Palindromic k-Factorization in Pure Linear Time

Mikhail Rubinchik
Ural Federal University, Ekaterinburg, Russia
mikhail.rubinchik@gmail.com

Arseny M. Shur
Ural Federal University, Ekaterinburg, Russia
arseny.shur@urfu.ru

Abstract

Given a string s of length n over a general alphabet and an integer k, the problem is to decide whether s is a concatenation of k nonempty palindromes. Two previously known solutions for this problem work in time $O(kn)$ and $O(n \log n)$ respectively. Here we settle the complexity of this problem in the word-RAM model, presenting an $O(n)$-time online deciding algorithm. The algorithm simultaneously finds the minimum odd number of factors and the minimum even number of factors in a factorization of a string into nonempty palindromes. We also demonstrate how to get an explicit factorization of s into k palindromes with an $O(n)$-time offline postprocessing.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Mathematics of computing → Combinatorial algorithms

Keywords and phrases stringology, palindrome, palindromic factorization, online algorithm

Digital Object Identifier 10.4230/LIPIcs.MFCS.2020.81

Related Version A full version of the paper is available at `https://arxiv.org/abs/2002.03965`.

Funding Supported by the Ministry of Science and Higher Education of the Russian Federation (Ural Mathematical Center project No. 075-02-2020-1537/1).

1 Introduction

Factorization, or representation of a string as a concatenation of substrings, is a key tool in both algorithmic and combinatorial studies on strings. For example, both the Lempel–Ziv factorization [13] and the Lyndon factorization [10] are ubiquitous in combinatorics on words and stringology. Factorization into palindromes also attracted the attention of researchers since 1970s. Recall that a string $s = a_1a_2 \cdots a_n$ is a *palindrome* if it is equal to its *reversal* $\overleftarrow{s} = a_n \cdots a_2a_1$; *palindromic (k-)factorization* is a representation of a string as a concatenation of (k) nonempty palindromes. There is a bunch of results concerning palindromic factorization. One of them is a hardness result: deciding the existence of a factorization into distinct palindromes is NP-complete [2]; for all other natural types of palindromic factorization, sooner or later linear-time algorithms in the word-RAM model of computation were designed. Knuth, Morris, and Pratt [11] presented such an algorithm deciding whether a string has a factorization into even-length palindromes. Galil and Seiferas [8] did the same for the factorization into palindromes of length >1, and also for 2-, 3-, and 4-factorization. The existence of k-factorization was shown to be decidable in $O(kn)$ time [12]. Most of the recent results were related to *palindromic length* of a string, which is the minimum number of factors in its palindromic factorization. There are several combinatorics papers, see e.g. [7, 6, 17], studying the conjecture that every aperiodic infinite string has finite factors of arbitrarily big palindromic length. On the algorithmic side, the palindromic length of a string of length n was shown to be computable in $O(n \log n)$ time [4, 9, 16]. In [3], the optimal $O(n)$ bound was reached, using bit compression and range operations.

A linear-time algorithm for palindromic length gave a hope for better results on palindromic k-factorization, because deciding the latter is equivalent to computing the minimum even and minimum odd number of factors in a palindromic factorization. An $O(n \log n)$ algorithm for palindromic length [16, Prop. 4.2] can be transformed into an $O(n \log n)$ algorithm for even/odd palindromic length [16, Prop. 4.9]. However, the properties of palindromic length used in the linear-time algorithm of [3] do not hold for even/odd palindromic length. In this paper we show how to overcome the technical difficulties and present the following optimal result.

▶ **Theorem 1.** *There exists an online algorithm deciding, in $O(n)$ time independent of k, whether a length-n input string over a general alphabet admits a palindromic k-factorization.*

In addition, we show how to find a palindromic k-factorization explicitly within the linear time (but offline). The paper is organized as follows: after preliminaries on strings and palindromes, we give an $O(n \log n)$-algorithm for computing the even/odd palindromic length of a string in Section 2. The main section is Section 3 where a linear-time algorithm is described. We conclude with an algorithm finding an explicit k-factorization (Section 4). Asterisks $(*)$ indicate that proofs or additional details can be found in the full version.

Preliminaries. For any i, j, $[i..j]$ denotes the range $\{k \in \mathbb{Z} \colon i \le k \le j\}$; we abbreviate $[i..i]$ as $[i]$. We use ranges, in particular, to index strings and arrays. For strings we write $s = s[1..n]$, where $n = |s|$ is the length of s. The *empty string* is denoted by ε. A string u is a *substring* of s if $u = s[i..j]$ for some i, j ($j < i$ means $u = \varepsilon$). Such pair (i, j) is not necessarily unique; i specifies an *occurrence* of u at *position* i. A substring $s[1..j]$ (resp., $s[i..n]$) is a *prefix* (resp. *suffix*) of s. An integer $p \in [1..n]$ is a *period* of s if $s[1..n-p] = s[p+1..n]$. We write s^k for the concatenation of k copies of s (thus s^k has period $|s|$). We write $\overleftarrow{s}$ for a string or array obtained from s by reversing the order of elements (thus the equality $s = \overleftarrow{s}$ defines a *palindrome*). A substring (resp. suffix, prefix) that is a palindrome is called a *subpalindrome* (resp. *suffix-palindrome*, *prefix-palindrome*). If $s[i..j]$ is a subpalindrome of s, the numbers $(j + i)/2$ and $\lfloor (j - i + 1)/2 \rfloor$ are respectively the *center* and the *radius* of $s[i..j]$. Subpalindromes of odd (even) length have integer (resp., half-integer) centers.

A (k-)*pal-factorization* of s is a representation $s = w_1 \cdots w_k$, where $w_1, \ldots, w_k$ are palindromes. The minimum k such that a k-pal-factorization of s exists is the *palindromic length* of s, denoted by $\mathsf{pl}(s)$. We also introduce the *even palindromic length* $\mathsf{pl}^0(s)$ and *odd palindromic length* $\mathsf{pl}^1(s)$ as the minimum even (resp., odd) k among all such factorizations of s. If s has no pal-factorization with even (odd) number of factors, we write $\mathsf{pl}^0(s) = \infty$ (resp., $\mathsf{pl}^1(s) = \infty$). For example, $\mathsf{pl}(abcba) = \mathsf{pl}^1(abcba) = 1$, $\mathsf{pl}^0(abcba) = \infty$; $\mathsf{pl}(acaaba) = \mathsf{pl}^0(acaaba) = 2$, $\mathsf{pl}^1(acaaba) = 5$. A pal-factorization $s = w_1 \cdots w_k$ with $k \le |s| - 2$ can be easily transformed into a $(k{+}2)$-pal-factorization: either $|w_i| \ge 3$ for some i, so $w_i = aua$ for some letter a and palindrome u, or $|w_i| = |w_j| = 2$ for some i, j, so w_i and w_j can be replaced by four 1-letter factors. This leads to the following crucial observation.

▶ **Lemma 2** ([16, Sect. 4.1])**.** *(1) A string s has a k-pal-factorization iff $\mathsf{pl}^{k \bmod 2}(s) \le k$. (2) A k-pal-factorization of s can be obtained in $O(|s|)$ time from its $\mathsf{pl}^{k \bmod 2}(s)$-pal-factorization.*

▶ Remark 3. Due to Lemma 2, throughout the paper we study the problem of existence of a k-pal-factorization for a string s as the problem of computing $\mathsf{pl}^0(s)$ and $\mathsf{pl}^1(s)$.

Recall that the method for computing $\mathsf{pl}(s)$ is dynamic programming: we compute the array $\mathsf{pl}[1..n]$ such that $\mathsf{pl}[i] = \mathsf{pl}(s[1..i])$ using an artificial initial value $\mathsf{pl}[0] = 0$ and the rule

$\mathsf{pl}[k] = 1 + \min_{i \in S_k} \mathsf{pl}[i-1]$, where S_k is the set of positions of all suffix-palindromes of $s[1..k]$. This rule can be easily adapted to compute $\mathsf{pl}^0(s)$ and $\mathsf{pl}^1(s)$ (due to symmetry, we always write pl^j assuming $j \in \{0,1\}$). We define the arrays $\mathsf{pl}^j[1..n]$ and employ the scheme

$$\mathsf{pl}^0[0] = 0,\ \mathsf{pl}^1[0] = \infty,\ \mathsf{pl}^j[k] = 1 + \min_{i \in S_k} \mathsf{pl}^{1-j}[i-1],\ \text{where } S_k = \{i : s[i..k] = \overleftarrow{s[i..k]}\} \quad (1)$$

Henceforth, s is the input string of length n and $\mathsf{PL}[i] = (\mathsf{pl}^0[i], \mathsf{pl}^1[i])$. All considered algorithms work in the unit-cost word-RAM model with $\Theta(\log n)$-bit machine words and standard operations like in the C language. Since the only operation used on alphabetic symbols is testing two symbols for equality, the algorithms are valid for the general unordered alphabet. All algorithms except Algorithm 2 are online and work in *iterations*: ith iteration begins with reading the symbol $s[i]$ and ends before reading $s[i+1]$.

2 An $O(n \log n)$ Algorithm

Palindromic Iterator. We process the input string using a data structure called (palindromic) *iterator* [12], which stores a string s and answers the following queries in $O(1)$ time:
- $\mathsf{rad}(x)$ / $\mathsf{len}(x)$ returns the radius / length of the longest palindrome in s with the center x;
- maxPal returns the center of the longest suffix-palindrome of s;
- $\mathsf{nextPal}(x)$ returns the center of the longest proper suffix-palindrome of the suffix-palindrome of s with the center x.

The iterator stores an array of radii for all possible centers of palindromes and a list of centers of suffix-palindromes in increasing order. The update query $\mathsf{add}(a)$ appends letter a to s. Performing this query, the iterator emulates an iteration of Manacher's algorithm [14] and updates the list of suffix-palindromes, all within $O(1 + \mathsf{maxPal}_{new} - \mathsf{maxPal}_{old})$ time, which is $O(n)$ for n updates to the originally empty structure. Below we assume that $\mathsf{add}(a)$ returns the list of deleted centers in the form $\mathsf{dead}(x) = (x, \text{answers to all queries about } x)$. We also write $\mathsf{cntr}(d) = n - (d-1)/2$ for the center of the length-d suffix-palindrome of $s[1..n]$.

With the iterator, the rule (1) can be implemented to work in time proportional to the number of subpalindromes in s ($\Omega(n^2)$ in the worst case); one iteration is as follows:

```
1: add(s[n]); pl^j[n] ← +∞
2: for (x ← maxPal; x ≠ n + 1/2; x ← nextPal(x)) do
3:     pl^j[n] ← min{pl^j[n], 1 + pl^{1−j}[n − len(x)]}
```

Series of Palindromes. Let $u_1, \dots, u_\ell$ be all non-empty suffix-palindromes of a string s in the order of decreasing length. For any $i < j$, since u_j is a suffix of u_i, any period of u_i is a period of u_j. Hence the sequence of minimal periods of $u_1, \dots, u_l$ is non-increasing. The groups of suffix-palindromes with the same minimal period are *series of palindromes* (of s):

$$\underbrace{u_1, \dots, u_{i_1}}_{p_1}, \underbrace{u_{i_1+1}, \dots, u_{i_2}}_{p_2}, \dots, \underbrace{u_{i_{t-1}+1}, \dots, u_\ell}_{p_t}.$$

A *p-series* is a series with the period p. The longest and the shortest palindrome in a p-series are its *head* $\mathsf{head}(p)$ and *tail* $\mathsf{tail}(p)$ respectively (they coincide in a 1-element series).

▶ **Lemma 4** ([4, 9, 12]; (∗)). *For any string s, if ℓ is the length of a tail of a series, then the head of the next series has length less than $2\ell/3$. In particular, if $p_1 > \ldots > p_t$ are periods of all series of s, then $t = O(\log p_1)$ and $p_1 + \cdots + p_t = O(p_1)$.*

Note that length-n strings with $\Omega(\log n)$ series for $\Omega(n)$ prefixes do exist [4]. The structure of series is described in the following lemma.

▶ **Lemma 5** ([3]; (∗)). *For a string s and $p \geq 1$, let U be a p-series of palindromes, $r = \#U$. There exist unique palindromes u, v with $|uv| = p$, $v \neq \varepsilon$ such that one of the conditions holds:*
1) *$U = \{(uv)^{r+1}u, (uv)^r u, \ldots, (uv)^2 u\}$ and uvu is the head of the next series,*
2) *$U = \{(uv)^r u, (uv)^{r-1}u, \ldots, uvu\}$ and u is the head of the next series,*
3) *$U = \{v^r, v^{r-1}, \ldots, v\}$, $p = 1$, $|v| = 1$, $u = \varepsilon$.*
With the notion of series, rule (1) can be rewritten as

$$\mathsf{pl}^j[n] = 1 + \min_U \min_{u \in U} \mathsf{pl}^{1-j}[n-|u|], \text{ where } U \text{ runs through all series of } s. \tag{2}$$

Our Algorithm 1 below makes use of Lemma 5 to compute each internal minimum in (2) in $O(1)$ time at the expense of some additional storage. By Lemma 4, this means computing of both arrays $\mathsf{pl}^j[1..n]$ in $O(n \log n)$ time. Algorithm 1 is an adaptation of the algorithm of [3] for palindromic length; but since it serves as a base for the linear-time solution, we provide all necessary details. To maintain series, we define an auxiliary array $\mathsf{left}[1..n]$: for $p \in [1..n]$, if $s[1..n]$ has a p-series, then $\mathsf{left}[p]$ is such that $s[\mathsf{left}[p]{+}1..n]$ is the longest suffix (which is not necessarily a palindrome) of $s[1..n]$ with period p; otherwise, $\mathsf{left}[p]$ is undefined.

▶ **Example 6.** If $s[1..n] = \cdots aaabaaba$ and $p = 3$, the longest 3-periodic suffix is $s[n{-}6..n] = aabaaba$ and $\mathsf{left}[3] = n - 7$. If we extend s by b, this will break period 3 and make $\mathsf{left}[3]$ undefined. If we then append ba, the resulting string $s \cdot bba = \cdots aaabaab\mathbf{abba}$ of length $n+3$ will have a suffix-palindrome of period 3 **again**, and $\mathsf{left}[3]$ will get the new value $n - 2$.

▶ Remark 7. We do *not* explicitly make $\mathsf{left}[p]$ undefined if it was defined earlier. We compute it at the iterations where a p-series is present. If the new value differs from the old one, we conclude that period p broke since we saw the previous p-series.

▶ **Lemma 8** (∗). *Suppose that the iterator contains a string $s[1..n]$ having a p-series. Given p and $|\mathsf{head}(p)|$, $\mathsf{left}[p]$ can be computed in $O(1)$ time.*

Internal minima in (2) are computed in $O(1)$ time as follows. Let $U = \{(uv)^r u, \ldots, uvu\}$, where $r > 1$, be a p-series for $s[1..n]$ (other cases from Lemma 5 are similar). In (2) we compute $m = \min\{\mathsf{pl}^{1-j}[n{-}rp{-}|u|], \ldots, \mathsf{pl}^{1-j}[n{-}p{-}|u|]\}$ to update $\mathsf{pl}^j[n]$. Note that $s[1..n]$ ends with $(uv)^r u$ but not with $(uv)^{r+1}u$: otherwise, the latter string would belong to U. Then $s[1..n{-}p]$ ends with $(uv)^{r-1}u$ but not with $(uv)^r u$ and thus has the p-series $U' = \{(uv)^{r-1}u, \ldots, uvu\}$. Thus, at $(n{-}p)$th iteration we computed $m' = \min\{\mathsf{pl}^{1-j}[n{-}rp{-}|u|], \ldots, \mathsf{pl}^{1-j}[n{-}2p{-}|u|]\}$ to update $\mathsf{pl}^j[n{-}p]$ and saved m' into an auxiliary array. Then $m = \min\{m', \mathsf{pl}^{1-j}[n{-}p{-}|u|]\}$ is computed in constant time, as required. We store all precomputed minima in two arrays $\mathsf{pre}^j[1..n]$, where $j \in \{0, 1\}$ and each $\mathsf{pre}^j[p]$ is, in turn, an array $\mathsf{pre}^j[p][0..p{-}1]$ such that

$$\mathsf{pre}^j[p][i] = \min_t \mathsf{pl}^j[t], \text{ where } t \in [\mathsf{left}[p]..n{-}p{-}1] \textbf{ and } (t - \mathsf{left}[p]) \bmod p = i$$
$$\textbf{and } s[t{+}1..n] \text{ begins with a palindrome of minimal period } p \tag{3}$$

(see the example in Fig. 1). If the minimum in (3) is taken over the empty set, $\mathsf{pre}^j[p][i]$ is undefined. Let $\{u_1 = \mathsf{head}(p), \ldots, u_r\}$ be a p-series for $s[1..n]$, $i = n - |u_1| - \mathsf{left}[p]$. Then $0 \leq i < p$ and $(n - |u_\ell| - \mathsf{left}[p]) \bmod p = i$ for all ℓ. By (3), $\mathsf{pre}^{1-j}[p][i] = \min_{\ell \in [1..r]} \mathsf{pl}^{1-j}[n{-}|u_\ell|]$, which is exactly the value m mentioned above. We denote $\mathsf{PRE}[p][i] = (\mathsf{pre}^0[p][i], \mathsf{pre}^1[p][i])$.

If $s[1..n]$ has a suffix-palindrome w centered at x, we say that w *survives* the next iteration if x is the center of a suffix-palindrome of $s[1..n{+}1]$. Otherwise, w (or x) *dies* at that iteration. We refer to the number of future iterations x survives as its *time-to-live*, denoted by $\mathsf{pttl}_n(x)$. The same notions apply to any series of $s[1..n]$ and to its period p; we write $\mathsf{ttl}_n(p)$ for the time-to-live of p. If p dies, then p-series also dies, but not vice versa.

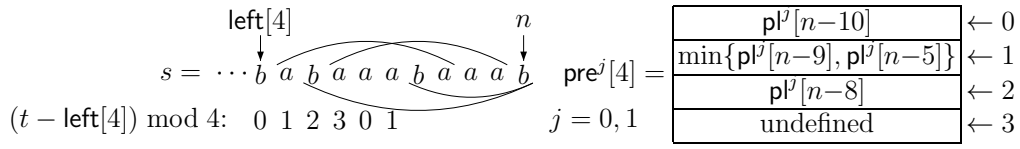

Figure 1 Precomputed values for the 4-series at the nth iteration; see (3). Residues modulo 4 are shown for all positions $t \in [\mathsf{left}[4]..n-4-1]$. Each position followed by a palindrome of minimum period 4 (an arc) contributes to the computation of an element of $\mathsf{pre}^j[4]$. The palindromes following the only position marked by 3 have minimum periods 1 (aa) and 3 ($aabaa$), but not 4.

More precisely, while p is alive, p-series evolves as follows (see Fig. 2; $(*)$). A p-series appears at nth iteration as a single palindrome $\mathsf{head}(p) = uvu$ centered at x. At the iteration $n{+}i$ such that $n + i - x = x - \mathsf{left}[p]$, x dies together with the series. At some iteration $n{+}i{+}\ell$, $0 \le \ell < p{-}i$, the series "reborns" as a palindrome centered at $x + \frac{p}{2}$ (if $\ell = 0$, we say that the series has not died). At the $(n{+}p)$th iteration, a palindrome centered at $x{+}p$ joins the series; every subsequent p iterations follow the same pattern, but the death of the head no longer means the death of the series. Knowing the structure of series allows one to store $\mathsf{PRE}[p]$ in dynamic arrays, avoiding the allocation of $\Omega(n^2)$ space. We omit the proof of the next lemma, since a stronger result (Lemma 19) is proved in Section 3.

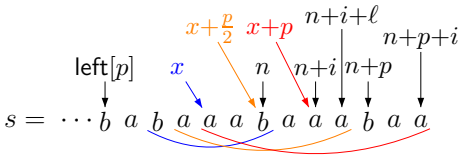

Figure 2 Evolution of a series ($p = 4, i = 2, j = 1$). 4-series appears at the nth iteration as a single palindrome $\mathsf{head}(p) = baaab$ centered at x (blue arc). At $(n{+}i)$th iteration it dies because $s[n{+}i] \ne s[\mathsf{left}[p]]$. At $(n{+}i{+}\ell)$th iteration it reappears as $\mathsf{head}(p) = aaabaaa$ centered at $x + \frac{p}{2}$ (orange arc). At $(n{+}p{+}i)$th iteration this center dies, but the series survives, because it was earlier joined by a palindrome centered at $x{+}p$. Currently $\mathsf{head}(p) = aabaaabaa$ (red arc).

▶ **Lemma 9.** *All arrays* $\mathsf{PRE}[p]$ *can be stored in a data structure requiring* $O(n \log n)$ *space,* $O(1)$ *time per deletion of an array, and amortized* $O(1)$ *time per operation with any element.*

Now we are ready for Algorithm 1. Given a new letter $s[n]$, we compute $\mathsf{PL}[n]$ as follows:

Algorithm 1 : $O(n \log n)$ algorithm, nth iteration.

```
1:  add(s[n]); PL[n] ← (∞, ∞);
2:  for (x ← maxPal; x ≠ n + 1/2; x ← nextPal(cntr(d))) do      ▷ goes to next head each time
3:      p ← len(x) − len(nextPal(x));                          ▷ min. period of the head centered at x
4:      d ← p + (len(x) mod p);                                ▷ length of candidate tail(p)
5:      if len(cntr(d)) − len(nextPal(cntr(d))) ≠ p then d ← d + p;   ▷ corrected length of tail(p)
6:      y ← left[p]; compute left[p]; i ← n−len(x)−left[p];    ▷ O(1) time by Lemma 8
7:      if left[p] > y then clear PRE[p];                      ▷ delete obsolete values
8:      if len(x) = d then PRE[p][i] ← PL[n−d];
9:      PRE[p][i] ← (min{pre^0[p][i], pl^0[n−d]}, min{pre^1[p][i], pl^1[n−d]});
10:     PL[n] ← (min{pl^0[n], 1 + pre^1[p][i]}, min{pl^1[n], 1 + pre^0[p][i]});
```

▶ **Proposition 10.** *Algorithm 1 correctly computes* $\mathsf{PL}[1..n]$ *in* $O(n \log n)$ *time.*

Proof. All add queries require $O(n)$ time in total, so we ignore them. Let x be the center of a processed suffix-palindrome $w = (uv)^r u$ with period p. Then $p = |uv| = \mathsf{len}(x) - \mathsf{len}(\mathsf{nextPal}(x))$ (see, e.g., [12, Lemmas 2,3]). Let $x' = \mathsf{cntr}(p + (\mathsf{len}(x) \bmod p))$ be the center of the suffix-palindrome uvu, $p' = \mathsf{len}(x') - \mathsf{len}(\mathsf{nextPal}(x'))$. By Lemma 5, $uvu = \mathsf{tail}(p)$ if $p' = p$ and $uvu = \mathsf{head}(p')$ otherwise. Thus in lines 3–5 Algorithm 1 computes, using $O(1)$ queries to the iterator, the period and the length of $\mathsf{tail}(p)$. This means, in particular, that the **for** loop iterates over all heads of series in s, which means $O(\log n)$ runs by Lemma 4.

If a symbol added to s breaks period p, all values in $\mathsf{pre}^j[p]$ become obsolete and should be deleted. Algorithm 1 handles this in lines 6–7, using Remark 7.

Let $\{u_1, \dots, u_r\}$ be a p-series, $i = n - |u_1| - \mathsf{left}[p]$. If $r = 1$, there was no p-series p iterations ago, so the undefined value $\mathsf{pre}^j[p][i]$ is set to $\mathsf{pl}^j[n-|u_r|]$ in line 8. Otherwise one has $\mathsf{pre}^j[p][i] = \min\{\mathsf{pl}^j[n-|u_1|], \dots, \mathsf{pl}^j[n-|u_{r-1}|]\}$ by (3), and this value is updated using $\mathsf{pl}^j[n-|u_r|]$ in line 9; so pre^j is correctly maintained. Finally, in line 10 the rule (2) is implemented. So the algorithm is correct and each run of the **for** loop takes $O(1)$ amortized time due to Lemma 9. The result now follows. ◀

3 Linear-Time Algorithm

Resources for speed-up. In some cases, dynamic programming can be sped up by a $\log n$ factor by a technique called *four Russians' trick* [1]. The idea is to store the DP array(s) in a compressed form requiring $O(1)$ bits per element and update $(\log n)$-size chunks of the compressed array using $O(1)$ operations on machine words. The key operations used are *table operations*: $f(..)$ is a table operation if it has $o(n)$ (typically $O(n^\alpha)$, where $\alpha < 1$) valid inputs and the results for all valid inputs can be computed in $o(n)$ time.

▶ Remark 11. We follow a usual scheme: all results for a constant number of table operations are computed in the $o(n)$-time preprocessing phase and stored in auxiliary tables.

This technique was used, in particular, for palindromic factorization in [12, 3] and for square factorization in [15]. To apply it to Algorithm 1, we should meet the following conditions: (i) the array PL can be compressed to $O(1)$ bits per element; (ii) each array $\mathsf{PRE}[p]$ can be compressed to $O(p + \log n)$ bits; (iii) updates of arrays $\mathsf{PL}, \mathsf{PRE}[p]$ (lines 8–10) can be performed without decompression, simultaneously for $\Omega(\log n)$ successive iterations with a constant number of operations over machine words; (iv) all intermediate states of arrays PL and $\mathsf{PRE}[p]$ during the course of the algorithm are valid inputs for the compression scheme.

For palindromic length [3] this works as follows. If w is a string and a is a letter, then $|\mathsf{pl}(wa) - \mathsf{pl}(w)| \le 1$ [16, Lemma 4.11]. Thus it is possible to store any subarray $\mathsf{pl}[i..j]$ as one number $\mathsf{pl}[i]$ followed by $(j-i)$ 2-bit codes for the differences $\mathsf{pl}[r] - \mathsf{pl}[r-1]$. The situation with the arrays $\mathsf{pre}[p]$ is subtle, but each of these arrays can be efficiently split into a constant number of chunks, where successive elements of the same chunk differ by at most one. For each chunk, the same encoding as for pl works. Range updates are based on the observation that if $s[1..n-1]$ has a suffix-palindrome with the center x, which survives next t iterations, then one can assign $\mathsf{pl}[n..n+t-1] \leftarrow \min\{\mathsf{pl}[n..n+t-1], 1 + \overleftarrow{\mathsf{pl}[2x-n-t-1..2x-n]}\}$.

Let $t = \lfloor \frac{\log n}{8} \rfloor$. Formally, a *chunk* is an array $A = A[1..h]$, where $h < t$, of $(\log n)$-bit numbers such that $|A[i] - A[i-1]| \le 1$ for all i; it is stored as a $(\log n)$-bit number followed by h 2-bit codes encoding these differences. If the length of a chunk is less than t, the unused 2-bit code is added to the end. In chunks $\overleftarrow{A}$ and $\min\{A, B\}$ the consecutive elements differ by at most 1, so they can be compressed. The next lemma allows fast performance.

▶ **Lemma 12** ([3];(∗)). *The following operations can be performed in $O(1)$ time using table operations: (1) incrementing all elements of a chunk, (2) extracting an element from a chunk, (3) extracting a chunk at any position, (4) reversing a chunk, (5) concatenating two chunks, (6) extending a chunk with dummy values, (7) taking the minimum of two chunks.*

The algorithm of [3] groups iterations into *phases*. Each phase begins immediately after the end of the previous phase and continues until one of three conditions is met: t iterations passed, the input string ended, or the longest suffix-palindrome will die at the next iteration. In the beginning of a phase, a "prediction" is made that maxPal survives the next t iterations. Under this assumption, time-to-live's of periods are computed in $O(1)$ time and range updates to pre and pl are performed for the corresponding number of iterations, using $O(1)$ operations from Lemma 12. After processing all series, actual t letters are added one by one; each time the iterator is updated, one or two new centers for palindromes appear. These palindromes are used to update pl and pre; their time-to-live's are computable in $O(1)$ time. When $s[i]$ is processed, $\mathsf{pl}[i]$ gets its true value. If an input symbol changes maxPal, the phase is aborted, unfinished updates are deleted, and a new phase is started from the current symbol.

Compression and smoothing. The following lemma allows one to compress the array PL.

▶ **Lemma 13.** *If w is a string, a, b are letters, $j \in \{0,1\}$, then $\mathsf{pl}^j(wab) \in \{\mathsf{pl}^{1-j}(wa)+1, \mathsf{pl}^{1-j}(wa)-1, \mathsf{pl}^{1-j}(w)+1\}$.*

Proof. Let $k = \mathsf{pl}^j(wab)$. Consider all palindromic factorizations of the form $wab = w_1 \cdots w_k$. Three cases are possible.
1. There is a factorization with $w_k = b$. Then $k = \mathsf{pl}^{1-j}(wa)+1$.
2. There is a factorization with $w_k = bub$, $u \neq \varepsilon$, and no factorization with $w_k = b$. Then wa has no $(k-1)$-factorization, but has a $(k+1)$-factorization $w_1 \cdots w_{k-1} bu$; so $k = \mathsf{pl}^{1-j}(wa)-1$.
3. $w_k = ab$ in each factorization (so $a = b$). Then $k = \mathsf{pl}^{1-j}(w)+1$. ◀

As above, we set $t = \lfloor \frac{\log n}{8} \rfloor$. *Double chunks* (called just *chunks* if no confusion arises) are segments $A = A[1..h] = (A^0[1..h], A^1[1..h])$ of PL or $\mathsf{PRE}[p]$. We encode them using $O(t) = O(\log n)$ bits. The difference with the case of palindromic length is that we store four explicit values: $A^0[1], A^1[1], A^0[2], A^1[2]$. Subsequent elements are encoded by 2-bit codes associated with the cases of Lemma 13, the unused 2-bit code indicates the end of a shorter chunk. Below we demonstrate a problem with this encoding and show the solution.

Let us consider the graphs of two functions $f_1(i) = \mathsf{pl}^{i \bmod 2}[i]$ and $f_2(i) = \mathsf{pl}^{(i+1) \bmod 2}[i]$ ($f_2 = \infty$ for small values of i until two consecutive equal letters occur in s). At most points, $|f_j(i) - f_j(i-1)| = 1$; if $f_j(i) - f_j(i-1) < -1$ (and thus $f_j(i) = f_{3-j}(i-2)$ by Lemma 13, case 3), we say that f_j has a *drop at* i. In the left graph in Fig. 3, drops are dash lines.

The drops cause the following problem: taking minimum of two chunks can result in a chunk having no valid encoding, violating condition (iv). For example, taking the minimum of two valid chunks $\begin{array}{|c|c|c|c|}\hline \cdots & 3 & 7 & 3 \\ \hline \cdots & 6 & 4 & 4 \\ \hline \end{array}$ and $\begin{array}{|c|c|c|c|}\hline \cdots & 1 & 7 & 3 \\ \hline \cdots & 6 & 2 & 6 \\ \hline \end{array}$, we get the chunk $\begin{array}{|c|c|c|c|}\hline \cdots & 1 & 7 & 3 \\ \hline \cdots & 6 & 2 & \mathbf{4} \\ \hline \end{array}$. The marked element does not satisfy any alternative from Lemma 13. To remedy this, we consider the following *smoothing* operation on the computed array $\mathsf{PL}[1..n-1]$: for each $i \in [1..n-2]$, $j \in \{0,1\}$, replace $\mathsf{pl}^j[i]$ by $\min\{\mathsf{pl}^j[i], \mathsf{pl}^{(j+1) \bmod 2}[i+1]+1, \ldots, \mathsf{pl}^{(j+n-1-i) \bmod 2}[n-1]+n-1-i\}$. The result of smoothing can be seen in Fig. 3 (right graph). The main property of smoothing is

▶ **Lemma 14.** *Smoothing of the array $\mathsf{PL}[1..n-1]$ does not affect the value $\mathsf{PL}[n]$ computed by rule* (1).

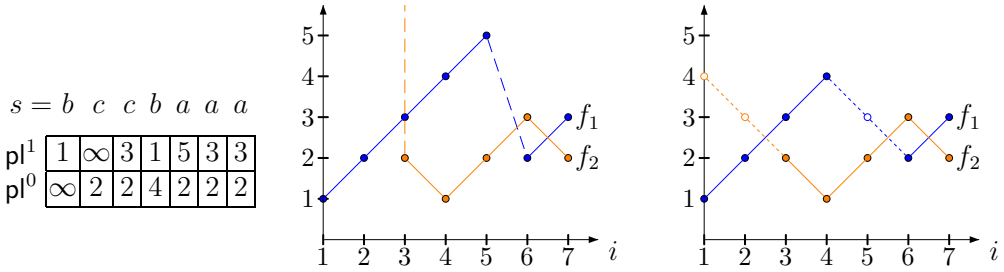

Figure 3 even/odd palindromic length visualized by $f_1(i) = \mathsf{pl}^{i \bmod 2}[i]$ and $f_2(i) = \mathsf{pl}^{(i+1) \bmod 2}[i]$. On the right, the graphs of f_1 and f_2 after smoothing the PL array.

Proof. Assume that we applied smoothing and then computed $k = \mathsf{pl}^{k \bmod 2}[n]$ by rule (1) getting the minimum $k-1$ as the element $\mathsf{pl}^{(k-1) \bmod 2}[r]$. Since the minimum cannot increase after smoothing, we just need to check that $s[1..n]$ indeed has a palindromic k-factorization. This is obvious if $\mathsf{pl}^{(k-1) \bmod 2}[r]$ was not changed by smoothing. Otherwise, $k-1 = \mathsf{pl}^{(k-1+i-r) \bmod 2}[i] + i - r$ for some $i > r$. Since $s[r+1..n]$ is a palindrome (see (1)), we factor it as $uv\overleftarrow{u}$, where $u = s[r+1..i]$. So, $s[1..n] = s[1..i]v\overleftarrow{u}$ indeed has a k-factorization: $s[1..i]$ can be factored into $k-1-i+r$ palindromes and $\overleftarrow{u}$ into $i-r$ 1-symbol palindromes. ◀

▶ Remark 15.

1) We extend smoothing to subarrays of the form $\mathsf{PL}[l..r]$: for each $i \in [l..r-1]$, $j \in \{0,1\}$, replace $\mathsf{pl}^j[i]$ by $\min\{\mathsf{pl}^j[i], \mathsf{pl}^{(j+1) \bmod 2}[i+1] + 1, \ldots, \mathsf{pl}^{(j+r-i) \bmod 2}[r] + r - i\}$. The resulting values are always between the values from the original PL and those from smoothed PL, so Lemma 14 stays true for this *local* smoothing.
2) From formula (3) one can conclude that smoothing also works for the arrays $\mathsf{PRE}[p]$. Namely, if $\mathsf{pre}^j[p][i]$ and $\mathsf{pre}^{1-j}[p][i-1]$ are used for computing $\mathsf{pl}^{1-j}[n]$ and $\mathsf{pl}^j[n+1]$ respectively (i.e, a p-series survives during these two iterations), then $\mathsf{pre}^{1-j}[p][i-1]$ can be replaced by $\min\{\mathsf{pre}^{1-j}[p][i-1], \mathsf{pre}^j[p][i] + 1\}$. This replacement can be iterated for a range.
3) Observations 1), 2) allow one to use the following principles: extracting a chunk from PL or PRE, smooth it; concatenating two chunks in PRE, smooth the result; taking the minimum of two smoothed chunks of different length, extend the shorter chunk with dummy values satisfying $f_j(i) - f_j(i+1) = 1$ (resp., $= -1$) for left (resp., right) extensions.

The success of the approach described in Remark 15 relies on two lemmas.

▶ **Lemma 16.** *Smoothing a (double) chunk can be done in $O(1)$ time.*

Proof. Suppose that a chunk $A[1..h], h \le t$, is given, so we know the numbers $f_1(1), f_2(1)$, $f_1(2), f_2(2)$, and the 2-bit codes $b_3^1, \ldots, b_h^1, b_3^2, \ldots, b_h^2$, where the functions f_1, f_2 are defined as above and b_i^j encodes the case from Lemma 13 for $f_j(i)$. We write $\tilde{A}, \tilde{f}_1, \tilde{f}_2$ for the chunk and the functions after smoothing. If f_j has a drop at 2, we can replace $f_j(1)$ with $f_j(2)+1$ without affecting $\tilde{f}_j$. With this reservation, we can restore f_1, f_2 from $f_1(2), f_2(2), b_2^1, \ldots, b_h^1, b_2^2, \ldots, b_h^2$. Indeed, compute $f_j(1)$ for $j = 1, 2$ from $f_j(2)$ and b_2^j; then consider $f_j(3)$. If f_j has no drop at 3, it is computed from $f_j(2)$, b_3^j. If it has this drop, then $f_{3-j}(2) \le f_{3-j}(1) + 1 = f_j(3) \le f_j(2) - 3 \le f_j(0) - 1$ (by 0 we mean the position preceding the first position in A), and so f_{3-j} has no drop at 2. Then the computed value $f_{3-j}(1)$ is true, and we put $f_j(3) = f_{3-j}(1) + 1$. Knowing $f_j(2)$ and $f_j(3)$, it is easy to reconstruct the rest.

For any constant c, one can obtain $\tilde{f}_j + c$ from $f_j + c$. So to get a table operation we replace $f_2(2)$ with 0 and $f_1(2)$ with $\delta = f_1(2) - f_2(2)$. Now observe that if $|\delta| \geq 2h$, only two cases are possible, and they are easy to distinguish: either there are no drops, so $\tilde{f}_j = f_j$ for $j = 1, 2$, or the leftmost drop is in f_j at position i, and then $\tilde{f}_j(i - r) = f_j(i) + r$ for all $r < i$. So in this case only the sign of δ matters, and we assign $\delta = \pm\infty$. Thus we have got a table operation which, given $(2t - 2)$ 2-bit codes and a number δ with $O(t)$ distinct values, returns a compressed chunk; adding $f_2(2)$ to the explicit values in this chunk, one gets $\tilde{A}$. ◀

▶ **Lemma 17** (∗). *The following operations with (double) chunks can be performed in $O(1)$ time: (1) incrementing all elements of a chunk, (2) extracting an element from a chunk, (3) extracting a chunk at any position, (4) concatenating two chunks, (5) extending a smoothed chunk with dummy values, (6) taking the minimum of two smoothed chunks.*

Algorithm EOPL. We describe one phase of Algorithm EOPL computing $\mathsf{pl}^0(s)$ and $\mathsf{pl}^1(s)$ for a string s. The definition of phase and the idea to "predict" the next t input symbols and perform range updates by means of table operations are the same as for palindromic length.

Prerequisites. P1. Arrays PL and PRE[p] are stored as sequences of compressed length-t chunks (the last chunk in a sequence can be short). We maintain a *work chunk* W to compute the reversal of a new chunk of PL during the current phase; we move symbols from W to PL one by one and thus avoid the reversal operation. All chunks applied to W are aligned to its current right end (corresponding to the current iteration). List wait stores new palindromes and palindromes that changed periods, to perform PRE updates at the end of the phase.

P2. Using Lemma 14 and Remark 15, we smooth chunks extracted from PL and extend short smoothed chunks before applying min operations. Each array PRE[p] consists of smoothed chunks and W is also smoothed; drops may occur between the last element of PL and the first element of W as a result of processing 2-letter suffix-palindromes at step F3a.

P3. *Big* p-series ($p \geq t$) and *small* p-series ($p < t$) are processed separately.

First iteration. F1. Read the next symbol (say, $s[n]$), perform add($s[n]$); at this moment

- **a.** the iterator stores $s[1..n]$ and returns the list dead;
- **b.** the array PL[$1..n-1$] and the arrays PRE[p] for $s[1..n-1]$ are correctly computed;
- **c.** the chunk W is initialized by ∞'s, the list wait is empty;
- **d.** the maximum number of iterations in the phase is set to $t' = \min\{n - \mathsf{len}(\mathsf{maxPal}), t\}$ (if the first number is smaller, maxPal must change after at most t' iterations).

F2. Loop through the list of big p-series of $s[1..n-1]$:

- **a.** compute left[p] and clear PRE[p] if necessary;
- **b.** compute $\mathsf{ttl}'[p] = \min\{\mathsf{ttl}_{n-1}[p], t'\}$ using symmetry inside palindromes;
- **c*.** using left[p] and $\mathsf{ttl}'[p]$, update PRE[p], W, and wait (details in a separate item below).

F3. Process new suffix palindromes with centers $n - \frac{1}{2}$ (if $s[n-1] = s[n]$) and n:

- **a.** update W directly using the chunks of PL ending at position $n-1$ (and possibly $n-2$);
- **b.** using symmetry again, check whether n and/or $n - \frac{1}{2}$ remain centers of suffix-palindromes after the last iteration of the phase; add the center(s) with the answer "yes" to the list wait.

F4. Small series and finalization:

- **a.** extract from W its last element $m = (m_0, m_1)$ and delete it from W;

b. assign $(\mathsf{pl}^0[n], \mathsf{pl}^1[n]) = (m_0, m_1)$ and process small p-series of $s[1..n{-}1]$ directly by Algorithm 1, getting the final value of $\mathsf{PL}[n]$;

c. update arrays $\mathsf{PRE}[p]$ with all palindromes from the list dead.

Subsequent iterations. S1. Read $s[i]$, compare it to $s[i{-}1{-}\mathsf{len}(\mathsf{maxPal})]$.

S2. If $s[i]$ extends the longest suffix-palindrome, the phase continues: perform $\mathsf{add}(s[i])$ and then the operations from steps F3 and F4, replacing n by i.

S3. If $s[i]$ breaks the longest suffix-palindrome, the phase is aborted:

a. update the lists $\mathsf{PRE}[p]$ using suffix-palindromes from the list wait, clear wait;

b. start the next phase with $\mathsf{add}(s[i])$.

S4. If there was no abortion and all t' symbols are processed, the phase is terminated:

a. update the lists $\mathsf{PRE}[p]$ using suffix-palindromes from wait, clear wait;

b. start the next phase with $\mathsf{add}(s[n{+}t'])$.

F2c detailed. Let x be the center of $\mathsf{tail}(p)$, $\mathsf{pttl}'(x) = \min\{\mathsf{pttl}_{n-1}(x), t'\}$, $d = \min\{\mathsf{ttl}'(p), \mathsf{pttl}'(x)\}$. Update $\mathsf{PRE}[p]$ with a length-d chunk from PL and W with the updated length-d chunk from $\mathsf{PRE}[p]$. If $\mathsf{ttl}'(p) < \mathsf{pttl}'(x)$ (after d iterations period p dies, but $\mathsf{head}(p) = \mathsf{tail}(p)$ survives, becoming a palindrome with a bigger period), additionally update W with a chunk of length $\mathsf{pttl}'(x)$; if $\mathsf{pttl}'(x) \geq t'$, add x to wait. Note that if p-series gets new tail $x{+}\frac{p}{2}$ during the phase, then at the start of the phase $x{+}\frac{p}{2}$ was the center of the head of some p'-series, survived the death of that series, and thus was used for an additional update of W. Necessary updates of $\mathsf{PRE}[p]$ are carried when the lists dead and wait are processed. All the same stays true for $x{+}p$ which is the next potential tail of the p-series.

▶ **Lemma 18.** *Algorithm EOPL correctly computes the array* PL.

Proof. The computation of PL by Algorithm EOPL differs from the correct computation by Algorithm 1 in a few points. The use of smoothed chunks is justified by Lemmas 14, 16, so below we take smoothing into account speaking about correctness of the arrays $\mathsf{PRE}[p]$. We prove the lemma by induction, with an obvious base and F1b as the hypothesis; more precisely, we assume that at the start of a new phase with $s[n]$, the array $\mathsf{PL}[1..n{-}1]$ and all arrays $\mathsf{PRE}[p]$ are correct, where p runs through the set of live periods of $s[1..n{-}1]$ (i.e., $s[1..n{-}1]$ has a p-periodic suffix with at least one p-periodic palindrome in it).

In an aborted phase, some predictions in F2c are made beyond the actual end of the phase (the phase processes $s[n..n{+}i{-}1]$, and updates are made for more than i elements of W and $\mathsf{PRE}[p]$). For W, this does not matter, because W is translated to PL one element per iteration (F4a) and is initialized at the beginning of each phase (F1c). For $\mathsf{PRE}[p]$, the situation means that p will become dead at the $(n{+}i)$th iteration: the actual symbol $s[n{+}i]$ differs from the predicted symbol which preserved the period.

Next, if p is alive for $s[1..n{+}i{-}1]$, all necessary updates for $\mathsf{PRE}[p]$ were made at steps F2c, F4c (useful if a palindrome dies while its period survives), and S3a/S4a. Note that repeated updates using the same palindrome (say, first at step F2c and then at F4b) cannot harm. Finally, all palindromes ending in $s[n{-}1..n{+}i{-}2]$ were used to update W, so $\mathsf{PL}[n..n{+}i{-}1]$ is computed according to rule (1). The result now follows. ◀

Details and Analysis of Algorithm EOPL. To prove Theorem 1, it remains to show that the details of computation can be organized to provide the linear-time performance. The key part is maintaining $\mathsf{PRE}[p]$ arrays.

▶ **Lemma 19.** *All arrays* $\mathsf{PRE}[p]$ *can be stored in a compressed form in a data structure requiring* $O(n)$ *space,* $O(1)$ *time per deletion of an array, and amortized* $O(1)$ *time per operation with any chunk.*

Proof. We use dynamic arrays (like *vectors* in C^{++}). Such an array has *size* (space in use) and *capacity* (allocated space). When the size increases and reaches the capacity, the latter doubles; this step rebuilds the array to provide a constant-time access to its elements and takes the time linear in its size. Thus, adding an element can be done in amortized $O(1)$ time, an existing element can be modified in $O(1)$ time, an array can be cleared by setting its size to 0, and the total allocated space is proportional to the maximum size reached.

For each array $\mathsf{PRE}[p]$, we store integers R_1, I_1, R_2, I_2 and dynamic arrays F_1 and F_2, both initially of size 0 and capacity 1. We also perform (amortized) constant-time chunk operations listed in Lemma 17. When we need to extract a chunk for updating the array W, we take the minimum of the corresponding chunks from F_1 and F_2. We refer to the evolution of p-series described at p. 5. Note that $\mathsf{PRE}[p]$ is always filled right to left (see Fig. 1, 2): starting with the index $i-1$ in the first pass, with the index $p-\ell-1$ in the second pass, and with the index $p-1$ in subsequent passes. During its time-to-live, p-series is processed in three stages. To decide the current stage, the indicators described below are used.

1. A p-series appears at nth iteration as a palindrome uvu centered at x; at this or one of subsequent iterations we get a first chunk to add to $\mathsf{PRE}[p]$. We set R_1 to the index of the first element added to $\mathsf{PRE}[p]$ at nth iteration; it is $\mathsf{ttl}_n(x)$ or $i-1$ in terms of Fig. 2. Then we write the first chunk to F_1 and set I_1 to the last used index. After that, we possibly add more chunks until x dies, updating I_1 respectively. (Each chunk is concatenated with the previous one such that all of them except the last one have length t.) At this moment, $I_1 = 0$ and the series dies. The indicator of stage 1 is $(\mathit{size}(F_2) = 0) \wedge (I_1 > 0)$.

2. At one of subsequent iterations ($n+i+\ell$ in Fig. 2) the series reborns with the new head/tail $x+\frac{p}{2}$. Getting the next chunk, we set R_2 to the index $p-\ell-1$, write the chunk, set I_2 to the last used index, and proceed with subsequent chunks, using concatenation and updating I_2. At some point we get $I_2 \le R_1+1 = i$. This means that the tail of the series has changed to the palindrome centered at $x+p$ either during the phase or immediately after it. All updates made after this change make no sense (we have written to F_2 the elements already written to F_1); so we set $I_2 = i$ and wait for the next chunk (if the series survives the current phase, the chunk will appear when the list wait will be processed). From this point, we take minimum of the new chunks with the corresponding chunks of F_1 and then add the result to F_2. The stage continues until $I_2 = 0$. The indicator of stage 2 is $(I_1 = 0) \wedge (R_1 < n-1)$.

3. Getting $I_2 = 0$ at the previous stage, we set $R_1 = n-1, I_1 = n$, and clear F_1. All subsequent chunks are written to F_1, in the way described in the previous stages. When we get $I_1 \le i$ for the first time (change of the tail of the series), we set I_1 to i and truncate the last chunk. When $I_1 = 0$ is reached, we reset $I_1 = p$ and set a flag telling that each next chunk should be written as the minimum of the new chunk and the existing chunk at the same positions of the array. The indicator of stage 3 is $R_1 = n-1$.

Thus, we perform each update in $O(1)$ time (amortized, due to the properties of dynamic arrays). For $j = 1, 2$, all chunks in F_j, except for the last one, have length t, so it is easy to find the argument for the "extract a chunk" operation in $O(1)$ time using R_j. We finally note that the total number of stored chunks is $O(n)$, each requiring $O(1)$ machine words. ◀

Let us prove time bounds for all steps of Algorithm EOPL. All add queries require $O(n)$ time in total. By Lemma 8, F2a requires $O(1)$ time; F2b is covered by Lemma 20 below.

▶ **Lemma 20.** *For a big* p*-series, the value* $\mathsf{ttl}'[p]$ *can be computed in* $O(1)$ *time.*

Proof. Let x be the center of the head $(uv)^r u$ of the p-series. We compute $|u| = \mathsf{len}(x) \bmod p$ and $y = \mathsf{cntr}(|u|)$ as in Lemma 8; $y' = 2\mathsf{maxPal} - y$ is the center of the prefix-palindrome u of the longest suffix-palindrome $s[i..n{-}1]$ of $s[1..n{-}1]$. Note that the longest suffix-palindrome of $s[1..n{+}t'{-}1]$ is $z = s[i{-}t'..n{+}t'{-}1]$ because maxPal does not change during a phase. Let w be the longest palindrome with the center y'; using the query $\mathsf{rad}(y')$ we determine whether w is located inside z or begins outside it. In the latter case, z has a p-periodic prefix ending with $(uv)^r u$ and then a p-periodic suffix beginning with $(uv)^r u$. So $\mathsf{ttl}_{n-1}[p_i] \geq t'$ and then $\mathsf{ttl}'[p_i] = t'$. In the former case, $w = \overleftarrow{w_1} u w_1$, where w_1 is a proper prefix of vu because $|vu| = p_i > t' \geq |w_1|$. Hence the period p_i breaks on the left of w, which means that $\mathsf{ttl}_{n-1}[p_i] = |w_1| = (\mathsf{len}(y') - \mathsf{len}(y))/2$. All computations above take $O(1)$ time. The result now follows. ◀

F3b uses the same idea as Lemma 20: for the center $n{+}i$, compare $\mathsf{rad}(2\mathsf{maxPal} - n - i)$ to the number of remaining iterations. Both F3a and F4a require, per iteration, $O(1)$ constant-time chunk operations from Lemma 17. Each update in step F4b can be also performed by $O(1)$ such operations: extract elements of $\mathsf{PRE}[p]$ and PL, take minima, and "return" the updated element to $\mathsf{PRE}[p]$, extending 1-element chunk and using min. Hence, by Lemma 21 below, F4b takes $O(p + i)$ time per phase of i iterations, where p is the longest small period.

▶ **Lemma 21.** *In a phase of i iterations, the number of times the **for** loop of Algorithm 1 processes series which existed at the start of the phase and have periods $\leq p$, is $O(p + i)$.*

Proof. We prove two claims.

▷ Claim 1. If $s[1..n]$ has p-series and q-series such that $p > q$ and $\mathsf{ttl}_n(p) \geq p$, then $\mathsf{ttl}_n(q) < p$.

Let $\mathsf{head}(p) = (uv)^r u$. If $\mathsf{ttl}(q) \geq p$, the string $s[n{-}q{+}1..n{+}p]$ of length $p + q$ has periods p and q. Hence it has period $d = \gcd(p, q)$ by the Fine–Wilf theorem [5]. Thus vu is a (p/d)-power of a shorter word; so $(uv)^r u$ has period $d < p$, contradicting the definition of p-series. The claim is proved.

▷ Claim 2. Let a string $s[1..n]$ have series with periods $p = p_1 > p_2 > \cdots > p_\ell$ and let $i > 0$. Then $\sum_{c=1}^{\ell} \min\{\mathsf{ttl}_n(p_c), i\} = O(p + i)$.

Divide the periods in two groups: those with $\mathsf{ttl}_n(p) < p$ and the rest. In the first group, the sum of ttl's is upper bounded by $\sum_{c=1}^{\ell} p_c$, which is $O(p)$ by Lemma 4. For the second group, $\mathsf{ttl}_n(p_c)$ is smaller than the previous period from this group by Claim 1. So we can take $i + \sum_{c=1}^{\ell} p_c = O(p + i)$ as the upper bound. The claim now follows.

The statement of the lemma is immediate from Claim 2 and definitions. ◀

Let P be the minimum period of the longest suffix-palindrome of s at the beginning of the phase. In F2c, the updates to W and wait require $O(1)$ time per series, which is $O(\log P)$ per phase by Lemma 4. The updates to $\mathsf{PRE}[p]$ arrays are considered in Lemma 22 below.

▶ **Lemma 22.** *During a phase of length i, all range updates of $\mathsf{PRE}[p]$ at steps F2c and S3a/S4a spend $O(\log P + i)$ time in total.*

Proof. Altogether, there are $O(\log P)$ series of palindromes in a phase, $O(\log P)$ (from F2c) plus $O(i)$ (from F3b) palindromes in the list wait in a phase. Due to Lemma 17, it suffices to prove that each update requires $O(1)$ chunks. For big periods, this is obvious from the description of Algorithm EOPL. Consider small periods. For each palindrome w, it is enough to update the range of $\mathsf{PRE}[p]$ corresponding to the iterations where $w = \mathsf{tail}(p)$. If this range

begins with $\mathsf{PRE}[p][\ell]$ then it ends not later than $\mathsf{PRE}[p][0]$ is reached (recall that $\mathsf{PRE}[p]$ is filled right to left). Thus we can cut the range computed from time-to-live so that it will fit into one chunk. The lemma now follows. ◀

Finally, for the updates of PRE at step F4c, the argument of Lemma 22 about $O(1)$ chunks per update also works. All lists dead for the total run of Algorithm EOPL contains $O(n)$ palindromes, because the iterator works in $O(n)$ time. Thus, for F4c we get the cumulative $O(n)$ time bound. For all remaining steps, the time bound we proved is $O(\log P + i)$ per phase, except $O(p + i)$ per phase in Lemma 21. If $i = t$ (the phase is terminated), then we spend $O(t)$ time for t iterations. Completing a short phase (either aborted or satisfying $t' < t$), we increase maxPal by at least $P/2$ in time $O(P)$. Since maxPal never decreases, we get the overall $O(n)$ time bound. Theorem 1 is proved.

4 Computing k-factorization

A run of linear-time Algorithm EOPL leaves the iterator containing the input string $s = s[1..n]$ and also the arrays $\mathsf{pl}^0[1..n]$, $\mathsf{pl}^1[1..n]$ for s. Let us use them to construct an explicit k-factorization in additional linear time. Let $k' = \mathsf{pl}^{k \bmod 2}[n]$. By Lemma 2, it suffices to build a k'-factorization of s in $O(n)$ time, which can be done as follows.

Algorithm 2 Computing k'-factorization.

1: **for** $(i \leftarrow 1; i < k'; i{+}{+})$ **do** build a list $\mathsf{list}[i]$ of all positions $j : \mathsf{pl}^{i \bmod 2}[j] = i$
2: $end \leftarrow n$; $factors \leftarrow \{\}$
3: **for** $(i \leftarrow k' - 1; i > 1; i{-}{-})$ **do** ▷ main cycle: building the k'-factorization
4: **for** $j \in \mathsf{list}[i]$ **do**
5: **if** $\mathsf{len}(\frac{j+1+end}{2}) \geq end - j$ **then** break ▷ $s[j{+}1..end]$ is a palindrome
6: add $s[j{+}1..end]$ to $factors$; $end \leftarrow j$
7: add $s[1..end]$ to $factors$ ▷ after last assignment in line 6, $end \in \mathsf{list}[1]$

Algorithm 2 discovers factors of a k'-factorization of s in reversed order, using an observation that if a string w has even (odd) palindromic length i then $w = uv$, where u has odd (resp., even) palindromic length $i{-}1$ and v is a nonempty palindrome. All lists are built in parallel in linear time. In the main cycle, each position serves as j at most once, so the algorithm performs $O(n)$ len queries to the iterator. Thus we proved

▶ **Theorem 23.** *There is a linear-time word-RAM algorithm which, given a string s over a general alphabet and a number k, builds a palindromic k-factorization of s or reports that no such factorization exists.*

References

1 V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economical construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk SSSR*, 194(11):1209–1210, 1970.

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 *Developments in Language Theory - 19th International Conference, DLT 2015, Liverpool, UK, July 27-30, 2015, Proceedings*, volume 9168 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 2015. doi:10.1007/978-3-319-21500-6_6.

3 Kirill Borozdin, Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Palindromic length in linear time. In *28th Annual Symposium on Combinatorial Pattern Matching, CPM*

2017, July 4-6, 2017, Warsaw, Poland, volume 78 of *LIPIcs*, pages 23:1–23:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.23.

4 G. Fici, T. Gagie, J. Kärkkäinen, and D. Kempa. A subquadratic algorithm for minimum palindromic factorization. *J. of Discrete Algorithms*, 28:41–48, 2014. doi:10.1016/j.jda.2014.08.001.

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

6 Anna E. Frid. Sturmian numeration systems and decompositions to palindromes. *Eur. J. Comb.*, 71:202–212, 2018. doi:10.1016/j.ejc.2018.04.003.

7 Anna E. Frid, Svetlana Puzynina, and Luca Q. Zamboni. On palindromic factorization of words. *Adv. Appl. Math.*, 50(5):737–748, 2013. doi:10.1016/j.aam.2013.01.002.

8 Z. Galil and J. Seiferas. A linear-time on-line recognition algorithm for "Palstar". *J. ACM*, 25:102–111, 1978.

9 T. I, S. Sugimoto, S. Inenaga, H. Bannai, and M. Takeda. Computing palindromic factorizations and palindromic covers on-line. In *CPM 2014*, volume 8486 of *LNCS*, pages 150–161. Springer, 2014.

10 R. C. Lyndon K.-T. Chen, R. H. Fox. Free differential calculus. iv. the quotient groups of the lower central series. *Annals of Mathematics, 2nd Ser.*, 68(1):81–95, 1958. doi:10.2307/1970044.

11 D. E. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Comp.*, 6:323–350, 1977.

12 D. Kosolobov, M. Rubinchik, and A. M. Shur. Pal^k is linear recognizable online. In *SOFSEM 2015*, volume 8939 of *LNCS*, pages 289–301. Springer, 2015. doi:10.1007/978-3-662-46078-8_24.

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

14 G. Manacher. A new linear-time on-line algorithm finding the smallest initial palindrome of a string. *J. of the ACM*, 22(3):346–351, 1975.

15 Yoshiaki Matsuoka, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, and Florin Manea. Factorizing a string into squares in linear time. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016*, volume 54 of *LIPIcs*, pages 27:1–27:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

16 Mikhail Rubinchik and Arseny M. Shur. EERTREE: an efficient data structure for processing palindromes in strings. *Eur. J. Comb.*, 68:249–265, 2018. doi:10.1016/j.ejc.2017.07.021.

17 Aleksi Saarela. Palindromic length in free monoids and free groups. In *Combinatorics on Words - 11th International Conference, WORDS 2017, Montréal, QC, Canada, September 11-15, 2017, Proceedings*, volume 10432 of *Lecture Notes in Computer Science*, pages 203–213. Springer, 2017. doi:10.1007/978-3-319-66396-8_19.