Factorizing a String into Squares in Linear Time

Yoshiaki Matsuoka[1], Shunsuke Inenaga[2], Hideo Bannai[3], Masayuki Takeda[4], and Florin Manea[5]

1 **Department of Informatics, Kyushu University, Japan**
`yoshiaki.matsuoka@inf.kyushu-u.ac.jp`
2 **Department of Informatics, Kyushu University, Japan**
`inenaga@inf.kyushu-u.ac.jp`
3 **Department of Informatics, Kyushu University, Japan**
`bannai@inf.kyushu-u.ac.jp`
4 **Department of Informatics, Kyushu University, Japa**
`takeda@inf.kyushu-u.ac.jp`
5 **Department of Computer Science, Kiel University, Germany**
`flm@informatik.uni-kiel.de`

Abstract

A *square factorization* of a string w is a factorization of w in which each factor is a square. Dumitran et al. [SPIRE 2015, pp. 54-66] showed how to find a square factorization of a given string of length n in $O(n \log n)$ time, and they posed a question whether it can be done in $O(n)$ time. In this paper, we answer their question positively, showing an $O(n)$-time algorithm for square factorization in the standard word RAM model with machine word size $\omega = \Omega(\log n)$. We also show an $O(n + (n \log^2 n)/\omega)$-time (respectively, $O(n \log n)$-time) algorithm to find a square factorization which contains the maximum (respectively, minimum) number of squares.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Squares, Runs, Factorization of Strings

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.27

1 Introduction

Factorization problems are one of the important topics in the study of string algorithms and combinatorics on strings and their applications. Essentially, the task is to efficiently identify a decomposition of a string into factors of a specific given form. For instance, we recall here the various forms of Lempel-Ziv factorizations of a string [21, 22, 19, 20]; this class of factorizations found many applications in data-compression but also in the efficient detection of repetitive structures in strings [18, 14]. Similarly, the standard factorization of strings (also called Lyndon factorization) [17, 8] found applications in data compression, in variants of the Burrows-Wheeler transform [16]. Both these factorizations were defined in very simple ways, starting from basic combinatorial concepts: repeats (or repeated occurrences of the same factor) in a string, or lexicographically minimal factors of a string; they can be both computed in linear time; see [5] and [8], respectively.

Some other factorizations of strings, whose factors are defined by well-studied combinatorial objects, were proposed and analyzed as well in the literature. Closer to the topic of this paper, we recall here palindromic factorizations of a string (where we want to split that string into an arbitrary number of non-trivial palindromic factors, or into a minimal or fixed number of such factors), analyzed already in the seminal paper of Knuth, Morris, and Pratt [13], as well as in a series of more recent papers in [9, 15, 2, 12]. In [1], Bakobeh et al. consider

algorithms for computing closed substrings in strings; a string is closed if it contains proper substring that occurs in it as a prefix and a suffix, but not elsewhere; more precisely, the problem of greedily factorizing a string into a sequence of longest closed substrings is solved. Even more relevant to our study, in [7] it was shown how factorizations into highly repetitive factors (e.g., repetitions with an exponent greater than 2) can be efficiently computed.

On the other hand, the study of repetitive structures occurring in strings is also one of the central topics in combinatorics on strings and stringology. Main and Lorentz [18] proposed an algorithm that decides whether a string w of length n contains a square (i.e., two consecutive occurrences of some factor, called root) in $O(n \log n)$ time. Their result was improved by Crochemore [4], who showed how to identify all the squares with a primitive root of a string w in $O(n \log n)$ time. These results hold for general alphabets and are optimal in a comparison-based model, but if we use the (realistic) RAM model with logarithmic word size (see [11] and the references therein for a survey of relevant results and techniques related to this model), and we are only interested in inputs over integer alphabets, we can actually find all runs of a string in linear time [14, 3]. Thus, we can also construct a succinct representation of all primitively rooted squares of a string within the same time complexity.

Following these two research directions, in [7], the task of deciding in linear time whether a string can be split into squares was left as an open problem. In this paper we show that a square factorization of a string can be indeed computed in linear time, using the RAM model with logarithmic word size and working under the assumption that the string is over an integer alphabet. We extend this result by proposing an efficient algorithm for the computation of such square factorizations with a maximum or minimum number of factors. Finally, we discuss several connected problems.

2 Preliminaries

Let Σ be an alphabet. An element of Σ^* is called a *string*. The empty string ε is the string of length 0. Let Σ^+ be the set of non-empty strings, i.e., $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For any strings x and y, we denote by $x \cdot y$ the concatenation of x and y. For a string $w = x \cdot y \cdot z$, x, y and z are called a *prefix*, *substring*, and *suffix* of w, respectively. A prefix x of w is called a *proper prefix* of w if $x \neq w$. The length of a string w is denoted by $|w|$. The i-th character of a string w is denoted by $w[i]$ for each $1 \leq i \leq |w|$. 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 integers i and j with $i \leq j$, we denote $[i, j] = \{i, i+1, \ldots, j\}$.

For a string x and positive integer k, let $x^0 = \varepsilon$, and $x^k = x^{k-1} \cdot x$. A string w is called *primitive* if there does not exist a string x and an integer $k \geq 2$ such that $w = x^k$. For any non-empty string x, a repetition x^2 is called a *square*. A square x^2 is called a *primitively rooted square* if x is primitive.

A positive integer p is called a *period* of string w if $w[i] = w[i+p]$ for all $1 \leq i \leq |w| - p$. For a string w, a triplet (p, s, e) is called a *run* in w if $p \leq (e - s + 1)/2$, p is the smallest period of $w[s..e]$, $s = 1$ or $w[s-1] \neq w[s-1+p]$, and $e = |w|$ or $w[e+1] \neq w[e+1-p]$.

We call a sequence $F = (f_1, ..., f_m)$ of m non-empty strings a *square factorization* of a string w if f_i is a square for each $1 \leq i \leq m$ and the concatenation $f_1 \cdots f_m$ is equal to w; the integer m is called the size of the factorization. Also, we call F a *largest square factorization* (resp. a *smallest square factorization*) of w if the size m is largest (resp. smallest) among all square factorizations of w.

▶ **Example 1.** $F_1 = (\mathtt{abaababaab}, \mathtt{bb}, \mathtt{aa}, \mathtt{bb}, \mathtt{bb})$, $F_2 = (\mathtt{abaababaab}, \mathtt{bb}, \mathtt{aa}, \mathtt{bbbb})$ and $F_3 =$

(abaaba, baabbbaabb, bb) are the square factorizations of a string $w =$ abaababaabbbaabbbb. F_1 is the largest square factorization of w and F_3 is the smallest one.

Notice that a string can have more than one largest square factorization and/or one smallest square factorization. For instance, string $w =$ aabaabaa has two largest square factorizations: (aa, baabaa) and (aabaab, aa). Notice that they are also smallest square factorizations of the string.

Let w be a string of length n over an alphabet $\Sigma = [1, n]$. Our model of computation is a standard word RAM model of machine word size $\omega = \Omega(\log n)$, where the following operations can be performed in $O(1)$ time: Let X, Y be bit arrays of length $m \leq \omega$ each, and let k be a non-negative integer. We denote by $X \,\&\, Y$, $X \mid Y$, and $X \oplus Y$, the *bitwise and*, *bitwise or*, and *bitwise exclusive or* of X and Y, respectively. We denote by $\sim X$ the *bitwise negation* of X. We denote by $X \gg k$ the *k-bit logical right shift* of X. We can also see X as an unsigned m-bit integer where the most (least) significant bit is $X[1]$ (respectively, $X[m]$); arithmetic operations on such integers take constant time.

3 Algorithms

3.1 A linear time algorithm for computing a square factorization

In this subsection we propose an $O(n)$-time algorithm for computing a square factorization of a given string w of length n. Note that if a square factorization of w exists, then there clearly exists a square factorization such that each factor is a primitively rooted square. Therefore we only consider primitively rooted squares in w.

For any run $\lambda = (p, s, e)$ in w, we denote $\rho(\lambda) = p$, $SqBegRange(\lambda) = [s, e - 2p + 1]$ and $SqEndRange(\lambda) = [s+2p, e+1]$; namely, for any position $k \in [1, |w|+1]$, $k \in SqBegRange(\lambda)$ iff $w[k..k+2\rho(\lambda)-1]$ is a primitively rooted square, and $k \in SqEndRange(\lambda)$ iff $w[k-2\rho(\lambda)..k-1]$ is a primitively rooted square. Also, we denote by R all runs in w.

▶ **Lemma 2** ([3, 14]). *$|R| < n$. Also, R can be computed in $O(n)$ time.*

▶ **Lemma 3** ([6]). *For any string v, the number of prefixes of v which are also primitively rooted squares is $O(\log |v|)$.*

▶ **Corollary 4.** $\sum_{\lambda \in R} |SqBegRange(\lambda)| = \sum_{\lambda \in R} |SqEndRange(\lambda)| = O(n \log n)$.

Proof. Clearly, both $\sum_{\lambda \in R} |SqBegRange(\lambda)|$ and $\sum_{\lambda \in R} |SqEndRange(\lambda)|$ are equal to the number of primitively rooted squares in w. By Lemma 3, the number of primitively rooted squares beginning at each position of w is $O(\log n)$. Thus we obtain $\sum_{\lambda \in R} |SqBegRange(\lambda)| = \sum_{\lambda \in R} |SqEndRange(\lambda)| = O(n \log n)$. ◀

Let C be a bit array of length $n + 1$ such that $C[i] = 1$ iff $w[i..n]$ can be factorized into squares. For convenience, let $C[i] = 0$ if $i < 1$ or $i > n + 1$. Algorithm 1 is a simple solution by dynamic programming, which is essentially equivalent to the approach of [7].

Let τ be some integer parameter such that $1 \leq \tau \leq \omega$. We split C into blocks of length τ. For each $1 \leq j \leq \lceil (n + 1)/\tau \rceil$, we call $C[(j - 1)\tau + 1..j\tau]$ as the j-th block of C.

Let $SPR_\tau = \{\lambda \in R \mid 2\rho(\lambda) < \tau\}$ and $LPR_\tau = \{\lambda \in R \mid 2\rho(\lambda) \geq \tau\}$. We call each element of SPR_τ and LPR_τ a *short period run* and a *long period run*, respectively. Also, for each position i, we denote $S_{\tau,i} = \{\lambda_{\mathrm{SP}} \in SPR_\tau \mid i \in SqEndRange(\lambda_{\mathrm{SP}})\}$.

For each $1 \leq j \leq \lceil (n + 1)/\tau \rceil$, let $B_{\tau,j} = \{\lambda_{\mathrm{LP}} \in LPR_\tau \mid [(j-1)\tau+1, j\tau] \cap SqBegRange(\lambda_{\mathrm{LP}}) \neq \emptyset\}$ and $E_{\tau,j} = \{\lambda_{\mathrm{LP}} \in LPR_\tau \mid [(j - 1)\tau + 1, j\tau] \cap SqEndRange(\lambda_{\mathrm{LP}}) \neq \emptyset\}$. Also, for each

Algorithm 1: A simple algorithm for determining whether w can be factorized into squares.

```
Input: String w of length n.
Compute R;
C[1..n] ← 0;
C[n + 1] ← 1;
for i = n + 1 down to 1 do
    if C[i] = 1 then
        foreach λ ∈ R such that i ∈ SqEndRange(λ) do
            C[i − 2ρ(λ)] ← 1;
```

$i \in [1, n+1]$, let $P_{\tau,i}$ be a bit array of length $\tau - 1$ such that for each $k \in [1, \tau - 1]$, $P_{\tau,i}[k] = 1$ iff there exists $\lambda_{\mathrm{SP}} \in S_{\tau,i}$ such that $\tau - 2\rho(\lambda_{\mathrm{SP}}) = k$.

▶ **Lemma 5.** *For any positive* τ, $\sum_{j=1}^{\lceil (n+1)/\tau \rceil} |B_{\tau,j}| = O(n + \frac{n}{\tau} \log n)$ *and* $\sum_{j=1}^{\lceil (n+1)/\tau \rceil} |E_{\tau,j}| = O(n + \frac{n}{\tau} \log n)$.

Proof. For each $\lambda \in LPR_\tau$, let $x_{\tau,\lambda}$ be the number of integers j such that $[(j-1)\tau + 1, j\tau] \cap SqBegRange(\lambda) \neq \emptyset$, and $y_{\tau,\lambda}$ be the number of integers j' such that $[(j'-1)\tau + 1, j'\tau] \subseteq SqBegRange(\lambda)$. Clearly, $x_{\tau,\lambda} \leq 2 + y_{\tau,\lambda}$ and $\tau y_{\tau,\lambda} \leq |SqBegRange(\lambda)|$ for each $\lambda \in LPR_\tau$. We obtain $\sum_{j=1}^{\lceil (n+1)/\tau \rceil} |B_{\tau,j}| = \sum_{\lambda \in LPR_\tau} x_{\tau,\lambda} \leq \sum_{\lambda \in LPR_\tau} (2 + y_{\tau,\lambda}) = 2|LPR_\tau| + \sum_{\lambda \in LPR_\tau} y_{\tau,\lambda} \leq 2|LPR_\tau| + (\sum_{\lambda \in LPR_\tau} |SqBegRange(\lambda)|)/\tau = O(n + \frac{n}{\tau} \log n)$. We can obtain $\sum_{j=1}^{\lceil (n+1)/\tau \rceil} |E_{\tau,j}| = O(n + \frac{n}{\tau} \log n)$ similarly. ◀

▶ **Lemma 6.** *For any parameter* τ *with* $1 \leq \tau \leq \omega$, *all bit arrays* $P_{\tau,1}, \ldots, P_{\tau,n+1}$ *can be computed in* $O(n)$ *time.*

Proof. Initially let $P_{\tau,i} \leftarrow 0$ for all $1 \leq i \leq n+1$. Also, for simplicity, we regard $P_{\tau,n+2} = 0$. Then, for each $\lambda_{\mathrm{SP}} \in SPR_\tau$, flip $P_{\tau,s}[\tau - 2\rho(\lambda_{\mathrm{SP}})]$ and $P_{\tau,e+1}[\tau - 2\rho(\lambda_{\mathrm{SP}})]$ where $[s, e] = SqEndRange(\lambda_{\mathrm{SP}})$. Finally, let $P_{\tau,i} \leftarrow P_{\tau,i} \oplus P_{\tau,i-1}$ for $i = 2$ to $n+1$, which can be done in $O(1)$ time for each operation since $\tau \leq \omega$. This algorithm takes $O(n + |SPR_\tau|) = O(n)$ time. Its correctness follows from the fact that for two different runs λ_1 and λ_2 in w, if $\rho(\lambda_1) = \rho(\lambda_2)$, then $SqEndRange(\lambda_1)$ and $SqEndRange(\lambda_2)$ do not overlap. ◀

In our algorithm, we process the blocks of C in descending order, from the $\lceil (n+1)/\tau \rceil$-th block to the first block of C. Suppose that we are going to process the j-th block of C. Here we assume that Algorithm 1 has already computed $C[j\tau + 1..n+1]$ correctly.

First, we handle short period runs. We process each $i \in [(j-1)\tau + 1, j\tau]$ in descending order. We assume that we have already computed $C[i..n+1]$ correctly. In Algorithm 1, if $C[i] = 1$, then we perform $C[i - 2\rho(\lambda_{\mathrm{SP}})] \leftarrow 1$ for each $\lambda_{\mathrm{SP}} \in S_{\tau,i}$. We can confirm that by the definition of $P_{\tau,i}$, it is equivalent to performing $C[i-\tau+1..i-1] \leftarrow C[i-\tau+1..i-1] \,|\, P_{\tau,i}$. Thus we can update the short period runs in $O(1)$ time for each position i.

After processing these short period runs, it is guaranteed that $C[(j-1)\tau + 1..n+1]$ is computed correctly. Next, we handle each long period run $\lambda_{\mathrm{LP}} \in E_{\tau,j}$. Let s, e be integers such that $[s + 2\rho(\lambda_{\mathrm{LP}}), e + 2\rho(\lambda_{\mathrm{LP}})] = [(j-1)\tau + 1, j\tau] \cap SqEndRange(\lambda_{\mathrm{LP}})$. In Algorithm 1, we perform $C[s+k] \leftarrow C[s+k] \mid C[s+k+2\rho(\lambda_{\mathrm{LP}})]$ for each $k \in [0, e-s]$. Note that from the definition of long period runs, we obtain $e < s + \tau \leq s + 2\rho(\lambda_{\mathrm{LP}})$, which means that $[s, e]$ and $[s + 2\rho(\lambda_{\mathrm{LP}}), e + 2\rho(\lambda_{\mathrm{LP}})]$ do not overlap. Thus the operation

Algorithm 2: An $O(n + \frac{n}{\tau}\log n)$-time algorithm of determining whether w can be factorized into squares.

Input: String w of length n, and a parameter τ with $1 \leq \tau \leq \omega$.

Compute SPR_τ and LPR_τ;
Compute $E_{\tau,1}, \ldots, E_{\tau,\lceil (n+1)/\tau \rceil}$ from LPR_τ;
Compute $P_{\tau,1}, \ldots, P_{\tau,n+1}$ from SPR_τ;
$C[1..n] \leftarrow 0$; $C[n+1] \leftarrow 1$;
for $j = \lceil (n+1)/\tau \rceil$ **down to** 1 **do**
  for $i = \min\{j\tau, n+1\}$ **down to** $(j-1)\tau + 1$ **do**
    if $C[i] = 1$ **then**
      $C[i-\tau+1..i-1] \leftarrow C[i-\tau+1..i-1] \mid P_{\tau,i}$;
  foreach $\lambda_{\mathrm{LP}} \in E_{\tau,j}$ **do**
    Let s, e be integers such that
    $[s + 2\rho(\lambda_{\mathrm{LP}}), e + 2\rho(\lambda_{\mathrm{LP}})] = [(j-1)\tau + 1, j\tau] \cap \mathit{SqEndRange}(\lambda_{\mathrm{LP}})$;
    $C[s..e] \leftarrow C[s..e] \mid C[s + 2\rho(\lambda_{\mathrm{LP}})..e + 2\rho(\lambda_{\mathrm{LP}})]$;

$C[s+k] \leftarrow C[s+k] \mid C[s+k+2\rho(\lambda_{\mathrm{LP}})]$ for each k can be done in parallel. Hence we perform $C[s..e] \leftarrow C[s..e] \mid C[s+2\rho(\lambda_{\mathrm{LP}})..e+2\rho(\lambda_{\mathrm{LP}})]$, which can be done in $O(1)$ time since $|[s,e]| \leq \tau \leq \omega$. Therefore it takes $O(|E_{\tau,j}|)$ time for long period runs in the j-th block of C. By Lemma 5, the computation on long period runs for all blocks can be done in $O(n + \frac{n}{\tau}\log n)$ time.

From above, we obtain Algorithm 2 and Lemma 7.

▶ **Lemma 7.** *For any parameter $1 \leq \tau \leq \omega$, Algorithm 2 determines whether w can be factorized into squares in $O(n + \frac{n}{\tau}\log n)$ time.*

Now we describe how to compute a square factorization of w. For a position i, we assume that $C[i] = 1$, i.e., $w[i..n]$ can be factorized into squares. First, we determine whether there exists $l \in [1, \tau-1]$ such that $C[i+l] = 1$ and $P_{\tau,i+l}[\tau - l] = 1$. This means that there exists some square factorization of $w[i..n]$ whose first factor is $w[i..i+l-1]$. In this case, we can spend $O(l)$ time to find such l, if any.

Next, we consider the case where there is no $l \in [1, \tau-1]$ s.t. $C[i+l] = 1$ and $P_{\tau,i+l}[\tau - l] = 1$. Then, there exists no short period run $\lambda_{\mathrm{SP}} \in S_{\tau,i+l}$ which satisfies $2\rho(\lambda_{\mathrm{SP}}) = l$ and $C[i+l] = 1$ for any l. In such a case, from the fact that $C[i] = 1$, there must exist some long period run $\lambda_{\mathrm{LP}} \in LPR_\tau$ such that $i \in \mathit{SqBegRange}(\lambda_{\mathrm{LP}})$ and $C[i + 2\rho(\lambda_{\mathrm{LP}})] = 1$. We scan all long period runs in $B_{\tau,\lceil i/\tau \rceil}$ and find such λ_{LP} in $O(|B_{\tau,\lceil i/\tau \rceil}|)$ time. Then, we use $w[i..i + 2\rho(\lambda_{\mathrm{LP}}) - 1]$ in the square factorization of $w[i..n]$. From above, we obtain Algorithm 3.

▶ **Lemma 8.** *For any parameter $1 \leq \tau \leq \omega$, Algorithm 3 computes a square factorization of w in $O(n + \frac{n}{\tau}\log n)$ time if it exists.*

Proof. We analyze the time complexity of Algorithm 3. For a position i with $C[i] = 1$, if there exists any short period run $\lambda_{\mathrm{SP}} \in B_{\tau,i+l}$ such that $2\rho(\lambda_{\mathrm{SP}}) = l$ and $C[i+l] = 1$ for any $l \geq 1$, we can compute $l = 2\rho(\lambda_{\mathrm{SP}})$ by scanning $P_{\tau,i+1}[\tau-1], P_{\tau,i+2}[\tau-2], \ldots, P_{\tau,i+l}[\tau-l]$ one by one. Note that if such $l \in [1, \tau-1]$ exists, then i increases by l, and hence we can afford to spend $O(l)$ time to find such l. Otherwise, we compute $l = 2\rho(\lambda_{\mathrm{LP}})$ for some $\lambda_{\mathrm{LP}} \in LPR_\tau$ such that $i \in \mathit{SqBegRange}(LPR_\tau)$ and $C[i + 2\rho(\lambda_{\mathrm{LP}})] = 1$; it takes $O(|B_{\tau,\lceil i/\tau \rceil}|)$

Algorithm 3: A linear-time algorithm of factorizing w into squares.

Input: String w of length n, and a parameter τ with $1 \leq \tau \leq \omega$.
Output: A square factorization of w if it exists; otherwise *nil*.

Compute SPR_τ and LPR_τ;
Compute $B_{\tau,1}, \ldots, B_{\tau,\lceil (n+1)/\tau \rceil}$ from LPR_τ;
Compute $P_{\tau,1}, \ldots, P_{\tau,n+1}, C$ by Algorithm 2;
if $C[1] = 1$ **then**
    $F \leftarrow ()$;
    $i \leftarrow 1$;
    while $i \leq n$ **do**
        $l \leftarrow 1$;
        while $l < \tau$ **do**
            if $C[i+l] = 1 \wedge P_{\tau,i+l}[\tau - l] = 1$ **then**
                break;
            $l \leftarrow l + 1$;
        if $l \geq \tau$ **then**
            Find any $\lambda_{\mathrm{LP}} \in B_{\tau,\lceil i/\tau \rceil}$ such that $i \in \mathit{SqBegRange}(\lambda_{\mathrm{LP}})$ and $C[i + 2\rho(\lambda_{\mathrm{LP}})] = 1$;
            `/* It is guaranteed that such` λ_{LP} `exists. */`
            $l \leftarrow 2\rho(\lambda_{\mathrm{LP}})$;
        Append $w[i..i+l-1]$ to the end of F;
        $i \leftarrow i + l$;
    return F;
else return *nil*;

time. Then we use $w[i..i+l-1]$ for a square factorization of $w[i..n]$ and increase i by l. Note that in this case, i increases by at least τ. Hence we can afford to spend $O(\tau)$ time before deciding to scan $B_{\tau,\lceil i/\tau \rceil}$. Moreover, for any $1 \leq j \leq \lceil (n+1)/\tau \rceil$, $B_{\tau,j}$ is scanned at most once. Therefore, using Lemma 5, we can show that Algorithm 3 takes $O(n + \frac{n}{\tau} \log n)$ time. ◀

Optimally, we choose $\tau = \omega$. Since $\omega = \Omega(\log n)$, by Lemma 8, we obtain Theorem 9.

▶ **Theorem 9.** *A square factorization of a string of length n can be computed in $O(n)$ time, if it exists.*

3.2 An algorithm for computing a largest square factorization

In this subsection we propose an algorithm for computing a largest square factorization of w. Note that any largest square factorization of w consists only of primitively rooted squares, since otherwise there exist a larger square factorization of w.

Let τ be some integer parameter such that $1 \leq \tau \leq \omega/(\lfloor \log n \rfloor + 1)$. As with Section 3.1, we define C, SPR_τ, LPR_τ, $S_{\tau,i}$ for each position $i \in [1, n+1]$, and $B_{\tau,j}$ and $E_{\tau,j}$ for each $j \in [1, \lceil (n+1)/\tau \rceil]$.

For each position i of string w, let us denote by $T[i]$ the size of largest square factorization of $w[i..n]$ if it exists; otherwise $T[i] = 0$. Algorithm 4 is a simple algorithm which computes the size of a largest square factorization of each suffix of w in $O(n \log n)$ time.

Algorithm 4: A simple algorithm for computing the size of largest square factorization of each suffix of w.

```
Input: String w of length n.
1 Compute R;
2 C[1..n] ← 0; C[n + 1] ← 1;
3 T[i] ← 0 for each i ∈ [1, n + 1];
4 for i = n + 1 down to 1 do
5     if C[i] = 1 then
6         foreach λ ∈ R such that i ∈ SqEndRange(λ) do
7             C[i − 2ρ(λ)] ← 1;
8             T[i − 2ρ(λ)] ← max{T[i − 2ρ(λ)], T[i] + 1};
```

Let $b = \lfloor \log n \rfloor$. For each position $i \in [1, n+1]$, let $P'_{\tau,i}$ be a bit array of length $(\tau-1)(b+1)$ such that for each $k \in [1, (\tau-1)(b+1)]$, $P'_{\tau,i}[k] = 1$ iff there exists $\lambda_{\mathrm{SP}} \in S_{\tau,i}$ such that $(\tau - 2\rho(\lambda_{\mathrm{SP}}))(b+1) = k$. We remark that $P'_{\tau,1}, \ldots, P'_{\tau,n+1}$ can be computed in $O(n)$ time in a similar way to Lemma 6. Also, let U be an array of bit arrays. For each $1 \leq i \leq n+1$, let $U[i]$ be a bit array of length $b+1$ such that $U[i][1] = C[i]$, and $U[i][2..b+1]$ is the binary representation of $T[i]$. Note that $T[i]$ can be represented as an unsigned b-bit integer since $T[i] \leq n/2$. For convenience, we regard $U[i] = 0$ if $i < 1$ or $i > n+1$. In addition, for two integers s, e with $s \leq e$, we denote by $U[s..e]$ the concatenation $U[s] \cdot U[s+1] \cdots U[e]$, which we also regard as an unsigned $((b+1)(e-s+1))$-bit integer where the most significant bit is $U[s][1]$ and the least significant bit is $U[e][b+1]$. To obtain a largest square factorization quickly, we compute U instead of T and C.

For each $1 \leq j \leq \lceil (n+1)/\tau \rceil$, we call $U[(j-1)\tau + 1..j\tau]$ the j-th block of U. As with Algorithm 2, we process the blocks of U in descending order, from the $\lceil (n+1)/\tau \rceil$-th block to the first block of U. Suppose that we are going to process the j-th block of U. Here we assume that our algorithm has already computed $U[j\tau + 1..n+1]$ correctly.

See Algorithm 5 for computing a largest square factorization, where M serves as a fixed-length $(\tau(b+1))$ bit array to specify the runs to be processed at once (Lines 7-8).

First, we handle short period runs. We process each $i \in [(j-1)\tau+1, j\tau]$ in descending order. We assume that we have already computed $U[i..n+1]$ correctly. As with Algorithm 4, if $C[i] = 1$, then we perform $C[i - 2\rho(\lambda_{\mathrm{SP}})] \leftarrow 1$ and $T[i - 2\rho(\lambda_{\mathrm{SP}})] \leftarrow \max\{T[i - 2\rho(\lambda_{\mathrm{SP}})], T[i] + 1\}$ for each $\lambda_{\mathrm{SP}} \in S_{\tau,i}$. In other words, if $U[i][1] = 1$, then we perform $U[i - 2\rho(\lambda_{\mathrm{SP}})][1] \leftarrow 1$ and $U[i - 2\rho(\lambda_{\mathrm{SP}})][2..b+1] \leftarrow \max\{U[i - 2\rho(\lambda_{\mathrm{SP}})][2..b+1], U[i][2..b+1] + 1\}$ for each $\lambda_{\mathrm{SP}} \in S_{\tau,i}$. It is equivalent to performing $U[i - 2\rho(\lambda_{\mathrm{SP}})] \leftarrow \max\{U[i - 2\rho(\lambda_{\mathrm{SP}})], U[i] + 1\}$ if $U[i][1] = 1$. We process all short period runs in $S_{\tau,i}$ in parallel. Note that since $2\rho(\lambda_{\mathrm{SP}}) < \tau$ for any $\lambda_{\mathrm{SP}} \in S_{\tau,i}$, we update $U[i - \tau + 1..i - 1]$ by taking $U[i]$ and $S_{\tau,i}$ into consideration. We show the method in Lines 12–17 of Algorithm 5, where M' in Line 13 serves as a bit array of length $(\tau-1)(b+1)$. The fact that $(\tau-1)(b+1) < \omega$ implies that the operations in Lines 12–17 can be performed in constant time.

▶ **Example 10.** Here, we explain the situation with some i and $\lambda_{\mathrm{SP}} \in S_{\tau,i}$ using a concrete example. Let $Y = (U[i] + 1)P'_{\tau,i}$ where Y is a bit array of length $(\tau-1)(b+1)$. In this example, we consider the case when $b = 4, \tau = 7, P'_{\tau,i} = 00001\,00000\,00000\,00000\,00001\,00000$ in binary representation and $U[i] = 10110$ in binary representation, which means that $C[i] = 1$ and $T[i] = 6$. Then we obtain $Y = 10111\,00000\,00000\,00000\,10111\,00000$. Intuitively, we *copy* $U[i] + 1$ to the appropriate positions. After that, in order to update $U[i - \tau + k]$

with $\max\{U[i-\tau+k], U[i]+1\}$ for each $k \in [1, \tau-1]$ with $P'_{\tau,i}[k(b+1)] = 1$, we perform $U[i-\tau+k] \leftarrow \max\{U[i-\tau+k], Y[(k-1)(b+1)+1..k(b+1)\}$ for each $k \in [1, \tau-1]$. This remaining part is almost same as Lines 23–26, which we will explain in Example 11.

After processing these short period runs, it is guaranteed that $U[(j-1)\tau+1..n+1]$ is computed correctly. Next, we handle long period runs in Lines 19–26, where again M' serves as a variable-length (up to $\tau(b+1)$) bit array. Consider each long period run $\lambda_{\mathrm{LP}} \in E_{\tau,j}$. Let s, e be integers such that $[s+2\rho(\lambda_{\mathrm{LP}}), e+2\rho(\lambda_{\mathrm{LP}})] = [(j-1)\tau+1, j\tau] \cap \mathit{SqEndRange}(\lambda_{\mathrm{LP}})$. In Algorithm 4, for each $k \in [0, e-s]$, we perform $C[s+k] \leftarrow 1$ and $T[s+k] \leftarrow \max\{T[s+k], T[s+k+2\rho(\lambda_{\mathrm{LP}})]+1\}$ if $C[s+k+2\rho(\lambda_{\mathrm{LP}})] = 1$. Note that from the definition of long period runs, we obtain $e < s+\tau \leq s+2\rho(\lambda_{\mathrm{LP}})$, which means that $[s, e]$ and $[s+2\rho(\lambda_{\mathrm{LP}}), e+2\rho(\lambda_{\mathrm{LP}})]$ do not overlap. Thus we process all k's in parallel. We next analyze Lines 19–26 of Algorithm 5. Since $e-s+1 \leq \tau$, we obtain $(e-s+1)(b+1) \leq \tau(b+1) \leq \omega$, which means that we can perform Lines 19–26 in constant time.

▶ **Example 11.** Here, we explain the situation with some j and $\lambda_{\mathrm{LP}} \in E_{\tau,j}$ using a concrete example. Let s, e be integers such that $[s+2\rho(\lambda_{\mathrm{LP}}), e+2\rho(\lambda_{\mathrm{LP}})] = [(j-1)\tau+1, j\tau] \cap \mathit{SqEndRange}(\lambda_{\mathrm{LP}})$. Also, let $s' = s+2\rho(\lambda_{\mathrm{LP}})$ and $e' = e+2\rho(\lambda_{\mathrm{LP}})$. In Lines 19–26, we update $U[s..e]$ by taking $U[s'..e']$ into consideration.

1. Let $X = U[s..e]$ and $Y = U[s'..e']$. In this example, we consider the case when $b = 4, |[s, e]| = 5, X = 10010\,00000\,00000\,00000\,11010$ in binary representation and $Y = 10110\,00000\,11000\,00000\,10101$ in binary representation, which means that $C[s..e] = (1,0,0,0,1), T[s..e] = (2,0,0,0,10), C[s'..e'] = (1,0,1,0,1)$ and $T[s'..e'] = (6,0,8,0,5)$. Also, let $M' = 10000\,10000\,10000\,10000\,10000$ in binary representation.
2. Let $Y' = Y + ((Y \,\&\, M') \gg b)$. Then we obtain $Y' = 10111\,00000\,11001\,00000\,10110$. Intuitively, it represents $T[s'+k]+1$ for each k with $C[s'+k] = 1$.
3. Let $D = (Y' \mid M') - (X \,\&\, {\sim}M')$. Then we obtain $D = 10101\,10000\,11001\,10000\,01100$. Intuitively, it represents $(T[s'+k]+1) - T[s+k]$ for each k.
4. Let $D' = ((D \,\&\, M') \gg b)(2^b - 1)$. Then we obtain $D' = 01111\,01111\,01111\,01111\,00000$. Intuitively, it indicates all positions k such that $T[s'+k]+1 \geq T[s+k]$.
5. Let $Z = D \,\&\, D'$. Then we obtain $Z = 00101\,00000\,01001\,00000\,00000$. Intuitively, it represents $\max\{(T[s'+k]+1) - T[s+k], 0\}$ for each k.
6. Let $Z' = Z + X$. Then we obtain $Z' = 10111\,00000\,01001\,00000\,11010$. Intuitively, it represents $\max\{T[s'+k]+1, T[s+k]\}$ for each k.
7. Compute $Z'' = Z' \mid (Y \,\&\, M')$ to set $C[s..e]$ appropriately. Then we obtain $Z'' = 10111\,00000\,11001\,00000\,11010$. Finally, we substitute Z'' for $U[s..e]$. Then we obtain $C[s..e] = (1,0,1,0,1)$ and $T[s..e] = (7,0,9,0,10)$ as a result.

After computing $U[1..n+1]$, we obtain a largest square factorization of w as in Algorithm 3.

▶ **Lemma 12.** *Algorithm 5 computes a largest square factorization of w in $O(n + \frac{n}{\tau}\log n)$ time for any parameter $1 \leq \tau \leq \omega/(\lfloor \log n \rfloor + 1)$.*

Proof. Clearly, Algorithm 5 requires $O(n + \sum_{j=1}^{\lceil (n+1)/\tau \rceil} |B_{\tau,j}| + \sum_{j=1}^{\lceil (n+1)/\tau \rceil} |E_{\tau,j}|)$ time. Therefore, from Lemma 5, Algorithm 5 runs in $O(n + \frac{n}{\tau}\log n)$ time in total. ◀

The optimal strategy is to choose $\tau = \lfloor \omega/(\lfloor \log n \rfloor + 1) \rfloor$. Thus, we obtain Theorem 13.

▶ **Theorem 13.** *A largest square factorization of a string of length n can be computed in $O(n + \frac{n}{\omega}\log^2 n)$ time.*

Algorithm 5: An algorithm for computing a largest square factorization of w.

Input: String w of length n, and a parameter τ with $1 \leq \tau \leq \omega/(\lfloor \log n \rfloor + 1)$.
Output: A largest square factorization of w if it exists; otherwise *nil*.

Compute SPR_τ and LPR_τ;
Compute $B_{\tau,1}, \ldots, B_{\tau,\lceil (n+1)/\tau \rceil}, E_{\tau,1}, \ldots, E_{\tau,\lceil (n+1)/\tau \rceil}$ from LPR_τ;
Compute $P'_{\tau,1}, \ldots, P'_{\tau,n+1}$ from SPR_τ;
$U[i][1..b+1] \leftarrow 0$ for each $i \in [1, n+1]$;
$U[n+1][1] \leftarrow 1$; `/* equivalent to` $C[n+1] \leftarrow 1$ `*/`
$b \leftarrow \lfloor \log n \rfloor$;
$M[1..\tau(b+1)] \leftarrow 0$ where M is a bit array of length $\tau(b+1)$;
$M[k(b+1)-b] \leftarrow 1$ for each $k \in [1, \tau]$;
for $j = \lceil (n+1)/\tau \rceil$ **down to** 1 **do**
  for $i = \min\{j\tau, n+1\}$ **down to** $(j-1)\tau + 1$ **do**
    if $U[i][1] = 1$ **then**
      $Y \leftarrow (U[i]+1)P'_{\tau,i}$ where Y is a bit array of length $(\tau-1)(b+1)$;
      $M' \leftarrow M[1..(\tau-1)(b+1)]$;
      $X \leftarrow U[i-\tau+1..i-1]$;
      $D \leftarrow (Y \mid M') - (X \,\&\, {\sim}M')$;
      $D' \leftarrow ((D \,\&\, M') \gg b)(2^b - 1)$;
      $U[i-\tau+1..i-1] \leftarrow ((D' \,\&\, D) + X) \mid (Y \,\&\, M')$;
  foreach $\lambda_{\mathrm{LP}} \in E_{\tau,j}$ **do**
    Let s, e be integers such that
    $[s + 2\rho(\lambda_{\mathrm{LP}}), e + 2\rho(\lambda_{\mathrm{LP}})] = [(j-1)\tau + 1, j\tau] \cap \mathit{SqEndRange}(\lambda_{\mathrm{LP}})$;
    $M' \leftarrow M[1..(e-s+1)(b+1)]$;
    $Y \leftarrow U[s + 2\rho(\lambda_{\mathrm{LP}})..e + 2\rho(\lambda_{\mathrm{LP}})]$;
    $Y' \leftarrow Y + ((Y \,\&\, M') \gg b)$;
    $X \leftarrow U[s..e]$;
    $D \leftarrow (Y' \mid M') - (X \,\&\, {\sim}M')$;
    $D' \leftarrow ((D \,\&\, M') \gg b)(2^b - 1)$;
    $U[s..e] \leftarrow ((D' \,\&\, D) + X) \mid (Y \,\&\, M')$;
if $U[1][1] = 1$ **then** `/* equivalent to` $C[1] = 1$ `*/`
  $F \leftarrow ()$;
  $i \leftarrow 1$;
  while $i \leq n$ **do**
    $l \leftarrow 1$;
    while $l < \tau$ **do**
      if $U[i+l] + 1 = U[i] \wedge P'_{\tau,i+l}[(\tau-l)(b+1)] = 1$ **then break**;
      $l \leftarrow l + 1$;
    if $l \geq \tau$ **then**
      Find any $\lambda_{\mathrm{LP}} \in B_{\tau,\lceil i/\tau \rceil}$ such that $i \in \mathit{SqBegRange}(\lambda_{\mathrm{LP}})$ and
      $U[i + 2\rho(\lambda_{\mathrm{LP}})] + 1 = U[i]$;
      $l \leftarrow 2\rho(\lambda_{\mathrm{LP}})$;
    Append $w[i..i+l-1]$ to the end of F;
    $i \leftarrow i + l$;
  return F;
else return *nil*;

4 Other problems and further work

In this section we discuss several connected problems. The first such problem is that of computing a smallest square factorization of a string (i.e., a square factorization with a minimum number of factors). Unlike the factorizations produced as solutions to the previous problems, in such a factorization the square factors we use are no longer necessarily primitively rooted. Thus, it seems that a slightly different strategy might be needed to solve this problem. Next, we propose an $O(n \log n)$ time algorithm for computing a smallest square factorization of a string w of length n, over $\Sigma = [1, n]$.

Following Lemma 3, let us assume that $x_1^2, \ldots, x_k^2$ are all the primitively rooted squares starting at position i of w, with $|x_j| < |x_{j+1}|$ for $1 \leq j \leq k-1$. If there exists a position i' of w where x_h^2 starts, for some $h \leq k$, we have that the primitively rooted squares whose root is shorter than x_h^2 starting at i' are exactly $x_1^2, \ldots, x_{h-1}^2$, so x_h^2 is the h-th primitively rooted square occurring at both positions i and i', in the list of such squares ordered increasingly w.r.t. their length.

Further, we can produce for each $i \leq n$ the list of all primitively rooted squares starting at i in $O(n \log n)$ time; there are at most $2 \log n$ such squares for each i. We define the $(n+1) \times (1 + 2 \log n)$ matrix Q, where, for $1 \leq i \leq n$ and $1 \leq j \leq 2 \log n$ we have that $Q[i][j]$ is the number of factors in a smallest square factorization of $w[i..n]$, such that the first square of this factorization is a power of the j-th primitively rooted square in the list of primitively rooted squares starting at i, ordered increasingly w.r.t. their length (or undefined, if there are less than j primitively rooted squares starting at i). Moreover, $Q[i][0] = \min\{Q[i][j] \mid 1 \leq j \leq 2 \log n\}$, and $Q[n+1][j] = 0$ for all $0 \leq j \leq 2 \log n$.

The values stored in this matrix can be computed by dynamic programming. Assume we are computing $Q[i][j]$ with $i \leq n$ and $1 \leq j \leq 2 \log n$, and there are at least j primitively rooted squares starting at position i of w; at this point in our computation we already know all the values stored in the arrays $Q[i'][\cdot]$ for $i' > i$. When x_j^2 occurs also at position $i + 2|x_j|$, by the preliminary remark we made, we get that x_j^2 is the j-th primitively rooted square in the list of such squares occurring at position $i + 2|x_j|$; so we can compute $Q[i][j]$ as the minimum between $Q[i + 2|x_j|][0] + 1$ and $Q[i + 2|x_j|][j]$. Indeed, either the first square in the factorization of $w[i..n]$ is x_j^2, and then we continue with the smallest square factorization of $w[i + 2|x_j|..n]$; or the first square in the factorization of $w[i..n]$ is x_j^{2k} for some $k > 1$, and the smallest square factorization of $w[i + 2|x_j|..n]$ started with x_j^{2k-2}. The case when x_j^2 does not occur at $i + 2|x_j|$ is much simpler: we just set $Q[i][j]$ as $Q[i + 2|x_j|][0] + 1$. Finally, after computing $Q[i][j]$ for all $1 \leq j \leq 2 \log n$, we set $Q[i][0]$ as their minimum. Clearly, this process can be easily implemented in $O(n \log n)$ time, with the help of data structures allowing us to test whether some primitively rooted square x_j^2 occurring at position i also occurs at some other position $i + 2|x_j|$, like, e.g., the data structures *SqBegRange* and *SqEndRange*.

The number of factors in a smallest square factorization of w can be now found in $Q[1][0]$, while this factorization can be effectively obtained by tracing back the computation of $Q[1][0]$ via dynamic programming. Thus, we have shown the following result.

▶ **Theorem 14.** *A smallest square factorization of w can be obtained in $O(n \log n)$ time.*

We conjecture that a more efficient implementation of the above solution can be obtained using and extending the ideas in the previous section.

Two other open problems connected to square factorizations of strings are the following. The first one is inspired by the work of [15]: given a string w and a number k decide whether w has a square factorization with exactly k factors. The second one follows the line of

research in [2]: given a string w decide whether there exists a square factorization of w whose factors are each two distinct (i.e., a diverse square factorization of w). While we expect that the first problem can be solved efficiently, we conjecture that the second one is NP-Complete.

The strategy employed in Section 3.1 seems to also lead to an improvement in deciding whether a string w_1, of length n, can be obtained from other string w_2, of length $m < n$, by iterated prefix-suffix duplication [10]. Prefix-suffix duplication is a string operation that rewrites a string $u = xwy$ into xu (prefix-duplication) or uy (suffix-duplication). Accordingly, in the respective problem one asks whether there exists a sequence of prefix-suffix duplications that can be applied to w_1 so that in the end we get w_2; state-of-the-art algorithms [10] solved this problem in $O(n^2 \log n)$ time or, alternatively, in $O(n \log n)$ time if we allow only suffix-duplications or only prefix-duplications to be applied in order to obtain w_1 from w_2. We conjecture that using the ideas of Section 3.1 we can shave a $\log n$ factor from both of these complexities. For instance, if we consider the case of only suffix-duplications, we basically have to decide the existence of a factorization of w into $w = x_0 \cdots x_k$ such that $x_0 = w_1$ and x_i is a primitive string which is a suffix of $x_0 \cdots x_{i-1}$; in other words, x_i is a primitively rooted square centered at position $|x_0 \cdots x_{i-1}| + 1$. This problem greatly resembles to the problem of factoring a string into squares and we conjecture that it can be solved by the same methods, within the same linear time complexity.

References

1. Golnaz Badkobeh, Hideo Bannai, Keisuke Goto, Tomohiro I, Costas S. Iliopoulos, Shunsuke Inenaga, Simon J. Puglisi, and Shiho Sugimoto. Closed factorization. In *Proc. PSC 2014*, pages 162–168, 2014.
2. Hideo Bannai, Travis Gagie, Shunsuke Inenaga, Juha Kärkkäinen, Dominik Kempa, Marcin Piatkowski, Simon J. Puglisi, and Shiho Sugimoto. Diverse palindromic factorization is np-complete. In *Proc. DLT 2015*, volume 9168 of *Lecture Notes in Comput. Sci.*, pages 85–96. Springer, 2015.
3. Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "Runs" Theorem. *CoRR*, abs/1406.0263, 2014. URL: `http://arxiv.org/abs/1406.0263`.
4. Maxime Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.*, 12(5):244–250, 1981.
5. Maxime Crochemore, Lucian Ilie, and William F. Smyth. A simple algorithm for computing the lempel ziv factorization. In *Proc. DCC 2008*, pages 482–488. IEEE, 2008.
6. Maxime Crochemore and Wojciech Rytter. Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13(5):405–425, 1995. `doi:10.1007/BF01190846`.
7. Marius Dumitran, Florin Manea, and Dirk Nowotka. On prefix/suffix-square free words. In *Proc. SPIRE 2015*, volume 9309 of *Lecture Notes in Comput. Sci.*, pages 54–66. Springer, 2015.
8. Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983. `doi:10.1016/0196-6774(83)90017-2`.
9. Gabriele Fici, Travis Gagie, Juha Kärkkäinen, and Dominik Kempa. A subquadratic algorithm for minimum palindromic factorization. *J. Discrete Algorithms*, 28:41–48, 2014.
10. Jesús García-López, Florin Manea, and Victor Mitrana. Prefix-suffix duplication. *J. Comput. Syst. Sci.*, 80(7):1254–1265, 2014. `doi:10.1016/j.jcss.2014.02.011`.
11. Torben Hagerup. Sorting and searching on the word RAM. In *Proc. STACS 1998*, volume 1373 of *Lecture Notes in Comput. Sci.*, pages 366–398. Springer, 1998.

12 Tomohiro I, Shiho Sugimoto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing palindromic factorizations and palindromic covers on-line. In *Proc. CPM 2014,*, volume 8486 of *Lecture Notes in Comput. Sci.*, pages 150–161. Springer, 2014.

13 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

14 Roman Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *Proc. FOCS 1999*, pages 596–604. IEEE, 1999.

15 Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Pal^k is linear recognizable online. In *SOFSEM 2015*, volume 8939 of *Lecture Notes in Comput. Sci.*, pages 289–301. Springer, 2015.

16 Manfred Kufleitner. On bijective variants of the Burrows-Wheeler transform. In *Proc. PSC 2009*, pages 65–79, 2009.

17 Roger C. Lyndon. On Burnside's problem. *Trans. Amer. Math. Soc.*, 77(2):202–215, 1954. URL: http://www.jstor.org/stable/1990868.

18 Michael G. Main and Richard J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. Algorithms*, 5(3):422–432, 1984.

19 James A. Storer and Thomas G. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982. doi:10.1145/322344.322346.

20 Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

21 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, 23(3):337–343, 1977.

22 Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory*, 24(5):530–536, 1978.