Enumeration of LCP values, LCP intervals and Maximal repeats in BWT-runs Bounded Space

Takaaki Nishimoto
RIKEN Center for Advanced Intelligence Project, Japan
takaaki.nishimoto@riken.jp

Yasuo Tabei
RIKEN Center for Advanced Intelligence Project, Japan
yasuo.tabei@riken.jp

Abstract

Lcp-values, lcp-intervals, and maximal repeats are powerful tools in various string processing tasks and have a wide variety of applications. Although many researchers have focused on developing enumeration algorithms for them, those algorithms are inefficient in that the space usage is proportional to the length of the input string. Recently, the run-length-encoded Burrows-Wheeler transform (RLBWT) has attracted increased attention in string processing, and various algorithms on the RLBWT have been developed. Developing enumeration algorithms for lcp-intervals, lcp-values, and maximal repeats on the RLBWT, however, remains a challenge. In this paper, we present the first such enumeration algorithms with space usage not proportional to the string length. The complexities of our enumeration algorithms are $O(n \log \log(n/r))$ time and $O(r)$ words of working space for string length $n$ and RLBWT size $r$.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases Enumeration algorithm, Burrows-Wheeler transform, Maximal repeats, LCP array

1 Introduction

For the notion of the longest common prefix (LCP or lcp), lcp-values, lcp-intervals, and maximal repeats are powerful tools for practical string processing [23, 1, 13], and they have a wide variety of applications [30, 5, 3, 21, 16, 20]. Although many researchers have focused on developing enumeration algorithms for them, those algorithms are inefficient in that the space usage is proportional to the length of the input string. Recently, the run-length-encoded Burrows-Wheeler transform (RLBWT) has attracted increased attention, and various algorithms on the RLBWT have been developed. Developing space-efficient enumeration algorithms for lcp-intervals, lcp-values, and maximal repeats on the RLBWT, however, remains a challenge. Hence, we present the first such enumeration algorithms in this paper.

An lcp-value is the length of the LCP of lexicographically adjacent suffixes. Enumeration of all lcp-values is useful for constructing an LCP array and a succinct permuted LCP (succinct PLCP) array [22, 33].

An lcp-interval is an interval that represents all occurrences of a right-maximal repeat on a suffix array [27] (i.e., an integer array storing the positions of all the suffixes of a given string in lexicographic order). Enumeration of the lcp-intervals on a given string is useful for constructing a compressed suffix tree [20] and enumerating several characteristic substrings, e.g., maximal repeats, minimal absent words, and minimal unique substrings [5]. Kasai et al. [23] presented the first enumeration algorithm for lcp-intervals without the suffix tree of a given string. Belazzougui [4] showed that lcp-intervals can be enumerated in $O(n \log \log(n/r))$ time and $O(r)$ bits of additional working space, where $\sigma$ is the alphabet size. Very
recently, Prezza and Rosone [32] proposed an enumeration algorithm for lcp-intervals that runs in $O(n(\log \sigma + \epsilon^{-1}\log \log n))$ time and uses $n \log \sigma(\epsilon + o(1))$ bits of additional working space for a user-defined parameter $0 < \epsilon \leq 1$.

A maximal repeat is a left- and right-maximal repeat, i.e., a substring that occurs at least twice and has all its extensions occurring fewer times. Maximal repeats are used in various applications, e.g., lossless data compression [10], document clustering and classification [30, 28], and computational biology (e.g., [3, 21]). So far, many enumeration algorithms for maximal repeats have been proposed (e.g., [30, 26, 7, 5]). Beller et al. [10] and Belazzougui et al. [6] showed that the lcp-intervals representing the maximal repeats of a given string can be enumerated in $O(n \log \sigma)$ time and with $O(n \log \sigma)$ bits by using wavelet trees. Belazzougui and Cunial [5, 4] also showed that the lcp-intervals representing maximal repeats can be enumerated in $O(n)$ time and $O(n \log \sigma)$ bits of additional working space.

The Burrows-Wheeler transform (BWT) [12] is a reversible transformation for lossless data compression and string processing algorithms. The BWT of a string is defined as a permutation of the string and consists of the last characters of the sorted circular strings of the string. Moreover, the run-length BWT (RLBWT) is a recent, popular lossless data compression algorithm (or format). It is defined as the string computed by applying the run-length encoding to the BWT of a given string. The RLBWT practically achieves a high compression ratio for highly repetitive texts such as genome sequences for the same species, version-controlled documents, and source code repositories. Very recently, Kempa and Kociumaka [25] showed an upper bound on the size of the RLBWT by using a measure of repetitiveness, $\delta$. Several compressed data structures using the RLBWT structure have been proposed for string processing algorithms (e.g., [6, 17, 29, 2, 24]), and those algorithms use $O(r \polylog n)$ bits. Furthermore, the space usage of those data structures can drop to $o(n)$ bits when the RLBWT is very small (i.e., when $r \ll n$), where $r$ is the number of runs in the RLBWT and $n$ is the length of the input string.

Finally, the succinct PLCP array is a bit array of length $2n$ storing a compressed representation of the LCP array of a given string of length $n$. It can support random access on the LCP array by combining the bit array with the suffix array or compressed suffix array [29] for the given string. Belazzougui [4] showed that the succinct PLCP array can be constructed in $O(n)$ time and $O(n \log \sigma)$ bits of working space. Very recently, Prezza and Rosone [32] showed that the succinct PLCP array can be constructed in $O(n(\log \sigma + \epsilon^{-1}\log \log n))$ time and $n \log \sigma(\epsilon + o(1))$ bits of additional working space by using an enumeration algorithm for lcp-values, where $0 < \epsilon \leq 1$ is a user-defined parameter.

Our contribution. Our contribution. We present three new enumeration algorithms for the lcp-values, lcp-intervals, and maximal repeats on the RLBWT of a string $T$. In addition, we present an enumeration algorithm for the lcp-intervals representing maximal repeats and a construction algorithm for the succinct PLCP array. For a given string $T$ of length $n$, all the algorithms run in $O(n(\log r + \log \log_w(n/r)))$ time with the same space usage: $O(r \log n)$ bits of working space, where $w = \Theta(\log n)$ is the machine word size, and $r$ is the number of runs in the RLBWT of $T$. Moreover, for a given RLBWT of string $T$, all the algorithms run in $O(n(\log \log_w(n/r)))$ time and $O(r \log n)$ bits of working space. This is because the RLBWT of string $T$ can be built in $O(n \log r)$ time and $O(r \log n)$ bits of working space [29, 31]. Our algorithms are the first such algorithms for enumeration of lcp-values, lcp-intervals, and maximal repeats and construction of the succinct PLCP array that use $O(r \log n)$ bits of working space; furthermore, the working space can be smaller than that of previous algorithms when $r$ is small. Table [1] summarizes the existing and proposed enumeration algorithms for lcp-intervals and maximal repeats and construction algorithms.
Table 1 Summary of the running time and working space of enumeration algorithms for lcp-intervals and maximal repeats and construction algorithms for the succinct PLCP array. The input of these algorithms is the BWT or RLBWT of a string $T$ of length $n$. In addition, $\sigma$ is the alphabet size of $T$, $w = \Theta(\log n)$ is the machine word size, $0 < \epsilon \leq 1$ is a user-defined parameter, and $r$ is the number of runs in the RLBWT of $T$.

| Algorithm (lcp-intervals) | Running time | Working space (bits) |
|---------------------------|--------------|----------------------|
| Belazzougui (Lemma 5 [4]) | $O(n)$       | $O(n \log \sigma)$  |
| Prezza and Rosone [32]    | $O(n(\log \sigma + \epsilon^{-1} \log \log n))$ | $n \log \sigma(\epsilon + o(1))$ |

| Algorithm (maximal repeats) | Running time | Working space (bits) |
|-----------------------------|--------------|----------------------|
| Beller et al. [6]           | $O(n \log \sigma)$ | $O(n \log \sigma)$  |
| Belazzougui and Cunial [5]  | $O(n)$       | $O(n \log \sigma)$  |

| Algorithm (succinct PLCP array) | Running time | Working space (bits) |
|---------------------------------|--------------|----------------------|
| Belazzougui [4]                 | $O(n)$       | $O(n \log \sigma)$  |
| Prezza and Rosone [32]          | $O(n(\log \sigma + \epsilon^{-1} \log \log n))$ | $n \log \sigma(\epsilon + o(1))$ |
| Dominik Kempa (Theorem 5.2 [24]| $O(n/\log n + r \log^{1+\epsilon} n)$ | $O(n + r \log^{3} n)$ |

| This study                     | $O(n \log \log_w (n/r))$ | $O(r \log n)$ |
|--------------------------------|---------------------------|---------------|

for the succinct PLCP array.

In addition, we present a practical enumeration algorithm for lcp-intervals that runs in $O(n \log \log_w (n/r))$ time and $O(r + K) \log n$ bits of working space, where $K$ is the maximal number of elements in the algorithm’s stack, with $K \ll r$ in many cases. We then implement our enumeration algorithm for maximal repeats by using this practical enumeration algorithm for lcp-intervals, and we show its effectiveness on benchmark datasets of highly repetitive texts.

2 Preliminaries

Let $\Sigma$ be an ordered alphabet of size $\sigma$, $T$ be a string of length $n$ over $\Sigma$, and $|T|$ be the length of $T$. Let $T[i]$ be the $i$-th character of $T$, and $T[i..j]$ be the substring of $T$ that begins at position $i$ and ends at position $j$. For two strings $T$ and $P$, $T \prec P$ means that $T$ is lexicographically smaller than $P$. We assume the following two conditions: (i) The last character of $T$ is a special character $\$ not occurring in substring $T[1..n-1]$ and it is lexicographically smaller than any other character (i.e., $\$ $\prec c$ for any character $c \in \Sigma \setminus \{\$\}). (ii) $\sigma = O(n)$. For two integers $b$ and $c$ ($b \leq c$), $[b, c]$ is the set $\{b, b+1, \ldots, c\}$ called interval. We also call $b$ and $c$ left boundary and right boundary of the interval, respectively. Let $\text{Count}_T(P)$ be the number of occurrences of a given string $P$ in $T$, i.e., $\text{Count}_T(P) = |\{i \mid P = T[i..(i+|P|-1)], i \in [1,n-|P|+1]\}|$.

For a substring $P$ of $T$, we call $P$ a repeat if $\text{Count}_T(P) \geq 2$. Similarly, we call $P$ left-maximal (respectively, right-maximal) if $\text{Count}_T(cP) < \text{Count}_T(P)$ (respectively, $\text{Count}_T(Pc) < \text{Count}_T(P)$) for any character $c \in \Sigma$. Lastly, we call $P$ a maximal repeat when it is both a left- and right-maximal repeat. For example, the maximal repeats of $T = \text{banana}$ are $a$ and $ana$.

Our computation model is a unit-cost word RAM with a machine word size of $w = \Theta(\log_2 n)$ bits. We evaluate the space complexity in terms of the number of machine words. A bitwise evaluation of the space complexity can be obtained with a multiplicative factor.
of \( \log_2 n \). We assume the base-2 logarithm throughout this paper when the base is not indicated.

### 2.1 Predecessor and interval queries

For an integer \( x \) and a set \( S \) of integers, a predecessor query \( \text{pred}(S, x) \) returns the number of elements that are no more than \( x \) in \( S \) (i.e., \( \text{pred}(S, x) = |\{ y \mid y \in S \text{ s.t. } y \leq x \}| \)). The predecessor data structure \([8]\) for \( S \) enables any predecessor query on \( S \) in \( O(\log \log_w (u/m)) \) time and with \( O(m) \) words of space, where \( m \) is the number of elements in \( S \), and \( u \) is the size of the universe of elements. The predecessor data structure can be constructed in \( O(m \log \log_w (u/m)) \) time and \( O(m) \) words of working space \([17]\) by processing the set \( S \).

Next, for an integer \( x \) and a set \( S' \) of intervals, a report query \( \text{report}(S', x) \) reports the intervals containing \( x \) in \( S' \). A delete query \( \text{delete}(S', x) \) deletes the intervals containing \( x \) in \( S' \). The semi-dynamic interval tree for \( S' \) supports report and delete queries on \( S' \) in \( O((1 + k) \log u) \) time, where \( k \) is the number of reported or removed elements in \( S' \), and \( u \) is the maximal right boundary of the intervals in \( S' \) (i.e., \( u = \max \{ e \mid [b, e] \in S' \} \)). It can be implemented using an interval tree \([14]\) and a linked list. We can construct the semi-dynamic interval tree in \( O(|S'| \log u) \) time and \( O(|S'|) \) words of working space by preprocessing the set \( S' \). We denote a semi-dynamic interval tree as \( \Gamma \). See Appendix A for a more detailed description of it.

### 2.2 Suffix array (SA) and longest common prefix (LCP) array

The suffix array (SA) \([27]\) of string \( T \) is an integer array of size \( n \) such that \( \text{SA}[i] \) stores the starting position of the \( i \)-th suffix of \( T \) in lexicographical order. Formally, \( \text{SA} \) is a permutation of \([1, n]\) such that \( T[\text{SA}[1]..n] < \cdots < T[\text{SA}[n]..n] \). The LCP array (LCP) of \( T \) is an integer array of size \( n \) such that \( \text{LCP}[1] = 0 \) and \( \text{LCP}[i] \) stores the length of the LCP of the two suffixes \( T[\text{SA}[i]..n] \) and \( T[\text{SA}[i-1]..n] \) for \( i \in [2, n] \). We call the values in the suffix array and LCP array \( sa-values \) and \( lcp-values \), respectively. Figure 1 depicts the suffix array and LCP array of a string \( T \).

Moreover, let \( \text{LF} \) be a function such that (i) \( \text{SA}[\text{LF}(i)] = \text{SA}[i] - 1 \) for any integer \( i \in [1, n] \) such that \( \text{SA}[i] \neq 1 \); and (ii) \( \text{SA}[\text{LF}(i)] = n \) for \( i \in [1, n] \) such that \( \text{SA}[i] = 1 \). Let \( \text{FL} \) be the inverse function of \( \text{LF} \), i.e., for any integer \( i \in [1, n] \), \( \text{FL}(i) \) returns an integer \( j \) such that \( \text{LF}(j) = i \).

### 2.3 SA-intervals and lc-intervals

For a substring \( P \) of \( T \), the \( sa \)-interval of \( P \) is a 3-tuple \((b, e, |P|)\) such that \( \text{SA}[b..e] \) represents all the occurrence positions of \( P \) in \( T \); that is, for any integer \( p \in [1, n] \), \( T[p..p+|P|-1] = P \) if and only if \( p \in \text{SA}[b..e] \). The length of the substring is called depth of the sa-interval. For two sa-intervals \( J = (b, e, d) \) and \( J' = (b', e', d') \), \( J \subset J' \) if \([b, e] \subset [b', e'] \). The sa-interval of a right-maximal repeat \( P \) is then called an lc-interval. Let \( I \) be the set of the lc-intervals on the suffix array of \( T \).

### 2.4 BWT and RLBWT

The Burrows-Wheeler transform (BWT) \([12]\) of a string \( T \) is a permutation \( L \) of \( T \) built as follows. We sort all the \( n \) rotations of \( T \) in lexicographical order and take the last character of each rotation in the sorted order. Formally, let \( L \) be the permutation of \( T \) such that
$L[i] = T[\text{LF}(\text{SA}[i])]$ for any $i \in [1, n]$, and let $F$ be a permutation of $T$ that consists of the first character of each rotation in the sorted order, i.e., $F[i] = T[\text{SA}[i]]$ for any $i \in [1, n]$.

Then, the RLBWT of $T$ is the BWT encoded by a run-length encoding, i.e., a partition of $L$ into $r$ substrings $L_1, L_2, \ldots, L_r$ such that each $L_i$ is a maximal repetition of the same character in $L$. We call such a maximal repetition an $L$-run. The RLBWT can be stored in $2r$ words, because we can represent each $L$-run in 2 words. $L^+$ is defined as the set of starting positions of $L$-runs, i.e., $L^+ = \{1, |L_1| + 1, \ldots, \sum_{k=1}^{r-1} |L_k| + 1\}$, and each element of $L^+$ is denoted as $\ell^+_i$, i.e., $\ell^+_i = 1 + \sum_{k=1}^{i-1} |L_k|$. Similarly, $L^-$ is defined as the set of ending positions of $L$-runs, i.e., $L^- = \{\sum_{k=1}^{i} |L_k|, \sum_{k=1}^{i} |L_k| + |L_{i+1}|, \ldots, \sum_{k=1}^{r} |L_k|\}$, and each element of $L^-$ is denoted as $\ell^-_i$, i.e., $\ell^-_i = \sum_{k=1}^{i} |L_k|$.

### 2.5 Adjacent relation on SA and incremental relation on LCP array

An adjacent relation on the SA string $T$ means that the previous sa-values on adjacent sa-values for any $L$-run occur as adjacent sa-values in the SA, i.e., $\text{LF}(x + 1) = \text{LF}(x) + 1$ for any $x \in [\ell^+_i, \ell^-_i]$. An incremental relation on the LCP array of string $T$ means that the length of the LCP for the previous sa-values is 1 plus the length of the LCP for the adjacent sa-values, i.e., $\text{LCP}[\text{LF}(x + 1)] = \text{LCP}[x + 1]$ for any $x \in [\ell^+_i, \ell^-_i]$.

The adjacent and incremental relations hold for the following reason. Consider two adjacent positions $x$ and $x + 1$ in the SA of string $T$ and their suffixes $S_x$ and $S_{x+1}$ (i.e., $S_x = T[\text{SA}[x]..n]$ and $S_{x+1} = T[\text{SA}[x+1]..n]$). Obviously, there is no suffix $S$ such that $S_x \prec S \prec S_{x+1}$, because $S_x$ and $S_{x+1}$ are adjacent in the SA. If the $L[x]$ and $L[x+1]$ are the same character $c$, then $cS_x$ and $cS_{x+1}$ are also suffixes of $T$, where $L$ is the BWT of $T$. Because the suffix $S$ does not exist, $cS_x$ and $cS_{x+1}$ are also adjacent in the SA. Similarly, the LCP of suffixes $cS_x$ and $cS_{x+1}$ is obviously 1 plus the LCP of $S_x$ and $S_{x+1}$.

The permutation $F$ for the BWT of $T$ can be considered as a permutation of $L$-runs, because (i) $\text{LF}$ is a bijective function from $[1, n]$ to $[1, n]$, and (ii) it maps the interval $[\ell^+_i, \ell^-_i]$ on any $L$-run $L_i$ to the interval on the permutation $F$ by the adjacent relation on the SA. We call the substring corresponding to each $L$-run on $F$ an $F$-run, and we denote the $i$-th $F$-run as $F_i$. Formally, let $p_1, p_2, \ldots, p_r$ be the permutation of $[1, r]$ such that $\text{LF}(\ell^+_{p_{j-1}}) < \text{LF}(\ell^+_{p_{j}})$ for any integer $j \in [2, r]$; then, $F_i = \text{LF}(p_i) \cdot \text{LF}(p_i) + |L_{p_i}|-1$. $F^+$ is defined as the set of starting positions of $F$-runs, i.e., $F^+ = \{1, |F_1| + 1, \ldots, \sum_{k=1}^{r-1} |L_k| + 1\}$, and each element of $F^+$ is denoted as $f^+_i$, i.e., $f^+_i = \sum_{k=1}^{i-1} |L_k| + 1$.

Figure 1 depicts the SA, LCP array, BWT, and circular strings of a string $T = \text{abaababaabab}$.

| $i$ | SA | LCP | F | L(BWT) |
|-----|-----|-----|-----|--------|
| 1   | 16  | 0   | $\text{abaabababaa}$ | b |
| 2   | 8   | 0   | $\text{abaababab}$ | b |
| 3   | 11  | 0   | $\text{abaabababab}$ | b |
| 4   | 3   | 5   | $\text{abaababababab}$ | b |
| 5   | 14  | 1   | $\text{abaabababababab}$ | b |
| 6   | 6   | 2   | $\text{abaababababab}$ | b |
| 7   | 9   | 0   | $\text{abaabababababab}$ | b |
| 8   | 1   | 7   | $\text{abaababababababab}$ | b |
| 9   | 12  | 3   | $\text{abaababababababab}$ | a |
| 10  | 4   | 4   | $\text{abaabababababab}$ | a |
| 11  | 15  | 0   | $\text{abaababababababab}$ | a |
| 12  | 1   | 1   | $\text{abaabababababababab}$ | a |
| 13  | 10  | 3   | $\text{abaababababababababab}$ | a |
| 14  | 2   | 6   | $\text{abaabababababababab}$ | a |
| 15  | 13  | 2   | $\text{abaababababababababab}$ | a |
| 16  | 5   | 3   | $\text{abaabababababababababab}$ | a |
LCP[LF(3)] = LCP[3] + 1 = 5 hold by the adjacent relation on the SA and the incremental relation on the LCP array. We can construct a data structure Z with \(O(r)\) words to support the LF and FL functions by using the adjacent relation on the SA (e.g., \[17\]). Here, Z uses predecessor queries on \(L^+\) and \(F^+\). Then, the following lemma holds.

The algorithm is built on the following five data structures: (i) the data structure \(F\) \(p\ Φ \[\[\]\]\) \(∈\) way:

\[\text{Lemma 1. There exists a data structure } Z \text{ of } O(r) \text{ words to support (i) the LF and FL functions, (ii) accesses to } F[i] \text{ and } L[i] \text{ for a given integer } i \in [1, n], (iii) predecessor queries on } F^+ \text{ and } L^+, \text{ and (iv) accesses to positions } f_i^+ \text{ and } ℓ_i^+ \text{ for a given integer } i \in [1, r]. \] \(Z\) can execute the first three operations and the last operation in \(O(\log \log \sigma(n/r))\) and \(O(1)\) time, respectively. We can construct \(Z\) in \(O(n + r \log \log \sigma(n/r))\) time and \(O(r)\) words of working space by preprocessing the RLBWT of \(T\) over an alphabet of size \(σ = O(n)\).

Proof. See Appendix A. ▷

3 Enumeration of lcp-values on RLBWT

We present two new algorithms for enumerating all the lcp-values for \(T\) using the RLBWT of \(T\). The first algorithm runs in \(O(n \log \log \sigma(n/r) + r \log n)\) time and \(O(r \log n)\) bits of working space, and it runs in \(O(n \log \log \sigma(n/r))\) time if \(r < \frac{n}{\log n}\). The second algorithm runs in \(O(n \log \log \sigma(n/r))\) time and \(O(n + r \log n)\) bits of working space, and it uses \(O(n + r \log n) = O(r \log n)\) bits of working space if \(r \geq \frac{n}{\log n}\). Thus, all the lcp-values can be enumerated in \(O(n \log \log \sigma(n/r))\) time and \(O(r \log n)\) bits of working space for any string by combining these two algorithms.

3.1 Enumeration algorithm in \(O(n \log \log \sigma(n/r) + r \log n)\) time and \(O(r \log n)\) bits of working space

The algorithm is built on the following five data structures: (i) the data structure \(Z\) supporting the LF function and predecessor queries on \(L^+\) in Lemma 1, (ii) a semi-dynamic interval tree \(Γ\) storing a set of at most \(r\) intervals in the LCP array; (iii) a bit array \(V\) of length \(2r\); (iv) an integer array \(D\) storing the positions with lcp-value 0 in increasing order; and (v) intervals \(B_1, B_2, ..., B_{2r}\) defined as a partition of interval \([1, n]\) such that \(B_{2i−1} = [ℓ_i^+, ℓ_i^−]\) and \(B_{2i} = [ℓ_i^+, ℓ_i^−]\) for any integer \(i \in [1, r]\). The space usage of these five data structures is \(O(r)\) words in total.

A set of intervals in the LCP array is computed by using the FL function in the following way: \(\{\text{FL}(f_i^+ − 1) + 1, \text{FL}(f_i^+), \text{FL}(f_i^+ − 1) + 1, \text{FL}(f_i^+), ..., \text{FL}(f_i^+ − 1) + 1, \text{FL}(f_i^+)\}\). We call each interval \([\text{FL}(f_i^+ − 1) + 1, \text{FL}(f_i^+)]\) an FL-interval. The semi-dynamic interval tree \(Γ\) stores a set \(Ψ\) of FL-intervals such that the lcp-value at position \(f_i^+\) in each FL-interval \([\text{FL}(f_i^+ − 1) + 1, \text{FL}(f_i^+)\]) is no less than 1 (i.e., \(Ψ = \{\text{FL}(f_i^+ − 1) + 1, \text{FL}(f_i^+)\} \mid f_i^+ \in F^+ \text{ s.t. } LCP[f_i^+] \geq 1\}\)). Each \(i\)-th bit in bit array \(V\) is set to 1 if \(Γ\) does not store any FL-interval overlapping with interval \(B_i\), and it is set to 0 otherwise.

A basic idea behind our algorithm is to enumerate all lcp-values in increasing order by using the above five data structures. Positions with lcp-value \(p\) are gradually computed from positions with lcp-value \(p − 1\) in the LCP array for any integer \(p \in [1, n]\) by using the FL-intervals, the LF function, and a property of primary and secondary positions, explained as follows.

A primary position with lcp-value \(p\) is defined as \(f_i^+\) such that (i) \(LCP[f_i^+] ≥ 1\), and (ii) FL-interval \([\text{FL}(f_i^+ − 1) + 1, \text{FL}(f_i^+)]\) contains lcp-value \(p − 1\) as the smallest lcp-value in the interval on the LCP array. The set of all primary positions with lcp-value \(p\) is denoted by \(Φ_p\), i.e., \(Φ_p = \{x \mid x \in F^+ \text{ s.t. } LCP[x] ≥ 1 \text{ and } \min(LCP[\text{FL}(x − 1) + 1..\text{FL}(x)]) = p − 1\}\). Πₚ
is the set of secondary positions defined as positions computed by applying the LF function to positions with lcp-value \( p - 1 \), except for the positions in \( L^+ \), i.e., \( \Pi_p = \{ \text{LF}(x) \mid x \in [1, n] \text{ s.t. } x \notin L^+ \text{ and } \text{LCP}[x] = p - 1 \} \).

A property of the primary and secondary positions is that the union of the sets of primary and secondary positions with lcp-value \( \ell \) results in the set of positions with lcp-value \( p \), as presented in the following lemma.

**Lemma 2.** \( \Phi_p \cup \Pi_p = \{ x \mid x \in [1, n], \text{LCP}[x] = p \} \) for any integer \( p \in [1, n] \).

**Proof.** See Appendix B.

Lemma 2 shows that we can enumerate lcp-value \( p \) by taking the union of the primary positions \( \Phi_p \) and the secondary positions \( \Pi_p \) computed with the previous lcp-value \( p - 1 \), \( FL \)-intervals, and LF function.

To find the primary positions with lcp-value \( p \), our algorithm uses a semi-dynamic interval tree \( \Gamma \) storing the set \( \Psi_p \) of FL-intervals such that the lcp-value at the primary position of each FL-interval is no less than \( p \) (i.e., \( \Psi_p = \{ \text{FL}(f^+_{i, j} - 1) + 1, \text{FL}(f^+_{i, j}) \mid f^+_{i, j} \in F^+ \text{ s.t. } \text{LCP}[f^+_{i, j}] \geq p \} \}). The report query of \( \Gamma \) for a position with lcp-value \( p - 1 \) on set \( \Psi_p \) using \( \Gamma \) reports the FL-intervals such that each FL-interval contains lcp-value \( p - 1 \) as the smallest lcp-value in the FL-interval on the LCP array, because it reports the FL-intervals such that each FL-interval contains the lcp-value \( p - 1 \), and any FL-interval in set \( \Psi_p \) does not contain lcp-values less than \( p - 1 \). Therefore, we can obtain the primary positions with lcp-value \( p \) (i.e., \( \Phi_p \)) by using the report query for the input positions with lcp-value \( p - 1 \) on set \( \Psi_p \), because (i) set \( \Psi_p \) contains all FL-intervals with the smallest lcp-value \( p - 1 \), (ii) the report query reports the FL-intervals with the smallest lcp-value \( p - 1 \), and (iii) the lcp-value at the primary position of any FL-interval with the smallest lcp-value \( p - 1 \) is \( p \).

Then, we immediately remove the reported FL-intervals from set \( \Psi_p \) by a delete query on \( \Gamma \) for the input positions with lcp-value \( p - 1 \) to obtain a semi-dynamic interval tree \( \Gamma \) storing the set \( \Psi_{p + 1} \), so as not to report the same FL-intervals multiple times. Thus, we can find the primary positions with lcp-value \( p \) by using \( \Gamma \) and the positions of lcp-value \( p - 1 \) for any integer \( p \in [1, n] \).

To reduce the number of executions for the report queries to \( O(r) \), the algorithm skips the report query for a given query position \( x \) if an interval \( B_i \) contains \( x \) (i.e., \( x \in B_i \)) and the \( i \)-th bit is 1 in the bit array \( V \). The results of the skipped report queries are always empty, because (i) the interval \( B_i \) contains the query positions \( x \), (ii) semi-dynamic interval tree \( \Gamma \) does not store the FL-intervals containing the interval \( B_i \) if \( V[i] = 1 \), and (iii) any FL-interval consists of consecutive elements of intervals \( B_1, B_2, \ldots, B_r \), i.e., \( \forall [b, e] \in \Psi, \exists [t, t'] \subseteq [1, 2r], [b, e] = B_t \cup B_{t+1} \cup \cdots \cup B_r \). Therefore, we can find all the FL-intervals stored in semi-dynamic interval tree \( \Gamma \) with at most \( 2r \) report queries.

Here, we show an example of computing the positions with lcp-value 4 in Figure 1. \( F^+ = \{ 1, 2, 3, 11 \} \), \( L^+ = \{ 1, 7, 8, 9 \} \), the sets of positions with lcp-values 3 and 4 are \( \{ 9, 16 \} \) and \( \{ 3, 10 \} \), respectively. \( \Psi = \{ [8, 9] \} \), and intervals \( B_1, B_2, \ldots, B_8 \) are \( B_1 = [1, 1] \), \( B_2 = [2, 6] \), \( B_3 = [7, 7] \), \( B_4 = \emptyset \), \( B_5 = [8, 8] \), \( B_6 = \emptyset \), \( B_7 = [9, 9] \), and \( B_8 = [10, 16] \). The set \( \Phi_4 \) of primary positions with lcp-value 4 is \( \{ 3 \} \), because the smallest lcp-value in FL-interval \([8, 9]\) is 3, and the primary position of the FL-interval is 3. The set \( \Pi_4 \) of secondary positions with lcp-value 4 is \( \{ 10 \} \), because the only position with lcp-value 3 such that each position is not contained in \( L^+ \) is 16, and \( \text{LF}(16) = 10 \). Therefore, \( \Phi_4 \cup \Pi_4 \) represents the set of positions with lcp-value 4 by Lemma 2. In addition, the FL-interval \([8, 9]\) consists of consecutive intervals \( B_5, B_6, \) and \( B_7 \), i.e., \([8, 9] = B_5 \cup B_6 \cup B_7 \).
The total running time and working space of our enumeration algorithm are \(O(n \log \log w_r(n/r) + (r + |\Psi|) \log n)\) and \(O(r + \max(|\Phi_1 \cup \Pi_1|, |\Phi_2 \cup \Pi_2|, \ldots, |\Phi_n \cup \Pi_n|))\) words, respectively, except for the construction algorithm for the five data structures \(Z, \Gamma, V, D,\) and \(B_1, B_2, \ldots, B_2r\). This is because the algorithm uses the \(O(n)\) LF function and predecessor queries on \(L^\dagger\) and the \(O(r)\) report and delete queries on the semi-dynamic interval tree \(\Gamma\), and it must temporarily store the positions with lcp-value \(p - 1\) to compute those with lcp-value \(p\) for any integer \(p \in [1, n]\). Here, \(|\Psi| \leq r\) and \(\max(|\Phi_1 \cup \Pi_1|, |\Phi_2 \cup \Pi_2|, \ldots, |\Phi_n \cup \Pi_n|) \leq r\), as shown in Lemma 3. See Appendix B for the details of our enumeration algorithm and construction algorithms. Finally, we obtain the following lemma and theorem.

\[\textbf{Lemma 3.} \text{The following two statements hold.} (i) \{x \mid x \in [1, n] \text{ s.t. } \text{LCP}[x] = p\} \leq n \text{ for any integer } p \in [0, n]. (ii) For any position } x \in [1, n] \text{ and FL-interval } [b, e) \in \Psi, x \in [b, e) \Leftrightarrow B_i \subseteq [b, e), \text{ where } i \text{ is an integer such that } x \in B_i.\]

\[\textbf{Proof.} \text{See Appendix B.}\]

\[\textbf{Theorem 4.} \text{We can enumerate all the lcp-values in the LCP array of a string } T \text{ in } O(n \log \log w_r(n/r) + r \log n) \text{ time and } O(r) \text{ words of working space by preprocessing the RLBWT of } T.\]

\[\textbf{Proof.} \text{See Appendix B.}\]

### 3.2 Enumeration algorithm in \(O(n \log \log w_r(n/r))\) time and \(O(n + r \log n)\) bits of working space

Our algorithm leverages Beller et al.'s algorithm \[11\] that does not enumerate lcp-values but can enumerate Weiner-intervals, defined as sa-intervals \((b, e, d)\) having one-to-one correspondence to an lcp-value on an LCP array and a property that \(\text{LCP}[e + 1] = d - 1\).

Formally, the Weiner-interval with right boundary \(e\) is the sa-interval \((b, e, \text{LCP}[e + 1] + 1)\) of substring \(T[\text{SA}[e] \ldots \text{SA}[e + \text{LCP}[e + 1]]]\) for any integer \(e \in [0, n]\), where \(\text{LCP}[n + 1] = 0\). First, we introduce Beller et al.'s enumeration algorithm running in \(O(n \log \sigma)\) time and \(O(n \log \sigma + r \log n)\) bits of working space. Then, we present our enumeration algorithm running in \(O(n \log \log w_r(n/r))\) time and \(O(n + r \log n)\) bits of working space by modifying the data structures used in their algorithm.

Beller et al.'s algorithm is built on the following three data structures: (i) a data structure \(U\) supporting Weiner queries; (ii) a queue \(Q\) storing all Weiner-intervals with depth 1 in any order; and (iii) a bit array \(V'\) of size \(n\). A Weiner query \(\text{weiner}(b, e, |P|)\) returns a set of all the sa-intervals of substrings \(c_1P, c_2P, \ldots, c_\sigma P\) for a given sa-interval \((b, e, |P|)\) of substring \(P\), i.e., \(\text{weiner}(b, e, |P|) = \{\text{interval}(c^P) \mid c \in \Sigma \text{ s.t. } \text{Count}_\Sigma(c^P) \geq 1\}\), where \(\text{interval}(c^P)\) is the sa-interval of the string \(cP\), and \(\Sigma = \{c_1, c_2, \ldots, c_\sigma\}\). Data structure \(U\) supports a Weiner query in \(O(k \log \sigma)\) time, and the space usage is \(O(n \log \sigma)\) bits \[11\], where \(k\) is the number of outputs by the Weiner query. Each \(i\)-th bit in bit array \(V'\) is set to 1 if the Weiner-interval with right boundary \(i\) has already been outputted by the algorithm, and it is set to 0 otherwise. The space usage of the data structures is \(O(n \log \sigma + m \log n)\) bits in total, where \(m\) is the maximal number of Weiner-intervals stored in queue \(Q\).

Beller et al.'s algorithm enumerates all the Weiner-intervals by using a filtered set of sa-intervals and leveraging its key property. The filtered set \(\text{filter}(w)\) for Weiner interval \(w\) is a set of sa-intervals such that each is obtained by a Weiner query for \(w\), and the corresponding bit of the sa-interval with right boundary \(e'\) is 0 in bit array \(V'\) (i.e., \(V[e'] = 0\)). The key property of a filtered set of sa-intervals is that the union of filtered sets of sa-intervals for
Weiner intervals with depth $d - 1$ represents the set of Weiner-intervals with depth $d$, which enables computing Weiner-intervals with depth $d$ from Weiner-intervals with depth $d - 1$.

Formally, let $W_d$ and $V_d'$ be, respectively, the set of Weiner-intervals with depth $d$ and the bit array $V'$ after enumerating Weiner-intervals with depth no more than $d$ (i.e., $V_d'[e] = 1$ if and only if the depth of a Weiner-interval with right boundary $e$ is no more than $d$ for any integer $e \in [1, n]$). Let \( \text{filter}(w) = \{(b', e', d) \mid (b', e', d) \in \text{weiner}(b, e, d - 1) \text{ s.t. } V_d'[e'] = 0\} \) for any Weiner-interval $w = (b, e, d - 1)$. Then, the following lemma holds.

**Lemma 6.** \( W_d = \bigcup_{w \in W_{d-1}} \text{filter}(w) \) holds for any integer $d \in [1, n]$.  

The details of their enumeration algorithm are as follows. Before it enumerates the Weiner-intervals with depth $d$, queue $Q$ stores the Weiner-intervals with depth $d - 1$, and bit array $V'$ corresponds to bit array $V_{d-1}'$. The algorithm computes Weiner-intervals with depth $d$ by using Weiner-intervals with depth $d - 1$ stored in queue $Q$, by Lemma 6. Then, it updates bit array $V'$ and replaces the elements stored in queue $Q$ with the enumerated Weiner-intervals with depth $d$.

The total running time of the algorithm is \( O(n \log \sigma) \), because it executes at most $k'$ Weiner queries for the number $k'$ of sa-intervals obtained by Weiner queries in the algorithm (i.e., $k' = \sum_{(b, e, d) \in W_1 \cup W_2 \cup \ldots \cup W_n} |\text{weiner}(b, e, d)|$). Beller et al. showed that $k' = O(n \log \sigma)$. The maximal number $m$ of elements stored in queue $Q$ can be bounded by \( \max\{|W_1|, |W_2|, \ldots, |W_n|\} \), where \( \max\{|W_1|, |W_2|, \ldots, |W_n|\} \leq r \), because Weiner-intervals with depth $d$ correspond to positions with lcp-value $d - 1$ for any integer $d$, and the number of positions with lcp-value $d$ is \( O(r) \) by Lemma 5. Thus, the algorithm runs in \( O(n \log \sigma) \) time and \( O(n \log \sigma + r \log n) \) bits of working space. Then, we following lemma holds.

**Lemma 5 (\[11\]).** \( \sum_{(b, e, d) \in W_1 \cup W_2 \cup \ldots \cup W_n} |\text{weiner}(b, e, d)| = O(n \log \sigma) \), and \( |W_d| = O(r) \) for any integer $d \in [0, n]$.  

**Our data structure.** The term \( O(n \log \sigma) \) in Beller et al.’s running time and working space is due to the data structure $U$. We thus developed a new data structure $U'$ of \( O(r \log n) \) bits supporting Weiner queries in \( O((k + 1) \log \log(n/r)) \) time. As a result, we present a new enumeration algorithm running in \( O(n \log \log(n/r)) \) time and \( O(n + r \log n) \) bits of working space by replacing the data structure $U$ with $U'$.

Our data structure $U'$ is implemented using data structure $Z$ (Lemma 1) and a new data structure $\Lambda$ supporting range color listing with first and last occurrences (RCLFL) queries on a string $L'$ of length $r$. Here, the string $L'$ consists of the first $r$ characters of L-runs (i.e., $L' = L[1]_r, L[2]_r, \ldots, L[l_+^r]$). An RCLFL query RCLFL($T, b, c$) on a string $T$ returns the set of triplets of each distinct character and its first and last occurrences in substring $T[b..c]$ for any interval $[b, c] \in [1, n]$. The data structure $\Lambda$ supports an RCLFL query on $L'$ in \( O(1) \) time per output element, while its space usage can be bounded by \( O(r) \) words.

The data structure $U'$ answers a query \( \text{weiner}(b, e, \{P\}) \) by using the following two relations among queries $\text{weiner}(b, e, \{P\})$, RCLFL($L, b, e$), and RCLFL($L', b', e'$) for the sa-interval $\{b, e, \{P\}\}$ of a substring $P$, where $L_{b'}$ and $L_{e'}$ are L-runs containing characters $L[b]$ and $L[e]$, respectively. The first relation is that the left boundary $b''_c$ (respectively, right boundary $e''_c$) of the sa-interval of substring $cP$ is equal to the result of the LF function for the first occurrence position $b_c$ (respectively, the last occurrence position $e_c$) of character $c$ in substring $L[b..c]$ for any character $c$, i.e., $b''_c = \text{LF}(b_c)$ (respectively, $e''_c = \text{LF}(e_c)$). The second relation is that we consider the case such that the two positions $b$ and $e$ are the starting and ending positions of two L-runs $L_{b'}$ and $L_{e'}$, respectively (i.e., $b = l'_b$ and $e = l'_e$); the first occurrence position $b''_c$ (respectively, the last occurrence position $e''_c$) of character $c$ in substring $L[b'..e']$
corresponds to the first occurrence position of the same character in substring $L[b..e]$, i.e., $b_c = \ell^-_c$ (respectively, $e_c = \ell^+_c$), because the first characters of the L-runs correspond on a one-to-one basis to the characters of $L'$. The two relations enable us to compute query $\text{weiner}(b_c, e_c, P)$ in $O(\log \log n(n/r))$ time per output element by using the LF function, set $L^+$, and query $\text{RCLFL}(L', b', e')$. See Appendix C for more details on the data structure $U'$ and our construction algorithm for the three data structures $U'$, $Q$, and $V'$. Formally, we obtain the following theorem.

\textbf{Theorem 7.} We can enumerate the lcp-values on the LCP array of a string $T$ in $O(n \log \log n(n/r))$ time and $O(n + r \log n)$ bits of working space by preprocessing the RLBWT of $T$. in $O(n \log \log n(n/r))$ time and $O(n + r \log n)$ bits of working space by preprocessing the RLBWT of $T$.

\textbf{Proof.} See Appendix C.

\section{Enumeration algorithm for LCP intervals in $O(r)$ words}

We next present a new enumeration algorithm for lcp-intervals with \textit{end pairs of sa-values} on string $T$ that runs in $O(n \log \log n(n/r))$ time and $O(r)$ words of working space using the RLBWT of $T$. An \textit{end pair of sa-values} (EPS) in an lcp-interval $\langle b, e, d \rangle$ is defined as a pair $\langle \text{SA}[b], \text{SA}[e] \rangle$ of sa-values $\text{SA}[b]$ and $\text{SA}[e]$. Our algorithm categorizes lcp-intervals into four types, and it enumerates all the lcp-intervals with EPSs of each type.

Specifically, each lcp-interval $I = \langle b, e, d \rangle$ is categorized as (i) type-1 if interval $[b - 1, e + 1]$ is contained in an L-run (i.e., there exists $i \in [1, r]$ such that $[b - 1, e + 1] \subseteq [\ell^+_i, \ell^-_i]$); (ii) type-2 if it is not categorized into type-1 and contains at least two positions in $L^+$ (i.e., $[b, e] \cap L^+ \geq 2$); (iii) type-3 if it is not categorized into type-1 and contains exactly one position in $L^+$; and (iv) type-4 for any other case (i.e., if $I$ is not categorized into type-1 and does not contain any position in $L^+$).

Figure 2 illustrates the four types of lcp-intervals in the example of Figure 1. The gray, yellow, orange, and blue rectangles represent lcp-intervals of type-1, type-2, type-3, and type-4, respectively.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{The lcp-intervals in the example of Figure 1. The gray, yellow, orange, and blue rectangles represent lcp-intervals of type-1, type-2, type-3, and type-4, respectively.}
\end{figure}

In this section, we only present the enumeration algorithms for lcp-intervals of type-1 and type-3 because of the space limitation. The other two enumeration algorithms are similar to the one for lcp-intervals of type-3, and they are presented in the Appendix E. Formally, we give the following theorem for our enumeration algorithm.
Theorem 8. For a given RLBWT of a string $T$, we can enumerate all the lcp-intervals with an EPS in $O(n \log \log n(n/r))$ time and $O(r)$ words of working space.

Proof. See Appendix E.

4.1 Enumeration algorithm for type-3 lcp-intervals with EPS

The enumeration algorithm for lcp-intervals of type-3 is built on the following four data structures: (i) the data structure $Z$ in Lemma 1; (ii) an array $S$ storing $r$ intervals; (iii) array $S'$ of length $r$ and storing $5$-tuples of integers to represent the intervals stored in $S$; and (iv) a run-length extended suffix array (RLESA) as proposed by Gagie et al. [17]. Each $S[i]$ stores an interval $[\ell_T^i, \ell_R^i]$. Each $S'[i]$ stores a $5$-tuple of integers $(SA[b], SA[e], \min(\text{LCP}[b+1..e]), \text{LCP}[b], \text{LCP}[e+1])$ for interval $S[i] = [b, e]$, where the third element $\min(\text{LCP}[b+1..e])$ means the length of the LCP of $SA[b..e]$. The RLESA is a data structure that supports two queries for a given $i$-th sa-value in the following way: (i) a next-access query returning two values $SA[i+1]$ and $\text{LCP}[i+1]$, and (ii) a previous-access query returning two values $SA[i-1]$ and $\text{LCP}[i-1]$. Those two queries are supported in $O(\log \log n(n/r))$ time and $O(r)$ words. Therefore, the space usage of these four data structures is $O(r)$ words in total.

For any integers $i \in [1, r]$ and $t \in [1, n]$, let $b_{i,t}$ be the smallest position in the LCP array such that (i) interval $[b_{i,t}, b_{i,t}+t-1]$ contains position $\ell_T^i$ (i.e., $\ell_T^i \in [b_{i,t}, b_{i,t}+t-1]$) and (ii) the length of the LCP of $SA[b_{i,t}, b_{i,t}+t-1]$ is no less than the length of the LCP of $SA[b', b'+t-1]$ for any $b' \in [\ell_T^i - t + 1, \ell_T^i]$. Formally, $b_{i,1} = \ell_T^i$ and $b_{i,t} = b_{i,t-1} - 1$ for any integers $t \in [2, n]$ and $i \in [1, r]$ if $\text{LCP}[b_{i,t-1}] \geq \text{LCP}[b_{i,t}+t-1]$, otherwise, $b_{i,t} = b_{i,t-1}$. Interval $[b_{i,t}, b_{i,t}+t-1]$ has the following two properties: (i) any lcp-interval $I = (b, b+t-1, d)$ containing position $\ell_T^i$ is equal to interval $[b_{i,t}, b_{i,t}+t-1]$, and (ii) any interval $[b_{i,t}, b_{i,t}+t-1]$ is an lcp-interval if and only if $\text{LCP}[b_{i,t}], \text{LCP}[b_{i,t}+t] < \min(\text{LCP}[b_{i,t}+1..b_{i,t}+t-1])$. This is because lcp-interval $(b, b+t-1, d)$ satisfies $\text{LCP}[b], \text{LCP}[b+t] < d$ and $d = \min(\text{LCP}[b+1..b+t-1])$. Therefore, the algorithm finds type-3 lcp-intervals that are equal to $[b_{i,t}, b_{i,t}+t-1]$ for any integer $i \in [1, r]$ by extending interval $[b_{1,1}, b_{1,1}]$ in a way such as $[b_{i,1}, b_{i,1}] \rightarrow [b_{i,2}, b_{i,2}+1] \rightarrow [b_{i,3}, b_{i,3}+2] \rightarrow \cdots$ for each $i$. All the type-3 lcp-intervals can be found by extending interval $[b_{i,1}]$ for all $i \in [1, r]$, because any type-3 lcp-interval contains a single position in set $L^+ = \{\ell_T^1, \ell_T^2, \ldots, \ell_T^r\}$. Formally, the following corollary and lemma hold.

Corollary 9. For any integers $b, e \in [1, n]$ and $d = \min(\text{LCP}[b+1..e])$, interval $[b, e]$ is an lcp-interval with depth $d$ if and only if $\text{LCP}[b], \text{LCP}[e+1] < d$.

Lemma 10. Let $\mathcal{I}^3_r$ be the set of type-3 lcp-intervals containing position $\ell_T^i$. Then, $[b_{i,t}, b_{i,t}+t-1] = [b, e]$ for any $i \in [1, r]$ and $(b, b+t-1, d) \in \mathcal{I}^3_r$.

The details of our enumeration algorithm for type-3 lcp-intervals with EPSs containing position $\ell_T^i$ is as follows. Before the algorithm finds the type-3 lcp-interval of length $t$ containing position $\ell_T^i$, $S[i]$ and $S'[i]$ store interval $[b_{i,t}, b_{i,t}+t-1]$ and the $5$-tuple for $[b_{i,t}, b_{i,t}+t-1]$ (i.e., $(SA[b_{i,t}], SA[b_{i,t}+t-1], \min(\text{LCP}[b_{i,t}+1..b_{i,t}+t-1]), \text{LCP}[b_{i,t}], \text{LCP}[b_{i,t}+t])$, respectively. The algorithm repeats the following two steps until interval $[b_{i,t}, b_{i,t}+t-1]$ contains position $\ell_T^i$ or $\ell_T^i-1$: (i) It verifies whether interval $[b_{i,t}, b_{i,t}+t-1]$ is an lcp-interval in constant time by using integers $\text{LCP}[b_{i,t}], \text{LCP}[b_{i,t}+t]$, and $\min(\text{LCP}[b_{i,t}+1..b_{i,t}+t-1])$ stored in $S'[i]$ (Corollary 9). Then, it outputs the interval $[b_{i,t}, b_{i,t}+t-1]$ as a type-3 lcp-interval with an EPS only if it satisfies that condition. (ii) It replaces the interval $[b_{i,t}, b_{i,t}+t-1]$ stored in $S[i]$ with interval $[b_{i,t+1}, b_{i,t+1}+t]$, and it also replaces the $5$-tuple for interval $[b_{i,t}, b_{i,t}+t-1]$ stored in $S'[i]$ with the $5$-tuple for interval $[b_{i,t+1}, b_{i,t+1}+t]$ in $O(\log \log n(n/r))$ time by using the next- and previous-access queries.
The running time of our enumeration algorithm for type-3 lcp-intervals with EPSs containing position $t_k^+$ is $O((|L_{k-1}| + |L_k|) \log \log_w (n/r))$. Hence, the total running time of the whole algorithm is $O(n \log \log_w (n/r))$, because $O(|L_0| + |L_1| + \cdots + |L_{r+1}|) = O(n)$ for $|L_0| = 0$ and $|L_{r+1}| = 0$. The four data structures, $Z$, $S$, $S'$, and the RLESA, can be constructed in $O(n \log \log_w (n/r))$ time and $O(r)$ words of working space by processing the RLBWT of a string $T$. This is because the RLESA consists of $O(r)$ lcp-values and sa-values for the starting and ending positions of L-runs and two predecessor data structures on $O(r)$ sa-values, and these sa-values and lcp-values can be enumerated using the data structure $Z$ and the algorithm presented in Section 3. More details of the RLESA and the construction algorithm for the four data structures are presented in the Appendixes. Here, we have the following lemma.

Lemma 11. We can construct a RLESA for a string $T$ in $O(n \log \log_w (n/r))$ time and $O(r)$ words of working space by processing the RLBWT of $T$.

Proof. See Appendix D.

4.2 Enumeration algorithm of type-1 lcp-intervals with EPS

This algorithm finds all the type-1 lcp-intervals by leveraging the incremental relation on the LCP array and using data structure $Z$ in Lemma 1. Consider a type-1 lcp-interval $I = [b, c, d]$. Then $\langle LF(b), LF(e), d + 1 \rangle$ is also an lcp-interval and interval $[LF(b) - 1, LF(e) + 1]$ is contained in an F-run by the following three observations: (i) LCP$[b]$, LCP$[e + 1] < d - 1$ and min(LCP$[b, e + 1]) = d$ since $I$ is an lcp-interval. (ii) LCP$[LF(x)] = LCP[x] + 1$ for any $x \in [b, e + 1]$ by the incremental relation on the LCP array because interval $[b - 1, e + 1]$ is contained in an L-run. (iii) LCP$[LF(b)]$, LCP$[LF(e + 1)] < d$ and min(LCP$[LF(b), LF(e + 1)]) = d + 1$ by the above observations (i) and (ii), and $\langle LF(b), LF(e), d + 1 \rangle$ is an lcp-interval by Corollary 3. Thus, any type-1 lcp-interval $[b, c, d]$ can be found using an lcp-interval $\langle LF(b), LF(e), d + 1 \rangle$ and the FL function.

The enumeration algorithm computes type-1 lcp-intervals by recursively applying the FL function to lcp-intervals of the other types (i.e., type-2, type-3, and type-4 lcp-intervals). Formally, let $T_1$, $T_2$, $T_3$, and $T_4$ be the sets of type-1, type-2, type-3, and type-4 lcp-intervals, respectively. Let track($I$) be the set of type-1 lcp-intervals obtained by recursively applying the FL function to a given lcp-interval $I = [b, c, d]$. That is, track($I$) = $\{\langle LF(b), LF(e), d - 1 \rangle \} \cup \text{track}(\langle LF(b), LF(e), d - 1 \rangle)$ if $[b, e + 1] \cap F^+ = \emptyset$; otherwise, track($I$) = $\emptyset$. Then, the following lemma holds.

Lemma 12. $T_1 = \bigcup_{I \in \mathcal{I}} \text{track}(I)$.

The enumeration algorithm processes type-2, type-3, and type-4 lcp-intervals with EPSs enumerated by the other three enumeration algorithms for lcp-intervals. Then, it computes set track($I$) for each type-2, type-3, or type-4 lcp-interval $I$ with an EPS by using the FL function, a predecessor query on $F^+$, and Lemma 12. The running time of the enumeration algorithm is $O(n \log \log_w (n/r))$, because it uses $O(n)$ FL functions and predecessor queries on $F^+$. The working space is $O(r)$ words, because the algorithm can process its input lcp-intervals in an online manner.

5 Practical enumeration algorithm for lcp-intervals in $O(r + K)$ words

Next, we present a practical enumeration algorithm for lcp-intervals with EPSs, which runs in $O(n \log \log_w (n/r))$ time and $O(|\text{RLESA}| + K)$ words of working space. Here, the term
|RLESA| = O(r) denotes the space usage of an RLESA supporting only next-access queries, which is smaller than that of an RLESA supporting both next- and previous-access queries. The term \( K = O(n) \) denotes the number of maximal elements in the stack data structure used in our algorithm, such that \( K \) is smaller than \( r \), which happens in practice. Thus, the space usage of the algorithm is smaller than that of the enumeration algorithm presented in Section 4. Our algorithm leverages Kasai et al.’s algorithm that can enumerate lcp-intervals by reading the LCP array of \( T \) in left-to-right order. First, we introduce Kasai et al.’s enumeration algorithm running in \( O(n) \) time and \( O(n+K) \) words of working space. Then, we present our enumeration algorithm running in \( O(n \log \log w \cdot (n/r)) \) time and \( O(|RLESA| + K) \) words of working space by replacing the LCP array with the RLESA supporting only next-access queries.

Kasai et al.’s algorithm is built on the LCP array and a stack \( X \). Let \( \Upsilon_i \) be a set of lcp-intervals such that (i) the left and right boundaries of each lcp-interval \( \langle i, e, d \rangle \) are contained in intervals \( [i, i] \) and \( [i, n] \), respectively (i.e., \( b \in [i, i] \) and \( e \in [i, n] \)); and (ii) the smallest lcp-value is contained in \( LCP[1..i] \) (i.e., \( \Upsilon_i = \{ \langle b, e, d \rangle \mid \langle b, e, d \rangle \in I \mathrm{ s.t. } b \in [i, i], e \in [i, n] \mathrm{ and } \mathrm{Rmq}(LCP, b + 1, e) \in [i, i] \cup \{0, n, 0\} \})

Their algorithm enumerates all the lcp-intervals in increasing order of their right boundaries by using sets of lcp-intervals, \( \Upsilon_1, \Upsilon_2, \ldots, \Upsilon_n \). For simplicity, the lcp-intervals in each set \( \Upsilon_i \) are sorted in decreasing order of their depths. These sets \( \Upsilon_1, \Upsilon_2, \ldots, \Upsilon_n \) have the following four properties: (i) Set \( \Upsilon_i \) contains all the lcp-intervals with right boundary \( i \), which correspond to the first lcp-intervals with a depth of at least \( LCP[i+1] + 1 \) in set \( \Upsilon_i \). (ii) Set \( \Upsilon_i \) consists of at most one lcp-interval \( I_i \) and the lcp-intervals in set \( \Upsilon_{i-1} \), except for lcp-intervals with right boundary \( i-1 \). (iii) An lcp-interval \( I_i \) exists if and only if \( \mathrm{Lcp}[i] > d_{top} \), where \( d_{top} \) is the depth of the first lcp-interval in set \( \Upsilon_{i-1} \), except for lcp-intervals with right boundary \( i-1 \). (iv) The left boundary and depth of lcp-interval \( I_i \) are \( b_{last} \) and \( \mathrm{Lcp}[i] \), respectively, if \( I_i \) exists. Here, \( b_{last} \) is the left boundary of the last lcp-interval with right boundary \( i-1 \) in set \( \Upsilon_{i-1} \) if there exists at least one lcp-interval with right boundary \( i-1 \); otherwise, \( b_{last} = i-1 \). These four properties indicate that we can compute lcp-intervals with right boundary \( i-1 \), and the lcp-intervals in \( \Upsilon_i \) except for right boundaries, by using lcp-value \( \mathrm{Lcp}[i] \) and the lcp-intervals in \( \Upsilon_{i-1} \) (except for right boundaries). Therefore, stack \( X \) stores the pair \( (b, d) \) of each lcp-interval \( \langle b, e, d \rangle \) in \( \Upsilon_{i-1} \) before reading lcp-value \( \mathrm{Lcp}[i] \). Their algorithm computes the lcp-intervals with right boundary \( i-1 \) and updates stack \( X \) by using the stack and the lcp-value \( \mathrm{Lcp}[i] \).

The details of their enumeration algorithm are as follows. Before it computes lcp-intervals with right boundary \( i \), stack \( X \) stores a pair \( (b, d) \) of the left boundary and depth of each lcp-interval \( \langle b, e, d \rangle \) in \( \Upsilon_i \) in decreasing order of their depths. The algorithm then executes the following steps to compute lcp-intervals with right boundary \( i \): (i) It reads \( \mathrm{Lcp}[i+1] \). (ii) It removes pairs for lcp-intervals with right boundary \( i \) from stack \( X \) and outputs the lcp-intervals by using pop operations. (iii) It computes a pair of the left boundary and depth of a new lcp-interval \( I_{i+1} \), and it pushes the pair onto stack \( X \) if \( I_{i+1} \) exists. The number of push and pop operations for stack \( X \) is \( O(n) \) in total, and the running time of the algorithm is thus \( O(n) \). More details of their algorithm are presented in the Appendix F.

As described above, we can replace the LCP array used in their algorithm with an RLESA supporting only next-access queries, because their algorithm only reads the LCP array in left-to-right order. We can also output the EPS of each lcp-interval by using an RLESA, because it enables us to read an SA in left-to-right order. Therefore, we can enumerate lcp-intervals with EPSs in \( O(n \log \log w \cdot (n/r)) \) time and \( O(r + K) \) words of working space. Formally, we obtain the following theorem.
Theorem 13. We can enumerate lcp-intervals with EPSs in $O(n \log \log_w (n/r))$ time and $O(r + K)$ words of working space by preprocessing the RLBWT of a string $T$, where $K = \max \{ \text{nest}(1), \text{nest}(2), \ldots, \text{nest}(n) \}$ for the number nest$(i)$ of lcp-intervals containing position $i$, i.e., nest$(i) = |\{(b,e,\ell) \mid (b,e,\ell) \in I \text{ s.t. } i \in [b,e]\}|$.

Proof. See Appendix F.

 Enumeration of the sa-intervals of maximal repeats with EPSs in $O(r)$ words

A maximal repeat in a string $T$ has the following properties: a substring $P$ is a maximal repeat if and only if (i) an sa-interval $\langle b, e, |P| \rangle$ of $P$ is an lcp-interval, and (ii) the sa-interval contains at least two distinct characters in the BWT $L$ of $T$, i.e., $[b + 1, e] \cap L^+ \neq \emptyset$ (e.g., [10], see Appendix G). These properties indicate that the sa-interval of each maximal repeat is an lcp-interval. Thus, the algorithm enumerates the sa-interval of each maximal repeat from the lcp-intervals with EPSs enumerated by the algorithm presented in Section 4.

The details of our enumeration algorithm for the sa-intervals of maximal repeats is as follows. It processes each lcp-interval with an EPS that is enumerated by the algorithm presented in Section 4. It then verifies whether each lcp-interval with an EPS contains at least two distinct characters in permutation $L$ by using one predecessor query supported by data structure $Z$ on set $L^+$. Next, it outputs the lcp-interval with an EPS as the lcp-interval of a maximal repeat with an EPS if the lcp-interval contains at least two distinct characters. Finally, the algorithm obtains the lcp-intervals of all maximal repeats with EPSs by verifying all lcp-intervals. The running time is $O(n \log \log_w (n/r))$, except for the executed enumeration algorithm for lcp-intervals, because the algorithm uses $O(n)$ predecessor queries on $F^+$.

The maximal repeat $P$ in an sa-interval $\langle b, e, |P| \rangle$ with an EPS is recovered by using data structure $Z$ (Lemma 1). The sa-interval has the following property: Let $b_1 = b$ and $b_i = \text{FL}(b_{i-1})$ for any integer $i \in [2, |P|]$. Then permutation $F$ stores the characters of $P$ in positions $b_1, b_2, \ldots, b_{|P|}$ (i.e., $P = F[b_1], F[b_2], \ldots, F[b_{|P|}]$), because suffix $T\langle b, \ldots, n \rangle$ has $P$ as a prefix and the FL function returns integer $i'$ such that $SA[i'] = SA[i] + 1$ for any integer $i \in [1, n - 1]$. The property indicates that we can compute the maximal repeat $P$ by recursively applying the FL function $|P| - 1$ times to position $b$ and accessing $|P|$ characters $F[b_1], F[b_2], \ldots, F[b_{|P|}]$. The computation time is $O(|P| \log \log_w (n/r))$ by using data structure $Z$.

The occurrence positions of maximal repeat $P$ in $T$ are recovered from sa-interval $\langle b, e, |P| \rangle$ with an EPS by using an RLESA. $SA[b..e]$ stores all the occurrence positions of maximal repeat $P$ in string $T$. The sa-interval with an EPS has sa-value $SA[b]$. We then compute $SA[b+1..e]$ by recursively applying a next-access query $e - b$ times to sa-value $SA[b]$. Thus, the computation time is $O((e - b + 1) \log \log_w (n/r))$ in total by using the RLESA.

Finally, we obtain the following theorem.

Theorem 14. We can enumerate the sa-intervals of maximal repeats with EPSs in a string $T$ in $O(n \log \log_w (n/r))$ time and $O(r)$ words of working space by processing the RLBWT of $T$. We can also recover a string $P$ and its occurrences in string $T$ from the sa-interval with the sa-value of $P$ in $O(|P| \log \log_w (n/r))$ time and $O(k \log \log_w (n/r))$ time by using data structure $Z$ or an RLESA, respectively, where $k$ is the number of occurrences of string $P$ in string $T$. 

7 Construction of succinct PLCP array in $O(r)$ words

The PLCP array $PLCP$ of string $T$ is the permutation of the LCP array of $T$ in text order (i.e., $PLCP[SA[i]] = LCP[i]$ for integer $i \in [1, n]$). Moreover, the succinct PLCP array $PLCP_{succ}$ of $T$ is a bit array of size $2n$ representing the PLCP array, and the position of each $j$-th 1 is $2j + PLCP[j]$ in the bit array. The succinct PLCP array can support random access on the LCP array by combining it with an SA or compressed SA (see [22] for a more detailed description).

Our construction algorithm for the succinct PLCP array is built on (i) data structure $Z$, (ii) an array $V$ storing $r$ lcp-values $LCP[f_r^+], LCP[f_{r-1}^+], \ldots, LCP[f_1^+]$, and (iii) a pair of lcp-value $PLCP[n]$ and its corresponding position $x_n$ in the SA (i.e., $SA[x_n] = n$). Array $V$ is constructed in $O(n \log \log_w(n/r))$ time by using our enumeration algorithm for lcp-values that was presented in Section 3. Thus, these three data structures are constructed in $O(n \log \log_w(n/r))$ time in total.

Specifically, our construction algorithm for the succinct PLCP array outputs the array’s bits in right-to-left order by enumerating lcp-values $PLCP[n], PLCP[n-1], \ldots, PLCP[1]$. This is because any pair of adjacent lcp-values $PLCP[i]$ and $PLCP[i-1]$ indicates that $PLCP_{succ}[2(i-1) + PLCP[i-1], 2i + PLCP[i]] = 1, 0, 0, \ldots, 0, 0, 1$. The algorithm computes lcp-value $PLCP[i-1]$ and its corresponding position $x_{i-1}$ in the SA (i.e., $SA[x_{i-1}] = i - 1$) by using lcp-value $PLCP[i]$ and its corresponding position $x_i$ in the SA. If lcp-value $PLCP[i]$ has the incremental relation on the LCP array (i.e., $x_i \notin L^+$), then $LCP[x_{i-1}] = LCP[x_i] + 1$, i.e., $PLCP[i-1] = PLCP[i] + 1$; otherwise, $x_{i-1}$ is position $f_r^+$ in set $F^+$ such that $i' = \text{pred}(F^+, LF(x_i))$, and $V[i']$ stores $LCP[x_{i-1}]$, i.e., $V[i'] = PLCP[i-1]$. Thus, each lcp-value can be computed in $O(\log \log_w(n/r))$ time by using the LF function, a predecessor query on set $L^+$, and a predecessor query on set $F^+$. Our algorithm runs in $O(n \log \log_w(n/r))$ time in total. Formally, we obtain the following theorem.

▶ Theorem 15. We can output the bits organizing the succinct PLCP array of a string $T$ in right-to-left order in $O(n \log \log_w(n/r))$ time and $O(r)$ words of working space by preprocessing the RLBWT of $T$.

8 Experiments

8.1 Method

In this section, we demonstrate the effectiveness of our enumeration algorithm for maximal repeats on a benchmark dataset of highly repetitive texts. We used real, repetitive collections in the Pizza & Chili corpus downloadable from [http://pizzachili.dcc.uchile.cl](http://pizzachili.dcc.uchile.cl). We enumerated the lcp-intervals of the maximal repeats in a given string and used the memory consumption and complete execution time as evaluation measures. We implemented the enumeration algorithm by using the SDSL Library [15]. We performed all the experiments on one core of a quad-core Intel(R) Xeon(R) E5-2680 v2 (2.80 GHz) CPU with 256 GB of memory.

We compared our enumeration method with the enumeration method proposed by Okanohara and Tsujii (OT method) [30]. Our implementation constructed the RLBWT of a given string by applying the compression algorithm proposed by Ohno et al. [29], and it enumerated lcp-intervals by using the enumeration algorithm presented in Section 5. The enumeration algorithm of Okanohara and Tsujii constructs the LCP array of a given string via the SA, enumerates lcp-intervals by using Kasai et al.’s enumeration algorithm, and
detects the lcp-intervals of maximal repeats by using the BWT; hence, it runs in $O(n)$ time and $O(n)$ words of working space.

### 8.2 Results

Table 2 lists the execution time and memory consumption for our method and the OT method. The memory used by our method was smaller than that used by the OT method on all the benchmark strings. For a given benchmark string, the OT method used exactly 24.4 bytes per character, while our method used approximately 50-60 bytes per run in the string’s RLBWT. Therefore, the ratio of the memory use for our method to that for the OT method was proportional to the compression ratio of the benchmark strings (i.e., $n/r$). For example, the memory for our method was approximately 651 times smaller than that for the OT method on the file einstein.en.txt ($n/r \approx 1611$). As another example, the memory for our method was approximately 4 times smaller than that for the OT method on the file Escherichia Coli ($n/r \approx 7$).

On the other hand, the execution time for our method was longer than that for the OT method on all the benchmark strings. Our method and the OT method required about 0.83-4.14 and 0.23 microseconds per character, respectively, and our execution time per character was approximately inversely proportional to the compression ratio of the benchmark strings. For example, our method took 0.88 microseconds per character and was approximately 4 times slower than the OT method was for the file einstein.en.txt; likewise, our method took 4.14 microseconds per character and was approximately 16 times slower than the OT method was for the file Escherichia Coli.

In conclusion, our method was at most 16 times slower than the OT method in the worst case, while its memory use was 651 times smaller than that for the OT method in the best case. Therefore, our method is significantly better than the OT method for highly repetitive texts.

### References

1. Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004.
2. Hideo Bannai, Travis Gagie, and Tomohiro I. Online LZ77 parsing and matching statistics with RLBWTs. In *Proceedings of CPM*, pages 7:1–7:12, 2018.
Verónica Becher, Alejandro Deymonnaz, and Pablo Ariel Heiber. Efficient computation of all perfect repeats in genomic sequences of up to half a gigabyte, with a case study on the human genome. *Bioinformatics*, 25(14):1746–1753, 2009.

Djamal Belazzougui. Linear time construction of compressed text indices in compact space. In *Proceedings of STOC*, pages 148–193, 2014.

Djamal Belazzougui and Fabio Cunial. Space-efficient detection of unusual words. In *Proceedings of SPIRE*, pages 222–233, 2015.

Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In *Proceedings of CPM*, pages 26–39, 2015.

Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Versatile succinct representations of the bidirectional burrows-wheeler transform. In *Proceedings of ESA*, pages 133–144, 2013.

Djamal Belazzougui and Gonzalo Navarro. New lower and upper bounds for representing sequences. In *Proceedings of ESA*, pages 181–192, 2012.

Djamal Belazzougui, Gonzalo Navarro, and Daniel Valenzuela. Improved compressed indexes for full-text document retrieval. *J. Discrete Algorithms*, 18:3–13, 2013.

Timo Beller, Katharina Berger, and Enno Ohlebusch. Space-efficient computation of maximal and supermaximal repeats in genome sequences. In *Proceedings of SPIRE*, pages 99–110, 2012.

Timo Beller, Simon Gog, Enno Ohlebusch, and Thomas Schnattinger. Computing the longest common prefix array based on the burrows-wheeler transform. *J. Discrete Algorithms*, 18:22–31, 2013.

Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. *Technical report*, 1994.

Maxime Crochemore and Renaud Vérin. On compact directed acyclic word graphs. In *Structures in Logic and Computer Science, A Selection of Essays in Honor of Andrzej Ehrenfeucht*, pages 192–211, 1997.

Herbert Edelsbrunner. *Dynamic data structures for orthogonal intersection queries*. Universität Graz/Forschungszentrum Graz. Institutfür Informationsverarbeitung, 1980.

Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.

Isamu Furuya, Takuya Takagi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Takuya Kida. MR-RePair: Grammar compression based on maximal repeats. In *Proceedings of DCC*, pages 508–517, 2019.

Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1), 2020.

Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

Roberto Grossi, Alessio Orlandi, and Rajeev Raman. Optimal trade-offs for succinct string indexes. In *Proceedings of ICALP*, pages 678–689, 2010.

Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In *Proceedings of CPM*, pages 181–192, 2009.

Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of CPM*, pages 181–192, 2001.

Dominik Kempa. Optimal construction of compressed indexes for highly repetitive texts. In *Proceedings of SODA*, pages 1344–1357, 2019.
25 Dominik Kempa and Tomasz Kociumaka. Resolution of the burrows-wheeler transform conjecture. CoRR, abs/1910.10631, 2019.

26 M. Oguzhan Külekci, Jeffrey Scott Vitter, and Bojian Xu. Efficient maximal repeat finding using the burrows-wheeler transform and wavelet tree. IEEE/ACM Trans. Comput. Biology Bioinform., 9(2):421–429, 2012.

27 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. SIAM J. Comput., 22(5):935–948, 1993.

28 Tomonari Masada, Atsuhiro Takasu, Yuichiro Shibata, and Kiyoshi Oguri. Clustering documents with maximal substrings. In Proceedings of ICEIS, pages 19–34, 2011.

29 Tatsuya Ohno, Kenson Sakai, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. A faster implementation of online RLBWT and its application to LZ77 parsing. J. Discrete Algorithms, 52-53:18–28, 2018.

30 Daisuke Okanohara and Jun’ichi Tsujii. Text categorization with all substring features. In Proceedings of the SDM, pages 838–846, 2009.

31 Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. Algorithmica, 80(7):1986–2011, 2018.

32 Nicola Prezza and Giovanna Rosone. Space-efficient construction of compressed suffix trees. CoRR, abs/1908.04686, 2019.

33 Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In Proceedings of SODA, pages 225–232, 2002.

34 Kunihiko Sadakane. Compressed suffix trees with full functionality. Theory Comput. Syst., 41(4):589–607, 2007.
Appendix A: Proofs for Section 2

Details of semi-dynamic interval tree. A semi-dynamic interval tree $\Gamma$ for a set $S$ of intervals is a recursive data structure. It consists of the following four data structures: (i) The left boundary $b_{\text{half}}$ of interval $[b_{\text{half}}, e_{\text{half}}]$, where $[b_{\text{half}}, e_{\text{half}}]$ is the $\lceil |S|/2 \rceil$-th interval among the intervals sorted in increasing order of their left boundaries in set $S$. (ii) A semi-dynamic interval tree $\Gamma_L$ for set $S_L = \{[b, e] \mid [b, e] \in S \text{ s.t. } e < b_{\text{half}}\}$ if $S_L \neq \emptyset$. (iii) A semi-dynamic interval tree $\Gamma_R$ for set $S_R = \{[b, e] \mid [b, e] \in S \text{ s.t. } b_{\text{half}} < b\}$ if $S_R \neq \emptyset$. (iv) Two doubly linked lists $L$ and $R$ storing the intervals of set $S_C = \{[b, e] \mid [b, e] \in S \text{ s.t. } b_{\text{half}} \in [b, e]\}$. The intervals in $L$ are sorted in increasing order of their left boundaries, and each cell storing an interval $[b, e]$ also stores a pointer to the cell storing $[b, e]$ in $R$. Similarly, the intervals in $R$ are sorted in decreasing order of their right boundaries, and each cell storing an interval $[b, e]$ also stores a pointer to the cell storing $[b, e]$ in $L$. Set $S$ is equal to the union of $S_L$, $S_C$, and $S_R$, and hence, any interval in $S$ is stored in the two doubly linked lists of a semi-dynamic interval tree stored in $\Gamma$. The recursion depth of $\Gamma$ is at most $\log |S|$, because $|S_L|, |S_R| \leq |S|/2$. The space usage of the tree is $O(|S|)$ words, because the doubly linked lists $L$ and $R$ use $O(|S_C|)$ words and $|S_C| \geq 1$.

The algorithms of the report query $\text{report}(S, x)$ and delete query $\text{delete}(S, x)$ use the following two observations: (i) Let $S'_C$ be the intervals containing position $x$ in set $S_C$. If $x \leq b_{\text{half}}$, then only the first $|S'_C|$ intervals in $\hat{L}$ contain position $x$, because (a) the intervals in $\hat{L}$ are sorted in increasing order of their left boundaries and (b) any interval in $S_C$ contains position $b_{\text{half}}$. Otherwise, only the first $|S'_C|$ intervals in $\hat{R}$ contain position $x$. (ii) If $x \leq b_{\text{half}}$, then any interval in set $S_R$ does not contain position $x$, because the left boundary of any interval in $S_R$ is larger than $b_{\text{half}}$. Otherwise, any interval in set $S_L$ does not contain position $x$, because the right boundary of any interval in $S_L$ is smaller than $b_{\text{half}}$.

The algorithm of the report query $\text{report}(S, x)$ consists of the following two steps: (i) If $x \leq b_{\text{half}}$, then we sequentially read linked list $\hat{L}$ in left-to-right order until the obtained interval does not contain position $x$, and we output the first $|S'_C|$ intervals as the intervals in set $S'_C$. Otherwise, we sequentially read linked list $\hat{R}$ in left-to-right order until the obtained interval does not contain position $x$, and we output the first $|S'_C|$ intervals as the intervals in set $S'_C$. (ii) If $x \leq b_{\text{half}}$, then we execute steps (i) and (ii) by using the semi-dynamic interval tree $\Gamma_L$. Otherwise, we execute steps (i) and (ii) by using $\Gamma_R$. The algorithm obtains all the intervals containing position $x$ in set $S$. The running time is $O((1 + k) \log |S|)$, where $k$ is the number of intervals containing position $x$ in set $S$.

The algorithm of the delete query $\text{delete}(S, x)$ consists of the following two steps: (i) We obtain the cells of the intervals containing position $x$ by using $\text{report}(S, x)$. (ii) We remove each reported cell $z$ from the linked list storing the cell, and we also remove cell $\text{pointer}(z)$ from the linked list storing cell $\text{pointer}(z)$, where $\text{pointer}(z)$ is the cell associated with the pointer stored in cell $z$. Therefore, the intervals containing position $x$ are removed from $\Gamma$, and the running time is $O((1 + k) \log |S|)$.

The construction algorithm for the semi-dynamic interval tree $\Gamma$ of set $S$ is as follows. (i) We construct two doubly linked lists $L_S$ and $R_S$ storing the intervals in set $S$ by sorting the intervals. The intervals stored in $L_S$ are sorted in increasing order of their left boundaries. Similarly, the intervals stored in $R_S$ are sorted in decreasing order of their right boundaries. (ii) We find interval $[b_{\text{half}}, e_{\text{half}}]$ by reading list $L_S$ in left-to-right order. (iii) We construct lists $\hat{L}$ and $\hat{R}$ by reading lists $L_S$ and $R_S$, respectively. (iv) We remove the intervals in set $S_C$ from lists $L_S$ and $R_S$. (v) We construct semi-dynamic interval trees $\Gamma_L$ and $\Gamma_R$ by using lists $L_S$ and $R_S$, i.e., we repeat steps (ii-v) by using $L_S$ and $R_S$.

Next, we analyze the running time of the construction algorithm. The first step takes
Enumeration of maximal repeats in BWT-runs Bounded Space

Let $\Gamma_d$ be the set of semi-dynamic trees generated by the recursive construction algorithm instances with depth $d$, and let $\text{Set}(\Gamma')$ be the set of intervals stored in a semi-dynamic interval tree $\Gamma'$ (i.e., the semi-dynamic interval tree can support report and delete queries on $\text{Set}(\Gamma')$). Then, the data structures of each semi-dynamic interval tree $\Gamma'$ in $\Gamma_d$ are constructed in $O(|\text{Set}(\Gamma')|)$ time, except for the two trees $\Gamma_L$ and $\Gamma_R$. The semi-dynamic interval trees in $\Gamma_d$ store distinct intervals with respect to each other, i.e., $S \supseteq \bigcup_{\Gamma' \in \Gamma_d} \text{Set}(\Gamma')$. This indicates that we can construct the semi-dynamic interval trees in $\Gamma_d$ in $O(|S|)$ time, except for the left and right semi-dynamic interval trees. Therefore, the construction time is $O(|S| \log |S|)$ in total, because the recursion depth of semi-dynamic interval tree $\Gamma$ is at most $\log |S|$.

Details of Section 2.5 Formally, we show the adjacent relation on SA and the incremental relation on the LCP array. Let $p'_1, p'_2, \ldots, p'_i$ be the permutation of $[1,r]$ that represents the lexicographical order of the first characters of L-runs, i.e., for any integer $i \in [1, r - 1]$, $L[\ell^+_i] < L[\ell^+_i']$ if $L[\ell^+_i'] \neq L[\ell^+_i + 1]$; otherwise, $p'_i < p'_{i+1}$. For example, $p'_1, p'_2, p'_3, p'_4 = 3, 2, 4, 1$ in Figure 1 since $L[\ell^+_1], L[\ell^+_2], L[\ell^+_2], L[\ell^+_3] = b, a, S, a$. Then, the following lemmas hold.

\textbf{Lemma 16.} For two integers $x, y \in [1,n]$, $\text{LF}(x) < \text{LF}(y)$ if and only if (a) $L[x] < L[y]$, or (b) $L[x] = L[y]$ and $x < y$.

\textbf{Proof.} Consider two suffixes $S_{\text{LF}(x)}$ and $S_{\text{LF}(y)}$, where $S_i$ is the suffix $T[\text{SA}[i]..n]$. $S_{\text{LF}(x)} < S_{\text{LF}(y)}$ if and only if $L[x] < L[y]$ for two integers $x$ and $y$ such that $L[x] \neq L[y]$, because $S_{\text{LF}(x)}[1] = L[x] \neq L[y] = S_{\text{LF}(y)}[1]$. Next, $S_{\text{LF}(x)} < S_{\text{LF}(y)}$ if and only if $x < y$ for two integers $x$ and $y$ such that $L[x] = L[y]$, because (a) $S_{\text{LF}(x)}[1] = L[x] = L[y] = S_{\text{LF}(y)}[1]$, (b) $S_{\text{LF}(x)}[2..|S_{\text{LF}(x)}|] = S_x$ and $S_{\text{LF}(y)}[2..|S_{\text{LF}(y)}|] = S_y$, and (c) $S_x < S_y$ by $x < y$. ▶

\textbf{Lemma 17.} (i) $\text{LF}(\ell^+_i) = 1$ and $\text{LF}(\ell^+_i) = \text{LF}(\ell^+_i) + |L_{p'_i} - 1|$ for any integer $i \in [2,r]$.

\textbf{Proof.} $\text{LF}(\ell^+_i) = 1 < \text{LF}(\ell^+_i + 1) = \ldots < \text{LF}(\ell^+_i + |L_{p'_i} - 1|) < \text{LF}(\ell^+_i + |L_{p'_i} + 1|) < \ldots < \text{LF}(\ell^+_i + 1) < \text{LF}(\ell^+_i + |L_{p'_i} - 1|) < \ldots < \text{LF}(\ell^+_i + |L_{p'_i} + 1|)$ holds by Lemma 16. Therefore, we obtain Lemma 17. ▶

\textbf{Lemma 18.} $LCP[\text{LF}(x)] = LCP[x] + 1$ for any integer $x \not\in L^+$. $LCP[\text{LF}(x)]$ is the length of the LCP of two suffixes $S_{\text{LF}(x)-1}$ and $S_{\text{LF}(x)}$. $S_{\text{LF}(x)}[1] = S_{\text{LF}(x)-1}[1]$ because $x \not\in L^+$. The length of the LCP of $S_{\text{LF}(x)[2..n]}$ and $S_{\text{LF}(x-1)[2..n]}$ is equal to the lcp-value $LCP[x]$, and hence, Lemma 18 holds. ▶

\textbf{Proof of Lemma 1 (data structure Z).} Data structure $Z$ consists of the following four data structures: (i) Two sequences of integers, $F(f_i^+), F(f_i^+), \ldots, F(f_i^+)$, and $\text{LF}(f_i^+), \text{LF}(f_i^+), \ldots, \text{LF}(f_i^+)$. (ii) Two sequences of integers, $f_i^+, f_i^+, \ldots, f_i^+, \ldots, f_i^+, f_i^+, \ldots, f_i^+$. (iii) Two sequences of characters, $f[f_i^+], f[f_i^+], \ldots, f[f_i^+], \ldots, f[f_i^+], L[\ell^+_i], L[\ell^+_i], \ldots, L[\ell^+_i]$. (iv) Two predecessor data structures for sets $L^+$ and $F^+$. The space usage of $Z$ is $O(r)$ words in total.

Next, we show that $Z$ can support the queries described in Lemma 1, i.e., computing (i) $\text{pred}(F^+), \text{pred}(L^+)$, (ii) $f_i^+, \ell_i^+$, (iii) $F[i], L[i]$, and (iv) $\text{LF}(i), \text{FL}(i)$ for a given integer $i$. Obviously, $Z$ can support the queries $\text{pred}(F^+, i)$ and $\text{pred}(L^+, i)$ in $O(\log \log \log n)$ time using the predicate data structures. It can also support accessing $f_i^+$ and $\ell_i^+$ in $O(1)$ time. It can compute $F[i]$ and $L[i]$ in $O(\log \log \log n)$ time for any integer $i \in [1,n]$, because $F[i] = F[f_{\text{pred}(F^+,i)]}$ and $L[i] = L[\ell_{\text{pred}(L^+,i)]}$. Finally, it can compute $\text{LF}(i)$ and
Lemma 19. Assume that \( \sigma = O(n) \). Then, we can construct string \( L_{\text{shrunken}}' \) in \( O(n) \) time and \( O(r) \) words of working space by processing string \( L' \).

Proof. We consider the following two cases: (i) \( r \leq n/\log n \), and (ii) \( r > n/\log n \).

Case (i): The construction algorithm uses two arrays \( E \) and \( E' \) of length \( r \). Array \( E \) stores the pair \((L'[i],i)\) for each \( i \)-th character of \( L'[i] \), and the pairs are sorted in increasing order of their first elements. \( E'[i] \) stores the rank of the character stored in \( E[i] \) for any integer \( i \in [1,r] \), i.e., \( E'[i] = \text{pred}(\text{colors}(L'),L'[i]) \), where \( s_i \) is the second element in pair \( E[i] \), \( L_{\text{shrunken}}'[s_i] = E'[i] \) for any integer \( i \in [1,r] \), and hence, we can construct string \( L_{\text{shrunken}}' \) in \( O(r) \) time by processing the two arrays \( E \) and \( E' \). We construct array \( E \) in \( O(r \log r) \) time with a standard sorting algorithm. We can construct array \( E' \) in \( O(r) \) time by processing array \( E \), because \( E[i] = E[i-1] + 1 \) for any integer \( i \in [2,r] \) if \( L'[i] \neq L'[i-1] \); otherwise, \( E[i] = E[i-1] \). Therefore, we can construct string \( L_{\text{shrunken}}' \) in \( O(r \log r) = O(n) \) time and \( O(r) \) words.

Case (ii): We use (1) a bit array \( V \) of length \( \sigma \) such that \( V[c] = 1 \) if and only if if character \( c \) occurs in \( L' \) for any character \( c \in \Sigma \), and (2) a data structure \( H \) of \( O(|V|) \) bits supporting a rank query on bit array \( V \) in constant time [19], where a rank query \( \text{rank}_1(V,i) \) returns the number of 1s in bit array \( V[1..i] \) for any integer \( i \in [1,|V|] \). The space usage of \( V \) is \( O(r \log n) \) bits, and we can construct bit array \( V \) in \( O(n) \) time by reading \( L' \) in left-to-right order, because \( r \geq n/\log n \) and \( \sigma = O(n) \). We can construct data structure \( H \) in \( O(|V|) \) time and bits of working space by processing bit array \( V \) [19]. For any character \( c \), \( \text{rank}_1(V,c) \) represents the rank of the character in string \( L' \), and hence, we can construct string \( L_{\text{shrunken}}' \) in \( O(n) \) time by using a rank query, like in the construction algorithm for case (i). Therefore, Lemma [19] holds.

We also use the following relations to construct the data structure \( Z \): (i) \( \text{LF}(\ell^+_r) = 1 \) and \( \text{LF}(\ell^+_p) = \text{LF}(\ell^+_p) + |L_{p'_{i+1}}| \) for any integer \( i \in [2,r] \) (Lemma [17]) and (ii) \( F_i = L_{p'_{i}} \) for any integer \( i \in [1,r] \).

The construction algorithm of \( Z \) for the given RLBWT of \( T \) and string \( L_{\text{shrunken}}' \) consists of the following steps: (i) We construct two sequences \( \ell^+_1, \ell^+_2, \ldots, \ell^+_r \) and \( L[\ell^+_1], L[\ell^+_2], \ldots, L[\ell^+_r] \) in \( O(r) \) time by processing the RLBWT of \( T \). (ii) We construct permutation \( p'_1, p'_2, \ldots, p'_{r} \) by sorting \( L_{\text{shrunken}}' \) with bucket sort. The construction time and working space are \( O(r) \) and \( O(r) \), respectively, because the alphabet size of the shrunk string of \( L' \) is at most \( r \). (iii) We construct sequences \( f^+_1, f^+_2, \ldots, f^+_r \), \( F[f^+_1], F[f^+_2], \ldots, F[f^+_r], F(f^+_1), F(f^+_2), \ldots, F(f^+_r) \), and \( \text{LF}(f^+_1), \text{LF}(f^+_2), \ldots, \text{LF}(f^+_r) \) in \( O(r) \) time and words by using Lemma [17] and permutation \( p'_1, p'_2, \ldots, p'_{r} \). (iv) We construct two predecessor data structures for sets \( L' \) and \( F' \) in \( O(r \log \log_2(n/r)) \) time by using two sequences \( \ell^+_1, \ell^+_2, \ldots, \ell^+_r \) and \( f^+_1, f^+_2, \ldots, f^+_r \). Finally, we can construct data structure \( Z \) in \( O(n + r \log \log_2(n/r)) \) time and \( O(r) \) words of working space, and hence, we obtain Lemma [1].
Appendix B: Proofs for Section 3.1

Proof of Lemma 3(i). We prove Lemma 3(i) by using the following observation: For any two integers $i, j \in [1, n]$, $i = j$ if and only if $FL_{FLDist(i)}(i) = FL_{FLDist(j)}(j)$ and $LCP[i] = LCP[j]$. Here, the function $FL_d(i)$ is a recursive FL function applied $d$ times to a given position $i$ for any integer $i \in [1, n]$, i.e., $FL_d(i) = FL_{d-1}(FL(i))$ if $d \geq 1$; otherwise, $FL_d(i) = i$. The function $FL_{FLDist}(i)$ is the smallest positive integer $d$ such that $FL_d(i) \in F^+$. The above observation holds because (i) $LCP[FL_d(i)] = LCP[i] - d$ for any integers $i \in [1, n]$ and $d \in [0, FL_{Dist}(i)]$ by the incremental relation on the LCP array; and (ii) the positions in permutation $F$ correspond on a one-to-one basis to the positions in permutat $L$ by the FL function. The above observation means that each position with lcp-value $p$ corresponds to a distinct position in $F^+$, and hence, the number of positions with lcp-value $p$ (i.e., $\Phi_p \cup \Pi_p$) is at most $r$ for any integer $p \in [0, n]$. Therefore, we obtain Lemma 3(i).

Proof of Lemma 3(ii). The left boundary of any FL-interval $[FL(f_j^+ - 1) + 1, FL(f_j^+)]$ is the starting position of an L-run (i.e., $FL(f_j^+ - 1) + 1 \in L^+$), because $FL(f_j^+ - 1)$ is the ending position of an L-run by Lemma 17 (i.e., $FL(f_j^+ - 1) \in \{\ell_1, \ell_2, \ldots, \ell_r\}$). Similarly, the right boundary of any FL-interval is the starting position of an L-run, and hence, for any interval $FL(f_j^+ - 1) + 1, FL(f_j^+)]$, there exist two integers $x$ and $y$ such that $[\ell_x^+, \ell_y^+)$ has a unique integer $i \in [1, n]$. The left boundary of any FL-interval $LCP[Dist(i)] = FLDist(j) + 1$, $FL(f_j^+)$, and (ii) the positions in permutation $F$ correspond on a one-to-one basis to the positions in permutation $L$ by the FL function. The above observation means that each position with lcp-value $p$ corresponds to a distinct position in $F^+$, and hence, the number of positions with lcp-value $p$ (i.e., $\Phi_p \cup \Pi_p$) is at most $r$ for any integer $p \in [0, n]$. Therefore, we obtain Lemma 3(ii).

Proof of Lemma 2. We prove the following two statements for any position $x$ with lcp-value $p \geq 1$: (i) $x \in \Pi_p$ if and only if $x \not\in F^+$, and (ii) $x \in \Phi_p$ if and only if $x \in F^+$. We then obtain Lemma 2 by these two statements.

(i) Lemmas 17 and 18 indicate that the positions with lcp-value $p$ such that each position is contained in $F^+$ correspond on a one-to-one basis to the positions with lcp-value $p - 1$ such that each position is contained in $L^+$, i.e., $\{x \mid x \in [1, n] \text{ s.t. } LCP[x] = p \text{ and } x \in F^+\} = \{LF(x) \mid x \in [1, n] \text{ s.t. } x \not\in L^+ \text{ and } LCP[x] = p - 1\} = \Pi_p$. Therefore, statement (i) holds.

(ii) Consider a position $x$ with lcp-value $p \geq 1$ such that $x \in F^+$. The lcp-value for position $x$ is the length of the LCP of suffixes $S_x$ and $S_{x-1}$, where $S_t$ is the $t$-th suffix in the sorted suffixes corresponding to the SA (i.e., $S_t = T[SA[i]..n]$). Furthermore, $S_x = c_{FL(x)}$, $S_{x-1} = c_{FL(x-1)}$, and $S_{FL(x-1)} = S_{FL(x)}$ hold since the lcp-value is non-zero, where $c = F[x]$. The LCP of the next two suffixes $S_{FL(x)}$ and $S_{FL(x-1)}$ is the LCP among the suffixes $S_{FL(x-1)}$, $S_{FL(x-1)+1}, \ldots, S_{FL(x)}$, i.e., the length of the LCP is $\min(LCP[FL(x-1) + 1..FL(x)])$. Therefore, the lcp-value for position $x$ is 1 plus the smallest lcp-value in the FL-interval for position $x$, i.e., $p = 1 + \min(LCP[FL(x-1) + 1..FL(x)])$. On the other hand, recall that $\Phi_p$ is the set $\{x' \mid x' \in F^+ \text{ s.t. } LCP[x'] \geq 1 \text{ and } \min(LCP[FL(x') + 1..FL(x')]) = p - 1\}$. The set of the positions in $F^+$ with lcp-value $p \geq 1$ is equal to set $\Phi_p$, and hence the statement (ii) holds.

Details of our enumeration algorithm. The details of our enumeration algorithm are as follows. It processes each position $x$ with lcp-value $p - 1$ in the following two steps: (i) We verify that position $x$ is not contained in $L^+$ (i.e., $x \not\in L^+$) by using the predecessor query $pred(L^+, x)$. If $x \not\in L^+$, then we apply the LF function to the position $x$ and output the obtained position $LF(x)$ as the secondary position of lcp-value $p$. (ii) Next, we compute the interval $B_i$ containing the position $x$ by using arithmetic operations. If $V[i] = 0$, then we execute the report query for the given position $x$ with the semi-dynamic interval tree $\Gamma$ and output the primary positions of the reported FL-intervals. Afterward, we remove the reported FL-intervals from $\Gamma$ by a delete query for $x$ and set the bit $V[i]$ to 1. Steps (i) and (ii) output all secondary and primary positions with lcp-value $p$ by using the positions with lcp-value $p - 1$, because the semi-dynamic interval tree $\Gamma$ stores set $\Psi_{p-1}$. 
Construction algorithm for data structures. Recall that the enumeration algorithm is built on a data structure $Z$, a semi-dynamic interval tree $\Gamma$, a bit array $V$, an integer array $D$, and intervals $B_1, B_2, \ldots, B_{2r}$. The semi-dynamic interval tree $\Gamma$ stores a set $\Psi$ consisting of FL-intervals. The FL-intervals are $[\mathit{FL}(f_1^+ - 1) + 1, \mathit{FL}(f_1^+)]$, $[\mathit{FL}(f_2^+ - 1) + 1, \mathit{FL}(f_2^+)]$, $\ldots$, $[\mathit{FL}(f_r^+ - 1) + 1, \mathit{FL}(f_r^+)]$, and we can compute the intervals in $O(r \log \log_n(n/r))$ time by using data structure $Z$, because $Z$ stores the set $F^+$ and supports the FL function. Therefore, we can construct the semi-dynamic interval tree $\Gamma$ in $O(r \log n)$ time. Next, we can compute integer array $D$ by accessing $F[f_1^+ - 1], F[f_2^+], F[f_2^+ - 1], F[f_3^+], \ldots, F[f_r^+ - 1], F[f_r^+]$, because the array stores the positions with lcp-value 0 in increasing order, and the positions are a subset of $F^+$. The construction time for integer array $D$ is $O(r \log \log_n(n/r))$ by using data structure $Z$. We can also compute the intervals $B_1, B_2, \ldots, B_{2r}$ by using $Z$, because $Z$ stores set $L^+$. Finally, the data structure $Z$ can be constructed in $O(n \log \log_n(n/r))$ time by Lemma 1 and hence, the total construction time is $O(r \log n + n \log \log_n(n/r))$, and the working space is $O(r)$.

Appendix C.1: Details of data structure $\Lambda$ (for RCLFL queries)

Formally, let RCLFL($T, i, j$) be the set of triplets of each distinct character and its first and last occurrences in substring $T[i..j]$, i.e., RCLFL($T, i, j$) = \{$(c, b, e) \mid c \in \text{colors}(T[i..j])$\}, where $b_c$ and $e_c$ are the leftmost and rightmost occurrences of character $c$ in substring $T[i..j]$, respectively (i.e., $b_c = \min\{x \mid x \in [i, j] \text{ s.t. } T[x] = c\}$ and $e_c = \max\{x \mid x \in [i, j] \text{ s.t. } T[x] = c\}$), and \text{colors}($T$) = \{$T[i] \mid i \in [1, n]$\}. In this section, we prove the following lemma.

Lemma 20. For a string $T$ of length $n$ such that the alphabet size is $O(n)$, there exists a data structure $\Lambda$ of $O(r)$ words supporting any RCLFL query on string $L' = L[\ell_1^+, L[\ell_2^+, \ldots, L[\ell_r^+]$ of length $r$ in $O(k)$ time and $O(k)$ words of working space, where $k$ is the number of elements output by the RCLFL query, and $L$ is the BWT of $T$. We can construct the data structure in $O(n)$ time and $O(r)$ words of working space for a given $L'$.

Our algorithm for an RCLFL query is based on the algorithm for range color reporting with frequency query proposed by Belazzougui et al. [9]. We prove Lemma 20 by using the following lemma.

Lemma 21. For a string $T$, there exists a data structure $\Lambda'$ of $O(n + \sigma)$ words supporting any RCLFL query on a given string $T$ of length $n$ in $O(k)$ time and $O(k)$ words of working space, where $k$ is the number of elements output by the RCLFL query. We can construct the data structure in $O(n + \sigma)$ time and $O(n + \sigma)$ words of working space for a given $T$.

Proof of Lemma 21. Data structure $\Lambda'$ consists of (i) an array $W$ of size $\sigma$ and (ii) a data structure $X$ supporting range minimum queries (RmQ) on an array $\tilde{L}$ and range maximum queries (RMQ) on an array $\tilde{R}$. Every cell in array $W$ uses $\log n$ bits and is set to 0. The cells $\tilde{L}[i]$ and $\tilde{R}[i]$ store the previous and next occurrences, respectively, of the $i$-th character $T[i]$ in $T$ (i.e., $\tilde{L}[i] = \max\{\{x \mid x \in [1, i - 1] \text{ s.t. } T[x] = T[i]\} \cup \{0\}\}$ and $\tilde{R}[i] = \min\{\{x \mid x \in [i + 1, n] \text{ s.t. } T[x] = T[i]\} \cup \{n + 1\}\}$). The RmQ and RMQ of a given interval $[i, j]$ return the positions of the smallest value in $\tilde{L}[i..j]$ and the largest value in $\tilde{R}[i..j]$, respectively (i.e., $\text{RmQ}(T, i, j) = \min\{x \mid x \in [i, j] \text{ s.t. } T[x] = \min(T[b..e])\}$, and $\text{RMQ}(T, i, j) = \max\{x \mid x \in [i, j] \text{ s.t. } T[x] = \max(T[b..e])\}$). There exists a data structure of $O(|D|)$ words supporting RmQs and RMQs on an integer array $D$ in constant time, and it can be constructed in $O(|D|)$ time and space from the array $D$ (e.g., [13]). We use the data structure for $\tilde{L}$ and $\tilde{R}$ as data structure $X$, and we construct the two arrays in $O(n + \sigma)$
time and words of working space by processing $T$ with bucket sort. Therefore, the total preprocessing time is $O(n + \sigma)$, and the space usage of our data structures is $O(n + \sigma)$ words.

We compute the first and last occurrences of the distinct characters in substring $T[i..j]$ by using RmQs on array $\bar{L}$ and RMQs on array $\bar{R}$ for the RCLFL($T$, $i$, $j$) query. Let $c_1^+, c_2^+, \ldots, c_k^+$ be the sequence of distinct characters in the substring $T[i..j]$ in increasing order of their first occurrences in $T[i..j]$ (i.e., $\{c_1^+, c_2^+, \ldots, c_k^+\} = \text{colors}(T[i..j])$, and $b_c^+ < b_{c+1}^+$ for any integer $x \in [1, k - 1]$). Then, $\text{RmQ}(\bar{L}, b_{c-1}^+, b_c^+ + 1, j)$ is equal to the leftmost occurrence of character $c^+_x$ in substring $T[i..j]$ (i.e., $b^+_x$) for any integer $x \in [2, k]$, because characters $c_1^+, c_2^+, \ldots, c_{k-1}^+$ already occur in $T[i..b_{c-1}^+]$. Similarly, let $c_1^-, c_2^- \ldots, c_k^-$ be the sequence of distinct characters in substring $T[i..j]$ in decreasing order of their last occurrences in $T[i..j]$. Then, $\text{RMQ}(\bar{R}, i, e_{c-1}^- - 1)$ is equal to the rightmost occurrence of character $c_x^-$ for any integer $x \in [2, k]$. Formally, the following lemma holds.

**Lemma 22** ([9]). The following two statements hold. (i) $b_{c-1}^+ = \text{RmQ}(\bar{L}, i, j) \text{ and } b^+_x = \text{RmQ}(\bar{L}, b_{c-1}^+, b^+_x + 1, j)$ for any integer $x \in [1, k - 1]$. (ii) $e_{c-1}^- = \text{RmQ}(\bar{R}, i, j) \text{ and } e_x^- = \text{RMQ}(\bar{R}, i, e_{c-1}^- - 1)$ for any integer $x \in [1, k - 1]$.

We compute RCLFL($T$, $i$, $j$) in the following way: (i) Enumerate the distinct characters $c_1^+, c_2^+, \ldots, c_k^+$ and their first occurrences in the substring $T[i..j]$ by using $k + 1$ RmQ queries on $\bar{L}$. (ii) For each character $c_x^+$, we change $W[c_x^+]$ to its first occurrence $b^+_x$. (iii) Enumerate the distinct characters $c_1^-, c_2^- \ldots, c_k^-$ and their last occurrences in the substring $T[i..j]$ by using $k + 1$ RMQ queries on $\bar{R}$. (iv) For each character $c_x^-$, we obtain its first occurrence $b^-_x$ by accessing $W[c_x^-]$ and output the triplet $(e_x^-, b^-_x, e_x^-)$. Therefore, the running time is $O(k)$ in total, and we obtain Lemma 23.

**Proof of Lemma 20** Data structure $\Lambda$ consists of a data structure $\Lambda'$ for string $L_{\text{shrun}}$ (introduced in Appendix A) and an array $C'$ of length $r$. Each cell $C'[i] = c$ stores the original character corresponding to $L_{\text{shrun}}[i]$, i.e., $c \in \text{colors}(T)$ and $\text{pred}(\text{colors}(L'), c) = L_{\text{shrun}}[i]$. The space usage of $\Lambda$ is $O(r)$ words, because the alphabet size of $L_{\text{shrun}}$ is at most $r$. We can construct $\Lambda$ in $O(n)$ time and $O(r)$ words of working space, because (i) we can construct string $L_{\text{shrun}}$ in $O(n)$ time and $O(r)$ words of working space by Lemma 19, and (ii) we can construct array $C'$ in $O(n)$ time by using two strings $L_{\text{shrun}}$ and $L'$.

We compute the RCLFL($L'$, $i$, $j$) query using the RCLFL($L_{\text{shrun}}$, $i$, $j$) query and array $C'$. $L'[i] = C'[L_{\text{shrun}}[i]]$ for any integer $i \in [1, r]$, and hence, we compute RCLFL($L'$, $i$, $j$) in the following two steps: (i) We compute RCLFL($L_{\text{shrun}}$, $i$, $j$) = $\{(c_1, b_{c_1}, e_{c_1}), (c_2, b_{c_2}, e_{c_2}), \ldots, (c_k, b_{c_k}, e_{c_k})\}$. (ii) For each output element $(c_x, b_x, e_x)$ by using array $C'$. Therefore, the running time is still $O(k)$ in total, and finally, we obtain Lemma 20.

**Appendix C.2: Proofs for Section 3.2**

**Example for Section 3.2** We first give an example of computing Weiner-intervals with depth 2 in Figure 1. Here, $W_1 = \{(1, 1, |\$|), (2, 10, |\$|), (11, 16, |b|)\}$, and $W_2 = \{(1, 4, |\$|), (11, 11, |b|)\}$, where we let LCP[17] = 0. When we apply a Weiner query to each Weiner-interval with depth 1, we obtain a set of sa-intervals, $\{(1, 1, |\$|), (1, 4, |\$|), (5, 10, |\$|), (11, 11, |b|)\}$. Here, $V'_i[11] = 1, V'_i[4] = 0, V'_i[10] = 1$, and $V'_i[11] = 0$, and hence, $W_2$ corresponds to the set $\bigcup_{i,b,c,d,e\in\text{sa}}\{\langle b', c', 2 \rangle | \langle b', c', 2 \rangle \in \text{weiner}(b, e, 1) \text{ s.t. } V'_i[c'] = 0\}$.

**Details of Beller et al’s algorithm.** The details of their enumeration algorithm are as follows. We ensure that queue $Q$ stores Weiner-intervals with depth $d - 1$ in any order, and that bit array $V'$ corresponds to $V_{d-1}$, before the algorithm enumerates Weiner-intervals.
with depth \(d\). It repeats the following three steps until queue \(Q\) is empty: (i) It takes a Weiner-interval \(w = (b, c, d - 1)\) out of queue \(Q\). (ii) It computes set filter(\(w\)) by using a Weiner query and bit array \(V'\). (iii) It outputs each Weiner-interval \((b', c', d)\) in filter(\(w\)) and sets the bit \(V'[c']\) to 1. After enumerating the Weiner-intervals with depth \(d\), it pushes them into queue \(Q\).

**Details of data structure \(U'\).** Data structure \(U'\) consists of the data structure \(Z\) in Lemma 24 and data structure \(A\) for string \(L' = L[c^1], L[c^2], \ldots, L[c^r]\) in Lemma 23. The two data structures can be constructed in \(O(n \log \log_w(n/r))\) time and \(O(r)\) words of working space by processing the RLBWT of a string \(T\). Data structure \(U'\) answers a Weiner query by using the following two relations among the three queries \(\text{weiner}(b, c, d), \text{RCLFL}(L, b, e),\) and \(\text{RCLFL}(L', b', e')\).

\[\text{Lemma 23 (e.g., [11])}.\text{ For any sa-interval } (b, c, d), \text{weiner}(b, c, d) = \langle \{\text{LF}(b_1), \text{LF}(e_1), d+1\}, \{\text{LF}(b_2), \text{LF}(e_2), d+1\}, \ldots, \{\text{LF}(b_k), \text{LF}(e_k), d+1\} \rangle, \text{where } \text{RCLFL}(L, b, e) = \{(c_1, b_1, e_1), (c_2, b_2, e_2), \ldots, (c_k, b_k, e_k)\}.\]

\[\text{Lemma 24. For any sa-interval } (b, c, d), \text{RCLFL}(L, b, e) = \{(c_1, b_1, e_1), (c_2, b_2, e_2), \ldots, (c_k, b_k, e_k)\}. \text{ Here, (i) } \text{RCLFL}(L', b', e') = \{(c_1, b_1, e_1), (c_2, b_2, e_2), \ldots, (c_k, b_k, e_k)\}, (ii) b' = \text{pred}(L'^+, b), (iii) e' = \text{pred}(L'^+, e), \text{ and (iv) } x \text{ and } y \text{ are two integers such that } c_x = L[b] \text{ and } c_y = L[e], \text{ respectively.}\]

**Proof.** For each triplet \((\tilde{c}, \tilde{b}, \tilde{e}) \in \text{RCLFL}(L', b', e')\), \(L\)-run \(\tilde{L}_b\) is the first \(L\)-run representing repetitions of \(\tilde{c}\) on \(L[\tilde{b}^\ell, e^\ell]\), and at least one character in \(\tilde{L}_b\) is contained in \(L[\tilde{b}, e]\) (i.e., \([\ell^\ell_b, e^\ell] \cup [\tilde{b}, e] \neq \emptyset\)). This implies that the first occurrence of character \(\tilde{c}\) in \(L[\tilde{b}, e]\) is \(\min([\ell^\ell_b, e^\ell] \cup [\tilde{b}, e])\). If \(b' \neq \tilde{b}\), then \(\min([\ell^\ell_b, e^\ell] \cup [\tilde{b}, e]) = \ell^\ell_b\); otherwise, \(\min([\ell^\ell_b, e^\ell] \cup [\tilde{b}, e]) = b\). Similarly, the last occurrence of character \(\tilde{c}\) is \(\ell^\ell_e\) if \(e' \neq \tilde{e}\); otherwise, the last occurrence is \(e\). Therefore, we obtain Lemma 24.

**Lemma 25.** There exists a data structure \(U'\) supporting a Weiner query in \(O((1 + k) \log \log_w(n/r))\) time and \(O(k)\) words of working space. We can construct the data structure in \(O(n \log \log_w(n/r))\) time and \(O(r)\) words of working space by processing the RLBWT of a string \(T\).

**Construction algorithm for data structures.** Recall that our algorithm is built on the following three data structures: (i) a data structure \(U'\) supporting Weiner queries; (ii) a queue \(Q\) storing all Weiner-intervals with depth 1 in any order; and (iii) a bit array \(V'\) of size \(n\). We already showed that \(U'\) can be constructed in \(O(n \log \log_w(n/r))\) time and \(O(r)\) words of working space for the given RLBWT of a string \(T\) (Lemma 25). Obviously, the bit array \(V'\) can be constructed in \(O(n)\) time. We then need to compute all the Weiner-intervals with depth 1 to construct queue \(Q\). The Weiner-intervals with depth 1 are the sa-intervals of characters occurring in \(T\). The sa-intervals can be computed by reading permutation \(F\) in
left-to-right order, and we can access \( F \) in \( O(n \log \log_w(n/r)) \) time by using data structure \( Z \) (Lemma\(^7\)). Therefore, the construction time is \( O(n \log \log_w(n/r)) \) in total, and we obtain Theorem\(^7\).

**Appendix D: RLESA**

A run-length extended suffix array (RLESA) consists of (i) five arrays \( SA^- \), \( SA^+ \), \( DSA^- \), \( DSA^+ \), and \( LCP^- \) of size \( r \) and (ii) two predecessor data structures for \( SA^- \) and \( SA^+ \). \( SA^+ \) and \( SA^- \) store positions in sets \( L^+ \) and \( L^- \), respectively. \( LCP^+ \) stores the lcp-values at positions in set \( L^+ \). \( DSA^+ \) and \( DSA^- \) store the differences in adjacent sa-values at positions in sets \( L^+ \) and \( L^- \), respectively. Formally, we define the five arrays in the following way:

- \( SA^- = \ell_1^-, \ell_2^- , \ldots , \ell_r^- \),
- \( SA^+ = \ell_1^+, \ell_2^+, \ldots , \ell_r^+ \),
- \( LCP^+ = LCP[\ell_1^+ + 1], LCP[\ell_2^+ + 1], \ldots , LCP[\ell_r^+ + 1] \),
- \( DSA^- = SA[\ell_1^-] - SA[\ell_1^- - 1], SA[\ell_2^-] - SA[\ell_2^- - 1], \ldots , SA[\ell_r^-] - SA[\ell_r^- - 1] \), and
- \( DSA^+ = SA[\ell_1^+] - SA[\ell_1^+ - 1], SA[\ell_2^+] - SA[\ell_2^+ - 1], \ldots , SA[\ell_r^+] - SA[\ell_r^+ - 1] \).

Here, let \( SA[i + 1] = SA[i] \), \( SA[0] = SA[n] \), and \( LCP[i + 1] = LCP[1] \), and let \( \ell_1^-, \ell_2^-, \ldots , \ell_r^- \) and \( \ell_1^+, \ell_2^+, \ldots , \ell_r^+ \) be the positions in sets \( L^- \) and \( L^+ \) in increasing order of their sa-values, respectively, i.e., \( L^- = \{ \ell_1^-, \ell_2^-, \ldots , \ell_r^- \} \), \( SA[\ell_1^-] < SA[\ell_2^-] < \cdots < SA[\ell_r^-] \), \( L^+ = \{ \ell_1^+, \ell_2^+, \ldots , \ell_r^+ \} \), and \( SA[\ell_1^+] < SA[\ell_2^+] < \cdots < SA[\ell_r^+] \). Then, the following lemma holds.

**Lemma 26** [17]. For any integer \( i \in [1, n - 1] \), the following equations hold. (i) \( DSA^-[i^-] = SA[i] - SA[i + 1] \), and (ii) \( LCP^+[i^-] = LCP[i + 1] \). where \( i^- = \text{pred}(SA^-, SA[i]) \) and \( d = SA[i] - SA[i + 1] \). Similarly, for any integer \( i \in [2, n] \), \( DSA^+[i^+] = SA[i] - SA[i - 1] \), where \( i^+ = \text{pred}(SA^+, SA[i]) \).

Lemma\(^{26}\) implies that an RLESA can support next- and previous-access queries in \( O(\log \log_w(n/r)) \) time by using predecessor queries on \( SA^- \) and \( SA^+ \). The lemma also indicates that we can support next-access queries by using arrays \( DSA^- \), \( SA^- \), and \( LCP^- \) and the predecessor data structure for \( SA^- \).

**Proof of Lemma 11** For given enumerated lcp-values and sa-values, the five arrays can be constructed in \( O(n \log \log_w(n/r)) \) time and \( O(r) \) words of working space by using the predecessor queries on \( L^+ \) and the array \( L^+ \). For the given RLKBWT of string \( T \), we can enumerate sa-values in \( O(n \log \log_w(n/r)) \) time and \( O(r) \) words of working space in the order of \( SA[p] = n \), \( SA[LF[p]] = n - 1 \), \( SA[LF_2[p]] = n - 2 \), ... \( SA[LF_{n-1}[p]] = 1 \) by applying the LF function to the position \( p \), which is the position such that \( SA[p] = n \). Here, \( LF_d(i) \) is a recursive LF function applied \( d \) times to a given position \( i \), i.e., \( LF_d(i) = LF_{d-1}(LF(i)) \) if \( d \geq 1 \); otherwise, \( LF_d(i) = i \). We can also enumerate lcp-values in \( O(n \log \log_w(n/r)) \) time by using the algorithm presented in Section\(^3\). Therefore, an RLESA can be constructed in \( O(n \log \log_w(n/r)) \) time and \( O(r) \) words of working space.

**Appendix E.1: Enumeration algorithm for type-2 lcp-intervals**

The enumeration algorithm for lcp-intervals of type-2 is built on the following five data structures: (i) the RLESA for string \( T \); (ii) data structure \( Z \) (Lemma\(^7\)); (iii) an integer array \( D \) of length \( r \) such that each \( D[x] \) stores \( \min(LCP[\ell_x^+ + 1], \ell_x^{+1}) \); (iv) a data structure \( X \) of \( O(r) \) words supporting RunQ queries on array \( D \) in constant time; and (v) two empty stacks \( T^\text{left} \) and \( T^\text{right} \). Here, data structure \( X \) was introduced in Appendix C.1. The space usage of the data structures is \( O(r) \) words in total. We can construct array \( D \) in \( O(n \log \log_w(n/r)) \)
time by enumerating the lcp-values in left-to-right order by using the RLESA. Therefore, we
can construct the five data structures in $O(n \log \log_m (n/r))$ time by processing the RLBWT
of string $T$.

The enumeration algorithm finds type-2 lcp-intervals by using a type-2 lcp-interval tree
and sets of intervals $Y_1, Y_2, \ldots, Y_h$. The type-2 lcp-interval tree is a rooted tree such that each
node corresponds to a distinct type-2 lcp-interval in set $I^2$. Formally, a type-2 lcp-interval tree
is defined in the following way: (i) its nodes correspond to a one-to-one basis to intervals in
set $T = \{[b, e] \mid (b, c, d) \in I^2 \} \cup \{[1, n]\}$; (ii) its root corresponds to interval $[1, n]$; and (iii) the
parent of any interval $[b, e] \in T \setminus \{[1, n]\}$ is the interval $[b', e'] \in T$ such that $[b, e] \subset [b', e']$
and $e - b' + 1$ is smallest. Let $\text{parent}(I)$ and $\text{children}(I)$ be the parent and children of an interval $I$,
respectively, i.e., $\text{parent}(I) = \arg \min_{I'} |I'|$ and $\text{children}(I) = \{I' \mid I' \in T \text{ s.t. } \text{parent}(I') = I\}$,
where $V = \{I' \mid I' \in T \text{ s.t. } I' \supset I\}$.

Next, we define sets of intervals $Y_1, Y_2, \ldots, Y_h$. Set $Y_1$ consists of leaves in a type-2
lcp-interval tree (i.e., $Y_1 = \{I \mid I \in T \text{ s.t. } \text{children}(I) = \emptyset\}$). $Y_{i+1}$ is defined using set
$Y_i = \{I_1, I_2, \ldots, I_{|Y_i|}\}$ and integer $\tau_i$ for any integer $i \geq 1$. Here, $\tau_i$ is the smallest
integer such that the children of the parent of interval $I_{\tau_i}$ are contained in set $Y_i$, i.e.,
$\tau_i = \min\{x \mid x \in [1, |Y_i|] \text{ s.t. } \text{children}(\text{parent}(I_{\tau_i})) \subseteq Y_i\}$, where $I_{\tau_i}$ is the $x$-th interval in the
intervals in set $Y_i$ sorted in increasing order of their left boundaries. Then, $Y_{i+1}$ is the
set of intervals obtained by replacing the children of the parent of interval $I_{\tau_i}$ in set $Y_i$ with
the parent (i.e., $Y_{i+1} = (Y_i \setminus \text{children}(\text{parent}(I_{\tau_i}))) \cup \{\text{parent}(I_{\tau_i})\}$). Finally, let $h$ be the
smallest integer such that $Y_h = \{[1, n]\}$.

The union of the two sets $Y_1$ and set $\{\text{parent}(I_{1, \tau_1}), \text{parent}(I_{2, \tau_2}), \ldots, \text{parent}(I_{h-1, \tau_{h-1}})\}$
is equal to the set of nodes in the type-2 lcp-interval tree, because (i) $Y_1$ is the set of leaves in
the type-2 lcp-interval tree, (ii) $Y_h = \{[1, n]\}$ is the root of the tree, and (iii) $Y_i$ is obtained
by replacing an interval in $Y_{i+1}$ with its children for any integer $i \in [1, h-1]$. Formally, the
following lemmas hold.

**Lemma 27.** $\tau_i$ exists for any integer $i \in [1, h-1]$, i.e., $\{x \mid x \in [1, |Y_i|] \text{ s.t. } \text{children}(\text{parent}(I_{\tau_i})) \subseteq Y_i\} \neq \emptyset$.

**Proof.** We use the following observation: any rooted tree has at least one internal node such
that each child of the node is a leaf. Set $Y_1$ is the set of leaves in a type-2 lcp-interval tree,
and hence, $\tau_1$ exists by the above observation. If $Y_i$ represents the set of leaves in a tree $X$,
then $Y_{i+1}$ also represents the set of leaves in the subtree such that (i) its root is the root
of tree $X$ and (ii) its leaves are nodes in $Y_{i+1}$. This is because the set $Y_{i+1}$ is obtained by
replacing the children of a node in $Y_i$ with the node. Therefore, integers $\tau_2, \tau_3, \ldots, \tau_{h-1}$ also
exist by the above observation.

**Lemma 28.** The union of two sets $Y_1$ and $\{\text{parent}(I_{1, \tau_1}), \text{parent}(I_{2, \tau_2}), \ldots, \text{parent}(I_{h-2, \tau_{h-2}})\}$
is equal to the set of type-2 lcp-intervals, i.e., $T = Y_1 \cup \{\text{parent}(I_{1, \tau_1}), \text{parent}(I_{2, \tau_2}), \ldots,
\text{parent}(I_{h-1, \tau_{h-1}})\}$.

**Proof.** Set $Y_1$ represents the leaves in the type-2 lcp-interval tree, and $\{\text{parent}(I_{1, \tau_1}),
\text{parent}(I_{2, \tau_2}), \ldots, \text{parent}(I_{h-1, \tau_{h-1}})\}$ represents the internal nodes and root in the type-2
lcp-interval tree. Therefore, Lemma 28 holds.

The nodes in the type-2 lcp-interval tree represent the set of type-2 lcp-intervals, and hence,
we can enumerate the type-2 lcp-intervals by computing nodes in the type-2 lcp-interval tree.
The enumeration algorithm does not store the whole type-2 lcp-interval tree to enumerate
type-2 lcp-intervals, because the tree uses $O(n)$ words. Instead, the algorithm
We compute interval \( \text{parent}(I_{i,\tau_i}) \) for any integer \( i \in [1,h-1] \) by using integer \( \kappa_i \), set \( Y_i \), the 5-tuples for the intervals in set \( Y_i \), and our five data structures, where the 5-tuple for an interval \([b,e]\) is \((\text{SA}[b], \text{SA}[e], \min(\text{LCP}[b+1..e]), \text{LCP}[b], \text{LCP}[e+1])\). Here, \( \kappa_i \) is the smallest integer such that the parent of interval \( I_{i,\kappa_i} \) does not overlap with interval \( I_{i,\kappa_i+1} \) (i.e., \( \kappa_i = \min(\{x \mid x \in [1,Y_i]-1\} \) s.t. \( \text{parent}(I_{i,x}) \cap I_{i,x+1} = \emptyset \} \cup \{Y_i\})\).

Integer \( \kappa_i \) has the following three properties: (i) Interval \( I_{i,\kappa_i} \) is the rightmost child of interval \( I_{i,\tau_i} \) (i.e., \( \tau_i + |\text{children}(\text{parent}(I_{i,\tau_i}))| - 1 = \kappa_i \)). (ii) \( \kappa_i \) is equal to the smallest integer \( x \geq \tau_i - 1 \) such that \( D[z] \geq \min(\text{LCP}[b], \text{LCP}[e+1]) \), where \( I_{i,x} = [b,e] \) and \( z \) is the largest integer such that \( \ell_z^+ \in I_{i,x} \). (iii) \( \tau_i \) is equal to the smallest integer \( x \leq \kappa_i \) such that \( \min\{\min(\text{LCP}[b+1..e]), \min(\text{LCP}[b+1..e])\} \geq \max(\text{LCP}[b'], \text{LCP}[e'+1]) \), where \( I_{i,x} = [b,e] \), \( \kappa_i, = [b',e'] \), \( v \) is the largest integer such that \( \ell_v^+ \in [b,e] \), and \( v' \) is the smallest integer such that \( \ell_{v'}^+ \in [b',e'] \).

The first property indicates that we can compute the rightmost child of the parent of interval \( I_{i,\tau_i} \) by finding interval \( I_{i,\kappa_i} \) and computing \( |\text{children}(\text{parent}(I_{i,\kappa_i}))| \). The second property indicates that we can find interval \( I_{i,\kappa_i} \) by verifying whether a given interval is \( I_{i,\kappa_i} \) by using the 5-tuple for the given interval and array \( D \) while reading intervals \( I_{i,\tau_i-1}, I_{i,\tau_i-2}, \ldots, I_{i,Y_i} \) in left-to-right order. The third property indicates that we can find interval \( I_{i,\tau_i} \) by verifying whether a given interval is \( I_{i,\tau_i} \) by using the 5-tuple for the given interval and array \( D \) while reading intervals \( I_{i,1}, I_{i,2}, \ldots, I_{i,\kappa_i} \) in right-to-left order. After we find the two intervals \( I_{i,\tau_i} = [b,e] \) and \( I_{i,\kappa_i} = [b',e'] \) (i.e., the leftmost and rightmost children of \( \text{parent}(I_{i,\tau_i}) \)), we can compute interval \( \text{parent}(I_{i,\tau_i}) \) by using the RLESA, array \( D \), and data structure \( X \), because \( \text{parent}(I_{i,\tau_i}) \) is the longest interval \([b,e] \) such that its depth is equal to \( \min(\text{LCP}[b+1..e']) \) (i.e., \( \min(\text{LCP}[b+1..e]) = \min(\text{LCP}[b+1..e']) \)), and the interval contains \( I_{i,\tau_i} \) and \( I_{i,\kappa_i} \). Formally, the following five lemmas hold.

\textbf{Lemma 29.} We can compute interval \( \text{parent}(I_{i,\tau_i}) = [b,e] \) and the 5-tuple for the interval in \( O(|\text{parent}(I_{i,\tau_i}) \setminus [b,e]| + 1) \log \log \log(n/r)) \) time for two given intervals \( I_{i,\tau_i} = [b,e] \) and \( I_{i,\tau_i+|\text{children}(\text{parent}(I_{i,\tau_i}))|}-1 = [b',e'] \) by using (i) the 5-tuples for the given intervals, (ii) array \( D \), (iii) data structure \( X \), (iv) the RLESA for string \( T \), and (v) data structure \( Z \).

\textbf{Proof.} Interval \([b,e] \) is the longest interval such that its depth is equal to \( \min(\text{LCP}[b+1..e']) \) (i.e., \( \min(\text{LCP}[b+1..e]) = \min(\text{LCP}[b+1..e']) \)) and it contains the two given intervals. We compute \( \min(\text{LCP}[b+1..e']) \) by computing \( \min(\min(\text{LCP}[b+1..e]), \text{LCP}[\ell_{v^+}^+ + 1..\ell_{v^+}'^+]), \min(\text{LCP}[b+1..e]), \text{LCP}[\ell_{v^+}^+ + 1..\ell_{v^+}'^+]), \text{LCP}[\ell_{v^+}^+ + 1..\ell_{v^+}'^+]), \min(\text{LCP}[b+1..e]), \text{LCP}[\ell_{v^+}^+ + 1..\ell_{v^+}'^+]) \) in \( O(\log \log(n/r)) \) time, because (i) \( \min(\text{LCP}[b+1..e]) \) and \( \min(\text{LCP}[b+1..e']) \) are stored in the 5-tuples for the given intervals, (ii) \( \min(\text{LCP}[\ell_{v^+}^+ + 1..\ell_{v^+}'^+]) \) is equal to \( D[\text{Rmq}(D,v,v'-1)] \), and (iii) we can compute the two integers \( v \) and \( v' \) in \( O(\log \log(n/r)) \) time by using two predecessor queries on \( L^+ \).

Next, we compute the longest interval such that its depth is equal to \( \min(\text{LCP}[b+1..e']) \) and it contains interval \([b,e] \) by using the RLESA and the 5-tuples for the given intervals. We can compute the longest interval by using \( b \rightarrow b+2 \) previous-access queries and \( e \rightarrow e' + 2 \) next-access queries. We also obtain the 5-tuple for interval \([b,e] \) at the same time, and hence, the running time is \( O(|\text{parent}(I_{i,\tau_i}) \setminus [b,e]| + 1) \log \log(n/r)) \).

\textbf{Lemma 30.} Interval \( I_{i,\kappa_i} \) is the rightmost child of the parent of interval \( I_{i,\tau_i} \) for any integer \( i \in [1,h-1] \) (i.e., \( \tau_i = \kappa_i - |\text{children}(\text{parent}(I_{i,\kappa_i}))| + 1 \)).
Proof. Recall that set $Y_i$ represents the leaves in a tree. Integer $\kappa_i$ is the smallest integer such that $\text{parent}(I_{i,\kappa_i}) \cap I_{i,\kappa_i+1} = \emptyset$, and it indicates that (i) interval $I_{i,\kappa_i}$ is the rightmost child of interval $\text{parent}(I_{i,\kappa_i})$, and (ii) the parent of interval $I_{i,t}$ is equal to that of interval $I_{i,t+1}$ or an ancestor of $I_{i,t+1}$ for any integer $t \in [1, \kappa_i - 1]$ (i.e., $\text{parent}(I_{i,t}) \supseteq \text{parent}(I_{i,t+1}) \supseteq \cdots \supseteq \text{parent}(I_{i,\kappa_i})$). The two relations mean that $\text{parent}(I_{i,\kappa_i-\text{size}([\text{parent}(I_{i,\kappa_i})]+1}) = \text{parent}(I_{i,\kappa_i-\text{size}(\text{parent}(I_{i,\kappa_i}))}) = \cdots = \text{parent}(I_{i,\kappa_i})$, i.e., $\kappa_i$ is the smallest integer such that $\text{children}(\text{parent}(I_{i,\kappa_i})) \subset Y_i$ and interval $I_{i,\kappa_i}$ is the rightmost child of interval $\text{parent}(I_{i,\kappa_i})$. Therefore, Lemma 30 holds.

Lemma 31. $\tau_{i-1} \leq \kappa_i$ for any integer $i \in [1, h - 1]$, where $\tau_0 = 1$.

Proof. $\text{parent}(I_{i,x}) \cap I_{i,x+1} \neq \emptyset$ for any integer $x \in [1, \tau_i - 1]$, because (i) $I_{i,x} = I_{i-1,x}$, (ii) $I_{i-1,x-1} \cap I_{i-1,x+1} \neq \emptyset$, and (iii) $I_{i-1,\tau_i-1} \subset I_{i,\tau_i-1}$. Therefore, $\kappa_i$ is no less than $\tau_{i-1}$.

Lemma 32. For two integers $i \in [1, h - 1]$ and $x \in [1, |Y_i| - 1]$, $\text{parent}(I_{i,x}) \cap I_{i,x+1} \neq \emptyset \Leftrightarrow \max\{\text{LCP}[b, LCP[e+1]]\} = \mathcal{D}[z]$, where $I_{i,x} = [b, e]$, and $z$ is the largest integer such that $\ell_{z,1} \in I_{i,x}$.

Proof. Lemma 32 holds because (i) the parent of interval $I_{i,x}$ is the longest interval $[b', e']$ such that interval $[b', e']$ contains $I_{i,x}$, and its depth is max $\{\text{LCP}[b, LCP[e+1]]\}$; (ii) interval $I_{i,x+1}$ contains position $\ell_{z,1}$; and (iii) the parent contains position $\ell_{z,1}$ if and only if $\text{LCP}[\ell_{z,1} + 1..\ell_{z,1+1}] = \mathcal{D}[z] \geq \max\{\text{LCP}[b, LCP[e+1]]\}$.

Lemma 33. For any integer $i \in [1, h - 1]$, integer $\tau_i$ is equal to the smallest integer $x \leq \kappa_i$ such that $\min\{\min(\text{LCP}[b+1, e]), \min(\text{LCP}[\ell_{z,1} + 1..\ell_{z,1+1}]), \min(\text{LCP}[b'+1, e'])\} \geq \max\{\text{LCP}[b], \text{LCP}[e+1]\}$, where $I_{i,x} = [b, e]$, $I_{i,\kappa_i} = [b', e']$, $v$ is the largest integer such that $\ell_{v,1} \in [b', e']$, and $v'$ is the smallest integer such that $\ell_{v',1} \in [b', e']$.

Proof. Interval $\text{parent}(I_{i,\kappa_i})$ is the longest interval $[b, e]$ such that its depth is equal to $\min(\text{LCP}[b+1, e'])$ (i.e., $\min(\text{LCP}[b+1, e']) = \min(\text{LCP}[b'+1, e'])$), and the interval contains $I_{i,\kappa_i}$. The definition of the longest interval indicates that, for any integer $x \in [1, \kappa_i - 1]$, $\text{parent}(I_{i,x}) = \text{parent}(I_{i,\kappa_i})$ if and only if $\min(\text{LCP}[b+1, e']) \geq \{\text{LCP}[b'], \text{LCP}[e'+1]\}$, which is equal to $\min(\text{LCP}[b+1, e']) \geq \max\{\text{LCP}[b], \text{LCP}[e+1]\}$, where $I_{i,x} = [b, e]$, $I_{i,\kappa_i} = [b', e']$, $v$ is the largest integer such that $\ell_{v,1} \in [b', e']$, and $v'$ is the smallest integer such that $\ell_{v',1} \in [b', e']$. Therefore, Lemma 33 holds.

Next, the details of our enumeration algorithm for type-2 lcp-intervals with EPSs are as follows. Before the algorithm computes interval $\text{parent}(I_{i,\tau_i})$, two stacks $\mathcal{Y}_{\text{left}}$ and $\mathcal{Y}_{\text{right}}$ are set equal to two stacks $\mathcal{Y}_{\text{left}}$ and $\mathcal{Y}_{\text{right}}$, respectively. Here, $\mathcal{Y}_{\text{left}}$ stores $\tau_{i-1} - 1$ elements, and each $x$-th element is a pair of interval $I_{i,\tau_{i-1}-x}$ and the 5-tuple for the interval. Hence, the top element in $\mathcal{Y}_{\text{left}}$ is a pair of interval $I_{i,\tau_{i-1}-1}$ and the 5-tuple for the interval. Similarly, stack $\mathcal{Y}_{\text{right}}$ stores $|Y_i| - \tau_{i-1} + 1$ elements, and each $x$-th element is a pair of interval $I_{i,\tau_{i-1}-x+1}$ and the 5-tuple for the interval. Hence, the top element in $\mathcal{Y}_{\text{right}}$ is a pair of interval $I_{i,\tau_{i-1}}$ and the 5-tuple for the interval. The algorithm computing interval $\text{parent}(I_{i,\tau_i})$ consists of the following steps:

1. Remove intervals $I_{i,\tau_{i-1}}, I_{i,\tau_{i-1}+1}, \ldots, I_{i,\kappa_i}$ with the 5-tuples from $\mathcal{Y}_{\text{right}}$.
2. Push the removed intervals with the 5-tuples onto $\mathcal{Y}_{\text{left}}$.
3. Remove intervals $I_{i,\tau_i}, I_{i,\tau_{i+1}}, \ldots, I_{i,\kappa_i}$ with the 5-tuples from $\mathcal{Y}_{\text{left}}$.
4. Compute interval $\text{parent}(I_{i,\tau_i})$ and its 5-tuple, and push them onto $\mathcal{Y}_{\text{right}}$.
5. Output the type-2 lcp-interval $parent(I_{i, \tau_{i}})$ with an EPS by using interval $parent(I_{i, \tau_{i}})$ and its 5-tuple.

The algorithm outputs all the type-2 lcp-intervals in set $\{parent(I_{1, \tau_{1}}), parent(I_{2, \tau_{2}}), \ldots, parent(I_{k, \tau_{k}})\}$, and we can also output all the type-2 lcp-intervals in set $\{I_{1,1}, I_{1,2}, \ldots, I_{1,|Y_{1}|}\}$ by using two stacks $\mathcal{T}_{\text{left}}$ and $\mathcal{T}_{\text{right}}$. The union of the two sets is equal to the set of type-2 lcp-intervals, by Lemma 25 and hence, our algorithm can enumerate all the type-2 lcp-intervals with ESPs.

Next, we show that the running time is $O(n \log \log_{w}(n/r))$ in total. We can execute steps 1 and 2 in $O((B_{\text{right}} + h) \log \log_{w}(n/r))$ time in total for any integer $i \in [1, h - 1]$ by using Lemmas 21 and 22 where $B_{\text{right}}$ is the number of elements removed from stack $\mathcal{T}_{\text{right}}$. This is because we can compute the smallest integer $v$ and largest integer $v'$ such that $\ell_{v}^{+} \in I$ and $\ell_{v'}^{+} \in I$, respectively, by using data structure $Z$ for a given interval $I$ and its 5-tuple. Similarly, we can execute step 3 in $O((B_{\text{left}} + h) \log \log_{w}(n/r))$ time by using Lemma 23 and data structure $Z$, where $B_{\text{left}}$ is the number of elements removed from stack $\mathcal{T}_{\text{left}}$. We can execute step 4 in $O((\sum_{j=1}^{h-2} |G_{j}|) \log \log_{w}(n/r))$ time by using Lemma 29 where $G_{i} = parent(I_{i, \tau_{i}}) \setminus [b, e']$, $b$ is the left boundary of interval $I_{i, \tau_{i}}$ and $e'$ is the right boundary of interval $I_{k, \tau_{k}}$. The running time is $O(n \log \log_{w}(n/r))$ in total, except for the construction time for the two stacks $\mathcal{T}_{\text{left}}$ and $\mathcal{T}_{\text{right}}$, by the following observations: (i) Any interval in a type-2 lcp-interval tree is pushed onto $\mathcal{T}_{\text{right}}$ at most once, and hence, $B_{\text{right}} \leq |Z| = O(n)$. (ii)) Similarly, any such interval is pushed onto $\mathcal{T}_{\text{left}}$ at most once, and hence, we also have $B_{\text{left}} \leq |Z| = O(n)$. (iii) $h \leq |Z| = O(n)$. (iv) Sets $G_{1}, G_{2}, \ldots, G_{h-1}$ do not overlap each other, and hence, $\sum_{j=1}^{h-2} |G_{j}| = O(n)$.

Next, we show that we can construct the two stacks, $\mathcal{T}_{\text{left}}$ and $\mathcal{T}_{\text{right}}$, in $O(n \log \log_{w}(n/r))$ time and $O(r)$ words of working space by processing the RLBWT of a string $T$. The two stacks can be constructed in $O(n)$ time by using set $Y_{1}$ and the 5-tuples for the intervals in set $Y_{1}$. The 5-tuples can be constructed in $O(n \log \log_{w}(n/r))$ time by using set $Y_{1}$ and the RLESA of string $T$. Therefore, we show that set $Y_{1}$ can be constructed in $O(n \log \log_{w}(n/r))$ time and $O(r)$ words of working space by processing the RLBWT of $T$.

We use a sequence of intervals, $J_{1}, J_{2}, \ldots, J_{r-1}$, to construct set $Y_{1}$. Interval $J_{x} = [b, e]$ is a type-2 lcp-interval such that (i) it contains two positions $\ell_{x}^{+}$ and $\ell_{x}^{+}$, and (ii) its depth is largest, i.e., let $B_{x}^{+} = \{[b, e] | \langle b, e, d \rangle \in T \text{ s.t. } \ell_{x}^{+}, \ell_{x}^{+} \in [b, e]\}$, then $J_{x} = \arg \max(\min(\text{LCP}[b + 1, e + 1]))$.

Set $\{J_{1}, J_{2}, \ldots, J_{r-1}\}$ contains the leaves in the type-2 lcp-interval tree (i.e., $\{J_{1}, J_{2}, \ldots, J_{r-1}\} \supseteq Y_{1}$), because any leaf is a type-2 lcp-interval for some integer $x$ such that (i) it contains two positions $\ell_{x}^{+}$ and $\ell_{x}^{+}$ and (ii) it does not contain any other type-2 lcp-interval, i.e., its depth is largest. Formally, let $v_{1}, v_{2}, \ldots, v_{k}$ ($v_{1} < v_{2} < \cdots < v_{k}$) be the sequence of integers on interval $[1, r - 1]$ such that each interval $J_{v_{x}}$ is not equal to interval $J_{v_{x-1}}$, i.e., $\{v_{1}, v_{2}, \ldots, v_{k}\} = \{1\} \cup \{x | x \in [2, r - 1] \text{ s.t. } J_{x} \neq J_{x-1}\}$. Then set $Y_{1}$ is equal to the set of intervals in set $\{J_{v_{1}}, J_{v_{2}}, \ldots, J_{v_{k}}\}$ such that each interval $J_{v_{x}}$ does not contain the two intervals $J_{v_{x-1}}$ and $J_{v_{x+1}}$, i.e., $Y_{1} = \{J_{v_{x}} | x \in [1, k] \text{ s.t. } J_{v_{x}} \nsubseteq J_{v_{x-1}} \text{ and } J_{v_{x}} \nsubseteq J_{v_{x+1}}\}$, where $J_{0} = [0, 0]$ and $J_{n+1} = [n + 1, n + 1]$.

Next, we can compute intervals $J_{1}, J_{2}, \ldots, J_{r-1}$ by the following lemma.

**Lemma 34.** We can compute intervals $J_{1}, J_{2}, \ldots, J_{r-1}$ in $O(n \log \log_{w}(n/r))$ time and $O(r)$ words of working space by processing the RLBWT of string $T$.

**Proof.** Here, we use the following observation: $\min(\text{LCP}[b_{x}^{+} + 1, c_{x}^{+}]) = \min(\text{LCP}[\ell_{x}^{+} + 1, \ell_{x}^{+}])$ for any integer $x \in [1, r - 1]$, where $J_{x} = [b_{x}^{+}, c_{x}^{+}]$. This observation indicates that $b_{x}^{+}$ and $c_{x}^{+}$ are
the smallest and largest integers such that \( \min(\text{LCP}[b'_x + 1, \ell_{x,t+1}^+]) = \min(\text{LCP}[\ell_x^+ + 1, \ell_{x,t+1}^+]) \) and \( \min(\text{LCP}[\ell_x^++1, e'_t]) = \min(\text{LCP}[\ell_x^+ + 1, \ell_{x,t+1}^+]) \), respectively, because \( \text{LCP}[b], \text{LCP}[e+1] < \min(\text{LCP}[b',+1, e'_t]) \).

Our algorithm for computing intervals \( J_1, J_2, \ldots, J_{r-1} \) consists of the following steps: (i) Compute \( \min(\text{LCP}[\ell_1^+ + 1, \ell_2^+]), \min(\text{LCP}[\ell_2^+ + 1, \ell_3^+]), \ldots, \min(\text{LCP}[\ell_{r-1}^+ + 1, \ell_r^+]) \) by using the enumeration of lcp-values in left-to-right order. (ii) Compute \( e'_1, e'_2, \ldots, e'_{r-1} \) by using integers \( \min(\text{LCP}[\ell_1^+ + 1, \ell_2^+]), \min(\text{LCP}[\ell_2^+ + 1, \ell_3^+]), \ldots, \min(\text{LCP}[\ell_{r-1}^+ + 1, \ell_r^+]) \), the enumeration of lcp-values in left-to-right order, and a stack. (iii) Compute \( b'_1, b'_2, \ldots, b'_{r-1} \) by using integers \( \min(\text{LCP}[\ell_1^+ + 1, \ell_2^+]), \min(\text{LCP}[\ell_2^+ + 1, \ell_3^+]), \ldots, \min(\text{LCP}[\ell_{r-1}^+ + 1, \ell_r^+]) \), the enumeration of lcp-values in right-to-left order, and a stack. We can execute the three steps in \( O(n \log \log_w(n/r)) \) time, because the RLESA for string \( T \) can enumerate the lcp-values in \( O(n \log \log_w(n/r)) \) time in both left-to-right order and right-to-left order. Therefore, Lemma 34 holds, because we can construct the RLESA in \( O(n \log \log_w(n/r)) \) time and \( O(r) \) words of working space by processing the RLBWT of string \( T \).

Finally, our algorithm for computing set \( Y_1 \) consists of the following three steps: (i) Compute intervals \( J_1, J_2, \ldots, J_{r-1} \) by Lemma 34. (ii) Compute integers \( v_1, v_2, \ldots, v_r \) by using intervals \( J_1, J_2, \ldots, J_{r-1} \). (iii) Construct and output set \( Y_1 \) by using intervals \( J_{v_1}, J_{v_2}, \ldots, J_{v_r} \). The construction time is \( O(n \log \log_w(n/r)) \) in total, and the working space is \( O(r) \) words. Therefore, we can enumerate type-2 lcp-intervals in \( O(n \log \log_w(n/r)) \) time and \( O(r) \) words of working space by processing the RLBWT of string \( T \).

### Appendix E.2: Proofs for Section 4

**Example for Section 4.1** We present an example of computing the type-3 lcp-intervals containing \( \ell_4^+ \) in Figure 2. First, let \( a, b, \ldots, z \) be the smallest and largest integers such that \( \min(\text{LCP}[b_4 + 1, 10]) = \min(\text{LCP}[9 + 1, 10]) \) holds, because we can construct the RLESA in \( O(n \log \log_w(n/r)) \) time and \( O(r) \) words of working space by processing the RLBWT of string \( T \).

**Example for Section 4.2** Next, we present an example of computing the type-1 lcp-intervals in Figure 2 as follows. In the figure, each dotted arrow means that the lcp-interval at its head can be computed by applying the FL function to the lcp-interval at its tail. For example, \( (6,7, abaabab) \) and \( (13,14, baabab) \) are two lcp-intervals, because \( \text{LCP}[6], \text{LCP}[8] < \min(\text{LCP}[7,7]) = 7 \) and \( \text{LCP}[13], \text{LCP}[15] < \min(\text{LCP}[14,14]) = 6 \), respectively. Those two lcp-intervals are connected by a dotted arrow. The interval \( [6-1,7+1] \) of lcp-interval \( (6,7, abaabab) \) is contained in 3-th F-run. Then, we have FL(6) = 12. Thus, \( \text{LCP}[13] = \text{LCP}[6]-1, \text{LCP}[15] = \text{LCP}[8]-1 \), and \( \min(\text{LCP}[14,14]) = \min(\text{LCP}[7,7])-1 \) by the incremental relation on the LCP array, i.e., we can compute the lcp-interval \( (13,14, baabab) \) by applying the FL function to the lcp-interval \( (6,7, abaabab) \).

**Enumeration of type-4 lcp-intervals.** We enumerate type-4 lcp-intervals by modifying the enumeration algorithm for type-3 lcp-intervals. Any type-4 lcp-interval is contained in an L-run \( L_i \), and (i) the left boundary of the lcp-interval is \( \ell_{i,t}^- \) or (ii) the right boundary of the lcp-interval is \( \ell_{i,t}^+ \). We can find type-4 lcp-intervals satisfying the former condition by extending interval \([b'_{i,t}, b'_{i,t} + t - 1]\) as in \([b'_{i,1}, b'_{i,1}] \rightarrow [b'_{i,2}, b'_{i,2} + 1] \rightarrow \cdots \). Here, let \( b'_{i,t} \) be the smallest position in the LCP array such that (i) interval \([b'_{i,t}, b'_{i,t} + t - 1]\) contains position \( \ell_{i,t}^- \), and (ii) the length of the LCP of \( \text{SA}[b'_{i,t}, b'_{i,t} + t - 1] \) is no less than the length of the LCP of \( \text{SA}[b', b' + t - 1] \) for any \( b' \in [\ell_{i,t}^--t+2, \ell_{i,t}^+ + 1] \). Similarly, we can find type-4 lcp-intervals satisfying the latter condition by extending interval \([b''_{i,t}, b''_{i,t} + t - 1]\) as in \([b''_{i,1}, b''_{i,1}] \rightarrow [b''_{i,2}, b''_{i,2} + 1] \rightarrow \cdots \). In this case, let \( b''_{i,t} \) be the smallest position in the LCP
array such that (i) interval $[b'_{i,t}, b''_{i,t} + t - 1]$ contains position $\ell_{i}^t$, and (ii) the length of the LCP of $\text{SA}[b'_{i,t}, b''_{i,t} + t - 1]$ is no less than the length of the LCP of $\text{SA}[b', b' + t - 1]$ for any $b' \in [\ell_{i}^t - t + 1, \ell_{i}^{t+1}]$. Formally, the following lemma holds according to Lemma 10.

**Lemma 35.** Let $I_1^t$ and $I_2^t$ be the sets of type-4 lcp-intervals containing positions $\ell_{i}^t + 1$ and $\ell_{i}^{t+1}$, respectively. Then, $[b'_{i,t}, b''_{i,t} + t - 1] = [b, e]$ for any $i \in [1, r]$ and $(b, b + t - 1, d) \in I_1^t$. Similarly, $[b''_{i,t}, b''_{i+1,t} + t - 1] = [b, e]$ for any $i \in [1, r]$ and $(b, b + t - 1, d) \in I_2^t$.

Lemma 35 implies that we can enumerate type-4 lcp-intervals by using the strategy of the enumeration algorithm for type-3 lcp-intervals. Therefore, we can enumerate type-4 lcp-intervals in $O(n \log \log w(n/r))$ time and $O(r)$ working space by processing the RLBWT of $T$.

**Proof of Theorem 8.** We already showed that we can enumerate type-1, type-2, type-3, and type-4 lcp-intervals with an EPS in $O(n \log \log w(n/r))$ time and $O(r)$ words of working space by processing the RLBWT of string $T$. This immediately indicates that Theorem 8 holds.

**Appendix F: Detailed description for Section 5**

Here, we show the properties of sets $\Upsilon_1, \Upsilon_2, \ldots, \Upsilon_{n+1}$ described in Section 5. Let $B_i$ be the set of lcp-intervals with right boundary $i$ for any integer $i \in [1, n]$ (i.e., $B_i = \{b, e, d\}$ such that $(b, e, d) \in \mathcal{I}$ s.t. $e = i$). Let $I_{i,j}$ be the $j$-th lcp-interval in set $\Upsilon_i$, sorted in decreasing order of the depths of the lcp-intervals. Then, the following lemma holds.

**Lemma 36.** The following four statements hold. (i) Let $t_i$ be the largest integer such that $I_{i,t_i}$ is an lcp-interval with a depth of at least $\text{LCP}[i] + 1$ for any integer $i \in [1, n]$. Then, $\Upsilon_i \supseteq B_i$ and $B_i = \{I_{i,1}, I_{i,2}, \ldots, I_{i,t_i}\}$. (ii) For any integer $i \in [2, n + 1]$, let $d'$ be the depth of $I_{i-1}\{\text{B}_{i-1}\} = (b, e, d')$ (i.e., $d' = d$). Then $\Upsilon_{i-1} \setminus B_{i-1} = \{I_{i,2}, I_{i,3}, \ldots, I_{i,\Upsilon_{i}}\}$ (i.e., $\Upsilon_{i} = (\Upsilon_{i-1} \setminus B_{i-1}) \cup \{I_{i,1}\}$) if $\text{LCP}[i] > d'$; otherwise, $\Upsilon_{i} = \Upsilon_{i-1} \setminus B_{i-1}$. (iii) The left boundary and depth of $I_{i,1}$ are $b_{\text{last}}$ and $\text{LCP}[i]$, respectively, if $\Upsilon_{i} = (\Upsilon_{i-1} \setminus B_{i-1}) \cup \{I_{i,1}\}$. Here, $b_{\text{last}}$ is the left boundary of the last lcp-interval in set $B_{i-1}$ (i.e., $I_{i-1}\{\text{B}_{i-1}\}$) if $B_{i-1} \neq \emptyset$; otherwise, $b_{\text{last}} = i - 1$. (iv) $B_1 \cup B_2 \cup \cdots \cup B_n = \mathcal{I} \cup \{(0, n, 0)\}$.

**Proof.** (i) For any interval $(b, e, d) \in B_i$, $b < e = i \in [1, i]$, $e = i \in [i, n]$, and $\text{RmQ}(\text{LCP}, b + 1, e) \in [1, i]$ hold. Hence $\Upsilon_i \supseteq B_i$. Next, Corollary 9 indicates that the following two observations hold for any lcp-interval $(b, e, d)$: (a) $\text{LCP}[e + 1] < d$, and (b) $\text{LCP}[x] \geq d$ for any integer $x \in [b + 1, e]$. The right boundaries of the lcp-intervals $I_{i,1}, I_{i,2}, \ldots, I_{i,t_i}$ are $i$ by the above two observations. Therefore, $B_i = \{I_{i,1}, I_{i,2}, \ldots, I_{i,t_i}\}$, because $\Upsilon_i \supseteq B_i$.

(ii) Lemma 34(ii) holds if the following holds: (a) $\Upsilon_{i-1} \setminus B_{i-1} \subseteq \Upsilon_i$. Let $I'$ be the set of lcp-intervals such that each lcp-interval is contained in set $\Upsilon_i$ and it is not contained in set $\Upsilon_{i-1} \setminus B_{i-1}$. Then $|I'| = 1$ if $\text{LCP}[i] > d'$; otherwise $|I'| = 0$. (c) The lcp-interval in $I'$ is $I_{i,1}$ if $|I'| = 1$.

We prove the above three statements. (a) Any lcp-interval in set $\Upsilon_{i-1} \setminus B_{i-1}$ is a lcp-interval $(b, e, d)$ such that (1) $b \in [1, i - 1]$, (2) $e \in [i, n]$, and (3) $\text{RmQ}(\text{LCP}, b + 1, e) \in [1, i - 1]$. The lcp-interval also satisfies the following conditions: (1) $b \in [1, i]$, (2) $e \in [i, n]$, and (3) $\text{RmQ}(\text{LCP}, b + 1, e) \in [1, i]$. Therefore $\Upsilon_{i-1} \setminus B_{i-1} \subseteq \Upsilon_i$.

(b) Since $\Upsilon_{i-1} \setminus B_{i-1} \subseteq \Upsilon_i$, any lcp-interval $(b, e, d)$ in $I'$ satisfies the following conditions: (1) $b \in [1, i]$, (2) $e \in [i, n]$, and (3) $\text{RmQ}(\text{LCP}, b + 1, e) = i$. We consider the maximal interval $[b', e']$ satisfying the above three conditions, i.e., $b' \in [1, i]$ is the smallest integer such that $\text{RmQ}(\text{LCP}, b + 1, i) = i$, and $e' \in [i, n]$ is the largest integer such that $\text{RmQ}(\text{LCP}, i, e) = i$. If...
$\langle b', e', \text{LCP}[i] \rangle$ is an lcp-interval (i.e., $\text{LCP}[b'] < \text{LCP}[i]$), then it is contained in set $I'$ and $I' = \{ \langle b', e', \text{LCP}[i] \rangle \}$ holds, because two lcp-intervals are the same if they contain position $i$ and have the same depth. Otherwise (i.e., $\text{LCP}[b'] = \text{LCP}[i]$) $I'$ is the empty set. Next, if $\text{LCP}[i] > d'$, then the relation $\text{LCP}[b'] < \text{LCP}[i]$ holds by the following two observations: (1) $\text{LCP}[b'] < \text{LCP}[i]$ or $\text{LCP}[b'] = \text{LCP}[i]$ holds. (2) $\text{LCP}[b'] = \text{LCP}[i]$ if and only if there exists an lcp-interval with depth $d$ in $\mathcal{T}_{i-1} \setminus \mathcal{B}_{i-1}$. It means that $\text{LCP}[b'] = \text{LCP}[i]$ if and only if $\text{LCP}[i] \leq d'$. Similarly, the relation $\text{LCP}[b'] = \text{LCP}[i]$ holds if $\text{LCP}[i] \leq d'$. Therefore we obtain the observation (b).

(c) If $|I'| = 1$, then the depth of the lcp-interval in $I'$ is larger than $d'$ by the proof of the statement (b). Therefore the lcp-interval is $I_{i,1}$.

(iii) We consider the case such that the relation $\mathcal{T}_{i} = (\mathcal{T}_{i-1} \setminus \mathcal{B}_{i-1}) \cup \{ I_{i,1} \}$ holds. In this case, the depth of $I_{i,1}$ is $\text{LCP}[i]$ by the proof of Lemma 36 (ii), Next, we show that the left boundary of $I_{i,1}$ is $b_{\text{last}}$.

If $\mathcal{B}_{i-1} \neq \emptyset$, let $\langle b', e', d'' \rangle$ be the last interval in set $\mathcal{B}_{i-1}$ (i.e., $I_{i-1, \mathcal{B}_{i-1}} = \langle b', e', d'' \rangle$), then the left boundary of $I_{i,1}$ is $b''$ (i.e., $\text{LCP}[b''] < \text{LCP}[i]$ and $\min (\text{LCP}[b'' + 1..i-1]) > \text{LCP}[i]$) by the following observations: (1) The depth of the last interval is larger than $\text{LCP}[i]$ (i.e., $d'' > \text{LCP}[i]$), and hence $\min (\text{LCP}[b'' + 1..i-1]) > \text{LCP}[i]$. (2) The relation $\text{LCP}[b''] < \text{LCP}[i]$ holds by using the following two cases: (A) For the case $\text{LCP}[b''] = \text{LCP}[i]$, $\text{LCP}[i]$ holds by using the following two cases: (A) For the case $\text{LCP}[b''] = \text{LCP}[i]$.

Example for Section 5: Here, we present an example for set $\mathcal{T}_{i}$ in Figure 2 which illustrates the lcp-intervals for the example in Figure 1. The set of lcp-intervals is $\mathcal{I} = \{ \langle 1, 2, |a| \rangle, \langle 11, 16, |b| \rangle, \langle 5, 10, |ab| \rangle, \langle 12, 16, |ba| \rangle, \langle 6, 10, |aba| \rangle, \langle 15, 16, |bab| \rangle, \langle 2, 4, |aba| \rangle, \langle 9, 10, |abab| \rangle, \langle 3, 4, |aba| \rangle, \langle 12, 14, |baab| \rangle, \langle 6, 8, |abaab| \rangle, \langle 13, 14, |baabab| \rangle, \langle 7, 8, |baabab| \rangle \}$. Then, $\mathcal{T}_{1} = \{ \langle 0, 16, 0 \rangle \}, \mathcal{T}_{2} = \{ \langle 0, 16, 0 \rangle, \langle 2, 4, |aba| \rangle \}, \mathcal{T}_{3} = \{ \langle 0, 16, 0 \rangle, \langle 3, 4, |aba| \rangle \}, \mathcal{T}_{4} = \{ \langle 0, 16, 0 \rangle \}, \mathcal{T}_{5} = \{ \langle 0, 16, 0 \rangle, \langle 2, 10, |a| \rangle \}, \mathcal{T}_{6} = \{ \langle 0, 16, 0 \rangle, \langle 2, 10, |a| \rangle, \langle 5, 10, |ab| \rangle \}, \mathcal{T}_{7} = \{ \langle 0, 16, 0 \rangle, \langle 2, 10, |a| \rangle, \langle 5, 10, |ab| \rangle, \langle 6, 8, |abaab| \rangle \}, \mathcal{T}_{8} = \{ \langle 0, 16, 0 \rangle, \langle 2, 10, |a| \rangle, \langle 5, 10, |ab| \rangle, \langle 6, 8, |abaab| \rangle, \langle 7, 8, |baabab| \rangle \}, \mathcal{T}_{9} = \{ \langle 0, 16, 0 \rangle, \langle 2, 10, |a| \rangle, \langle 5, 10, |ab| \rangle, \langle 6, 10, |aba| \rangle \}, \mathcal{T}_{10} = \{ \langle 0, 16, 0 \rangle, \langle 2, 10, |a| \rangle, \langle 5, 10, |ab| \rangle, \langle 6, 10, |aba| \rangle \}, \mathcal{T}_{11} = \{ \langle 0, 16, 0 \rangle \}, \mathcal{T}_{12} = \{ \langle 0, 16, 0 \rangle, \langle 11, 16, |b| \rangle \}, \mathcal{T}_{13} = \{ \langle 0, 16, 0 \rangle, \langle 11, 16, |b| \rangle, \langle 12, 14, |baab| \rangle \}, \mathcal{T}_{14} = \{ \langle 0, 16, 0 \rangle, \langle 11, 16, |b| \rangle, \langle 12, 14, |baab| \rangle \}, \mathcal{T}_{15} = \{ \langle 0, 16, 0 \rangle, \langle 11, 16, |b| \rangle, \langle 12, 16, |ba|, \rangle \}, \mathcal{T}_{16} = \{ \langle 0, 16, 0 \rangle, \langle 11, 16, |b| \rangle, \langle 12, 16, |ba| \rangle, \langle 15, 16, |bab| \rangle \}, \text{and } \mathcal{T}_{17} = \emptyset$.

Details of Kasai et al.’s algorithm. Their algorithm reads the LCP array in left-to-right order. Before it reads $\text{LCP}[i]$ for any integer $i \in [2, n+1]$, stack $X$ stores a pair of the left boundary and depth of each lcp-interval in set $\mathcal{T}_{i-1}$ in decreasing order of their depths. When the algorithm reads lcp-value $\text{LCP}[i]$, it computes the lcp-intervals with right boundary $i-1$ (i.e., $I_{i-1,1}, I_{i-1,2}, \ldots, I_{i-1,i-1}$) by using stack $X$, Corollary 9 and Lemma 36. Afterward, it removes the lcp-intervals with right boundary $i-1$ from stack $X$ by using pop operations, and it pushes lcp-interval $I_{i,1}$ onto stack $X$ by using Lemma 36 if lcp-value $\text{LCP}[i]$
Enumeration of maximal repeats in BWT-runs Bounded Space

| i | SAI | LCP |
|---|-----|-----|
| 1 | 16  | 0   |
| 2 | 8   | 0   |
| 3 | 11  | 4   |
| 4 | 3   | 5   |
| 5 | 14  | 1   |
| 6 | 6   | 2   |
| 7 | 9   | 6   |
| 8 | 1   | 7   |
| 9 | 12  | 3   |
| 10| 4   | 4   |
| 11| 15  | 0   |
| 12| 7   | 1   |
| 13| 10  | 5   |
| 14| 2   | 6   |
| 15| 13  | 2   |
| 16| 5   | 3   |

$abaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababaababaabababa ab| is larger than the depth of $I_{i−1,t−1+1} = (b, e, d)$ (i.e., if $\text{LCP}[i] > d$). The total running time is obviously $O(n)$, because the number of executed push and pop operations on stack $X$ is $O(n)$.

Construction algorithm for our data structures. We already showed that a RLESA can be constructed in $O(n \log \log_w (n/r))$ time and $O(r)$ words of working space by processing the RLBWT of a string $T$. Initially, stack $X$ stores a pair of the left boundary and depth of each lcp-interval in set $\Upsilon_1$, and the set is $\{(0, n, 0)\}$, because $F[1] = \$ \neq F[2] \neq F[1]$, i.e., $\text{LCP}[1] = 0$. Therefore, we can construct the RLESA and stack $X$ in $O(n \log \log_w (n/r))$ time and $O(r)$ words of working space by processing the RLBWT of $T$. Finally, we obtain Theorem 13.

Appendix G: Notes for Section 6

We show that a substring $P$ of string $T$ is a maximal repeat if and only if (i) the sa-interval $(b, e, |P|)$ of $P$ is an lcp-interval and (ii) the sa-interval contains at least two distinct characters in permutation $L$, i.e., $[b + 1, e] \cap L^+ \neq \emptyset$. String $P$ is a right-maximal repeat if and only if its sa-interval is an lcp-interval, because any lcp-interval is an sa-interval that represents a right-maximal repeat. Then, string $P$ is left-maximal if and only if its sa-interval contains at least two distinct characters in the permutation, because the suffix $T[\text{SA}[x]..n]$ has string $P$ as a prefix for any integer $x \in [b, e]$, i.e., $T[\text{SA}[x]−1] = L[x]$, and $L[x]$ is the previous character of the suffix. Hence, $\text{Count}_T(cP) < \text{Count}_T(P)$ for any character $c \in \Sigma$ if and only if $L[b..e]$ contains at least two distinct characters. Therefore, the above statement holds.

Appendix H: Omitted experiments
Table 3 Execution time and memory for each method. Here, |RLBWT| is the number of runs in the RLBWT of a given benchmark string, i.e., $r$.

| Filename          | String length | |RLBWT| | Ours | OT | Execution time [sec] | Memory consumption [MB] |
|-------------------|----------------|------------------|------------------|------------------|------------------|------------------|------------------|
| dblp.xml.00001.1  | 104,857,600    | 172,489          | 87               | 27               | 0.30             | 10               | 2,561            | 0.004            |
| dblp.xml.00001.2  | 104,857,600    | 175,617          | 86               | 26               | 0.30             | 11               | 2,561            | 0.004            |
| dblp.xml.0001.1   | 104,857,600    | 240,535          | 99               | 27               | 0.26             | 14               | 2,561            | 0.005            |
| dblp.xml.0001.2   | 104,857,600    | 270,206          | 101              | 25               | 0.25             | 15               | 2,561            | 0.006            |
| sources.001.1     | 104,857,600    | 1,215,428        | 132              | 24               | 0.18             | 66               | 2,561            | 0.026            |
| dna.001.1         | 104,857,600    | 1,110,808        | 142              | 28               | 0.20             | 96               | 2,561            | 0.038            |
| proteins.001.1    | 104,857,600    | 1,278,201        | 135              | 30               | 0.22             | 80               | 2,561            | 0.033            |
| english.001.2     | 104,857,600    | 1,449,519        | 144              | 27               | 0.18             | 81               | 2,561            | 0.032            |
| einstein.de.txt   | 92,758,441     | 101,370          | 75               | 19               | 0.25             | 7                | 2,266            | 0.003            |
| einstein.en.txt   | 467,626,544    | 290,239          | 412              | 108              | 0.26             | 18               | 11,418           | 0.002            |
| world leaders     | 46,968,181     | 573,487          | 49               | 7                | 0.14             | 35               | 1,148            | 0.030            |
| influenza         | 154,808,555    | 3,022,822        | 230              | 36               | 0.15             | 162              | 3,781            | 0.043            |
| kernel            | 257,961,616    | 2,791,368        | 331              | 59               | 0.17             | 149              | 6,299            | 0.024            |
| cereus            | 461,289,644    | 11,574,641       | 1,123             | 117              | 0.10             | 576              | 11,263           | 0.031            |
| coreutils         | 205,281,776    | 4,084,400        | 359              | 46               | 0.13             | 237              | 5,013            | 0.049            |
| Escherichia Coti  | 112,689,515    | 15,044,487       | 467              | 30               | 0.06             | 754              | 2,752            | 0.274            |
| para              | 429,266,758    | 15,636,740       | 1,194             | 113              | 0.09             | 773              | 10,481           | 0.074            |