Abstract

The longest Lyndon substring of a string $T$ is the longest substring of $T$ which is a Lyndon word. $LLS(T)$ denotes the length of the longest Lyndon substring of a string $T$. In this paper, we consider computing $LLS(T')$ where $T'$ is an edited string formed from $T$. After $O(n)$ time and space preprocessing, our algorithm returns $LLS(T')$ in $O(\log n)$ time for any single character edit. We also consider a version of the problem with block edits, i.e., a substring of $T$ is replaced by a given string of length $l$. After $O(n)$ time and space preprocessing, our algorithm returns $LLS(T')$ in $O(l \log \sigma + \log n)$ time for any block edit where $\sigma$ is the number of distinct characters in $T$. We can modify our algorithm so as to output all the longest Lyndon substrings of $T'$ for both problems.

1 Introduction

A string $w$ is said to be a Lyndon word if $w$ is lexicographically smaller than any of its non-empty proper suffixes. An equivalent definition of Lyndon words is a string $w$ which is lexicographically smaller than any of its cyclic rotations. For instance, $aab$ is a Lyndon word, but its cyclic rotations $aba$ and $baa$ are not. Lyndon words have many important combinatorial properties in stringology, and have various applications in, e.g., musicology [7], bioinformatics [12], approximation algorithms [21], string matching [9, 6, 23], combinatorics on words [3, 15, 24], and free Lie algebras [20].
In stringology, Lyndon words are closely related to repetitive structures. A string \( w \) is said to be primitive if there do not exist an integer \( k \) and a string \( x \) such that \( w = x^k \). For any primitive string \( w \), \( w w \) contains one or two Lyndon words of length \( |w| \). Recently, Bannai et al. showed that the maximum number of maximal repetitions in a string of length \( n \) is less than \( n \) \[3\]. A key idea of their proof relied on the notion of the longest Lyndon word that starts at each position of the string. There are several recent studies on Lyndon trees and Lyndon arrays \[14, 10, 22\], which are closely related to longest Lyndon word because they represent all the longest Lyndon words in a given string. Although these structures take linear space and can be computed in linear time for an integer alphabet, they are not easy to maintain when allowing dynamic edit operations, since the structures may change a lot, even for a single character edit operation.

Although fully dynamic data structures are difficult in general, Amir et al. considered a new type of problem concerning the Longest Common Factor problem \[1\]. The goal there was to compute, given strings \( S \) and \( T \), the longest common factor of strings \( S \) and \( T \) where \( T' \) is a string which is obtained by a single character edit operation on \( T \). Their algorithm uses \( O(n \log^4 n) \) expected time and \( O(n \log^3 n) \) space for preprocessing, and then for any single character edit query, the LCF can be answered in \( O(\log^3 n) \) time. The important and interesting aspect of this problem setting is that all edit queries are on the original string \( T \), and the edited string is not maintained for subsequent edit queries.

In this paper, we consider the problem of computing the longest Lyndon substring after a single (character or block) edit operation. Let \( LLS(T) \) be the length of the longest Lyndon substring of a string \( T \) of length \( n \). We first consider the problem of computing \( LLS(T') \) for any single character edit (substitution, insertion, deletion) where \( T' \) is the string obtained after the edit operation on \( T \). We then extend the problem that asks for \( LLS(T') \) for any single block edit, where \( T' \) is the string obtained by replacing a substring of \( T \) with a given string of length \( l \) specified in the edit query. For single character edit operations, our algorithm runs in \( O(\log n) \) time for each edit query after \( O(n) \) time and space preprocessing. For block edit operations, our algorithm runs in \( O(l \log \sigma + \log n) \) time for each edit query after \( O(n) \) time and space preprocessing, where \( \sigma \) is the number of distinct characters in \( T \). We can modify our algorithm so as to output all the longest Lyndon substrings of \( T' \) for both problems.

The rest of this paper is organized as follows. In Section 2, we state some definitions and properties on strings. In Section 3, we propose our algorithm for a version of the problem with single character edits. In Section 4, we show our algorithm for a version of the problem with single block edits. Finally, we conclude in Section 5.

## 2 Preliminaries

### 2.1 Strings and model of computation

Let \( \Sigma \) be an ordered finite alphabet. An element of \( \Sigma^* \) is called a string. The length of a string \( w \) is denoted by \( |w| \). The empty string \( \varepsilon \) is a string of length 0. Let \( \Sigma^+ \) be the set of non-empty strings, i.e., \( \Sigma^+ = \Sigma^* - \{\varepsilon\} \). For a string \( w = xyz \), \( x \), \( y \) and \( z \) are called a prefix, substring, and suffix of \( w \), respectively. A prefix \( x \), a substring \( y \), and a suffix \( z \) of \( w \) are called a proper prefix, a proper substring, and a proper suffix of \( w \) if \( x \neq w \), \( y \neq w \), and \( z \neq w \), respectively. The \( i \)-th character of a string \( w \) is denoted by \( w[i] \), where \( 1 \leq i \leq |w| \). For a string \( w \) and two integers \( 1 \leq i \leq j \leq |w| \), let \( w[i..j] \) denote the substring of \( w \) that begins at position \( i \) and ends at position \( j \). For convenience, let \( w[i..j] = \varepsilon \) when \( i > j \). For any string \( w \) let \( w^1 = w \), and for any integer \( k \geq 2 \) let \( w^k = w w^{k-1} \), i.e., \( w^k \) is a \( k \)-times repetition of \( w \).
If character \( a \) is lexicographically smaller than another character \( b \), then we write \( a \prec b \).

For any strings \( x, y \), let \( lcp(x, y) \) be the length of the longest common prefix of \( x \) and \( y \). We write \( x \prec y \) if either \( x[lcp(x, y) + 1] \prec y[lcp(x, y) + 1] \) or \( x \) is a proper prefix of \( y \).

Our model of computation is the word RAM. We assume the computer word size is at least \( \lceil \log_2 |w| \rceil \), and hence, standard operations on values representing lengths and positions of string \( w \) can be manipulated in \( O(1) \) time. Space complexities will be determined by the number of computer words (not bits).

### 2.2 Lyndon words and Lyndon factorization of strings

A string \( w \) is said to be a Lyndon word, if \( w \) is lexicographically strictly smaller than all of its non-empty proper suffixes. The longest Lyndon substring of a string \( w \) is the longest substring of \( w \) which is a Lyndon word. \( LLS(w) \) denotes the length of the longest Lyndon substring of a string \( w \).

The Lyndon factorization of a string \( w \), denoted \( LF_w \), is the factorization \( \ell_1^{p_1}, \ldots, \ell_m^{p_m} \) of \( w \), such that each \( \ell_i \in \Sigma^+ \) is a Lyndon word, \( p_i \geq 1 \), and \( \ell_i \succ \ell_{i+1} \) for all \( 1 \leq i < m \). The size of \( LF_w \), denoted by \( |LF_w| \), is \( m \). \( LF_w \) can be represented by the sequence \((|\ell_1|, p_1), \ldots, (|\ell_m|, p_m)\) of integer pairs, where each pair \((|\ell_i|, p_i)\) represents the \( i \)-th Lyndon factor \( \ell_i^{p_i} \) of \( w \). Note that this representation requires \( O(m) \) space.

In the literature, the Lyndon factorization is sometimes defined to be a sequence of lexicographically non-increasing Lyndon words, namely, each Lyndon factor \( \ell^p \) is decomposed into a sequence of \( p \) \( \ell \)'s. In this paper, each Lyndon word \( \ell \) in the Lyndon factor \( \ell^p \) is called a decomposed Lyndon factor. We also refer to the factorization by decomposed Lyndon factors as the decomposed Lyndon factorization.

#### Lemma 1 ([13])

For any string \( w \), we can compute \( LF_w \) in \( O(|w|) \) time.

For any string \( w \), let \( LF_w = \ell_1^{p_1}, \ldots, \ell_m^{p_m} \). Let \( lfb_w(i) \) denote the position where the \( i \)-th Lyndon factor begins in \( w \), i.e., \( lfb_w(1) = 1 \) and \( lfb_w(i) = lfb_w(i-1) + |\ell_{i-1}^{p_{i-1}}| \) for any \( 2 \leq i \leq m \). For any \( 1 \leq i \leq m \), let \( lfs_w(i) = \ell_1^{p_1} \ell_{i+1}^{p_{i+1}} \cdots \ell_m^{p_m} \) and \( lfp_w(i) = \ell_1^{p_1} \ell_2^{p_2} \cdots \ell_i^{p_i} \). For convenience, let \( lfs_w(m + 1) = lfp_w(0) = \varepsilon \).

### 2.3 Lyndon tree

Given a Lyndon word \( w \) of length \(|w| > 1\), \((u, v)\) is the standard factorization [8, 19] of \( w \), if \( w = uv \) and \( v \) is the longest proper suffix of \( w \) that is a Lyndon word, or equivalently, the lexicographically smallest proper suffix of \( w \). It is well known that for the standard factorization \((u, v)\) of any Lyndon word \( w \), the factors \( u \) and \( v \) are also Lyndon words (e.g.[4]). The Lyndon tree of \( w \) is the full binary tree defined by recursive standard factorization of \( w \): \( w \) is the root of the Lyndon tree of \( w \), its left child is the root of the Lyndon tree of \( u \), and its right child is the root of the Lyndon tree of \( v \). The longest Lyndon word that starts at each position can be obtained from the Lyndon tree, due to the following lemma.

#### Lemma 2 (Lemma 5.4 of [3])

Let \( w \) be a Lyndon word with respect to \( \prec \). \( w[i..j] \) corresponds to a right node (or possibly the root) of the Lyndon tree with respect to \( \prec \) if and only if \( w[i..j] \) is the longest Lyndon word with respect to \( \prec \) that starts from \( i \).
2.4 Longest Common Extension

For any string $w$, the longest common extension query is, given two positions $1 \leq i, j \leq |w|$, to answer

$$LCE_w(i, j) = \max\{k \mid w[i..i+k-1] = w[j..j+k-1], i+k-1, j+k-1 \leq |w|\}.$$ 

By using suffix tree [26] of $w$ and the Lowest Common Ancestor query (also called Nearest Common Ancestor) [16] on the suffix tree, we can compute any LCE query in constant time after $O(|w|)$ time and space preprocessing.

3 Longest Lyndon substring after 1-edit

In this paper, we consider three edit operations, i.e., substitution, insertion and deletion. Let $T'$ be the string which was edited at a given position from a string $T$ of length $n$. A 1-edit longest Lyndon substring query (1-edit LLS) asks us to return LLS($T'$).

Firstly, we explain a basic property of LLS($T$) and give a naïve solution. The following lemma can be obtained by the definition of Lyndon factorization.

**Lemma 3.** For any string $T$, LLS($T$) is the length of the longest decomposed Lyndon factor of $LF_T$.

**Proof.** Let $x$ be the longest Lyndon substring of $T$. Suppose that $x$ is not a decomposed Lyndon factor of $LF_T$. If $x$ is a proper substring of a decomposed Lyndon factor $y$, then $y$ is a Lyndon substring which is longer than $x$. This implies that $x$ contains a boundary of consecutive decomposed factors. Let $x = stu$ where $s$ is a suffix of some decomposed Lyndon factor and $u$ is a prefix of some Lyndon decomposed factor ($s, u \in \Sigma^+, t \in \Sigma^*$). By the definition of Lyndon factorization, $s \geq u$ holds. Thus, $x$ is not a Lyndon word.

This fact can be obtained by Observation 3 of [14] in a different context. Due to this lemma, computing $LF_{T'}$ in $O(n)$ time by using Duval’s algorithm [13], we can compute LLS($T'$) in $O(n)$ time for each query.

**Example 4 (1-edit LLS).** Let $T = ababababab$. Since $LF_T = ab, (abc)^2, abac$, the longest Lyndon substring of $T$ is abac. (substitution) If the second $c$ is replaced by $b$, then the longest Lyndon substring of $T'$ is abababc since $LF_{T'} = ab, ababab, abac$. (insertion) If $a$ is inserted at the position preceded by the last $b$, then the longest Lyndon substrings of $T'$ are $ab, abc, aac$ since $LF_{T'} = ab, (abc)^2, ab, aac$. (deletion) If the second $a$ from the last is deleted, then the longest Lyndon substring of $T''$ is $abcabcbac$ since $LF_{T''} = ab, abcabcbac$.

Our goal of this paper is the following.

**Theorem 5.** After constructing an $O(n)$-space data structure of a given string $T$ in $O(n)$ time, we can compute LLS($T'$) in $O(\log n)$ time for each 1-edit query.

In this section, we explain our algorithm for substitution operations. We can solve our problem for the other two types of operations in a similar way.

More formally, for substitutions, let $T' = T[1..e - 1] \cdot \alpha \cdot T[e + 1..n] = T_p \cdot \alpha \cdot T_q$ where $\alpha \in \Sigma$. In our algorithm, we compute $LF_{T'}$ by concatenating $LF_{T_p}, LF_{\alpha}$, and $LF_{T_q}$. I et al. [18] showed an efficient algorithm to compute $LF_{uv}$ from $LF_u$ and $LF_v$ for any string $u, v$ (we will explain in Section 3.1). Hence, we can use this concatenation algorithm to compute $LF_{T'}$. The rest of this section is organized as follows. In Section 3.2, we show how to compute $LF_{T_p}$. In Section 3.3, we explain how to characterize $LF_{T_q}$. Finally, we summarize our algorithm in Section 3.4.
3.1 Overview of computing Lyndon factorization by concatenation

Here, we explain an overview of a Lyndon factorization algorithm which was proposed by I et al. [18]. This algorithm computes the Lyndon factorization of the concatenation of two strings by using their Lyndon factorizations.

For any string $u$ and $v$, let $LF_u = u_1^{p_1} \ldots u_m^{p_m}$ and $LF_v = v_1^{q_1} \ldots v_{m'}^{q_{m'}}$. Then, $LF_{uv}$ is characterized as follows.

**Lemma 6** ([2, 11]). $LF_{uv} = u_1^{p_1} \ldots u_{c'}^{p_{c'}} z^c v_1^{q_1} \ldots v_{c'm'+1}^{q_{c'm'+1}}$ for some $0 \leq c \leq m$, $1 \leq c' \leq m' + 1$ and $LF_{lfs_c(c+1)lfs_{c'(c'-1)}}(z^c) = z^k$.

This lemma implies that there is at most one new Lyndon factor $z^k$ (each of the other Lyndon factors of $LF_{uv}$ is also a Lyndon factor of $LF_u$ or $LF_v$). By a simple observation, we can consider three cases as follows.

- If $u_m > v_1$, then $LF_{uv} = LF_u, LF_v(z = 1)$.
- If $u_m = v_1$, then $LF_{uv} = u_1^{p_1} \ldots u_m^{p_m}, u_m^{p_m+q_1} v_2^{q_2} \ldots v_{m'}^{q_{m'}} (z = u_m = v_1)$.
- If $u_m < v_1$, then there exists the medial decomposed factor $z$ which begins in $u$ and ends in $v$.

In the first two cases, we can compute $LF_{uv}$ by one lexicographic string comparisons. In the third case, computing the medial decomposed Lyndon factor $z$ leads to computing $LF_{uv}$.

**Lemma 7** (Lemma 16 of [18]). Assume that $LF_u$ and $LF_v$ have been computed. Then, we can compute the medial decomposed Lyndon factor $z$ by $O(\log |LF_u| + \log |LF_v|)$ lexicographic string comparisons.

A key point of that result is that the medial decomposed Lyndon factor $z$ satisfies the following properties.

- The beginning position of $z$ is equal to $lfb_u(i)$ such that $lfs_u(1)v \succ \ldots \succ lfs_u(i)v \prec \ldots \prec lfs_v(m + 1)v$.
- The ending position of $z$ is equal to $lfb_v(j) − 1$ such that $lfs_u(1) \succ lfs_v(j − 1) \succ lfs_u(i)v \succ lfs_v(j) \succ \ldots \succ lfs_v(m' + 1)$.

From these monotonous conditions of suffixes which begin at the beginning position of some Lyndon factor, we can compute the beginning position and the ending position of $z$, respectively, by a binary search. After computing $z$, we can compute the Lyndon factor $z^k$ by checking whether $u_{i−1}$ ($v_j$) is equal to $z$ or not, respectively (i.e., $u_{i−1}$ and $v_j$ may be equal to $z$).

**Example 8.** Let $LF_u = \text{abb.}(ab)^2a$ and $LF_v = \text{bc.}ab\text{ababc}. \text{ab.}(a)^2$. Then, the medial decomposed Lyndon factor is $z = \text{abababc}$ obtained by Lyndon factors $(ab)^2, a, \text{bc.}$. Since the decomposed Lyndon factor succeeding to $z$ is also $\text{abababc}$, we need to pack them. Thus, $LF_{uv} = \text{abb.}(ab\text{ababc})^2, ab, (a)^2$.

In addition, we can modify the second property for the decomposed Lyndon factorization of $v$ by using the following property.

**Lemma 9.** Let $z = lfs_u(i)lfs_v(j)$ be the medial decomposed factor of $LF_{uv}$. Then, $lfs_u(j − 1) \succ v_{j−1}^{q_{j−1}} lfs_v(j) \succ \ldots \succ v_1^{q_1} lfs_v(j) \succ lfs_u(i)v \succ lfs_u(j)$ also holds.

3.2 Computing the Lyndon factorization of $T_p$

The following lemma is a well-known property of Lyndon words and Lyndon factorizations.
The Lyndon factorization of $\ell$. Let $w = x^kx'$. By Lemma 10, any prefix $w$ of a Lyndon word $\ell$ can be represented as $w = x^kx'$. For instance, $ababbababba = (abab)^2a$. Thus, we store $([x], |x|, |x'|) = (5, 2, 1)$ for this prefix of $\ell$.

**Lemma 10.** For any string $w$ which is a prefix of some Lyndon word, there exists a unique Lyndon word $x$ s.t. $w = x^kx'$ where $x'$ is a proper prefix of $x$ and an integer $k \geq 1$. Moreover, $LF_w = x^k, LF_{x'}$.

**Lemma 11.** We can compute $LF_{T_s}$ for any $1 \leq e \leq n$ in $O(\log n)$ time after $O(n)$ time and space preprocessing.

**Proof.** Let $LF_T = \ell_1^{p_1}, \ldots, \ell_m^{p_m}$. Assume that $|l_{b_T}(i) + |\ell_i^{k-1}| \leq e < |l_{b_T}(i) + |\ell_i^k|$, i.e., the edited position $e$ is in the $k$-th decomposed Lyndon factor of the $i$-th Lyndon factor. Then, $T_p = \ell_1^{p_1}, \ldots, \ell_i^{k-1}_{e'}, \ell_i^{k-1}$ where $\ell_i^e$ is a (possibly empty) proper prefix of $\ell_i$. It is easy to see that the Lyndon factorization of $\ell_i^{p_1}, \ldots, \ell_i^{k-1}$ is $\ell_i^{p_1}, \ldots, \ell_i^{k-1}, \ell_i^{k-1}$. On the other hand, from Lemma 10, $LF_{e'} = x^j, x'$ for some integer $j \geq 1$ and some Lyndon word $x$. Since $x'$ is also a prefix of Lyndon word $x$, we can consider $LF_{x'}$ in the same way. Because $x'$ must be shorter than half of $\ell_i$, it follows that $LF_{e'}$ consists of at most $\log |\ell_i|$ Lyndon factors. It is easy to see that $LF_{T_s} = \ell_1^{p_1}, \ldots, \ell_i^{k-1}, LF_{e'}$.

Based on this observation, we show our data structure for computing $LF_{T_s}$. We can compute $LF_T$ in $O(n)$ time and store it in $O(||LF_T||)$ space. Let $\ell$ be a decomposed Lyndon factor of $T$. For each prefix of $\ell$, we store a triple $([|x|, k, |x'|])$ based on Lemma 10. An example is shown in Figure 1. We note that all the triples for $T$ can be computed in $O(n)$ time by using Duval’s Lyndon factorization algorithm [13] (we can compute them together with the Lyndon factorization of $T$).

Now we explain how to compute $LF_{T_s}$ by using above data structures. The first $(i - 1)$ Lyndon factors of $LF_{T_s}$ are in $LF_T$. The $i$-th Lyndon factor of $LF_{T_s}$ is $\ell_i^{k-1}$ (changed only the exponent of $\ell_i$). Finally, we have to compute $LF_{e'}$. The form of $LF_{e'} = u^j, u'$ is stored as the $|\ell_i|$-th triple of $\ell_i$. If $u' = \varepsilon$ (i.e., the third entry of the triple is 0), then $u^j$ is a Lyndon factor of $LF_{T_s}$. Otherwise, since $u'$ is a prefix of $\ell_i$, the form of $LF_{u'}$ is stored as the $|u'|$-th triple of $\ell_i$. By repeating this recursively at most $\log |\ell_i|$ times, we can obtain $LF_{e'}$. Therefore, we can compute $LF_{T_s}$ in $O(\log n)$ time.

### 3.3 The Lyndon factorization of $T_s$ by Lyndon tree

In the previous subsection, we have computed the Lyndon factorization of a prefix of some Lyndon word, since the number of Lyndon factors of $T_p$ which are not in $LF_T$ is bounded by $\log n$. We also want to compute $LF_{T_s}$, but the size of the factorization can be large. Hence, we cannot compute $LF_{T_s}$ explicitly for each query in order to achieve an $O(\log n)$ time bound. To overcome this problem, we use the Lyndon tree of $T$, which can represent the Lyndon factorization of each suffix of $T$.

Let $LF_T = \ell_1^{p_1}, \ldots, \ell_m^{p_m}$. Assume that $|l_{b_T}(i) + |\ell_i^{k-1}| \leq e < |l_{b_T}(i) + |\ell_i^k|$, i.e., the edited position $e$ is in the $k$-th decomposed factor of the $i$-th Lyndon factor. Then, $T_s =$
time and space so that the left subtree. This preprocessing can also be done in $O(n)$ time and space.
3.4 Computing the longest Lyndon substring

In the rest of this section, we summarize our method.

Firstly, we compute \( LF_{T_e} \) based on Lemma 11 in \( O(\log n) \) time. From \( LF_{T_e} \) and \( \alpha \), we compute \( LF_{T_e, \alpha} \) by \( O(\log |LF_{T_e}|) \) lexicographic string comparisons by using Lemma 6 and 7. After that, we compute \( LF_{T'} \) from \( LF_{T_e, \alpha} \) and \( LF_{T_e} \). Let \( z \) be the medial decomposed Lyndon factor in this step. Since we know \( LF_{T_e, \alpha} \) and \( T_s \), we can compute the beginning position of \( z \) by \( O(\log |LF_{T_e, \alpha}|) \) lexicographic string comparisons on \( T' \). Then we compute the ending position of \( z \), by using Lemmas 7 and 9.

In order to compute the ending position, we access the necessary suffixes by considering the path \( P_e \), defined in Section 3.3, in the \( LTree(T) \). The key idea is that we can conduct a binary search on \( P_e \), and obtain \( z \) by \( O(\log h) \) lexicographic string comparisons on \( T' \). For any range of depths on \( P_e \), we can choose the middle node \( w \) in the range in constant time using Lemma 13. If the rightchild of \( w \) is on \( P_e \), we choose \( na(w) \) as \( w \). We then compare the suffix of \( T_s \) which begins at the beginning position of \( Rstr(w) \) and the suffix of \( T' \) which begins at the beginning position of \( z \), and recurse on the upper or lower half of the range depending on the result of the comparison.

Thus we can get \( LF_{T'} \) by \( O(\log n) \) string comparisons in total. The number of Lyndon factors of \( T' \) such that we should have explicitly is \( O(\log n) \) (new \( n \) factors in \( T_p \) and a new factor by concatenations). It is easy to see that we can compare lexicographic order between any substrings of \( T' \) by constant number of LCE queries on \( T \). Thus, we can compute \( LF_{T'} \) in \( O(\log n) \) time.

We have three candidates as the longest Lyndon substrings.

- Unchanged Lyndon factors at prefix.
- \( O(\log n) \) new Lyndon factors.
- Unchanged Lyndon factors at suffix.

Since we store \( O(\log n) \) new Lyndon factors explicitly, we can get the longest Lyndon factor in this part in \( O(\log n) \) time. To get the longest decomposed Lyndon factor in the first candidate, we precompute the rightmost longest Lyndon factor for each prefix of \( T \) which is a concatenation of Lyndon factors (i.e., for each \( \ell_1^p, \ldots, \ell_k^p \)). This can be computed in \( O(n) \) time and space. By using this information, we can see the length of longest Lyndon factor in the first part in constant time. For suffixes of \( T \), we precompute the same data structure as prefixes. Therefore, we obtain Theorem 5.

It is easy to see that we can return all the longest Lyndon substrings in unchanged part at prefix and at suffix in linear time w.r.t. the number of such factors. Then, we can get all the longest Lyndon substrings in \( T' \).

**Corollary 14.** After constructing an \( O(n) \)-space data structure of a given string \( T \) in \( O(n) \) time, we can compute all the longest Lyndon substrings of \( T' \) in \( O(\log n + occ) \) time for each 1-edit query where \( occ \) is the number of outputs.

4 Longest Lyndon substring after block edit

Here, we consider more general problem called 1-block-edit longest Lyndon substring query (1-block-edit LLS). Namely, a substring of \( T \) is replaced by a given string of length \( l \).

**Example 15** (1-block-edit LLS). Let \( T = acbacbcabacbac \). The longest Lyndon substring of \( T \) is \( abac \) since \( LF_T = acb, (abc)^2, abac \). When we are given an interval \([2, 3]\) of \( T \) and a string \( bac \), the longest Lyndon substring of \( T' \) is \( abacbcabacbac \) since \( LF_{T'} = abacbcabacbac \). When we are given \([8, 10]\) and an empty string, the longest Lyndon substring of \( T' \) is \( abac \) since \( LF_{T'} = acb, abc, abac \).
Theorem 16. After constructing an $O(n)$-space data structure of a given string $T$ in $O(n)$ time, we can compute $LLS(T')$ in $O(l \log \sigma + \log n)$ time for each 1-block-edit query.

This algorithm is almost similar to the 1-edit version. Let $\alpha$ be a given string of length $l$. Firstly, we need to compute $LF_\alpha$ in $O(l)$ time. After that we can concatenate three parts in the similar way. The key difference is that we conduct an additional $O(l \log \sigma)$ time and $O(l)$ space processing in order to compare any two substrings in $T'$ in constant time. Any comparisons on $T'$ can be separated to constant number of comparisons between

- a substring in $T$ and a substring in $T'$,
- a substring in $\alpha$ and a substring in $\alpha$,
- a substring in $T$ and a substring in $\alpha$.

The first one can be done by an LCE query on $T$. The second one can be done in constant time after constructing an LCE data structure for $\alpha$ in $O(l)$ time and space. Now we explain the last case. Assume that we have computed the suffix tree of $T$ in $O(n)$ time preprocessing. For each of suffixes $\alpha_i$ of $\alpha$, we compute the lowest node in the suffix tree which corresponds to some prefix of $\alpha_i$. This can be done in $O(l \log \sigma)$ time by using Ukkonen’s suffix tree construction algorithm [25]. Then we can compare a substring in $T$ and a substring in $\alpha$ by using LCA queries. Thus we can do any substring comparisons in constant time after constructing $O(l \log \sigma)$ time and space data structures. Therefore, we obtain Theorem 16.

In the similar way to Section 3, we can get the following.

Corollary 17. After constructing an $O(n)$-space data structure of a given string $T$ in $O(n)$ time, we can compute all the longest Lyndon substrings of $T'$ in $O(l \log \sigma + \log n + \text{occ})$ time for each 1-block-edit query where $\text{occ}$ is the number of outputs.

Remark. If $l$ is constant, we can compare the lexicographic order of any two substrings in $T'$ in constant time (by using constant number of LCE queries and constant number of character comparisons) without using suffix trees. Then the querying time of Theorem 16 turns out to be $O(\log n)$ time. Thus, this result includes Theorem 5.

5 Conclusion

We considered the problem of computing the longest Lyndon substring after 1-edit operation. We proposed an algorithm which uses $O(n)$ time and space so that for any single block edit query, the longest Lyndon substring can be answered in $O(l \log \sigma + \log n)$ time where $l$ is the length of a given query string and $\sigma$ is the number of distinct characters in $T$.

Our algorithm in this paper is almost the same for single characters edits and single block edits, and one of our interests is whether there is a more efficient solution at least for the case of single character edits.

References

Longest Lyndon Substring After Edit


