-run can be checked in $O(\tau)$ time by first verifying that the occurrences of the minimum fingerprint value are at most $\frac{\tau}{6}$ apart, then verifying that there is no fingerprint collision. If there is no collision then we now know the principal period length and the period itself. By extending up to at most $\tau$ positions to the left and to the right until the period breaks we can verify that the periodic substring is at least of length $\tau$, and that it contains the
candidate interval. Therefore, the candidate interval can be verified to be contained in a \((\tau, \frac{5}{6})\)-run in \(O(\tau)\) time.

For each interval \(I_\alpha\) we spend \(O(\tau)\) time and \(O(1)\) words of space to check if it is a candidate, and then another \(O(\tau)\) to verify if it is a part of \((\tau, \frac{5}{6})\)-run. Since there are \(O(\tau\rho)\) intervals, then all candidates can be found and verified in \(O(n)\) time, using \(O(\tau)\) words of space to store the verified candidates' indices. Notice that the algorithm does not assume the fingerprint has no collisions. The algorithm rather uses the fingerprint value to hint on positions of interest, and then the algorithm verifies the equality. Since the fingerprint function is collision free with high probability, then the algorithm is Las-Vegas, and with high probability its expected running time is \(O(n)\).

**Overlapping Runs** Remark that two runs with different periods may overlap if the ir periods share a common substring. We claim next that two \((\tau, \frac{5}{6})\)-runs may only overlap in less than \(\frac{5}{3}\) positions.

**Lemma C.3.** If \(u = S[i..j]\) and \(v = S[i'..j']\) are two overlapping \((\tau, \frac{5}{6})\)-runs, meaning \(i \leq i' \leq j\), then either \(j - i' + 1 < \frac{5}{3}\) or \(u = v\)

**Proof.** The second case is trivial. Therefore, assume \(i < i' \leq j < j'\). Let \(\rho_u, \rho_v\) be the lengths of the principal periods of \(u, v\) respectively. By way of contradiction, assume that \(j - i' + 1 \geq \frac{5}{3}\). Hence, \(w = S[i'..j]\) is a string such that both \(\rho_u\) and \(\rho_v\) are periods of \(w\). Since \(\frac{5}{3} \geq \rho_u + \rho_v\) then \(j - i' + 1 \geq \rho_u + \rho_v - \gcd(\rho_u, \rho_v)\). Therefore, due to Theorem F.1, \(w\) has a period length of \(\gcd(\rho_u, \rho_v)\).

\[w = S[i'..j]\] is a substring of both \(u\) and \(v\) of length greater than their period lengths, and also \(u\) has a period length that divides both \(\rho_u\) and \(\rho_v\). Hence, by Lemma F.2 \(\gcd(\rho_u, \rho_v)\) is a period length of both \(u\) and \(v\), but since \(\rho_u\) and \(\rho_v\) are the principal period length of \(u\) and \(v\) respectively, and hence the minimum period length, then \(\rho_u = \rho_v = \gcd(\rho_u, \rho_v)\).

Denote \(\rho = \rho_u = \rho_v\). From the assumption that \(i < i'\) we know that \(i \leq i' - 1\). We also know that \(S[i' - 1] = S[i' + \rho - 1]\) since \(u = S[i..j]\) is periodic and \(j \geq i' + \rho - 1\). But \(\rho\) is also a period of \(v\), hence the run \(v\) can be extended to the left without breaking its periodicity. This contradicts the maximality of \(v\) as a run of \(S\), hence it must be that \(j - i' + 1 < \frac{5}{3}\). \(\square\)

From Lemma C.3, the next corollary follows.

**Corollary C.4.** An interval \(I_\alpha\) may be fully contained in at most one \((\tau, \frac{5}{6})\)-run.

**Proof.** By way of contradiction, if \(I_\alpha\) is fully contained in at least two \((\tau, \frac{5}{6})\)-runs, then those two runs overlap with at most \(\frac{5}{3}\) characters, which contradicts Lemma C.3. \(\square\)

Putting all together, the next lemma follows.

**Lemma C.5.** At the end of the candidate interval verification process, for every \((\tau, \frac{5}{6})\)-run there is a verified candidate interval that is fully contained only in the run, and not in any other \((\tau, \frac{5}{6})\)-run.

**Proof.** From Observation C.1 there is an interval \(I_\alpha\) that is fully contained in \(u\). It follows from Claim C.2 that this interval matches the definition of a verified candidate interval and that in the process, the principal period of \(u\) is found. Finally, from Corollary C.4 follows that \(I_\alpha\) is contained only in \(u\). \(\square\)

From the above lemma we learn that for every \((\tau, \frac{5}{6})\)-run there is at least one verified candidate that is unique to it. Therefore, by finding all verified candidate intervals, we get at least one "certificate" for every \((\tau, \frac{5}{6})\)-run. Thus, all that is left is to mark all those runs.
From candidate intervals to runs-partitioning Our aim is to partition the string to long periodic substrings with small period lengths and to substrings that contain no such substrings. We will build a set \( Q \) that will define the partitioning of the string accordingly. After finding all candidate intervals, the algorithm stores all verified candidates, meaning all intervals that are contained in some \((\tau, \frac{\tau}{6})\)-run. The algorithm scans the verified candidates from right to left. We divide the scan into iterations, where at the end of each iteration a whole \((\tau, \frac{\tau}{6})\)-run is processed and its endpoints stored in \( Q \), perhaps without parts of his right margin in a case of overlapping runs.

Each iteration starts with the rightmost verified candidate interval that wasn’t marked. The algorithm extends the period to the right until the period breaks or until a previously marked run is found, and to the left as long as the period doesn’t break. Denote \( u_i = S[q_i..q'_i] \) be the run found in the \( i \)th iteration. First, the pair \((q_i, q'_i)\) is stored in the set \( Q \) and marked as a run. If the pair found in the previous iteration, \((q_{i-1}, q'_{i-1})\), is not adjacent to the current iteration pair - meaning \( q_{i-1} \neq q'_i + 1 \) - we add the pair \((q'_i + 1, q_i - 1)\) to \( Q \) and mark it as the endpoints of a non-run substring. Finally, every verified candidate interval that is fully contained in \( u_i \) is marked.

Since all verified candidate intervals were processed, it follows from Lemma C.5 that all \((\tau, \frac{\tau}{6})\)-runs were found, and are stored in \( Q \) together with their principal period. Since there are at most \( O(\frac{n}{\tau}) \) verified candidate intervals, then there are at most \( O(\frac{n}{\tau}) \) triplets contained in \( Q \).

This process scan each character at most once as a part of a \((\tau, \frac{\tau}{6})\)-run, and at most once as a character that breaks the run. The verified candidate intervals’ marking takes an additional negligible \( O(\frac{n}{\tau}) \) time. Therefore, the total running time is \( O(n) \). Regarding the space usage - each iteration used constant words of space, and maintaining \( Q \) takes \( O(\frac{n}{\tau}) \) words of space due to the bound on its size.

To conclude, using \( O(n) \) time and \( O(\frac{n}{\tau}) \) words of space we found all \((\tau, \frac{\tau}{6})\)-runs, and thus could partition it efficiently.

From runs-partitioning to partitioning-set After finding all the runs, we can construct a partitioning-set. Basically, the technique is to use the scanning algorithm from the proof of Lemma 4.3, with modifications regarding runs from \( Q \). We start with an empty set \( P \) and add positions to it, picked by the scanning algorithm, as follows. If the first pair in \( Q \) is non-run substring of length at least \( \tau \), the scanning algorithm runs on its positions and update \( P \) accordingly, and finish at the last interval of length \( \tau \) contained in the substring. If it is of a length shorter than \( \tau \), we pick no position for \( P \) from that substring. The next pair in \( Q \) marks the endpoints of a \((\tau, \frac{\tau}{6})\)-run, denoted \( u = S[q..q'] \). First we add \( q \) to \( P \). If the next pair in \( Q \) is marked as a run substring, we recursively run the process with the next \((\tau, \frac{\tau}{6})\)-run in \( Q \). Otherwise, \( q' + 1 \) is an endpoint of a non-run substring. We skip processing the characters of the string from position \( q \) until position \( q' - \tau + 1 \). Then we run the scanning algorithm from this position, picking positions for \( P \) from any interval of size \( \tau \) contained in the substring. Finally, we iterate using this algorithm on all pairs of \( Q \).

The only exception will be the last substring in \( Q \) if it is marked as a non periodic string. Let the second to last substring in \( Q \) be \( S[q..q'] \). If \( q' \in [n - \tau + 1..n - 1] \), we pick \( q' + 1 \) as the last position in \( P \). Otherwise, we run the scanning algorithm as described above from position \( q' - \tau + 1 \), until picking the last position of \( P \) from the interval \([n - 2\tau + 1..n - \tau]\), as was done in the non-periodic case.

By using the scan algorithm from the last \( \tau \) positions of a \((\tau, \frac{\tau}{6})\)-run, we select the periodic block end, and thus the next non-periodic short block start position according to the ID values. This will allow us to prove that the output \( P \) is a forward synchronized partitioning set. Firstly
we prove that at the end of this process, $P$ is a $(2\tau, 2\tau)$-partitioning-set. We then claim that this partitioning set is also forward synchronized. Finally, we show that $P$ can be computed in $O(n)$ expected time, and commit that $|P| = O(n^2/\tau)$.

**Lemma C.6.** Using the modified scan algorithm described above, the output $P$ is a $(2\tau, 2\tau)$-partitioning-set.

**Proof.** We show that $P$ has the properties of a $(2\tau, 2\tau)$-partitioning-set according to Definition 3.1.

**Property 1** Let $i, j \in [1 + 2\tau, n - 2\tau]$ such that $S[i - 2\tau..i + 2\tau] = S[j - 2\tau..j + 2\tau]$. Denote $u_i, u_j$ the substrings induced from the set $Q$ such that $i, j$ are positions within the substrings $u_i, u_j$ respectively.

Notice that $u_i$ is either fully contained in $S[i - 2\tau..i + 2\tau]$ or it intersects with at least $2\tau$ characters of it, and the same holds for $u_j$ and $S[j - 2\tau..j + 2\tau]$. Therefore, if $u_i$ is a $(\tau, \frac{6}{5})$-run, then position $i$ is contained in such a run. Thus, it follows that $u_i$ is a $(\tau, \frac{6}{5})$-run if and only if $u_j$ is a $(\tau, \frac{6}{5})$-run. Hence, both $u_i$ and $u_j$ are marked the same in $Q$. We shall differentiate each possibility.

1. $(u_i \text{ and } u_j \text{ are marked as non-run substrings})$ In this case, positions are picked for $P$ by their IDs. Since $S[i - 2\tau..i + 2\tau] = S[j - 2\tau..j + 2\tau]$ then from the ID definition it follows immediately that $i \in P \iff j \in P$.

2. $(u_i \text{ and } u_j \text{ are both marked as run substrings})$ Only two positions are picked for $P$ within a $(\tau, \frac{6}{5})$-run: The first position, and the position with the minimum ID out of the last $\tau$ positions of the run if the adjacent substring from $Q$ is marked as non-run substring. First, since $S[i - 2\tau..i + 2\tau] = S[j - 2\tau..j + 2\tau]$ then if $i$ is at the start of the run $u_i$ then $j$ is at the start of the run $u_j$, and vice versa. Therefore, in this case $i \in P \iff j \in P$.

Notice that, again, since $S[i - 2\tau..i + 2\tau] = S[j - 2\tau..j + 2\tau]$, then if $i$ is in the last $\tau$ positions of the run $u_i$, so does $j$ (regarding the run $u_j$). Moreover, they are at the same distance from the end of the run, and since the substrings from $Q$ consequent to $u_i, u_j$ respectively share at least $\tau$ characters, then those consequent substrings are of the same type. Therefore, either those consequent substrings are both $(\tau, \frac{6}{5})$-run, and thus neither $i$ nor $j$ are picked for $P$. If those substrings are both non-runs, then positions are picked by their ID values. As explained in the case of non-run substrings, since both positions will be selected by their ID value it yields that $i \in P \iff j \in P$.

**Property 2** Let $1 = p_0 < p_1 < \cdots < p_k < p_{k+1} = n + 1$ be all the positions in the partition $P \cup \{1, n+1\}$.

If $1 = p_0 \notin P$ then no $(\tau, \frac{6}{5})$-run start at $p_0$. Therefore, there is either a $(\tau, \frac{6}{5})$-run that starts at a position $k \in [1..\tau]$ or the algorithm will pick a position $k \in [1..\tau]$ for $P$ using the ID function. Hence, in the case where $1 = p_0 \notin P$, it follows that $p_1 - p_0 \leq 2\tau$.

As for the general case, let $p_j < p_{j+1}$ be two consecutive positions in $P \cup \{n+1\}$. We would like to prove that either $p_{j+1} - p_j \leq 2\tau$ or that $S[p_j..p_{j+1} - 1]$ is a periodic substring with period length smaller than $2\tau$. There are 3 cases: $p_j$ is the beginning of a periodic block, $p_j$ is a position selected for $P$ due to its ID value, or $p_j$ is the last position picked for $P$ - selected after a $(\tau, \frac{6}{5})$-run that ended in the last $\tau$ characters of $S$. The last case is trivial since by definition $p_{j+1} - p_j = n + 1 - p_j < \tau$.

Assume that $p_j$ is the start of a $(\tau, \frac{6}{5})$-run. Although it might be that $p_{j+1} - p_j > 2\tau$, since $S[p_j..p_{j+1} - 1]$ is a periodic substring with period smaller than $\frac{6}{5}$, then this block complies with the requirement of Property 2b of the partitioning set.
On the other hand, assume that \( p_j \) was selected for \( P \) due to its ID value. If \( p_{j+1} = n + 1 \) then \( p_j \) was selected for \( P \) due to its minimal ID value in the interval \([n - 2\tau - 1..n - \tau] \). If \( p_{j+1} \) was also selected due to its ID value, then from the ID selection criteria it must hold that \( p_{j+1} - p_j \leq 2\tau \). Otherwise, if \( p_{j+1} \) is the first position in a \((\tau, \frac{\tau}{6})\)-run, then \( p_j \) must be within at most \( 2\tau \) characters, and Property 2 holds.

We now prove that \( P \) is forward synchronized partitioning-set. Recall that due to Definition 6.1, we have to prove that for any pair \( p_i, p_j \in P \) such that \( \text{LCE}(p_i, p_j) > p_{i+1} - p_i + \delta \) then \( p_{i+1} - p_i \) is the first position in a \((\tau, \frac{\tau}{6})\)-run, then from the ID selection criteria it must hold that \( p_{j+1} - p_j \leq 2\tau \).

**Lemma C.7.** Using the modified scan algorithm described above, the output \( P \) is a forward synchronized \((2\tau, 2\tau)\)-partitioning-set.

**Proof.** Let \( p_i, p_j \in P \) such that \( \text{LCE}(p_i, p_j) > p_{i+1} - p_i + 2\tau \). Let \( \Delta = p_{i+1} - p_i \). Hence, \( S[p_i..p_{i+1} + 2\tau] = S[p_j..p_j + \Delta + 2\tau] \). There are four cases for \( p_i \) and \( p_{i+1} \):

1. Assume both \( p_i \) and \( p_{i+1} \) were picked to \( P \) by their ID value. Therefore, \( \text{ID}(p_{i+1}) \) was either the first ID after \( p_i \) such that \( \text{ID}(p_i) \geq \text{ID}(p_{i+1}) \) or it has the minimum value in the interval \([p_i + 1..p_i + \tau] \). In either case since \( S[p_i..p_{i+1} + 2\tau] = S[p_j..p_j + \Delta + 2\tau] \) then the IDs in that interval are equal and thus it also holds that \( p_j \) is not the start of a periodic block and that \( p_{j+1} = p_j + \Delta \).

2. Assume both \( p_i \) and \( p_{i+1} \) are the start of periodic blocks. Therefore, \( p_{i+1} \) is the start of a \((\tau, \frac{\tau}{6})\)-run with period that is different then the period of the \((\tau, \frac{\tau}{6})\)-run that starts at \( p_i \). But since \( S[p_i..p_{i+1} + 2\tau] = S[p_j..p_j + \Delta + 2\tau] \) then \( p_j \) is also a start of a \((\tau, \frac{\tau}{6})\)-run, and \( p_j + \Delta \) is the start of \((\tau, \frac{\tau}{6})\)-run with different period then the run that starts at \( p_j \). Therefore, \( p_{j+1} = p_j + \Delta \).

3. Assume that \( p_i \) is the start of a periodic block and \( p_{i+1} \) was selected for its ID value. By the position selection algorithm, it means that \( p_{i+1} \) is selected from the last \( \tau \) positions of the \((\tau, \frac{\tau}{6})\)-run that starts at \( p_i \), and thus the run length is less than \( \ell < \Delta + \tau \). By the selection algorithm selecting \( p_{i+1} \) means that \( \text{ID}(p_{i+1}) = \min\{\text{ID}(k) \mid k \in [p_i + \ell - \tau..p_i + \ell + 1]\} \). Those IDs are based solely on the characters in \([p_i + \ell - \tau..p_i + \ell + \tau - 2] \subseteq [p_i + \ell - \tau..p_i + \ell + \tau - 1]\). Since \( S[p_i..p_{i+1} + 2\tau] = S[p_j..p_j + \Delta + 2\tau] \), then \( p_j \) is also a start of a \((\tau, \frac{\tau}{6})\)-run, with the exact same length \( \ell \), and also \( \text{ID}(p_j + \Delta) = \min\{\text{ID}(k) \mid k \in [p_j + \ell - \tau..p_j + \ell - 1]\} \). Therefore, \( p_{j+1} = p_j + \Delta \).

4. Assume that \( p_i \) is selected for its ID value and that \( p_{i+1} \) is the start of a \((\tau, \frac{\tau}{6})\)-run. This means that \( \Delta < 2\tau \) and that \( \text{ID}(p_i) = \min\{\text{ID}(k) \mid k \in [p_i..p_{i+1} - \tau]\} \). Since \( S[p_i..p_{i+1} + 2\tau] = S[p_j..p_j + \Delta + 2\tau] \) then at \( p_j + \Delta \) a periodic string of length at least \( \tau \) and of period at most \( \frac{\tau}{6} \) starts, right after a non periodic sub-string. Hence, \( p_j + \Delta \) will be chosen for \( P \). In addition, it also follows that \( \text{ID}(p_j) = \min\{\text{ID}(k) \mid k \in [p_j..p_j + \Delta - \tau]\} \). Therefore, no other selection is picked between \( p_j \) and \( p_j + \Delta \). Thus, \( p_{j+1} = p_j + \Delta \).

Now that we ensured that \( P \) is a forward synchronized \((2\tau, 2\tau)\)-partitioning-set, finally we show that the process of selecting \( P \) using the runs-partition will take expected linear time and ensures that \( |P| = O(\frac{n}{\mu}) \). From Lemma 4.3, we know that using the ID function it is possible to select a partitioning-set of a non-periodic substring using time linear in the substring’s length, with an
additional $\mathcal{O}(\tau)$ time is needed for every step-back. When processing $S$ we skip all $(\tau, \frac{n}{\tau})$-runs completely, other than the last $\tau$ characters, and every position is processed as part of only one non-periodic substring. Therefore, similarly to the results in Section 4 a partitioning-set of size $\mathcal{O}(\frac{n}{\tau})$ is expected to be built in $\mathcal{O}(n)$ time.

Remark that the techniques introduced in Section 8 work similarly in linear time.

**Conclusion** In order to build a partitioning-set of a general string $S$ containing $(\tau, \frac{n}{\tau})$-runs we broke the problem to three steps. We used $\mathcal{O}(n)$ expected time with high probability and $\mathcal{O}(\frac{n}{\tau})$ space to find intervals that might contain a $(\tau, \frac{n}{\tau})$-run. Then another $\mathcal{O}(n)$ time and $\mathcal{O}(\frac{n}{\tau})$ space to find all $(\tau, \frac{n}{\tau})$-runs and building $Q$ to partition $S$ to different $(\tau, \frac{n}{\tau})$-runs and to contiguous substring that contain non of those. Finally, another $\mathcal{O}(n)$ expected time and $\mathcal{O}(\frac{n}{\tau})$ space were used to create a $(2\tau, 2\tau)$-partition-set, or $\mathcal{O}(n)$ expected time with high probability and $\mathcal{O}(\frac{n}{\tau} + \log n)$ space using the techniques of Section 8. Summing everything up we get the results of Theorem 1.1 for a general string $S$.

### C.3 Faster Partitioning Set Counting For High Probability Method

As described in Section 8, in order to reduce the time that is spent on stepping back while scanning the string in order to calculate $|P_i|$ for all $h_i$, we save future candidates in advance - up to $\mathcal{O}(\frac{n}{\tau \log n})$ per hash function.

The scanning algorithm will have to step-back after the $j^{th}$ iteration if $\text{ID}_{\phi, h_i}(j) < \text{ID}_{\phi, h_i}(k)$ for all $k \in [j + 1..j + \tau - 1]$. In this case the position with the minimal value in $[j + 1..j + \tau]$ need to be picked. By maintaining the candidates for several windows in advance, the algorithm avoids stepping back when the most recent candidate was out of the current window. A position $\ell$ is a candidate in the $j^{th}$ iteration if $j \leq \ell < j + \tau - 1$ and $\text{ID}_{\phi, h_i}(\ell) \leq \text{ID}_{\phi, h_i}(k)$ for all $k \in [\ell + 1..j + \tau - 1]$. Remark that by definition, if the candidates at a certain iteration are $\{\ell_1, \ldots, \ell_q\}$ such that $\ell_1 < \cdots < \ell_q$ then $\text{ID}_{\phi, h_i}(\ell_1) \leq \cdots \leq \text{ID}_{\phi, h_i}(\ell_q)$. Hence, at the end of each iteration the leftmost candidate $\ell_1$ is a part of the partitioning sets.

For a certain hash function $h_i$, the candidates and their IDs are maintained in a deque, denoted $D_i$. Candidates are pushed into $D_i$ from its right side in an ordered way in both position index and ID value. After calculating the ID of some position $\ell$ the algorithm peeks at the right end of $D_i$ and it pops all the obsolete candidates $(k, \text{ID}_{\phi, h_i}(k))$ such that $\text{ID}_{\phi, h_i}(\ell) < \text{ID}_{\phi, h_i}(k)$, and then pushes $(\ell, \text{ID}_{\phi, h_i}(\ell))$ to the right end of $D_i$. When $D_i$ reaches the point where it contains $\frac{c n}{\tau \log n}$ candidates, a flag denoted $f_i$ is risen. While $f_i$ is raised, no new candidates are added to $D_i$ unless they remove other candidates (meaning only better candidates are inserted). If such a better candidate is found, the flag $f_i$ is lowered back.

The algorithm maintains throughout its run the rightmost position picked for each $P_i$ - denoted $q_i$ and the current size $|P_i|$. It processes the string as follows for every $i$. Insert all the candidates from $[1..\tau]$ into $D_i$ according to the insertion rules from above. Remember the leftmost position in $D_i$ for the partitioning set $P_i$, and increment $|P_i|$ value. Afterwards the algorithm works in iterations. In the $j^{th}$ iteration check if the leftmost candidate in $D_i$ is the position $j - 1$. If so, pop it from the left end of $D_i$. If $D_i$ isn’t empty - insert $j + \tau - 1$ and its ID into $D_i$ according to the insertion rules from above. After inserting a new position to $D_i$, check if the leftmost position in $D_i$ has already been picked for $P_i$ by comparing it to $q_i$. If the leftmost position in $D_i$ wasn’t picked for $P_i$ yet, update $q_i$ with the new position and increment the current size $|P_i|$. Otherwise, if $D_i$ is empty, it must be because the deque $D_i$ was full, and its flag $f_i$ was raised and never lowered. So first - lower the flag $f_i$. Then remark that there is no knowledge on any candidates in the current
interval, thus the algorithm needs to step-back and recalculate the fingerprints and IDs of all the positions in \([j..j + \tau - 1]\), exactly as it did in the first iteration.

**Observation C.8.** A recalculation of \(\tau\) fingerprints and the IDs of candidates may occur at most \(\log \, n\) times.

**Proof.** A recalculation in the above algorithm happens only when the deque \(D_i\) is empty and \(f_i\) is still raised. This happens only when \(D_i\) was full at the end of some iteration, the \(j^{th}\), and afterwards no new candidates were inserted and all of them were removed since they weren’t relevant for the new intervals. Before a candidate is removed from the left end of the deque \(D_i\) it is picked for the partitioning set \(P_i\), and since \(D_i\) is empty it means that all the candidates that were in \(D_i\) at the \(j^{th}\) iteration were counted as positions in \(P_i\). There were \(\frac{c_n}{\tau \log n}\) positions at the \(j^{th}\) iteration, and therefore every time the deque \(D_i\) is empty, it means that \(\frac{c_n}{\tau \log n}\) positions are counted as positions in \(P_i\). Since the algorithm stops its run on \(h_i\) once it is determined that \(|P_i| > \frac{c_n}{\tau}\) then after \(\log n\) step-backs of the algorithm on the hash function \(h_i\), the run on \(h_i\) will stop. \(\square\)

**Lemma C.9.** Calculating \(|P_i|\) or determining that \(|P_i| > \frac{c_n}{\log n}\) for all \(h_i\) takes \(O(n + \tau \log^2 n)\) time using \(O(\frac{n}{\tau} + \log n)\) words of space.

**Proof.** When all function move forward simultaneously, we analysed earlier that it takes \(O(n)\) time. For each \(h_i\), using at least one word of space for the rightmost picked position, and at most \(O(\frac{n \log n}{\tau \log n})\) words of space for candidates, the run may have to step-back at most \(\log n\) times, and stepping back will cost \(O(\tau)\) time. This fact holds for all the \(\log n\) hash functions, thus in total the steps back may cost \(O(\tau \log^2 n)\) time in the worst case. Also, Notice that each position in \(P_0\) is inserted exactly once, and removed exactly once from the deque \(D_i\). Thus, there are exactly \(2n\) insertion and removal operations in all the deques throughout the whole run, costing \(O(n)\) time in total.

Summing all up, it costs \(O(n + \tau \log^2 n)\) time and \(O(\frac{n}{\tau} + \log n)\) words of space to run the above algorithm. \(\square\)

**Lemma C.10.** For \(\log^2 n < \tau < O(\frac{n}{\log \log n})\), a partitioning set of size \(O(\frac{n}{\tau})\) can be constructed in \(O(n)\) time with high probability using \(O(\frac{n}{\tau})\) words of space.

**Proof.** After calculating \(|P_i|\) or determining that it is too big for all \(h_i\) in linear time, with high probability there is at least one hash function \(h_k\) such that \(|P_k| < \frac{c_n}{\tau}\). We calculate \(P_k\) explicitly in \(O(n)\) time and declare it as the partitioning set. \(\square\)

## D  Missing Details for the Deterministic Selection

In this appendix we complete the algorithm for the deterministic selection of \((O(\tau), O(\tau \log^* n))\)-partitioning set from Section 5. In Appendix D.1, we prove that the selected positions are actually a \((O(\tau), O(\tau \log^* n))\)-partitioning set. Then, in Appendix D.2 we describe how to find these positions while using only \(O(\frac{n}{\tau} + \log n \cdot \log^* n)\) words of working space, which for \(\tau \leq O(\frac{n}{\log n \cdot \log^* n})\) is \(O(\frac{n}{\tau})\) words of space.

### D.1  Correctness Proof

In this section we prove that the selection we have described in Section 5 yields a \((O(\tau), O(\tau \log^* n))\)-partitioning set of \(S\), as defined in Definition 3.1. Recall that the partitioning of Section 5 is a
hierarchical decomposition of the range $[1,n]$, which its top-level is $\mu = \log_{3/2} \tau$. Let $P_\mu \subseteq [n]$ be the positions corresponding to level $\mu$ of the decomposition for any $\mu$ from $0$ to $\log_{3/2} \tau$. Notice that we have $P_{\lfloor \log_{3/2} \tau \rfloor} \subseteq \cdots \subseteq P_1 \subseteq P_0 = [n]$. So, our goal is to prove that $P_{\lfloor \log_{3/2} \tau \rfloor}$ is a $\mathcal{O}(\mathcal{O}(\tau),\mathcal{O}(\tau \log^* n))$-partitioning set of size $\mathcal{O}(\frac{n}{\tau})$. We will prove it as a special case, for the general case which is that for any $0 \leq \mu \leq \log_{3/2} \tau$ the set $P_\mu$ is a $(\mathcal{O}((3/2)^\mu),\mathcal{O}((3/2)^\mu \cdot \log^* n)))$-partitioning set of size $\mathcal{O}(\frac{n}{(3/2)^\mu})$.

For any $\mu$ we have to prove three properties of $P_\mu$: the two properties of Definition 3.1, and that the size of the set is $\mathcal{O}(\frac{n}{(3/2)^\mu})$. We prove each of these properties by induction.

We first prove that the size of $P_\mu$ is $\mathcal{O}(\frac{n}{(3/2)^\mu})$. This property is trivially deduced from the following lemma.

**Lemma D.1.** For any level $0 \leq \mu \leq \log_{3/2} \tau$ of the hierarchical decomposition, the length of any pair of consecutive blocks is at least $(3/2)^\mu$.

**Proof.** We prove the lemma by induction on the level of the hierarchy. For level $\mu = 0$, the lemma is trivial. We assume the lemma holds for level $\mu - 1$, and prove it holds for level $\mu$ as well. Let $[p_i,p_{i+1} - 1]$ and $[p_{i+1},p_{i+2} - 1]$ be two consecutive blocks at level $\mu$. If at least one of the blocks is a sub-block of type 1, then the claim obtained from the definition of type 1. Otherwise, if each one of the blocks is composed of at least two sub-blocks of level $\mu - 1$, the claim obtained by the induction hypothesis as their total length must be at least $2 \cdot (3/2)^{\mu - 1} > (3/2)^\mu$.

The only remaining case where a block is composed of only one sub-block, is the case that this sub-block is the only sub-block in a sequence of type 4. Since we already covered the case where one of the blocks is of type 1, let us assume that the other block is of type 2. Let us assume without loss of generality that $[p_i,p_{i+1} - 1]$ is the block of type 2 and that $[p_{i+1},p_{i+2} - 1]$ is also a block in level $\mu - 1$. We denote the length of each sub-block which assembles $[p_i,p_i+1 - 1]$ as $x$ (recall that all the sub-blocks that assembles $[p_i,p_{i+1} - 1]$ are identical, and in particular they have the same length), and the length of $[p_{i+1},p_{i+2} - 1]$ as $y$. By the induction hypothesis it must be that $x + y \geq (3/2)^{\mu - 1}$ and also that $2x \geq (3/2)^{\mu - 1}$, since in level $\mu - 1$ we have a pair of consecutive blocks of length $x$, and another pair of consecutive blocks of lengths $x$ and $y$. The length of $[p_i,p_{i+1} - 1]$ is at least $2x$, therefore the total length of the pair $[p_i,p_{i+1} - 1]$ and $[p_{i+1},p_{i+2} - 1]$ is at least $2x + y = (x + y) + x \geq (3/2)^{\mu - 1} + \frac{(3/2)^{\mu - 1}}{2} = (3/2)^\mu$. \hfill $\square$

Property 1 of Definition 3.1 states that in any partitioning set the selection of each position is local, which means it depends on a substring around this position of length $\mathcal{O}(\delta)$. In the following lemma we prove that the positions of the decomposition in each level of the hierarchy has this locality property. Let us denote $S_{a,\mu} = S[a - 6 \cdot (3/2)^{\mu} \cdot \log^* n..a + 6 \cdot (3/2)^{\mu} \cdot \log^* n]$ to be the substring of $S$ of length $12 \cdot (3/2)^{\mu} \cdot \log^* n + 1$ around position $a$. In addition, for any $p_i \in P_\mu$, let $p_{i+1}$ be the position following $p_i$ in $P_\mu$ (or 0 if $p_i$ is the first position in $P_\mu$), we say that $[p_i,p_{i+1} - 1]$ is the block of $p_i$ in level $\mu$.

**Lemma D.2.** For any level $0 \leq \mu \leq \log_{3/2} \tau$ of the hierarchical decomposition, if $a,b \in [n]$ are two indices such that $S_{a,\mu} = S_{b,\mu}$ then $a \in P_\mu \iff b \in P_\mu$

**Proof.** We prove the lemma by induction on the decomposition’s level. For level $\mu = 0$, the lemma is trivial, since $P_0 = [n]$. Let assume the lemma holds for $P_{\mu - 1}$, and we prove that it holds for $P_\mu$ as well. Let $a$ and $b$ be two positions such that $S_{a,\mu} = S_{b,\mu}$ and $a \in P_\mu$. Thus, in particular for any $2(3/2)^{\mu} \cdot \log^* n \leq \delta \leq 2(3/2)^{\mu} \cdot \log^* n$ we have $S_{a+\delta,\mu} = S_{b+\delta,\mu-1}$. Therefore, due to the induction hypothesis it must be that $a + \delta \in P_{\mu-1} \iff b + \delta \in P_{\mu-1}$. Moreover, since $P_\mu \subseteq P_{\mu-1}$ and $a \in P_\mu$, it must be that $a \in P_{\mu-1}$ and therefore $b \in P_{\mu-1}$ as well. We prove the lemma by partitioning the cases according to the type of the sequence contains $a$.
Type 1. Since the length of the sub-block that \( a \) is its leftmost index in level \( \mu - 1 \) is at least \((3/2)^\mu\), it must be that \([a, a + (3/2)^\mu - 1] \cap P_{\mu-1} = \{a\}\). Hence, it also must be that \([b, b + (3/2)^\mu - 1] \cap P_{\mu-1} = \{b\}\). Therefore, the block of \( b \) in level \( \mu - 1 \) is of length at least \((3/2)^\mu\) and therefore it is also a block in level \( \mu \). Hence, \( b \in P_\mu \).

Type 2. If the block of \( a \) in level \( \mu \) is not of type 1, then it must be that its sub-block length is less than \((3/2)^\mu\), let denote the length of the sub-block as \( \delta < (3/2)^\mu \), then the sub-block of \( a \) in level \( \mu - 1 \) is \([a, a + \delta - 1]\) and \((a + \delta) \in P_{\mu-1}\). Moreover, since \([a, a + \delta - 1]\) is in a sequence of type 2 it must be that the sub-block which follows \([a, a + \delta - 1]\) in level \( \mu - 1 \) is \([a + \delta, a + 2\delta - 1]\) and that \(S[a..a + \delta - 1] = S[a + \delta..a + 2\delta - 1]\). Thus, we have that \([a, a + 2\delta] \cap P_{\mu-1} = \{a, a + \delta, a + 2\delta\}\). In addition, since \( a \in P_\mu \), it must be that the preceding sub-block of \([a, a + \delta - 1]\) is of different length or \(S[a..a + \delta - 1] \neq S[a - \delta..a - 1]\). Recall that \( \delta < (3/2)^\mu \) and therefore \(2\delta < 2 \cdot (3/2)^\mu\), hence it must be that \([b, b + 2\delta] \cap P_{\mu-1} = \{b, b + \delta, b + 2\delta\}\). Therefore, in level \( \mu - 1 \) we have two equal sub-blocks \([b, b + \delta - 1]\) and \([b + \delta, b + 2\delta - 1]\) and the preceding sub-block must be differed. Therefore, it must be that \( b \) is the leftmost endpoint of the corresponding block in level \( \mu \), which means that \( b \in P_\mu \).

Type 3. Let us assume that the block of \( a \) in level \( \mu - 1 \) is not part of the last \( c \log^* n \) blocks in the sequence, otherwise the proof is similar to the case of type 4. Let \( p_0, p_1, \ldots, p_h \) be the positions of \( P_{\mu-1} \) sorted in ascending order and let \( i \) and \( j \) be the indices such that \( a = p_i \) and \( b = p_j \).

There are two cases that \( p_i \in P_{\mu-1} \) which is in a sequence of type 3 sub-blocks is moved to \( P_\mu \). Either \( p_i \) is the leftmost position in the sequence or its label is a local minimum. For the first case, it depends on at most \( 2(3/2)^\mu \) positions to its left, which are part of block created from type 1 or type 2. In both cases, since \( S_{p_i,\mu} = S_{p_j,\mu} \) it must be that this large block or periodic block appears also before \( p_j \) in level \( \mu - 1 \), and therefore \( b = p_j \in P_\mu \) as required.

Consider the second case, where \( p_i \in P_\mu \) since its label is a local minimum. Intuitively, we claim that the partition into blocks around \( p_i \) and \( p_j \) is the same by the induction hypothesis. Thus, the labels given for each block in this neighbourhood is also equal, and therefore the block of \( p_j \) must be also the local minimum. Formally, we have that \( \text{label}(i) \) is smaller than \( \text{label}(i + 1) \) and also smaller than \( \text{label}(i - 1) \), where \( \text{label}(x) = \text{label}_\mu(x, \log^* n) \) is the label of the block \([p_x..p_{x+1} - 1]\) after the last inner-iteration on the transformation from \( \mu - 1 \) to \( \mu \). Thus, we focus only on the values of \( \text{label}(p_{i-1}), \text{label}(p_i) \) and \( \text{label}(p_{i+1}) \). Each of these labels is dependent on the \( \log^* n \) sub-blocks to its right, and the length of each such a sub-block is at most \((3/2)^\mu\). Therefore, the first relevant sub-block begins at position which is at least \( p_i - (3/2)^\mu \) and the last relevant block ended at position which is at most at \( p_i + (c \log^* n + 2)(3/2)^\mu \). Since this range is fully contained in \([p_i - 2(3/2)^\mu c \log^* n, p_i + 2(3/2)^\mu c \log^* n]\) we are guaranteed by the induction hypothesis that the all the positions in the same distances from \( p_j \) are also induced partitioning of \( S \) in the same way. Therefore, the labels of \( \text{label}(p_{j-1}), \text{label}(p_j) \) and \( \text{label}(p_{j+1}) \) are exactly the same as \( \text{label}(p_{i-1}), \text{label}(p_i) \) and \( \text{label}(p_{i+1}) \). Thus, \( \text{label}(p_j) \) must be also a local minimum, and so \( b = p_j \in P_\mu \) as required.

Type 4. As in the proof of the previous type, let \( p_0, p_1, \ldots, p_h \) be the positions of \( P_{\mu-1} \) sorted in ascending order and let \( i \) and \( j \) be the indices such that \( a = p_i \) and \( b = p_j \). It is guaranteed that the closest type 2 or 1 block in \( P_\mu \) after \( p_i \) starts at most \( c \log^* n \) sub-blocks after \( p_i \). Since the length of each sub-block in this area is of length at most \((3/2)^\mu\), it must be that this position is at most \( p_i + (3/2)^\mu c \log^* n < p_i + 2(3/2)^\mu c \log^* n \). Therefore, by the induction hypothesis, the same blocks appear exactly the same also after \( p_j \). The only dependent on blocks to the left \( p_i \) is just
if there are at most 2 sub-blocks in the sequence of sub-blocks to the left of \( p_i \) which are not of type 1 or type 2. However, due to similar arguments to the above the number of such sub-blocks also must be the same before \( p_i \) and before \( p_j \). Therefore, \( p_j = b \) must be also a position in \( P_\mu \).

**Conclusion.** We proved that \( a \in P_\mu \Rightarrow b \in P_\mu \), the proof of the opposite direction is exactly the same. Hence, we have that \( a \in P_\mu \Leftrightarrow b \in P_\mu \). \( \square \)

Property 2 of Definition 3.1 states that any long block in the partitioning is a periodic substring of \( S \), which its period length is at most \( \tau \). In the following lemma we prove that the positions of the decomposition in each level of the hierarchy has this property. In particular, the set \( P_{\log_3 2 \tau} \) has Property 2 of Definition 3.1.

**Lemma D.3.** For any level \( 0 \leq \mu \leq \log_3 2 \tau \) of the hierarchic decomposition, if \( p_i < p_{i+1} \) are two consecutive positions in \( P_\mu \) such that \( p_{i+1} - p_i > 12 \cdot (3/2)^\mu \) then \( S[p_i..p_{i+1} - 1] \) has period length of at most \( (3/2)^\mu \)

**Proof.** We prove the lemma by induction on the level of the hierarchy. For level 0, the lemma is trivial, since \( P_0 = [n] \). We assume by induction that the lemma holds for \( P_{\mu-1} \), and we prove that it holds for \( P_\mu \) as well. Let \([p_i, p_{i+1} - 1]\) be a block in level \( \mu \), i.e. \([p_i, p_{i+1}] \cap P_\mu = \{p_i, p_{i+1}\}\) such that \(|[p_i, p_{i+1} - 1]| = p_{i+1} - p_i > 12 \cdot (3/2)^\mu \). If \([p_i, p_{i+1} - 1]\) is a block in level \( \mu - 1 \) as well, the lemma holds due to the induction hypothesis. Otherwise, it must be that \([p_i, p_{i+1} - 1]\) is composed of several sub-blocks of level \( \mu - 1 \) which each of them is of length at most \( (3/2)^\mu \) (since otherwise they correspond to a block of type 1). Hence, it must be that \([p_i, p_{i+1} - 1]\) is composed of more than 12 blocks of level \( \mu - 1 \). The only type of blocks in level \( \mu \) which composed of more than 12 sub-blocks of level \( \mu - 1 \) is type 2, and in this case all the substrings corresponds to the sub-blocks are equal. Therefore, the substring corresponds to \([p_i, p_{i+1} - 1]\) is \( uuu \cdots u \) where \( u \) is a string of length at most \( (3/2)^{\mu-1} \). Thus, the period length of \( S[p_i..p_{i+1} - 1] \) is at most \( (3/2)^\mu \). \( \square \)

Hence, by combining Lemmas D.1, D.2 and D.3 we receive the following corollary.

**Corollary D.4.** The set \( P_{\log_3 2 \tau} \) is \( \mathcal{O}(\tau), \mathcal{O}(\tau \log^* n) \)-partitioning set of size \( \mathcal{O}(\frac{\tau}{\tau}) \).

Or, by selecting a bit lower level, we get a truly \( (\tau, \mathcal{O}(\tau \log^* n)) \)-partitioning set of size \( \mathcal{O}(\frac{\tau}{\tau}) \).

**Corollary D.5.** The set \( P_{\log_3 (2 \tau/12)} \) is \( (\tau, \mathcal{O}(\tau \log^* n)) \)-partitioning set of size \( \mathcal{O}(\frac{\tau}{\tau}) \).

To complete the missing details for the deterministic construction we prove in the following lemma that \( P_{\log_3 (2 \tau/12)} \) is forward synchronized (see Definition 6.1). This lemma is required for the deterministic construction of the SST introduced in Section 6.

**Lemma D.6.** The set \( P_{\log_3 (2 \tau/12)} \) is a forward synchronized partitioning-set

**Proof.** Let \( q_1 < \cdots < q_6 \) be all the positions in \( P_\mu \) and \( p_1 < \cdots < p_6 \) be all the positions in \( P_{\mu+1} \). We prove the lemma by induction on the level of the hierarchy. For level 0, the lemma is trivial, since \( P_0 = [n] \). We assume by induction that the lemma holds for \( P_\mu \), and we prove that it holds for \( P_{\mu+1} \) as well. Let \( p_i, p_j \in P_{\mu+1} \) such that \( \text{LCE}(p_i, p_j) > p_{i+1} - p_i + 6(3/2)^{\mu+1} c \log^* n \). Since \( P_{\mu+1} \subseteq P_\mu \), \( p_i, p_j \in P_\mu \). Let \( q_i = p_i \) and \( q_j = p_j \) be the corresponding positions in the ordered set of positions \( P_\mu \). Remark that \( q_i < p_{i+1} \) and \( q_j < p_{j+1} \), thus

\[
\text{LCE}(q_i, q_j) > p_{i+1} - p_i + 6(3/2)^{\mu+1} c \log^* n > q_{i+1} - q_i + 6(3/2)^\mu c \log^* n.
\]
Therefore, by the induction hypothesis \( q_{i'} + 1 \) - \( q_i = q_{j'} + 1 - q_j \). Furthermore, for every \( k \geq 0 \) such that \( q_{i' + k + 1} \leq p_{i + 1} + 2(3/2)^k \log^* n \) it follows that

\[
\text{LCE}(q_{i' + k}, q_{j' + k}) = \text{LCE}(p_i, p_j) - (q_{i' + k} - p_i) \\
> (p_{i + 1} - p_i) - (q_{i' + k} - p_i) + 6(3/2)^{k+1} \log^* n \\
\geq q_{i' + k + 1} - 2(3/2)^k \log^* n - q_{j' + k} + 9(3/2)^k \log^* n \\
> q_{i' + k + 1} - q_{j' + k} + 6(3/2)^k \log^* n
\]

Thus by induction on \( k \), remark it follows that \( q_{i' + k + 1} - q_{i' + k} = q_{j' + k + 1} - q_{j' + k} \) for all \( k \geq 0 \) such that \( q_{i' + k + 1} \leq p_{i + 1} + 2(3/2)^k \log^* n \).

We continue the proof by partitioning the cases according to the type of the block \([p_i..p_{i+1}]\).

**Type 1.** This means that \([p_i..p_{i+1}]\) is longer than \((3/2)^{\mu+1}\), and it is also a block at level \( \mu \). Thus, \( q_{i' + 1} = p_{i+1} \). From the above remark it follows immediately that since \( q_{i' + 1} - p_i = q_{j' + 1} - p_j \) then \( q_{j' + 1} = p_{j+1} \) because it is a Type 1 block as well. Thus, the lemma holds.

**Type 2.** This means that for every \( k \geq 0 \) such that \( q_{i' + k} < p_{i + 1} \) all the blocks \([q_{i' + k}..q_{i' + k + 1}]\) are equal, and the block at level \( \mu \) that starts at \( p_{i + 1} \) is different. Let \( \delta = q_{i' + 1} - q_{i'} \) be the equal sub-blocks’ length. Notice that \( \delta < (3/2)^{\mu+1} \) since it is not a type 1 sub-block. From the above remark it follows that the same holds for all the blocks \([q_{j' + k}..q_{j' + k + 1}]\) for every \( k \geq 0 \) such that \( q_{j' + k} < p_j + p_{i + 1} - p_i \), and that the position defined as \( q_\ell = p_j + p_{i + 1} - p_i \) is a position in \( P_\mu \).

The only case where \( p_{j+1} \neq q_\ell \) is if \( [q_\ell..q_{\ell+1}] \) is equal to the previous sub-blocks. In that case since \( \delta < (3/2)^{\mu+1} \), if \( q_{\ell+1} = q_\ell + \delta \) then \([p_{i+1..p_{i+1} + \delta - 1}]\) is also a sub-block at level \( \mu \), and it is equal to the previous sub-blocks. This contradicts the fact that at \( p_{i + 1} \) starts a level \( \mu \) block that is different from its previous block. Therefore, at \( q_\ell \) starts a block that most be different from its previous equal sub-blocks. From all the above it follows that \( p_{j+1} = q_\ell = p_j + p_{i+1} - p_i \).

**Type 3.** As with the proof of Lemma D.2, we assume that the sub-block \( q_i \) is not of the last \( c \log^* n \) sub-blocks in the sequence of Type 3 sub-blocks. By definition, all sub-blocks in the sequence are shorter than \((3/2)^{\mu+1}\). Also, there are at least \( c \log^* n \) sub-blocks in the sequence after the sub-block that starts at \( q_i \). Therefore, for all \( 0 \leq k \leq c \log^* n \) it follows that \( q_{i' + k + 1} \leq p_{i + 1} + 2(3/2)^k \log^* n \). Hence, from the above remark it follows that the blocks partition at level \( \mu \) is forward synchronized for at least \( c \log^* n + 1 \) blocks from \( q_i \) and \( q_j \).

Notice that \( p_{i+1} = q_{i' + k} \) for some \( 1 \leq k \leq 11 < c \log^* n \) since the position of \( p_{i+1} \) was selected for level \( \mu + 1 \) either due to \( q_{i' + k} \) having a local minimum label value, or if \( q_{i' + k} \) is the right end-point of one of the last \( c \log^* n \) sub-blocks in the sequence. For the later case, the proof is similar to that of type 4.

If \( q_{i' + k} \) was chosen for level \( \mu + 1 \) due to its label value, it means that there are at least \( c \log^* n \) short sub-blocks in the sequence after \( q_{i' + k} \). Thus, the range of \( k \) such that \( q_{i' + k + 1} \leq p_{i + 1} + 2(3/2)^k \log^* n \) can be extended to reach up to \( \tilde{k} + c \log^* n \), and with it the range of forward synchronization of blocks from \( q_i \) and \( q_j \). Since the label of every position is dependent on the next \( c \log^* n \) blocks after it, then for every \( 0 \leq k \leq \tilde{k} + 1 \) it follows that \( \text{label}(q_{i' + k}) = \text{label}(p_{i+1}, k + 1) \). Since \( q_{i' + k} = p_{i+1} \) is the first block with local minimum label value after \( p_i \), then the same holds for \( q_{j' + k} \) and \( p_j \), thus \( p_{j+1} = p_j + p_{i+1} - p_i \).

**Type 4** Let \( q_{i' + k} \) for \( 1 \leq k \leq c \log^* n \) be the position at the end of the type 4 sub-blocks sequence. Since all blocks in the sequence are shorter than \((3/2)^{\mu+1}\), then as shown in the proof in the case of
type 3 it follows that the blocks at level $\mu$ are forward synchronized from $q_{i'}$ and $q_{j'}$ up to $q_{i'+\hat{k}}$ and $q_{j' + \hat{k}}$ respectively. Remark that if $q_{j'+\hat{k}+1}$ is the start of sub-block shorter than $(3/2)^{\mu+1}$, then an equal block must start at $q_{i'+\hat{k}}$. Since $q_{i'+\hat{k}}$ is the first position after a type 4 sequence, if it is the beginning if a short sub-block, then it must be the beginning of a type 2 sub-blocks sequence. In that case $q_{i'+\hat{k}+2} < p_{i'+1} + 2(3/2)^{\mu}\log^* n$ and thus from the forward synchronization assumption it follows that $q_{j'+\hat{k}}$ is also the beginning of a type 2 sub-blocks sequence. Therefore in any case $q_{j'+\hat{k}}$ is also the end of a type 4 sequence.

Since in the case of a type 4 sub-block, picking a position for the next level is dependent only on the distance from the end of the sequence and the sequence’s length, and since the positions of $p_{i}$ and $p_{j}$ are synchronized up to the end of the sequence, it follows that $p_{j+1} = p_{j} + \mu + 1, i + 1 - \mu + 1, i$.

\[ \square \]

D.2 Efficient Computation using only $\mathcal{O}(\frac{n}{\tau} + \log n \cdot \log^* n)$ Working Space

In this section we describe how to find the set $P_{\log 3/2 \tau}$ using only $\mathcal{O}(\frac{n}{\tau})$ words of space. We focus on the perspective of the hierarchical decomposition of $S$ as a tree. Since each block in level $\mu$ of the decomposition is composed of complete sub-blocks of level $\mu - 1$, the hierarchical decomposition induces a tree of blocks. Formally, for each block $[x, y - 1]$ in level $\mu$, we consider all the blocks of level $\mu - 1$ that compose $[x, y - 1]$ as its children. The important property for the algorithm is that the decision whether a block of level $\mu - 1$ is a leftmost endpoint of a block in level $\mu$ can be taken using information of only constant number of blocks to the left and right of the block. We will denote by $\text{label}_\mu([x, y - 1], j)$ the label of the sub-block $[x, y - 1]$ after the $j$th inner-iteration in the outer-iteration that computes the decomposition of level $\mu$, based on the decomposition of level $\mu - 1$.

**Labels list** We slightly generalize the definition of labels we gave in Section 5 in order to clarify how local is the transformation from $P_{\mu-1}$ to $P_{\mu}$. For each block $B = [x, y - 1]$ in level $\mu - 1$ we define an ordered list of $\mathcal{O}(\log^* n)$ labels $L_B$ as follows. If the length of $B$ is at least $(3/2)^{\mu}$ then $L_B = \emptyset$. Otherwise, let $B' = [y, z - 1]$ be the block following $B$ in level $\mu - 1$. If $S[x, y - 1] = S[y, z - 1]$, we define $L_B = \emptyset$ as well. On the other hand, if $S[x, y - 1] \neq S[y, z - 1]$, then the size of $L_B$ is $\min\{|L_{B'}| + 1, c \log^* n\}$. The $j$th element in $L_B$ is $\text{label}_\mu(B, j)$. Notice that for $j > 1$, given the $(j - 1)$ elements of $L_B$ and $L_{B'}$ the computation of $\text{label}_\mu(B, j)$ can be done in constant time without any additional information. The computation of $\text{label}_\mu(B, 1)$ is done in time which is linear in the length of $B$ and $B'$. The algorithm maintains with each block $B$ all the elements of $L_B$ explicitly, except for the label $\text{label}_\mu(B, 0)$, which is stored implicitly by the range of $B$ (recall that $\text{label}_\mu(B, 0)$ is the character of $\Pi$ corresponded to $S[x, y - 1]$). As explained above, each label stored in only $\mathcal{O}(1)$ words of space. Thus, each block stored by the algorithm uses $\mathcal{O}(\log^* n)$ words of space.

Now, using the labels lists, we introduce how to decide whether a block $B$ in level $\mu - 1$ is a leftmost sub-block of some block of level $\mu$, using only the information of one blocks to the left and four blocks to the right of $B$. First of all, we identify for each sub-block the type of sequence it belongs. Identifying of sub-blocks of type 1 is done by checking their length, regardless any information of other blocks. Let $B$ be a sub-block from level $\mu - 1$, and let $B'$ be the sub-block to the left of $B$. $B$ is in a sequence of type 2 if and only if $|B| < (3/2)^{\mu+1}$ and either $L_B = \emptyset$ or $L_{B'} = \emptyset$ and $|B| = |B'|$. Each block which is not of type 1 or 2 is obviously of type 3 or 4. Since the treat of the rightmost $c \log^* n$ sub-blocks in a sequence of type 3 is exactly as the treat for sub-blocks in a sequence of type 4 the algorithm is not required to distinguish between them.

Let $B$ be a block in level $\mu$. The algorithm sets $B$ as a leftmost sub-block in level $\mu$ due to the type of its sequence. If $B$ is in a sequence of type 1 it always moved to level $\mu + 1$. If $B$ is in a
sequence of type 2, the algorithm sets a block $B$ only if its preceding block is not in the sequence, which can be detected given the two successive blocks of $B$. If $B$ is in a sequence of type 3, and $|\mathcal{L}_B| = c \log^* n$ then we use the number of the block which is the last element in $\mathcal{L}_B$ and choose $B$ if this number is smaller than the numbers of the blocks to its left and right. The treat of margin cases is obvious. If $B$ is treated as a block in a sequence of type 4 (which includes the rightmost $c \log^* n$ blocks in a sequence of type 3), we take all the blocks that are in an even position in this sequence, which can be identified due to the parity of $|\mathcal{L}_B|$. The only exceptions are blocks near the ends of the sequence, which can be treated as described above, given the information of 4 blocks ahead.

The scheduling. Further to the discussion above, the algorithm maintains any time at most 6 blocks in any level of the hierarchical decomposition, where each block is maintained with its label set. The process is done in recursion from the bottom level to the top-level and from right to left. Whenever the algorithm detects a new block in level $\mu - 1$, the algorithm checks whether the block which is 4 blocks to the right of the current block is a leftmost sub-block in level $\mu$. If the answer is positive, the algorithm creates a new block in level $\mu$, from the end of the last block that was created in level $\mu$ until the leftmost end of the new block. After the recursive call ends, the algorithm deletes old blocks, so at any time each level maintains explicitly only constant number of blocks. The only exceptional level is of course the top-level, where the algorithm maintains all the blocks and in particular the corresponding positions which compose $P_{\log_{3/2} \tau}$.

Complexities. The algorithm maintains $\mathcal{O}(\log \tau)$ levels, where for each level the algorithm maintains $\mathcal{O}(1)$ elements with $\mathcal{O}(\log^* n)$ words of space per element. Hence, all the levels, except the top-level use $\mathcal{O}(\log \tau \cdot \log^* n)$ words of space. In the top-level the algorithm maintains all the blocks, which due to Corollary D.4 is a set of size $\mathcal{O}(n)$. Notice that the blocks in the top-level are maintained without any labels set. Thus, the total space usage of the algorithm is $\mathcal{O}(\frac{n}{\tau} + \log \tau \log^* n)$. Regarding the running time, we analyse the time required for each level of the hierarchical decomposition and summing among all the levels. So, for level $\mu$ of the algorithm, the time usage of the algorithm is dominated by to the $\mathcal{O}(\log^* n)$ inner iterations. The first inner-iteration in level $\mu$ takes $\mathcal{O}(n)$ time. Any other inner iteration takes time which is linear in the number of blocks in level $\mu$ which due to Lemma D.1 is $\mathcal{O}\left(\frac{n}{(3/2)^\mu}\right)$. Since there are $\mathcal{O}(\log^* n)$ inner iterations for level $\mu$, the time usage of the algorithm for level $\mu$ is $\mathcal{O}(n + \frac{n}{(3/2)^\mu} \log^* n)$. Summing over all the levels,

$$
\sum_{\mu=0}^{\log_{3/2} \tau} \mathcal{O}(n + \frac{n}{(3/2)^\mu} \log^* n) = \mathcal{O}(n \log \tau) + \mathcal{O}(n \log^* n) \sum_{\mu=0}^{\log_{3/2} \tau} \left(\frac{2}{3}\right)^\mu = \mathcal{O}(n \log \tau + n \log^* n).
$$

Thus, combining with Corollary D.5 and Lemma D.6 we receive the following corollary

**Corollary D.7.** For any $1 \leq \tau \leq \mathcal{O}(\frac{n}{\log n \log \log n})$ there exists a deterministic algorithm that computes a forward synchronized $(\tau, \mathcal{O}(\tau \log^* n))$-partitioning set of size $\mathcal{O}(\frac{n}{\tau})$ using only $\mathcal{O}(\frac{n}{\tau})$ words of working space.

**E Missing Details for the Improved Deterministic LCE Data Structure**

In this appendix we complete the details for the improved LCE construction from Section 7. The algorithm for the case $\tau < \sqrt{\log^* n}$ appears at the end of the section, thus we assume here $\tau \geq \sqrt{\log^* n}$. 

39
Recall, that we have defined the set $P$ as a $(O(\tau'), O(\tau' \log^* n))$-partitioning set ($\tau' = \frac{\tau}{\log^* n}$) and $Q \subseteq P$, as a special subset of $P$ that is used by the algorithm. The data structure maintains the SST of the set $Q$. Then, the query computation for the non-periodic case is similar to the first phase of the LCE data structure of Section 3. The algorithm just compares characters until it finds a mismatch or corresponding selected positions. However, in the case that there is no mismatch, the common prefix can be considered as composed of two parts. In the first part the algorithm reads $O(\delta) = O(\tau' \log^* n)$ characters, to reach synchronization due to positions of $P$. Then, in the second part the algorithm reads another $O(\tau' \log^* n)$ until it finds two corresponding positions that are actually in $Q$ and not only in $P$ (recall that the set $P$ is not maintained by the algorithm).

Let us focus on the second part, in this part the algorithm reads characters from at most $\log^* n$ blocks, and since we have assumed that each block is of length at most $\tau'$, the total length was $\tau' \log^* n$. However, when we have long periodic blocks, this assumption does not hold any more. Hence, we generalize our method. Instead of picking positions from $P$ based on only their rank, we add some "weights" to the task. For each block $[p_i, p_i+1 - 1]$ induced by $P$, let $\ell_i = p_i+1 - p_i$. We associate with the block of $p_i$, $\left\lfloor \frac{\ell_i+1}{\tau'} \right\rfloor$ tokens (we emphasize that the number of tokens of one block is dependent on the length of the preceding block), where the tokens are numbered $(1, 2, 3, \ldots)$. Then, we select all the positions that has a token which its index $i$ is an integer multiple of $\sqrt{\log^* n}$ or $i \mod \log^* n < \sqrt{\log^* n}$. Notice that this method is a generalization of the method that was introduced in Section 7, since there the length of each block is at most $\tau'$ and therefore each block has exactly one token.

First of all, we argue that the size of the set $Q$ produced by the algorithm is still $O(\frac{n}{\tau'})$.

**Lemma E.1.** $|Q| = O(\frac{n}{\tau'})$.

**Proof.** Recall that $|P| = O(\frac{n}{\tau'}) = O(\frac{n}{\tau'} \sqrt{\log^* n})$. First we bound the number of tokens among all blocks induced by $P$.

$$\sum_{i=1}^{|P|} \left\lfloor \frac{\ell_i+1}{\tau'} \right\rfloor \leq \sum_{i=1}^{|P|} 1 + \frac{\ell_i+1}{\tau'} \leq |P| + \frac{1}{\tau'} \sum_{i=1}^{|P|} \ell_i+1 \leq |P| + \frac{n}{\tau} = O\left(\frac{n}{\tau} \sqrt{\log^* n}\right)$$

Hence, the number of tokens divided by the algorithm is $O(\frac{n}{\tau'} \sqrt{\log^* n})$.

The algorithm adds to $P$ only positions which has token with number which is an integer multiple of $\sqrt{\log^* n}$ or that its remainder modulo $\log^* n$ is less that $\sqrt{\log^* n}$. Hence, the size of $Q$ is at most $O(\frac{\tau'}{\tau} \sqrt{\log^* n} / \sqrt{\log^* n}) = O\left(\frac{\tau'}{\tau}\right)$. Notice that, a case where one block has multiple tokens that should be selected just decreases the size of the set $Q$. \qed

As in Section 7, the algorithm creates the SST of the set $Q$ in the construction phase. If all the block induced by $P$ are of length at most $\tau \sqrt{\log^* n}$ then with analysis which is similar to the proof of Lemma 7.2, one can prove that the simple algorithm that compare characters until it finds a mismatch or corresponding positions will work and the query time will be $O(\tau \sqrt{\log^* n})$. However, as in the simple LCE data structure, we might have very long blocks, even due to $P$, and therefore we cannot hope to reach a corresponding positions in $O(\tau \sqrt{\log^* n})$ time. Therefore, for such cases we use similar techniques to the periodic case in the basic LCE data structure. Let $\tau' > 1$ be a constant such that $P$ is a $(\tau', \tau' \log^* n)$-partitioning set (of size $O(\frac{n}{\tau'})$), in the terms of Section 3 we have $\delta = \tau' \tau \log^* n = \tau' \sqrt{\log^* n}$.

**Lemma E.2.** If $q_i < q_i+1$ are two consecutive positions in $Q$, such that $q_{i+1} - q_i > 2\delta$ then it must be that $S[q_i + \delta..q_{i+1} - 1]$ is a periodic string with period length of at most $\tau' = \frac{\tau}{\sqrt{\log^* n}}$. 

40
Proof. Assume by contradiction that $S[q_i + \delta..q_{i+1} - 1]$ is not a periodic substring. Hence, it must be that in the partitioning of $S$ induced by $P$, there exists at least one position in the range $q_i + \delta < p < q_{i+1}$. We consider the number of tokens given for blocks beginning from the block following $q_i$ until the block of $p$. Let us denote the set of blocks in this range as $\hat{P}$, then the total number of tokens given for these blocks is at least

$$\sum_{i=1}^{\lceil \frac{p}{\tau'} \rceil} \frac{\ell_{i-1}}{\tau'} \geq \sum_{i=1}^{\lceil \frac{p}{\tau'} \rceil} \frac{\ell_{i-1}}{\tau'} \geq \frac{1}{\tau'} \sum_{i=1}^{\lceil \frac{p}{\tau} \rceil} \ell_{i-1} = \frac{\delta}{\tau'} = c' \log^* n > \log^* n$$

Since in every sequence of $\sqrt{\log^* n}$ tokens the algorithm inserts at least one position from $P$ to $Q$, it must be that there exists some position $q \in Q$ such that $q_i < q \leq p < q_{i+1}$, which contradicts the fact that $q_i$ and $q_{i+1}$ are consecutive positions in $Q$. Hence, it must be that $S[q_i + \delta..q_{i+1} - 1]$ is a periodic block with period length of at most $\tau' = \frac{\tau}{\sqrt{\log^* n}}$.

Now, using Lemma E.2 the algorithm works as follows. First the algorithm compares $4\delta$ characters. If the algorithm finds a mismatch or a selected corresponding positions in $P$, the algorithm continue as in the non-periodic case. Otherwise, due to Lemma 3.2 it must be that

$$\{p - i \mid p \in (P \cap [i + \delta..i + \ell - \delta - 1])\} = \{p - j \mid p \in (P \cap [j + \delta..j + \ell - \delta - 1])\}.$$ 

In particular, since $\ell > 4\delta$ we have that:

$$\Delta = \{p - i \mid p \in (P \cap [i + \delta..i + 3\delta - 1])\} = \{p - j \mid p \in (P \cap [j + \delta..j + 3\delta - 1])\}.$$ 

Hence, due to Lemma E.2 it must be that all the characters in $S[i + 2\delta..i + 3\delta - 1]$ and $S[j + 2\delta..j + 3\delta - 1]$ are in a long periodic blocks due to partitioning induced by $P$. Let $\alpha_i = \text{succ}_Q(i + 2\delta) - i$ and $\alpha_j = \text{succ}_Q(i + 2\delta) - i$. If $\alpha_i = \alpha_j$ the algorithm uses the SST on $Q$ to compute the $\text{LCE}(\text{succ}_Q(i + 2\delta), \text{succ}_Q(i + 2\delta))$ to compute the $\text{LCE}(i, j)$ query. Otherwise, due to Lemma E.2 we have that $S[i + 2\delta..i + \alpha_i] and $S[j + 2\delta..j + \alpha_j]$ are periodic substrings with the same period length and therefore if $\alpha = \min\{\alpha_i, \alpha_j\}$ then $S[i + \alpha] = S[j + \alpha]$. Moreover, as in Lemma B.1, it must be the case that $\text{LCE}(i, j) \leq \alpha + \delta$ and therefore, by reading another at most $\delta$ characters from $S[i + \alpha]$ and $S[j + \alpha]$ the algorithm is guaranteed to find the first mismatch between the suffixes.

**E.1 Algorithm for $\tau < \sqrt{\log^* n}$**

If $\tau < \sqrt{\log^* n}$ then we use even simpler algorithm. We define $P = \{1, 2, 3, \ldots, n\}$, and we use Lemma 7.1 with $t = \tau^2$. Let $Q \subseteq P$ be a set of size $|Q| = O(\frac{n}{\tau^2})$ such that for every $i, j \in [n - \tau^2]$ there exists $k \leq \tau^2$ such that both $i + k \in Q$ and $j + k \in Q$. Thus, the algorithm computes $Q$ and build the SST on $Q$. In addition the algorithm computes the LCA information for the SST. The query computation is straightforward. The algorithm compares at most $\tau^2$ pairs of corresponding characters. If a mismatch is found, the algorithm reports the LCE. Otherwise, the algorithm finds the smallest $k \geq 0$ such that $i + k, j + k \in Q$ and returns the sum of $k$ and $\text{LCE}(i + k, j + k)$. The space usage of this data structure is of course $O(\frac{n}{\tau})$ words of space, and the query time is $O(\tau^2)$. Since $\tau < \sqrt{\log^* n}$ we have $O(\tau^2) \subseteq O(\tau \sqrt{\log^* n})$, as required.

**F Useful Tools and Lemmas**

In this appendix we introduce several theorems and lemmas which we use as basic tools in our data structures and algorithms.
Strings Periodicity

The following theorem is one of the most basic theorems regarding periodicity of strings.

**Theorem F.1** (Fine and Wilf [14]). Let $u$ be a string. If $u$ has periods of length $p$ and $q$ and $|u| \geq p + q - \gcd(p, q)$, then $u$ also has a period of length $\gcd(p, q)$.

The following lemmas are used in several parts of the paper.

**Lemma F.2.** Let $u$ be a string with period length $p$. If $v$ is a substring of $u$ of length at least $p$, and $v$ has period length of $a|p$, then $u$ has period length of $q$.

**Proof.** If $p = q$ the lemma is trivial. Otherwise, $p \neq q$ and $q|p$, hence $q < p$.

Let $a$ be an index such that $v[1..p] = u[1+a..p+a]$. Notice that for each $i \in \mathbb{N}$ there exists $1 + a \leq j \leq p + a$ such that $i \mod p = j \mod p$.

Given any index $1 \leq i \leq |u| - q$ let $1+a \leq j+a \leq p+a$ be the index such that $i \mod p = (j+a) \mod p$.

Since $u$ has period length of $p$, $u[i] = u[j+a] = v[j]$. We treat two cases:

- **Case 1** $j+a+q \leq a+p$: then we have similarly that $u[i+q] = u[j+a+q] = v[j+q]$. In this case, since $v$ has period length of $q$ we have $[j] = v[j+q]$ and therefore $u[i] = v[j] = v[j+q] = u[i+q]$.

- **Case 2** $j+a+q > a+p$: hence $a \leq j + a + q - p < j + a \leq a + p$. We have that $i \mod p = j + a \mod p$ and therefore, $i + q \mod p = j + a + q \mod p = j + a + q - p \mod p$. Thus, since $u$ has period length of $p$, $u[i+q] = u[j+a+q-p] = v[j+q-p]$. Since $q|p$, we have also $q|(q-p)$, thus, since $v$ has period length of $q$ we have $v[j+q-p] = v[j]$. To conclude, $u[i] = v[j] = v[i+q] = u[i+q]$.

Hence, $u[i] = u[i+q]$ for any $1 \leq i \leq |u| - q$, and therefore $q$ is a period length of $u$.

**Lemma F.3.** Let $u$ and $v$ be two strings of length $|u| = |v| \geq 2t$ for some $t \in \mathbb{N}$, such that both $\rho_u \leq t$ and $\rho_v \leq t$. If $u[1..2t] = v[1..2t]$ then $u = v$.

**Proof.** We first prove that $\rho_u = \rho_v$, and then we use this fact to prove the lemma.

Assume by contradiction that $\rho_u \neq \rho_v$, and without loss of generality, we assume that $\rho_u < \rho_v$.

Let $w = u[1..2t] = v[1..2t]$. Due to Theorem F.1 (Fine and Wilf theorem), since $\rho_u$ and $\rho_v$ are both period lengths of $w$, we have that $\rho_w | \gcd(\rho_u, \rho_v)$ and therefore $\rho_w \leq \rho_u < \rho_v$. Due to Lemma F.2 since $w$ is a substring of $v$ length at least $2t \geq \rho_v$ and $\rho_u | \rho_v$, we have that $\rho_u$ is a period length of $v$, and therefore $\rho_v \leq \rho_w \leq \rho_u$, which contradicts the assumption. Hence $\rho_v = \rho_u$, and we denote this length as $\rho$.

Let $i \in [u]$ be an index in the strings, we have to prove that $u[i] = v[i]$. If $i \leq 2\rho$, the claim holds from the fact that $u[1..2t] = v[1..2t]$. Otherwise, since $\rho$ is a period length of $u$, we have that $u[i] = u[i+k \cdot \rho]$ for any integer $k \in \mathbb{Z}$. In particular $u[i] = u[\rho + (i \mod \rho)]$. Similarly, we have that $v[i] = v[\rho + (i \mod \rho)]$. Since $1 \leq \rho + (i \mod \rho) \leq 2\rho - 1 \leq 2t$ we have that $u[\rho + (i \mod \rho)] = v[\rho + (i \mod \rho)]$, and therefore $u[i] = v[i]$ as required.

**Sorting Strings**

The following lemmas are direct results of two facts stated in [19].

**Lemma F.4** ([19, Fact 3.1 and Fact 3.2]). For any $1 \leq \tau \leq n$, given random access to $O(n/\tau)$ strings of length $O(\tau \log^* n)$, the strings can be sorted in $O(n \log^* n)$ time and $O(n/\tau)$ words of space.

**Lemma F.5** ([19, Fact 3.1 and Fact 3.2]). For any $1 \leq \tau \leq n$, given random access to $O(n/\tau)$ strings of length $O(\tau)$, the strings can be sorted in $O(n)$ time and $O(n/\tau)$ words of space.
Lowest Common Ancestor

For any tree $T$, we denote by $LCA(u, v)$ the lowest common ancestor of nodes $u, v \in T$. The $LCA$ of any two leaves in a trie is the longest common prefix (LCP) of the corresponding strings. Hence, efficient data structure for $LCA$ queries implies an efficient data structure for $LCP$ queries.

**Lemma F.6** ([21, 2]). Any tree $T$, can be preprocessed in $O(|T|)$ time and space to support $LCA$ queries in constant time.

References

[1] Amihood Amir and Igor Nor. Real-time indexing over fixed finite alphabets. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1086–1095, 2008.

[2] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium*, pages 88–94, 2000.

[3] Philip Bille, Johannes Fischer, Inge Li Gørtz, Tsvi Kopelowitz, Benjamin Sach, and Hjalte Wedel Vildhøj. Sparse text indexing in small space. *ACM Trans. Algorithms*, 12(3):39:1–39:19, 2016.

[4] Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. Longest common extensions in sublinear space. In *Combinatorial Pattern Matching - 26th Annual Symposium, CPM*, pages 65–76, 2015.

[5] Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. Time-space trade-offs for longest common extensions. *J. Discrete Algorithms*, 25:42–50, 2014.

[6] Timo Bingmann, Johannes Fischer, and Vitaly Osipov. Inducing suffix and lcp arrays in external memory. In *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX*, pages 88–102, 2013.

[7] Dany Breslauer and Zvi Galil. Real-time streaming string-matching. *ACM Transactions on Algorithms*, 10(4):22:1–22:12, 2014.

[8] Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In *Combinatorial Pattern Matching, 14th Annual Symposium, CPM*, pages 55–69, 2003.

[9] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.

[10] Graham Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algorithms*, 3(1):2:1–2:19, 2007.

[11] Rene De La Briandais. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 295–298. ACM, 1959.

[12] Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS*, pages 137–143, 1997.

[13] Paolo Ferragina and Roberto Grossi. Improved dynamic text indexing. *J. Algorithms*, 31(2):291–319, 1999.
[14] Nathan J Fine and Herbert S Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965.

[15] Johannes Fischer and Volker Heun. Range median of minima queries, super-cartesian trees, and text indexing. In *Proceedings of the 19th International Workshop on Combinatorial Algorithms, IWOCA*, pages 239–252, 2008.

[16] Johannes Fischer, Tomohiro I, and Dominik Köppl. Deterministic sparse suffix sorting on rewritable texts. In *LATIN 2016: Theoretical Informatics - 12th Latin American Symposium*, pages 483–496, 2016.

[17] Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, 410(51):5354–5364, 2009.

[18] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[19] Pawel Gawrychowski and Tomasz Kociumaka. Sparse suffix tree construction in optimal time and space. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 425–439, 2017.

[20] Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 315–324, 1987.

[21] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[22] Piotr Indyk. A small approximately min-wise independent family of hash functions. *J. Algorithms*, 38(1):84–90, 2001.

[23] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003.

[24] Juha Kärkkäinen and Esko Ukkonen. Sparse suffix trees. In *Computing and Combinatorics, Second Annual International Conference, COCOON*, pages 219–230, 1996.

[25] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[26] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching, 12th Annual Symposium, CPM*, pages 181–192, 2001.

[27] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal pattern matching queries in a text and applications. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 532–551, 2015.

[28] Dmitry Kosolobov. Tight lower bounds for the longest common extension problem. *Inf. Process. Lett.*, 125:26–29, 2017.

44
[29] Mamoru Maekawa. A square root N algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985.

[30] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[31] Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

[32] Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumerical. *J. ACM*, 15(4):514–534, 1968.

[33] Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *Proceedings of 50th Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 315–323, 2009.

[34] Simon J. Puglisi and Andrew Turpin. Space-time tradeoffs for longest-common-prefix array computation. In *Algorithms and Computation, 19th International Symposium, ISAAC 2008, Gold Coast, Australia, December 15-17, 2008. Proceedings*, pages 124–135, 2008.

[35] Süleyman Cenk Sahinalp and Uzi Vishkin. Symmetry breaking for suffix tree construction. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, pages 300–309, 1994.

[36] Süleyman Cenk Sahinalp and Uzi Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract). In *37th Annual Symposium on Foundations of Computer Science, FOCS*, pages 320–328, 1996.

[37] Yuka Tanimura, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, Simon J. Puglisi, and Masayuki Takeda. Deterministic sub-linear space LCE data structures with efficient construction. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM*, pages 1:1–1:10, 2016.

[38] Yuka Tanimura, Takaaki Nishimoto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Small-space encoding LCE data structure with constant-time queries. *CoRR*, abs/1702.07458, 2017.

[39] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[40] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory*, pages 1–11. IEEE Computer Society, 1973.