Lempel–Ziv Parsing is Harder Than Computing All Runs

Dmitry Kosolobov

Ural Federal University, Ekaterinburg, Russia

Abstract

We study the complexity of computing the Lempel–Ziv decomposition and the set of all runs (= maximal repetitions) in the decision tree and RAM models on a general ordered alphabet. It is known that both these problems can be solved by RAM algorithms in $O(n \log \sigma)$ time, where n is the length of the input string and σ is the number of distinct letters in it. We prove an $\Omega(n \log \sigma)$ lower bound on the number of comparisons required to construct the Lempel–Ziv decomposition and thereby conclude that a popular technique of the computation of runs using the Lempel–Ziv decomposition cannot achieve an $o(n \log \sigma)$ time bound. In contrast with this, we exhibit an $O(n)$ decision tree algorithm finding all runs and, moreover, we describe a RAM algorithm computing all runs in $O(n \log^{\frac{2}{3}} n)$ time. Thus, the runs problem is easier than the Lempel–Ziv parsing in both decision tree and RAM models. In view of these results, we conjecture that there exists a linear RAM algorithm finding all runs on a general ordered alphabet.

Keywords: Lempel–Ziv decomposition, general alphabet, maximal repetitions, runs, decision tree, lower bounds

1. Introduction

String repetitions called runs and the Lempel–Ziv decomposition are structures that are of a great importance for data compression and play a significant role in stringology. Recall that a run of a string is a nonextendable (with the same minimal period) substring whose minimal period is at most half of its length. The definition of the Lempel–Ziv decomposition is given below. We consider algorithms finding these structures in the frameworks of two models: the RAM model, which is currently the most popular model of computation, and the decision tree model, which is widely used to obtain lower bounds on the time complexity of various algorithms. We prove that any algorithm finding the Lempel–Ziv decomposition on a general ordered alphabet must perform $\Omega(n \log \sigma)$[1] comparisons in the worst case, where n is the length of the input string and σ is the number of distinct letters in it. Since until recently, the only known efficient way to find all runs of a string was to use the Lempel–Ziv decomposition, one might expect that there is a nontrivial lower bound in the decision tree model on the number of comparisons required by algorithms finding all runs. These expectations were also supported by the existence of such bound in the case of unordered alphabet. In this paper we obtain a somewhat surprising fact: in the decision tree model with an ordered alphabet, there exists a linear algorithm finding all runs. This can be interpreted as one cannot have lower bounds in the decision tree model for algorithms finding runs (a similar result for another problem is provided in [UAH76] for example). Moreover, we describe a RAM algorithm finding all runs in $O(n \log^{\frac{2}{3}} n)$ time on a general ordered alphabet. These results strongly support the conjecture that there is a linear RAM algorithm finding all runs on a general ordered alphabet. All parts of the present work were published in [Kos15c, Kos15a].

Email address: dkosolobov@mail.ru (Dmitry Kosolobov)

[1] Throughout the paper, log denotes the logarithm with the base 2.

The Lempel–Ziv decomposition [LZ76] is a basic technique for data compression. It has several modifications used in various compression schemes. The decomposition considered in this paper is used in LZ77-based compression methods. All known efficient algorithms for the computation of the Lempel–Ziv decomposition on a general ordered alphabet work in $O(n \log \sigma)$ time (see [Cro86, RPE81, FG89]), although all these algorithms are time and space consuming in practice. However, for the case of polynomially bounded integer alphabet, there are efficient linear algorithms [AKO04, CPS08, CIS08], space efficient algorithms [Kos15b, KKP14, OS08, Sta12, YBIT13], and space efficient algorithms in external memory [KKP14].

String repetitions are fundamental objects in both stringology and combinatorics on words. The notion of run, introduced by Main in [Mai89], allows to grasp the whole periodic structure of a given string in a relatively simple form. In the case of unordered alphabet, there are some limitations on the efficiency of algorithms finding periodicities; in particular, it is known [ML85] that any algorithm that decides whether an input string over a general unordered alphabet has at least one run, requires $\Omega(n \log n)$ comparisons in the worst case. Kolpakov and Kucherov [KK99] proved that any string of length n contains $O(n)$ runs and proposed a RAM algorithm finding all runs in linear time provided the Lempel–Ziv decomposition is given. Thereafter much work has been done on the analysis of runs (e.g., see [CIT11, CKR$^+$12, KPPK14, BCT12]) but until the recent paper [BII$^+$14], all efficient algorithms finding all runs of a string on a general ordered alphabet used the Lempel–Ziv decomposition as a basis. Bannai et al. [BII$^+$14] use a different method based on Lyndon decomposition but, unfortunately, their algorithm spends $O(n \log \sigma)$ time too. Clearly, due to the found lower bound, our linear decision tree algorithm finding all runs does not use the Lempel–Ziv decomposition yet our approach differs from that of [BII$^+$14]. However, we could improve the solution of [BII$^+$14] to obtain an $O(n \log^{\frac{2}{3}} n)$-time RAM algorithm finding all runs on a general ordered alphabet.

The paper is organized as follows. Section 2 contains some basic definitions used throughout the text. In Section 3 we give a lower bound on the number of comparisons required to construct the Lempel–Ziv decomposition. In Section 4 we present additional definitions and combinatorial facts that are necessary for Section 5, where we describe our linear decision tree algorithm finding all runs. In Section 6 we describe a RAM algorithm finding all runs in $O(n \log^{\frac{2}{3}} n)$ time. Finally, we conclude with some remarks in Section 7.

2. Preliminaries

A *string of length* n over the alphabet Σ is a map $\{1, 2, \ldots, n\} \mapsto \Sigma$, where n is referred to as the length of w, denoted by $|w|$. We write $w[i]$ for the ith letter of w and $w[i..j]$ for $w[i]w[i{+}1] \ldots w[j]$. Let $w[i..j]$ be the empty string for any $i > j$. A string u is a *substring* (or a *factor*) of w if $u = w[i..j]$ for some i and j. The pair (i, j) is not necessarily unique; we say that i specifies an *occurrence* of u in w. A string can have many occurrences in another string. A substring $w[1..j]$ [respectively, $w[i..n]$] is a *prefix* [respectively, *suffix*] of w. An integer p is a *period* of w if $0 < p < |w|$ and $w[i] = w[i{+}p]$ for $i = 1, \ldots, |w|{-}p$. For any numbers i and j, the set $\{k \in \mathbb{Z} \colon i \le k \le j\}$ (possibly empty) is denoted by $[i..j]$. Denote $[i..j) = [i..j] \setminus \{j\}$, $(i..j] = [i..j] \setminus \{i\}$, and $(i..j) = (i..j] \cap [i..j)$.

One of the computational models used in this paper is the *decision tree* model. Informally, a decision tree processes input strings of given *fixed* length and each path starting at the root of the tree represents the sequence of pairwise comparisons made between various letters in the string. The computation follows an appropriate path from the root to a leaf; each leaf represents a particular answer to the studied problem.

More formally, a decision tree processing strings of length n is a rooted directed ternary tree in which each interior vertex is labeled with an ordered pair (i, j) of integers, $1 \le i, j \le n$, and edges are labeled with the symbols "$<$", "$=$", "$>$" (see Fig. 1). The *height* of a decision tree is the number of edges in the longest path from the root to a leaf of the tree. Consider a path p connecting the root of a fixed decision tree to some vertex v. Let t be a string of length n. Suppose that p satisfies the following condition: it contains a vertex labeled with a pair (i, j) with the outgoing edge labeled with $<$ (resp., $>$, $=$) if and only if $t[i] < t[j]$ (resp., $t[i] > t[j]$, $t[i] = t[j]$). Then we say that the vertex v is *reachable* by the string t or the string t *reaches* the vertex v. Clearly, each string reaches exactly one leaf of any given tree.

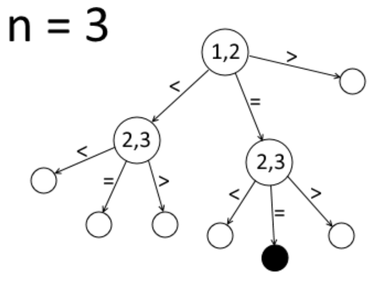


Figure 1: A decision tree of height 2 processing strings of length 3. The strings *aaa* and *bbb* reach the shaded vertex.

3. A Lower Bound on Algorithms Computing the Lempel–Ziv Decomposition

The Lempel–Ziv decomposition of a string t is the unique decomposition $t = t_1 t_2 \cdots t_k$, built by the following greedy procedure processing t from left to right:

- $t_1 = t[1]$;
- let $t_1 \cdots t_{i-1} = t[1..j]$; if $t[j{+}1]$ does not occur in $t[1..j]$, put $t_i = t[j{+}1]$; otherwise, t_i is the longest prefix of $t[j{+}1..n]$ that has an occurrence starting at some position $\le j$.

For example, the string *abababaabbbaaba* has the Lempel–Ziv decomposition *a.b.ababa.ab.bb.aab.a.* The substrings t_i are called the Lempel–Ziv factors.

Let t and t' be strings of length n. Suppose $t = t_1 t_2 \ldots t_k$ and $t' = t'_1 t'_2 \ldots t'_{k'}$ are their Lempel–Ziv decompositions. We say that the Lempel–Ziv decompositions of t and t' are equivalent if $k = k'$ and $|t_i| = |t'_i|$ for each $i \in [1..k]$. We say that a decision tree processing strings of length n finds the Lempel–Ziv decomposition if for any strings t and t' of length n such that t and t' reach the same leaf of the tree, the Lempel–Ziv decompositions of t and t' are equivalent.

Theorem 1. *The construction of the Lempel–Ziv decomposition for a string of length n with at most σ distinct letters requires $\Omega(n \log \sigma)$ comparisons of letters in the worst case.*

Proof. Let $a_1 < \ldots < a_\sigma$ be an alphabet. To obtain the lower bound, we construct a set of input strings of length n such that the construction of the Lempel–Ziv decomposition for these strings requires performing $\Theta(n)$ binary searches on the $\Theta(\sigma)$-element alphabet.

Without loss of generality, we assume that n and σ are even and $2 < \sigma < n/2$. Denote $s_1 = a_1 a_3 a_5 \ldots a_{\sigma-1}$, $s_2 = a_\sigma a_2 a_\sigma a_4 \ldots a_\sigma a_{\sigma-2} a_\sigma a_\sigma$, and $s = s_1 s_2$. We view s as a "dictionary" containing all letters a_i with even i. Note that $|s| = 1.5\sigma$. Consider a string t of the following form:

$$\begin{array}{l} a_\sigma a_{i_1} a_\sigma a_{i_2} \ldots a_\sigma a_{i_k} a_\sigma a_\sigma, \\ \text{where } k = \frac{n - 1.5\sigma - 2}{2} \text{ and } i_j \in [2..\sigma{-}2] \text{ is even for any } j \in [1..k] \ . \end{array} \tag{1}$$

Informally, the string t represents a sequence of queries to our "dictionary" s; any decision tree finding the Lempel–Ziv decomposition of the string st must identify each a_{i_j} of t with some letter of s. Otherwise, we can replace a_{i_j} with the letter $a_{i_j - 1}$ or $a_{i_j + 1}$ thus changing the Lempel–Ziv decomposition of the whole string; the details are provided below. Obviously, $|s| + |t| = n$ and there are $(\sigma/2 - 1)^k$ possible strings t of the form (1). Let us take a decision tree which computes the Lempel–Ziv decomposition for the strings of length n. It suffices to prove that each leaf of this tree is reachable by at most one string st with t of the form (1). Indeed, such decision tree has at least $(\sigma/2 - 1)^k$ leafs and the height of the tree is at least $\log_3((\sigma/2 - 1)^k) = k \log_3(\sigma/2 - 1) = \Omega(n \log \sigma)$.

Suppose to the contrary that some leaf of the decision tree is reachable by two distinct strings $r = st$ and $r' = st'$ such that t and t' are of the form (1); then for some $l \in [1..n]$, $r'[l] \ne r[l]$. Obviously $l = |s| + 2l'$ for some $l' \in [1..k]$ and therefore $r[l] = a_p$ for some even $p \in [2..\sigma{-}2]$. Suppose $r'[l] < r[l]$. Let $l_1 < \ldots < l_m$

be the set of all integers $l' > |s|$ such that for any string t_0 of the form (1), if the string $r_0 = st_0$ reaches the same leaf as the string r, then $r_0[l'] = r_0[l]$. Consider a string r'' that differs from r only in the letters $r''[l_1], \ldots, r''[l_m]$ and put $r''[l_1] = \ldots = r''[l_m] = a_{p-1}$. Let us first prove that the string r'' reaches the same leaf as r. Consider a vertex of the path connecting the root and the leaf reachable by r. Let the vertex be labeled with a pair (i, j). We have to prove that the comparison of $r''[i]$ and $r''[j]$ leads to the same result as the comparison of $r[i]$ and $r[j]$. The following cases are possible:

1. $i, j \neq l_q$ for all $q \in [1..m]$; then $r[i] = r''[i]$ and $r[j] = r''[j]$;
2. $i = l_q$ for some $q \in [1..m]$ and $r[i] < r[j]$; then since $r''[l_q] = a_{p-1} < a_p = r[l_q] = r[i]$ and $r[j] = r''[j]$, we obtain $r''[i] < r''[j]$;
3. $i = l_q$ for some $q \in [1..m]$ and $r[i] > r[j]$; then we have $j \neq p/2$ because $r[p/2] = r'[p/2] = a_{p-1} > r'[i]$ while $r'[i] > r'[j]$, and thus since $r[i] = a_p > r[j]$, we see that $a_{p-1} = r''[i] > r[j] = r''[j]$;
4. $i = l_q$ for some $q \in [1..m]$ and $r[i] = r[j]$; then, by definition of the set $\{l_1, \ldots, l_m\}$, $j = l_{q'}$ for some $q' \in [1..m]$ and $r''[i] = r''[j] = a_{p-1}$;
5. $j = l_q$ for some $q \in [1..m]$; this case is symmetric to the above cases.

Thus r'' reaches the same leaf as r. But the strings r and r'' have the different Lempel–Ziv decompositions: the Lempel–Ziv decomposition of r'' has one letter Lempel–Ziv factor a_{p-1} at position l_1 while r does not since $r[l_1{-}1..l_1{+}1] = a_\sigma a_p a_\sigma$ is a substring of $s = r[1..|s|]$. This contradicts to the fact that the analyzed tree computes the Lempel–Ziv decomposition. □

4. Runs

In this section we consider some combinatorial facts that will be useful in our decision tree algorithm described in the following section.

The *exponent* of a string t is the number $|t|/p$, where p is the minimal period of t. A *run* of a string t is a substring $t[i..j]$ of exponent at least 2 and such that both substrings $t[i{-}1..j]$ and $t[i..j{+}1]$, if defined, have strictly greater minimal periods than $t[i..j]$. A run whose exponent is greater than or equal to 3 is called a *cubic run*. For any fixed $d \geq 1$, a *d-short run* of a string t is a substring $t[i..j]$ which can be represented as xyx for nonempty strings x and y such that $0 < |y| \leq d$, $|xy|$ is the minimal period of $t[i..j]$, and both substrings $t[i{-}1..j]$ and $t[i..j{+}1]$, if defined, have strictly greater minimal periods.

Example. *The string $t = aabaabab$ has four runs $t[1..2] = aa$, $t[4..5] = aa$, $t[1..7] = aabaaba$, $t[5..8] = abab$ and one 1-short run $t[2..4] = aba$. The sum of exponents of all runs is equal to $2 + 2 + \frac{7}{3} + 2 \approx 8.33$.*

As it was proved in [KK99], the number of all runs is linear in the length of string. We use a stronger version of this fact.

Lemma 1 (see [BII+14, Theorem 9])**.** *The number of all runs in any string of length n is less than n.*

The following lemma is a straightforward corollary of [KPPK14, Lemma 1].

Lemma 2 (see [KPPK14])**.** *For fixed $d \geq 1$, any string of length n contains $O(n)$ d-short runs.*

We also need the following classical property of periodic strings.

Lemma 3 (see [FW65])**.** *Suppose a string w has periods p and q such that $p + q - \gcd(p, q) \leq |w|$; then $\gcd(p, q)$ is a period of w.*

Lemma 4. *Let t_1 and t_2 be substrings with the periods p_1 and p_2, respectively. Suppose t_1 and t_2 have a common substring of the length $p_1 + p_2 - \gcd(p_1, p_2)$ or greater; then t_1 and t_2 have the period $\gcd(p_1, p_2)$.*

Proof. It is immediate from Lemma 3. □

Unfortunately, the sum of the exponents of all runs with the minimal period p or greater in a string of length n is not equal to $O(\frac{n}{p})$ as the following example from [Kol12] shows: $(01)^k(10)^k$. Indeed, for any $p < 2k$, the string $(01)^k(10)^k$ contains at least $k - \lfloor p/2 \rfloor$ runs with the minimal period p or greater: $1(01)^i(10)^i1$ for $i \in [\lfloor p/2 \rfloor..k)$. However, it turns out that this property holds for cubic runs.

Lemma 5. *For any $p \geq 2$ and any string t of length n, the sum of the exponents of all cubic runs in t with the minimal period p or greater is less than $\frac{12n}{p}$.*

Proof. Consider a string t of length n. For any string u, $e(u)$ denotes the exponent of u and $p(u)$ denotes the minimal period of u. Denote by $\mathcal{R}$ the set of all cubic runs of t. Let $t_1 = t[i_1..j_1]$ and $t_2 = t[i_2..j_2]$ be distinct cubic runs such that $i_1 \leq i_2$. It follows from Lemma 4 that t_1 and t_2 cannot have a common substring of length $p(t_1) + p(t_2)$ or longer. Therefore, if $p(t_1)$ and $p(t_2)$ are sufficiently close, then positions i_1 and i_2 cannot be close. Let us describe it more precisely.

Let δ be a positive integer. Suppose $2\delta \leq p(t_1), p(t_2) \leq 3\delta$; then either $j_1 < i_2$ or $j_1 - i_2 < p(t_1) + p(t_2) \leq 2.5p(t_1)$. The latter easily implies $i_2 - i_1 > \delta$ and therefore $\rho = |\{u \in \mathcal{R}\colon 2\delta \leq p(u) \leq 3\delta\}| < \frac{n}{\delta}$. Moreover, we have $i_2 - i_1 \geq |t_1| - 2.5p(t_1) = (e(t_1) - 2.5)p(t_1) \geq (e(t_1) - 2.5)2\delta$ (see Fig. 2). Hence $\sum\limits_{u \in \mathcal{R}, 2\delta \leq p(u) \leq 3\delta} (e(u) - 2.5)2\delta \leq n$ and then $\sum\limits_{u \in \mathcal{R}, 2\delta \leq p(u) \leq 3\delta} e(u) \leq \frac{n}{2\delta} + 2.5\rho < \frac{3n}{\delta}$.

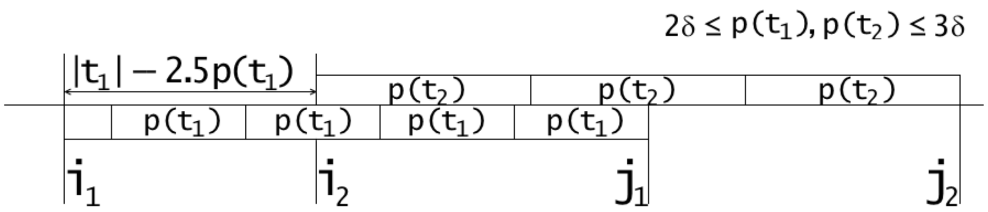


Figure 2: Overlapping cubic runs t_1 and t_2 such that $2\delta \leq p(t_1) < p(t_2) \leq 3\delta$.

Now it follows that if we have a sequence $\{\delta_i\}$ such that the union of the segments $[2\delta_i, 3\delta_i]$ covers the segment $[p, n]$, then the sum of the exponents of all cubic runs with the minimal period p or greater is less than $\sum_i \frac{3n}{\delta_i}$. Denote $\delta_i = (\frac{3}{2})^i$ and $k = \lfloor \log_{\frac{3}{2}} \frac{p}{2} \rfloor$. Evidently $\delta_k = (\frac{3}{2})^k \leq \frac{p}{2}$ and the union of the segments $\{[2\delta_i, 3\delta_i]\}_{i=k}^{\infty}$ covers $[p, n]$. Finally, we obtain $\sum\limits_{u \in \mathcal{R}, p(u) \geq p} e(u) < \sum_{i=k}^{\infty} \frac{3n}{\delta_i} = \sum_{i=k}^{\infty} 3n(\frac{2}{3})^i = 3n\frac{(2/3)^k}{1/3} \leq 9n\frac{4}{3p} = \frac{12n}{p}$. □

5. Linear Decision Tree Algorithm Finding All Runs

We say that a decision tree processing strings of length n *finds all runs with a given property P* if for any distinct strings t_1 and t_2 such that $|t_1| = |t_2| = n$ and t_1 and t_2 reach the same leaf of the tree, the substring $t_1[i..j]$ is a run satisfying P iff $t_2[i..j]$ is a run satisfying P for all $i, j \in [1..n]$.

We say that two decision trees processing strings of length n are equivalent if for each reachable leaf a of the first tree, there is a leaf b of the second tree such that any string t of length n reaches a iff t reaches b. The *basic height* of a decision tree is the minimal number k such that each path connecting the root and a leaf of the tree has at most k edges labeled with the symbols "<" and ">".

For a given positive integer p, we say that a run r of a string is *p-periodic* if $2p \leq |r|$ and p is a (not necessarily minimal) period of r. We say that a run is a *p-run* if it is q-periodic for some q such that q is a multiple of p. Note that any run is 1-run.

Example. *Let us describe a "naive" decision tree finding all p-runs in strings of length n. Denote by t the input string. Our tree simply compares $t[i]$ and $t[j]$ for all $i, j \in [1..n]$ such that $|i - j|$ is a multiple of p. The tree has the height $\sum_{i=1}^{\lfloor n/p \rfloor}(n - ip) = O(n^2/p)$ and the same basic height.*

Note that a decision tree algorithm finding runs does not report runs in the same way as RAM algorithms do. The algorithm only collects sufficient information to conclude where the runs are; once its knowledge of the structure of the input string becomes sufficient to find all runs without further comparisons of symbols, the algorithm stops and does not care about the processing of the obtained information. To simplify the construction of an efficient decision tree, we use the following lemma that enables us to estimate only the basic height of our tree.

Lemma 6. *Suppose the basic height of a decision tree processing strings of length n is k. Then there is an equivalent decision tree of the height $\leq k + n$.*

Proof. To construct the required decision tree of the height $\leq k + n$, we modify the given decision tree with the basic height k. First, we remove all unreachable vertices of this tree. Then, we contract each non-branching path into a single edge by removing all intermediate vertices and their outgoing edges. Indeed, the result of a comparison corresponding to such intermediate vertex is determined by the previous comparisons. So, it is straightforward that the resulting tree is equivalent to the original tree. Now it suffices to prove that there are at most $n-1$ edges labeled with the symbol "=" along any path connecting the root and some leaf.

Observe that if we perform $n-1$ comparisons on n elements and each comparison yields an equality, then either all elements are equal or the result of at least one comparison can be deduced by transitivity from other comparisons. Suppose a path connecting the root and some leaf has at least n edges labeled with the symbol "=". By the above observation, the path contains an edge labeled with "=" leaving a vertex labeled with (i, j) such that the equality of the ith and the jth letters of the input string follows by transitivity from the comparisons made earlier along this path. Then this vertex has only one reachable child. But this is impossible because all such vertices of the original tree were removed during the contraction step. This contradiction finishes the proof. □

Lemma 7. *For any integers n and p, there is a decision tree that finds all p-periodic runs in strings of length n and has basic height at most $2\lceil n/p\rceil$.*

Proof. Denote by t the input string. Note that any algorithm that processes t and is written on an imperative language or pseudocode can be transformed to a corresponding decision tree by an obvious procedure. So, our algorithm is as follows (the resulting decision tree contains only comparisons of letters of t):

1. assign $i \leftarrow 1$;
2. if $t[i] \neq t[i{+}p]$, then assign $i \leftarrow i + p$, $h \leftarrow \min\{i, n - p\}$ and for $i' = h{-}1, h{-}2, \ldots$, compare $t[i']$ and $t[i'{+}p]$ until $t[i'] \neq t[i'{+}p]$;
3. increment i and if $i \leq n - p$, jump to line 2.

Evidently, the algorithm performs at most $2\lceil n/p\rceil$ symbol comparisons yielding inequalities. Let us prove that the algorithm finds all p-periodic runs.

Let $t[j..k]$ be a p-periodic run. For the sake of simplicity, suppose $1 < j < k < n$. To discover this run, one must compare $t[l]$ and $t[l{+}p]$ for each $l \in [j{-}1..k{-}p{+}1]$. Let us show that the algorithm performs all these comparisons. Suppose, to the contrary, for some $l \in [j{-}1..k{-}p{+}1]$, the algorithm does not compare $t[l]$ and $t[l{+}p]$. Then for some i_0 such that $i_0 < l < i_0 + p$, the algorithm detects that $t[i_0] \neq t[i_0{+}p]$ and "jumps" over l by assigning $i = i_0 + p$ at line 2. Obviously $i_0 < j$. Then $h = \min\{i_0 + p, n - p\} < k$ and hence for each $i' = h{-}1, h{-}2, \ldots, j{-}1$, the algorithm compares $t[i']$ and $t[i'{+}p]$. Since $j - 1 \leq l < i_0 + p$, $t[l]$ and $t[l{+}p]$ are compared, contradicting to our assumption. □

Theorem 2. *There is a constant c such that for any integer n, there exists a decision tree of height at most cn that finds all runs in strings of length n.*

Proof. By Lemma 6, it is sufficient to build a decision tree with linear basic height. So, below we count only the comparisons yielding inequalities and refer to them as *inequality comparisons.* In fact we prove the following more general fact: for a given string t of length n and a positive integer p, we find all p-runs performing $O(n/p)$ inequality comparisons. To find all runs of a string, we simply put $p = 1$.

Let us outline the plot of the proof. Firstly, we briefly describe all steps of our decision tree algorithm finding all p-runs. Secondly, we discuss each of these steps: its correctness and the number of inequality comparisons performed; this is the largest part of the proof. Finally, we estimate the overall number of inequality comparisons; the main difficulty of the estimation is in the recursive nature of our algorithm.

The algorithm consists of five steps. Each step finds p-runs of t with a given property. Let us choose a positive integer constant $d \geq 2$ (the exact value is defined below.) The algorithm is roughly as follows:

1. find in a straightforward manner all p-runs having periods $\leq dp$;
2. using the information from step 1, build a new string t' of length n/p such that periodic factors of t and t' are strongly related to each other;
3. find p-runs of t related to periodic factors of t' with exponents less than 3;
4. find p-runs of t related to periodic factors of t' with periods less than d;
5. find p-runs of t related to other periodic factors of t' by calling steps 1–5 recursively for some substrings of t.

Step 1. Initially, we split the string t into n/p contiguous blocks of length p (if n is not a multiple of p, we pad t on the right to the required length with a special symbol which is less than all other symbols.) For each $i \in [1..n/p]$ and $j \in [1..d]$, we denote by $m_{i,j}$ the minimal $k \in [1..p]$ such that $t[(i-1)p+k] \neq t[(i-1)p+k+jp]$ and we put $m_{i,j} = -1$ if $ip+jp > n$ or there is no such k. To compute $m_{i,j}$, we simply compare $t[(i-1)p+k]$ and $t[(i-1)p+k+jp]$ for $k = 1, 2, \ldots, p$ until $t[(i-1)p+k] \neq t[(i-1)p+k+jp]$.

Example. *Let $t = bbba \cdot aada \cdot aaaa \cdot aaaa \cdot aada \cdot aaaa \cdot aaab \cdot bbbb \cdot bbbb$, $p = 4$, $d = 2$. The following table contains $m_{i,j}$ for $j = 1, 2$:*

i	1	2	3	4	5	6	7	8	9
$t[(i-1)p+1..ip]$	$bbba$	$aada$	$aaaa$	$aaaa$	$aada$	$aaaa$	$aaab$	$bbbb$	$bbbb$
$m_{i,1}, m_{i,2}$	$1, 1$	$3, 3$	$-1, 3$	$3, -1$	$3, 3$	$4, 1$	$1, 1$	$-1, -1$	$-1, -1$

To compute a particular value of $m_{i,j}$, one needs at most one inequality comparison (zero inequality comparisons if the computed value is -1.) Further, for each $i \in [1..n/p]$ and $j \in [1..d]$, we compare $t[ip-k]$ and $t[ip-k+jp]$ (if defined) for $k = 0, 1, \ldots, p-1$ until $t[ip-k] \neq t[ip-k+jp]$; similar to the above computation of $m_{i,j}$, this procedure performs at most one inequality comparison for any given i and j. Hence, the total number of inequality comparisons is at most $2dn/p$. Once these comparisons are made, all pq-periodic runs in the input string are determined for all $q \in [1..d]$.

Step 2. Now we build an auxiliary structure induced by $m_{i,j}$ on the string t. In this step, no comparisons are performed; we just establish some combinatorial properties required for further steps. We make use of the function:

$$\operatorname{sgn}(a, b) = \begin{cases} -1, & a < b, \\ 0, & a = b, \\ 1, & a > b \end{cases} .$$

We create a new string t' of length n/p. The alphabet of this string can be taken arbitrarily, we just describe which letters of t' coincide and which do not. For each $i_1, i_2 \in [1..n/p]$, $t'[i_1] = t'[i_2]$ iff for each $j \in [1..d)$, either $m_{i_1,j} = m_{i_2,j} = -1$ or the following conditions hold simultaneously:

$$\begin{aligned}
&m_{i_1,j} \neq -1, m_{i_2,j} \neq -1, \\
&m_{i_1,j} = m_{i_2,j}, \\
&\operatorname{sgn}(t[(i_1-1)p+m_{i_1,j}], t[(i_1-1)p+m_{i_1,j}+jp]) = \operatorname{sgn}(t[(i_2-1)p+m_{i_2}], t[(i_2-1)p+m_{i_2,j}+jp]) \ .
\end{aligned}$$

Note that the status of each of these conditions is known from step 1. Also note that the values $m_{i,d}$ are not used in the definition of t'; we computed them only to find all dp-periodic p-runs.

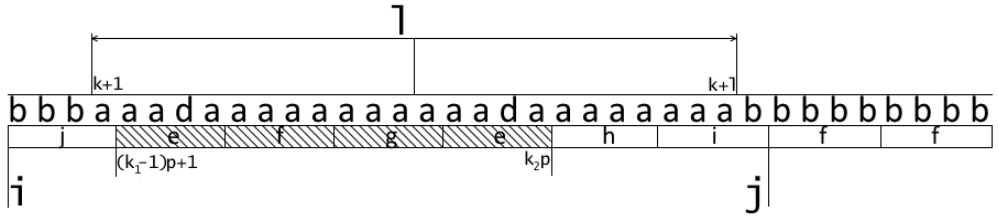
Example (continued). *Denote* $s_i = \mathrm{sgn}(t[(i{-}1)p{+}m_{i,1}], t[(i{-}1)p{+}m_{i,1}{+}p])$. *Let* $\{e, f, g, h, i, j\}$ *be a new alphabet for the string* t'. *The following table contains* $m_{i,1}$, s_i, *and* t':

i	1	2	3	4	5	6	7	8	9
$t[(i{-}1)p{+}1..ip]$	*bbba*	*aada*	*aaaa*	*aaaa*	*aada*	*aaaa*	*aaab*	*bbbb*	*bbbb*
$m_{i,1}$	1	3	-1	3	3	4	1	-1	-1
s_i	1	1	$-$	-1	1	-1	-1	$-$	$-$
$t'[i]$	j	e	f	g	e	h	i	f	f

If t contains two identical sequences of d blocks each, i.e., $t[(i_1{-}1)p{+}1..(i_1{-}1{+}d)p] = t[(i_2{-}1)p{+}1..(i_2{-}1{+}d)p]$ for some i_1, i_2, then $m_{i_1,j} = m_{i_2,j}$ for each $j \in [1..d)$ and hence $t'[i_1] = t'[i_2]$. This is why $t'[2] = t'[5]$ in Example 5. On the other hand, equal letters in t' may correspond to different sequences of blocks in t, like the letters $t'[3] = t'[8]$ in Example 5. The latter property makes the subsequent argument more involved but allows us to keep the number of inequality comparisons linear. Let us point out the relations between periodic factors of t and t'.

Let for some $q > d$, $t[k{+}1..k{+}l]$ be a pq-periodic p-run, i.e., $t[k{+}1..k{+}l]$ is a p-run that is not found in step 1. Denote $k' = \lceil k/p \rceil$. Since $t[k{+}1..k{+}l]$ is pq-periodic, t' has some periodicity in the corresponding substring, namely, $u = t'[k'{+}1..k'{+}\lfloor l/p \rfloor{-}d]$ has the period q (see example below). Let $t'[k_1..k_2]$ be the largest substring of t' containing u and having the period q. Since $2q \le \lfloor l/p \rfloor = |u| + d$, $t'[k_1..k_2]$ is either a d-short run with the minimal period q or a run whose minimal period divides q.

Example (continued). *Consider Fig. 3. Let* $k = 3$, $l = 24$. *The string* $t[k{+}1..k{+}l] = a \cdot aada \cdot aaaa \cdot aaaa \cdot aada \cdot aaaa \cdot aaa$ *is a p-run with the minimal period* $pq = 12$ *(here* $q = 3 > 2 = d$*). Denote* $k' = \lceil k/p \rceil = 1$, $k_1 = 2$, *and* $k_2 = 5$. *The string* $t'[k'{+}1..k'{+}\lfloor l/p \rfloor{-}d] = t'[k_1..k_2] = t'[2..5] = efge$ *is a d-short run of* t' *with the minimal period* $q = 3$.



l
k+1
k+l
b b b a a a d a a a a a a a a a a a d a a a a a a a a a b b b b b b b b
j
e
f
g
e
h
i
f
f
(k1-1)p+1
k2p
i
j

Figure 3: A p-run corresponding to d-short run $t'[k_1..k_2] = efge$, where $k_1 = 2$, $k_2 = 5$, $p = 4$, $d = 2$, $q = 3$, $k = 3$, $l = 2pq = 24$, $i = (k_1{-}2)p{+}1 = 1$, $j = (k_2{+}d)p = 28$.

Conversely, given a run or d-short run $t'[k_1..k_2]$ with the minimal period q, we say that a p-run $t[k{+}1..k{+}l]$ *corresponds to* $t'[k_1..k_2]$ (or $t[k{+}1..k{+}l]$ is a p-run *corresponding to* $t'[k_1..k_2]$) if $t[k{+}1..k{+}l]$ is, for some integer r, rpq-periodic and $t'[k'{+}1..k'{+}\lfloor l/p \rfloor{-}d]$, where $k' = \lceil k/p \rceil$, is a substring of $t'[k_1..k_2]$ (see Fig. 3 and Example 5).

The above observation shows that each p-run of t that is not found in step 1 corresponds to some run or d-short run of t'. Let us describe the substring that must contain all p-runs of t corresponding to a given run or d-short run $t'[k_1..k_2]$. Denote $i = (k_1 - 2)p + 1$ and $j = (k_2 + d)p$. Now it is easy to see that if $t[k{+}1..k{+}l]$ is a p-run corresponding to $t'[k_1..k_2]$, then $t[k{+}1..k{+}l]$ is a substring of $t[i..j]$.

Example (continued). *For* $k = 3$ *and* $l = 24$, *the string* $t[k{+}1..k{+}l] = a \cdot aada \cdot aaaa \cdot aaaa \cdot aada \cdot aaaa \cdot aaa$ *is a p-run corresponding to* $t'[k_1..k_2] = efge$, *where* $k_1 = 2$, $k_2 = 5$. *Indeed, the string* $t'[k'{+}1..k'{+}\lfloor l/p \rfloor{-}d] = t'[2..5]$, *for* $k' = \lceil k/p \rceil = 1$, *is a substring of* $t'[k_1..k_2]$. *Denote* $i = (k_1 - 2)p + 1 = 1$, $j = (k_2 + d)p = 28$. *Observe that* $t[k{+}1..k{+}l] = t[4..27]$ *is a substring of* $t[i..j] = t[1..28]$.

It is possible that there is another p-run of t corresponding to the string $t'[k_1..k_2]$. Consider the following example.

Example. *Let* $t = fabcdedabcdedaaifjfaaifjff$, $p = 2$, $d = 2$. *Denote* $s_i = \mathrm{sgn}(t[(i{-}1)p{+}m_{i,1}], t[(i{-}1)p{+}m_{i,1}{+}p])$. *Let* $\{w, x, y, z\}$ *be a new alphabet for the string* t'. *The following table contains* $m_{i,1}$, s_i, *and* t'*:*

i	1	2	3	4	5	6	7	8	9	10	11	12	13
$t[(i{-}1)p{+}1..ip]$	fa	bc	de	da	bc	de	da	ai	fj	fa	ai	fj	ff
$m_{i,1}$	1	1	2	1	1	2	1	1	2	1	1	2	-1
s_i	1	-1	1	1	-1	1	1	-1	1	1	-1	1	$-$
$t'[i]$	x	y	z	x	y	z	x	y	z	x	y	z	w

Note that p*-runs* $t[2..13] = abcded \cdot abcded$ *and* $t[14..25] = aaifjf \cdot aaifjf$ *correspond to the same* p*-run of* t'*, namely,* $t'[1..12] = xyz \cdot xyz \cdot xyz \cdot xyz$.

Thus to find for all $q > d$ all pq-periodic p-runs of t, we must process all runs and d-short runs of t'.

Step 3. Consider a noncubic run $t'[k_1..k_2]$. Let q be its minimal period. Denote $i = (k_1 - 2)p + 1$ and $j = (k_2 + d)p$. The above analysis shows that any p-run of t corresponding to $t'[k_1..k_2]$ is a p'-periodic run of $t[i..j]$ for some $p' = pq, 2pq, \dots, lpq$, where $l = \lfloor (j - i + 1)/(2pq) \rfloor$. Since $(k_2 - k_1 + 1)/q < 3$, we have $l = \lfloor (k_2 - k_1 + 2)/(2q) + d/(2q) \rfloor = O(d)$. Hence to find all p-runs of $t[i..j]$, it suffices to find for each $p' = pq, 2pq, \dots, lpq$ all p'-periodic runs of $t[i..j]$ using Lemma 7. Thus the processing performs $O(l(j - i + 1)/pq) = O(d^2) = O(1)$ inequality comparisons. Analogously we process d-short runs of t'. Therefore, by Lemmas 1 and 2, only $O(|t'|) = O(n/p)$ inequality comparisons are required to process all d-short runs and noncubic runs of t'.

Now it suffices to find all p-runs of t corresponding to cubic runs of t'.

Step 4. Let $t'[k_1..k_2]$ be a cubic run with the minimal period q. In this step we consider the case $q < d$. It turns out that such small-periodic substrings of t' correspond to substrings in t that are either periodic and discovered at step 1, or aperiodic. Therefore this step does not include any comparisons. The precise explanation follows.

Suppose that $m_{k,q} = -1$ for all $k \in [k_1..k_1{+}q)$. Then $m_{k,q} = -1$ for all $k = k_1, \dots, k_2$ by periodicity of $t'[k_1..k_2]$. Therefore by the definition of $m_{k,q}$, we have $t[k] = t[k{+}pq]$ for all $k \in [(k_1{-}1)p{+}1..k_2p]$. Hence the substring $t[(k_1{-}1)p{+}1..k_2p{+}pq]$ has the period pq. Now it follows from Lemma 4 that any p-run of t corresponding to $t'[k_1..k_2]$ is pq-periodic and therefore was found in step 1 because $pq < dp$.

Suppose that $m_{k,q} \neq -1$ for some $k \in [k_1..k_1{+}q)$. Denote $s = (k-1)p + m_{k,q}$, $l = \lfloor (k_2p - s)/pq \rfloor + 1$. Let $r \in [1..l]$. Since $t'[k] = t'[k{+}rq]$, we have $m_{k,q} = m_{k+rq,q}$ and $\mathrm{sgn}(t[s], t[s{+}pq]) = \mathrm{sgn}(t[s{+}rpq], t[s{+}(r{+}1)pq])$ (see Fig. 4). Therefore, one of the following sequences of inequalities holds:

$$\begin{array}{l} t[s] < t[s{+}pq] < t[s{+}2pq] < \dots < t[s{+}lpq], \\ t[s] > t[s{+}pq] > t[s{+}2pq] > \dots > t[s{+}lpq] \; . \end{array} \qquad (2)$$

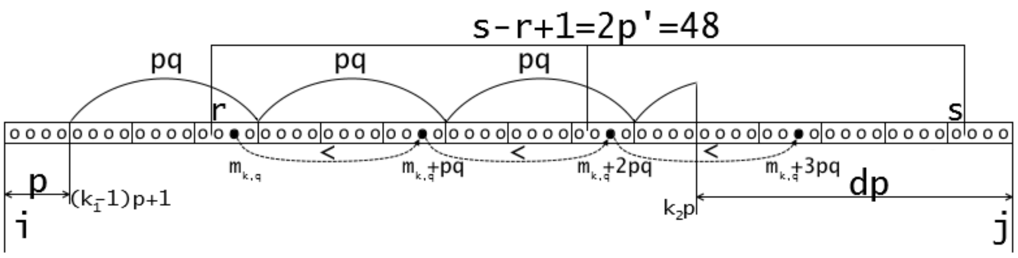


Figure 4: A cubic run of t' with the shortest period $q = 3 < d = 5$, where $p = 4$, $k_1 = 2$, $k_2 = 11$, $k = 4$, $m_{k,q} = 3$, $l = 3$, $p' = 2pq = 24$.

Let p' be a multiple of pq such that $p' > dp$. Now it suffices to show that due to the found “aperiodic chain”, there are no p'-periodic p-runs of t corresponding to $t'[k_1..k_2]$.

Suppose, to the contrary, $t[r..s]$ is a p'-periodic p-run corresponding to $t'[k_1..k_2]$ (see Fig. 4). Denote $r' = \lceil (r-1)/p \rceil$ and $l' = \lfloor (s-r+1)/p \rfloor$. By the definition of corresponding p-runs, $u = t'[r'{+}1..r'{+}l'{-}d]$ is a substring of $t'[k_1..k_2]$. Since $s-r+1 \geq 2p'$ and $p' > dp$, we have $|u| = l'-d \geq 2p'/p-d > p'/p$. Therefore, $r \leq r'p{+}m_{r'+1,q} < r'p{+}m_{r'+1,q}{+}p' \leq s$ and the inequalities (2) imply $t[r'p{+}m_{r'+1,q}] \neq t[r'p{+}m_{r'+1,q}{+}p']$, a contradiction.

Step 5. Let $t'[k_1..k_2]$ be a cubic run with the minimal period q such that $q \geq d$. Denote $i = (k_1-2)p+1$ and $j = (k_2+d)p$. To find all p-runs corresponding to the run $t'[k_1..k_2]$, we make a recursive call executing steps 1–5 again with new parameters $n = j-i+1$, $p = pq$, and $t = t[i..j]$.

After the analysis of all cubic runs of t', all p-runs of t are found and the algorithm stops. Now it suffices to estimate the number of inequality comparisons performed during any run of the described algorithm.

Time analysis. As shown above, steps 1–4 require $O(n/p)$ inequality comparisons. Let $t'[i_1..j_1], \ldots, t'[i_k..j_k]$ be the set of all cubic runs of t' with the minimal period d or greater. For $l \in [1..k]$, denote by q_l the minimal period of $t'[i_l..j_l]$ and denote $n_l = j_l - i_l + 1$. Let $T(n,p)$ be the number of inequality comparisons required by the algorithm to find all p-runs in a string of length n. Then $T(n,p)$ can be computed by the following formula:

$$T(n,p) = O\left(n/p\right) + T\left((n_1+d+1)p, pq_1\right) + \ldots + T\left((n_k+d+1)p, pq_k\right) \ .$$

For $l \in [1..k]$, the number n_l/q_l is, by definition, the exponent of $t'[i_l..j_l]$. It follows from Lemma 5 that the sum of the exponents of all cubic runs of t' with the shortest period d or larger is less than $\frac{12n}{dp}$. Note that for any $l \in [1..k]$, $n_l \geq 3q_l \geq 3d$ and therefore $n_l + d + 1 < 2n_l$. Thus assuming $d = 48$, we obtain $\frac{(n_1+d+1)p}{pq_1} + \ldots + \frac{(n_k+d+1)p}{pq_k} < \frac{2n_1}{q_1} + \ldots + \frac{2n_k}{q_k} \leq \frac{24n}{dp} = \frac{n}{2p}$. Finally, we have $T(n,p) = O(\frac{n}{2^0 p} + \frac{n}{2^1 p} + \frac{n}{2^2 p} + \ldots) = O(n/p)$. The reference to Lemma 6 ends the proof. □

6. RAM Algorithm Finding All Runs

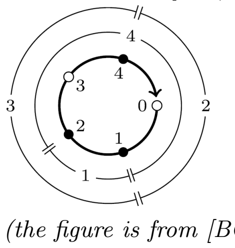
Hereafter, w denotes the input string of length n for our RAM algorithm. It turns out that the problem of the computation of all runs reduces to another well-known problem. In the *longest common extension (LCE)* problem one has the queries $LCE(i,j)$ returning for given positions i and j of w the length of the longest common prefix of the suffixes $w[i..n]$ and $w[j..n]$. It is well known that one can perform the LCE queries in constant time after a preprocessing of w requiring $O(n \log \sigma)$ time, where σ is the number of distinct letters in w (e.g., see [HT84]). It appears that the time consumed by the LCE queries is dominating in the algorithm of [BII$^+$14]; namely, one can easily prove the following lemma.

Lemma 8 (see [BII$^+$14, Alg. 1 and Sect. 4.2])**.** *Suppose we can compute any sequence of $O(n)$ LCE queries on w in $O(f(n))$ time for some function $f(n)$; then we can find all runs of w in $O(n + f(n))$ time.*

In what follows we describe an algorithm that computes $O(n)$ LCE queries in $O(n \log^{\frac{2}{3}} n)$ time and thus obtain the required time bound using Lemma 8. The key notion in our construction is a *difference cover.* Let $k \in \mathbb{N}$. A set $D \subset [0..k)$ is called a difference cover of $[0..k)$ if for any $x \in [0..k)$, there exist $y, z \in D$ such that $y - z \equiv x \pmod{k}$. Clearly $|D| \geq \sqrt{k}$. Conversely, for any $k \in \mathbb{N}$, there is a difference cover of $[0..k)$ with $O(\sqrt{k})$ elements and it can be constructed in $O(k)$ time (see [BK03]).

Example. *The set $D = \{1, 2, 4\}$ is a difference cover of* $[0..5)$.

x	y, z
0	1, 1
1	2, 1
2	1, 4
3	4, 1
4	1, 2



(the figure is from [BGSV14].)

Our algorithm utilizes the following interesting property of difference covers.

Lemma 9 (see [BK03]). *Let D be a difference cover of $[0..k)$. For any integers i, j, there exists $d \in [0..k)$ such that $(i - d) \bmod k \in D$ and $(j - d) \bmod k \in D$.*

At the beginning, our algorithm fixes an integer τ (the precise value of τ is given below). Let D be a difference cover of $[0..\tau^2)$ of size $O(\tau)$. Denote $M = \{i \in [1..n] \colon (i \bmod \tau^2) \in D\}$. Obviously, we have $|M| = O(\frac{n}{\tau})$. Our algorithm builds in $O(\frac{n}{\tau}(\tau^2 + \log n)) = O(\frac{n}{\tau}\log n + n\tau)$ time a data structure that can calculate $LCE(i, j)$ in constant time for any $i, j \in M$. To compute $LCE(i, j)$ for arbitrary $i, j \in [1..n]$, we simply compare $w[i..n]$ and $w[j..n]$ from left to right until we reach the positions $i + d$ and $j + d$ such that $i + d \in M$ and $j + d \in M$, and then we obtain $LCE(i, j) = d + LCE(i + d, j + d)$ in constant time. By Lemma 9, we have $d < \tau^2$ and therefore, the value $LCE(i, j)$ can be computed in $O(\tau^2)$ time. Thus, our algorithm can execute any sequence of $O(n)$ LCE queries in $O(\frac{n}{\tau}\log n + n\tau^2)$ time. Putting $\tau = \lceil\log^{\frac{1}{3}} n\rceil$, we obtain $O(\frac{n}{\tau}\log n + n\tau^2) = O(n\log^{\frac{2}{3}} n)$. Now it suffices to describe the data structure answering the LCE queries on the positions from M.

The data structure that we build in the preprocessing step is the minimal in the number of vertices compacted trie T such that for any $i \in M$, the string $w[i..n]$ can be spelled out on the path from the root to some leaf of T (see Figure 5). We store the labels on the edges of T as pointers to substrings of w. The trie T is commonly referred to as a *sparse suffix tree*. Obviously, T occupies $O(\frac{n}{\tau})$ space. For simplicity, we assume that $w[n]$ is a special letter that does not occur in $w[1..n{-}1]$, so, for each $i \in M$, the suffix $w[i..n]$ corresponds to some leaf of T.

Let $i, j \in M$. It is straightforward that $LCE(i, j)$ is equal to the length of the string written on the path from the root of T to the nearest common ancestor of the leaves corresponding to the suffixes $w[i..n]$ and $w[j..n]$. Using the construction of [HT84], one can preprocess T in $O(\frac{n}{\tau})$ time such that the nearest common ancestor of any two leaves can be found in constant time. So, to finish the proof, it remains to describe how to build T in $O(\frac{n}{\tau}(\tau^2 + \log n))$ time.

In general our construction is similar to that of [Kos15b]. We use the fact that the set M has the "period" τ^2, i.e., for any $i \in M$, we have $i + \tau^2 \in M$ provided $i + \tau^2 \le n$. Our algorithm consecutively inserts the suffixes $\{w[i..n] \colon i \in M\}$ in T from right to left. Suppose for some $k \in M$, we already have a compacted trie T that contains the suffixes $w[i..n]$ for all $i \in M \cap (k..n]$. We are to insert the suffix $w[k..n]$ in T. To perform the insertion efficiently, we maintain four additional data structures.

1. An order on the leaves of T. We store all leaves of T in a linked list in the lexicographical order of the corresponding suffixes. We maintain on this list the order maintenance data structure of [BCD⁺02] that allows to determine whether a given leaf precedes another leaf in the list in constant time. The insertion in this list takes constant amortized time. Hereafter, we say that a leaf x of T precedes [respectively, succeeds] another leaf y if x precedes [respectively, succeeds] y in the list of leaves.

2. Slow LCE queries. Denote by $i_1, i_2, \ldots, i_m$ the sequence of all positions $M \cap (k..n]$ in the increasing lexicographical order of the corresponding suffixes $w[i_1..n], w[i_2..n], \ldots, w[i_m..n]$. For each $i_p \in M \cap (k..n]$, we associate with the leaf corresponding to the suffix $w[i_p..n]$ the value $LCE(i_p, i_{p+1})$. It is easy to see that for any $i_p, i_q \in M \cap (k..n]$ such that $p < q$, we have $LCE(i_p, i_q) = \min\{LCE(i_p, i_{p+1}), LCE(i_{p+1}, i_{p+2}), \ldots, LCE(i_{q-1}, i_q)\}$. According to this observation, we store all leaves of T in an augmented balanced search tree C that allows to calculate $LCE(i_p, i_q)$ for any such i_p and i_q in $O(\log n)$ time. It is well known that the insertion in C of a new leaf with the associated LCE value requires $O(\log n)$ amortized time.

3. The "top" part of T. We maintain a compacted trie S that contains the strings $w[i..i{+}\tau^2]$ for all $i \in M \cap (k..n]$ (we assume $w[j] = w[n]$ for all $j > n$ and thus $w[i..i{+}\tau^2]$ is always well defined). Informally, S is the "top" part of T, so, we augment each vertex of S with a link to the corresponding vertex of T. We maintain on S the data structure of [FG04] supporting the insertions in $O(\tau^2 + \log n)$ amortized time. Let x be a leaf of S corresponding to a string $w[i..i{+}\tau^2]$. We augment x with a balanced search tree B_x that contains the leaves of T corresponding to all suffixes $w[j..n]$ such that $w[j{-}\tau^2..j] = w[i..i{+}\tau^2]$ in the order induced by the list of all leaves of T (see Figure 6). One can easily show that S together with the associated search trees occupies $O(\frac{n}{\tau})$ space in total.

4. Dynamic weighted ancestors. We maintain on T the *dynamic weighted ancestor* data structure of [KL07] that, for any given vertex x and an integer c, can find in $O(\log n)$ time the nearest ancestor of x such that the length of the string written on the path from the root to this ancestor is less than c. When we insert a new vertex in T, the modification of this structure takes $O(\log n)$ amortized time.

Example. *Let $\tau^2 = 4$. The set $D = \{0, 1, 3\}$ is a difference cover of $[0..\tau^2)$. Consider the string $w = \underline{a}b\underline{c}\underline{a}\underline{b}c\underline{a}\underline{b}\underline{a}b\underline{c}\underline{a}\underline{b}b\underline{\$}$; the underlined positions are from $M = \{i \in [1..n] \colon (i \bmod \tau^2) \in D\}$. The sparse suffix tree of w is presented in Figure 5. Figure 6 depicts the corresponding compacted trie S; each leaf of S is augmented with a balanced search tree of certain leaves of T (see the description above).*

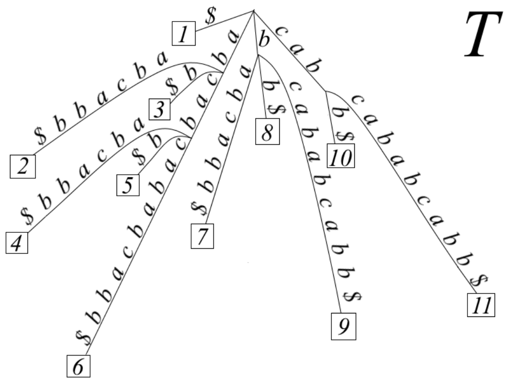


Figure 5: The sparse suffix tree T for $w = \underline{a}b\underline{c}\underline{a}\underline{b}c\underline{a}\underline{b}\underline{a}b\underline{c}\underline{a}\underline{b}b\underline{\$}$ (the underlined positions are from M).

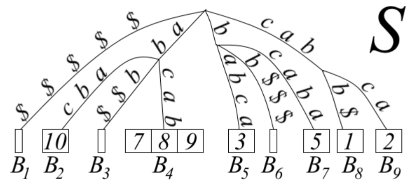


Figure 6: The balanced search trees $B_1, B_2, \ldots, B_9$ are augmented with the indices of leaves of T.

The construction of T. Now to insert $w[k..n]$ in T, we first insert $w[k..k{+}\tau^2]$ in S in $O(\tau^2 + \log n)$ time. If S does not contain $w[k..k{+}\tau^2]$, then we attach a new leaf in T using the links from S to T and modify in an obvious way all related data structures: the list of leaves of T, the newly created balanced search tree associated with the new leaf of S, the balanced search tree C, and the dynamic weighted ancestor data structure on T. The modifications require $O(\log n)$ amortized time.

Now suppose S contains $w[k..k{+}\tau^2]$. Denote by v the leaf of S corresponding to $w[k..k{+}\tau^2]$. Let y be the leaf of T corresponding to the suffix $w[k{+}\tau^2..n]$ (recall that $k{+}\tau^2 \in M$). In $O(\log n)$ time we obtain the immediate predecessor and successor of y in the search tree B_v, denoted by x and z, respectively. Notice that x is the immediate predecessor only in the set of all leaves contained in B_v but it may not be the immediate predecessor in the whole list of all leaves of T; the situation with z is similar. Let x and z correspond to suffixes $w[i_x..n]$ and $w[i_z..n]$, respectively. Since $w[i_x{-}\tau^2..i_x] = w[i_z{-}\tau^2..i_z] = w[k..k{+}\tau^2]$, it is straightforward that the suffixes $w[i_x{-}\tau^2..n]$ and $w[i_z{-}\tau^2..n]$ are, respectively, the immediate predecessor and successor of the suffix $w[k..n]$ in the set of all suffixes inserted in T. Hence, we must insert $w[k..n]$ between these suffixes.

It is easy to see that $LCE(k, i_x{-}\tau^2) = \tau^2 + LCE(k{+}\tau^2, i_x)$ and $LCE(k, i_z{-}\tau^2) = \tau^2 + LCE(k{+}\tau^2, i_z)$. The values $LCE(k{+}\tau^2, i_x)$ and $LCE(k{+}\tau^2, i_z)$ can be computed in $O(\log n)$ time using the balanced search tree C. Without loss of generality consider the case $LCE(k, i_x{-}\tau^2) \geq LCE(k, i_z{-}\tau^2)$. We find the position where we insert a new leaf in T using the weighted ancestor query on the value $LCE(k, i_x - \tau^2)$ and the leaf of T corresponding to the suffix $w[i_x - \tau^2..n]$. We finally modify all related data structures in an obvious way: the list of leaves of T, the balanced search trees B_v and C, and the dynamic weighted ancestor data structure on T. These modifications require $O(\log n)$ amortized time.

Time and space. The insertion of a new suffix in T takes $O(\tau^2 + \log n)$ amortized time. Thus, the construction of T consumes overall $O(\frac{n}{\tau}(\tau^2 + \log n))$ time as required. The whole data structure occupies $O(\frac{n}{\tau})$ space.

The above analysis together with Lemma 8 imply the following theorem.

Theorem 3. *For a general ordered alphabet, there is an algorithm that computes all runs in a string of length n in $O(n \log^{\frac{2}{3}} n)$ time and linear space.*

7. Conclusion

Lemma 6 expressing a nonconstructive property is a bottleneck for the conversion of our decision tree algorithm into a RAM algorithm. So, it remains an open problem whether there exists a linear RAM algorithm finding all runs in a string over a general ordered alphabet. Moreover, it is unknown if there is a linear RAM algorithm that decides whether a given string has runs (this problem was posed in [Bre, Chapter 4]). It seems that further improvements in the considered problem may be achieved by more and more efficient longest common extension data structures on a general ordered alphabet. One even might conjecture that there is a data structure that can execute any sequence of k LCE queries on a string of length n over a general ordered alphabet in $O(k + n)$ time. However, we do not yet have a theoretical evidence for such strong results. Another interesting direction is a generalization of our result for the case of online algorithms (e.g., see [HC08] and [Kos15d]).

Acknowledgement The author would like to thank Arseny M. Shur for many valuable comments and the help in the preparation of this paper.

[AKO04] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[BCD+02] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Algorithms-ESA 2002*, volume 2461 of *LNCS*, pages 152–164. Springer, 2002.

[BCT12] G. Badkobeh, M. Crochemore, and C. Toopsuwan. Computing the maximal-exponent repeats of an overlap-free string in linear time. In *String Processing and Information Retrieval*, pages 61–72. Springer, 2012.

[BGSV14] P. Bille, I. L. Gørtz, B. Sach, and H. W. Vildhøj. Time-space trade-offs for longest common extensions. *J. of Discrete Algorithms*, 25:42–50, 2014.

[BII+14] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta. The "runs" theorem. *arXiv preprint arXiv:1406.0263v4*, 2014.

[BK03] S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In *CPM 2003*, volume 2676 of *LNCS*, pages 55–69. Springer, 2003.

[Bre] D. Breslauer. *Efficient string algorithmics.* PhD thesis, Columbia University.

[CIS08] M. Crochemore, L. Ilie, and W. F. Smyth. A simple algorithm for computing the lempel-ziv factorization. In *Data Compression Conference (DCC'08)*, pages 482–488. IEEE, 2008.

[CIT11] M. Crochemore, L. Ilie, and L. Tinta. The "runs" conjecture. *Theoretical Computer Science*, 412(27):2931–2941, 2011.

[CKR+12] M. Crochemore, M. Kubica, J. Radoszewski, W. Rytter, and T. Waleń. On the maximal sum of exponents of runs in a string. *Journal of Discrete Algorithms*, 14:29–36, 2012.

[CPS08] G. Chen, S. J. Puglisi, and W. F. Smyth. Lempel–ziv factorization using less time & space. *Mathematics in Computer Science*, 1(4):605–623, 2008.

[Cro86] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.

[FG89] E. R. Fiala and D. H. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, 1989.

[FG04] G. Franceschini and R. Grossi. A general technique for managing strings in comparison-driven data structures. In *ICALP 2004*, volume 3142 of *LNCS*, pages 606–617. Springer, 2004.

[FW65] N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965.

[HC08] J.-J. Hong and G.-H. Chen. Efficient on-line repetition detection. *Theoretical Computer Science*, 407(1):554–563, 2008.

[HT84] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[KK99] R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science*, pages 596–604. IEEE, 1999.

[KKP14] J. Karkkainen, D. Kempa, and S. J. Puglisi. Lempel-ziv parsing in external memory. In *Data Compression Conference (DCC'14)*, pages 153–162. IEEE, 2014.

[KL07] T. Kopelowitz and M. Lewenstein. Dynamic weighted ancestors. In *SODA 2007*, pages 565–574. SIAM, 2007.

[Kol12] R. Kolpakov. On primary and secondary repetitions in words. *Theoretical Computer Science*, 418:71–81, 2012.

[Kos15a] D. Kosolobov. Computing runs on a general alphabet. *arXiv preprint arXiv:1507.01231*, 2015.

[Kos15b] D. Kosolobov. Faster lightweight Lempel–Ziv parsing. In *MFCS 2015*, volume 9235 of *LNCS*, pages 432–444. Springer-Verlag Berlin Heidelberg, 2015.

[Kos15c] D. Kosolobov. Lempel-Ziv factorization may be harder than computing all runs. In *STACS 2015*, volume 30 of *LIPIcs*, pages 582–593. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

[Kos15d] D. Kosolobov. Online detection of repetitions with backtracking. In *CPM 2015*, volume 9133 of *LNCS*, pages 295–306. Springer International Publishing, 2015.

[KPPK14] R. Kolpakov, M. Podolskiy, M. Posypkin, and N. Khrapov. Searching of gapped repeats and subrepetitions in a word. In *Combinatorial Pattern Matching*, pages 212–221. Springer, 2014.

[LZ76] A. Lempel and J. Ziv. On the complexity of finite sequences. *Information Theory, IEEE Transactions on*, 22(1):75–81, 1976.

[Mai89] M. G. Main. Detecting leftmost maximal periodicities. *Discrete Applied Mathematics*, 25(1):145–153, 1989.

[ML85] M. G. Main and R. J. Lorentz. Linear time recognition of squarefree strings. In *Combinatorial Algorithms on Words*, pages 271–278. Springer, 1985.

[OS08] D. Okanohara and K. Sadakane. An online algorithm for finding the longest previous factors. In *Algorithms-ESA 2008*, pages 696–707. Springer, 2008.

[RPE81] M. Rodeh, V. R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *Journal of the ACM (JACM)*, 28(1):16–24, 1981.

[Sta12] T. Starikovskaya. Computing lempel-ziv factorization online. In *Mathematical Foundations of Computer Science 2012*, pages 789–799. Springer, 2012.

[UAH76] J. D. Ullman, A. V. Aho, and D. S. Hirschberg. Bounds on the complexity of the longest common subsequence problem. *Journal of the ACM (JACM)*, 23(1):1–12, 1976.

[YBIT13] J. Yamamoto, H. Bannai, S. Inenaga, and M. Takeda. Faster compact on-line lempel-ziv factorization. *arXiv preprint arXiv:1305.6095v1*, 2013.