Pal^k Is Linear Recognizable Online

Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur

Ural Federal University, Ekaterinburg, Russia
dkosolobov@mail.ru, mikhail.rubinchik@gmail.com, arseny.shur@usu.ru

Abstract. Given a language L that is online recognizable in linear time and space, we construct a linear time and space online recognition algorithm for the language $L \cdot \mathrm{Pal}$, where Pal is the language of all nonempty palindromes. Hence for every fixed positive k, Pal^k is online recognizable in linear time and space. Thus we solve an open problem posed by Galil and Seiferas in 1978.

Keywords: linear algorithms, online algorithms, palindrome, factorization, palstar, palindrome decomposition

1 Introduction

In the last decades the study of palindromes constituted a notable branch in formal language theory. Recall that a string $w = a_1 \cdots a_n$ is a *palindrome* if it is equal to $\overleftarrow{w} = a_n \cdots a_1$. There is a bunch of papers on palindromes in strings. Some of these papers contain the study of strings "rich" in palindromes (see, e.g., [GJWZ]), some other present solutions to algorithmic problems like finding the longest prefix-palindrome [Man] or counting distinct subpalindromes [KRS].

For languages constructed by means of palindromes, an efficient recognition algorithm is often not straightforward. In this paper we develop a useful tool for construction of acceptors for such languages. Before stating our results, we recall some notation and known facts.

The language of nonempty palindromes over a fixed alphabet is denoted by Pal. Let $\mathrm{Pal}_{\mathrm{ev}} = \{w \in \mathrm{Pal} \colon |w| \text{ is even}\}$, $\mathrm{Pal}_{>1} = \{w \in \mathrm{Pal} \colon |w| > 1\}$. Given a function $f \colon \mathbb{N} \to \mathbb{N}$ and a language L, we say that an algorithm *recognizes L in $f(n)$ time and space* if for any string w of length n, the algorithm decides whether $w \in L$ using at most $f(n)$ time and at most $f(n)$ additional space. We say that an algorithm recognizes a given language *online* if the algorithm processes the input string sequentially from left to right and decides whether to accept each prefix after reading the rightmost letter of that prefix.

It is well known that every context-free language can be recognized by relatively slow Valiant's algorithm (see [Val]). According to [Lee], there are still no examples of context-free languages that cannot be recognized in linear time on a RAM computer. Some "palindromic" languages were considered as candidates to such "hard" context-free languages.

At some point, it was conjectured that the languages ${\mathrm{Pal}_{\mathrm{ev}}}^*$ and ${\mathrm{Pal}_{>1}}^*$, where * is a Kleene star, cannot be recognized in $O(n)$ (see [KMP, Section 6]).

But a linear algorithm for the former was given in [KMP] and for the latter in [GS]. The recognition of Pal^k appeared to be a more complicated problem. Linear algorithms for the cases $k = 1, 2, 3, 4$ were given in [GS]. Their modified versions can be found in [CR, Section 8]. In [GS] and [CR] it was conjectured that there exists a linear time recognition algorithm for Pal^k for arbitrary k. In this paper we present such an algorithm. Moreover, our algorithm is online. The main contribution is the following result.

Theorem. *Suppose a given language L is online recognizable in $f(n)$ time and space, for some function $f : \mathbb{N} \to \mathbb{N}$. Then the language $L \cdot \mathrm{Pal}$ can be recognized online in $f(n) + cn$ time and space for some constant $c > 0$ independent of L.*

Corollary. *For arbitrary k, Pal^k is online recognizable in $O(kn)$ time and space.*

Note that the related problem of finding the minimal k such that a given string belongs to Pal^k can be solved online in $O(n \log n)$ time [FGKK], and it is not known whether a linear algorithm exists.

The paper is organized as follows. Section 2 contains necessary combinatorial properties of palindromes; similar properties were considered, e.g., in [BG]. In Sect. 3 we describe an auxiliary data structure used in the main algorithm. An online recognition algorithm for Pal^k with $O(kn \log n)$ working time is given in Sect. 4. Finally, in Sect. 5 we speed up this algorithm to obtain the main result.

2 Basic Properties of Palindromes

A *string of length* n over the alphabet Σ is a map $\{1, 2, \ldots, n\} \mapsto \Sigma$. The length of w is denoted by $|w|$ and the empty string by ε. We write $w[i]$ for the ith letter of w and $w[i..j]$ for $w[i]w[i{+}1] \ldots w[j]$. Let $w[i..i{-}1] = \varepsilon$ for any i. A string u is a *substring* of w if $u = w[i..j]$ for some i and j. The pair (i, j) is not necessarily unique; we say that i specifies an *occurrence* of u in w. A string can have many occurrences in another string. A substring $w[1..j]$ (resp., $w[i..n]$) is a *prefix* [resp. *suffix*] of w. An integer p is a *period* of w if $w[i] = w[i{+}p]$ for $i = 1, \ldots, |w|{-}p$.

Lemma 1 (see [Lot, Chapter 8]). *Suppose v is both a prefix and a suffix of a string w; then the number $|w|{-}|v|$ is a period of w.*

A substring [resp. suffix, prefix] of a given string is called a *subpalindrome* [resp. *suffix-palindrome*, *prefix-palindrome*] if it is a palindrome. We write $w = (uv)^*u$ to state that $w = (uv)^k u$ for some nonnegative integer k. In particular, $u = (uv)^*u$, $uvu = (uv)^*u$.

Lemma 2. *Suppose p is a period of a nonempty palindrome w; then there are palindromes u and v such that $|uv| = p$, $v \neq \varepsilon$, and $w = (uv)^*u$.*

Proof. Let uv be a prefix of w of length p such that $v \neq \varepsilon$ and $w = (uv)^*u$. Since $w = \overleftarrow{w} = (\overleftarrow{u}\overleftarrow{v})^*\overleftarrow{u}$, we see that $u = \overleftarrow{u}$ and $v = \overleftarrow{v}$.

Lemma 3. *Suppose w is a palindrome and u is its proper suffix-palindrome or prefix-palindrome; then the number $|w|-|u|$ is a period of w.*

Proof. Let $w = vu$ for some v. Hence $vu = w = \overleftarrow{w} = \overleftarrow{u}\overleftarrow{v} = u\overleftarrow{v}$. It follows from Lemma 1 that $|v|$ is the period of w. The case of a prefix-palindrome is similar.

Lemma 4. *Let u, v be palindromes such that $v \neq \varepsilon$ and $uv = z^k$ for some string z and integer k; then there exist palindromes x and y such that $z = xy$, $y \neq \varepsilon$, $u = (xy)^*x$, and $v = (yx)^*y$.*

Proof. The case $k = 1$ is trivial. Suppose $k > 1$. Consider the case $|z| \leq |u|$. It follows from Lemma 2 that there exist palindromes x, y such that $z = xy$, $y \neq \varepsilon$, $u = (xy)^*x$. Since $z^k = (xy)^k = uv$, we have $v = (yx)^*y$. The case $|z| \leq |v|$ is similar.

A string is *primitive* if it is not a power of a shorter string. A string is called a *palindromic pair* if it is equal to a concatenation of two palindromes.

Lemma 5. *A palindromic pair w is primitive iff there exists a unique pair of palindromes u, v such that $v \neq \varepsilon$ and $w = uv$.*

Proof. Let w be a non-primitive palindromic pair. Suppose $w = z^k = uv$, where z is a string, $k > 1$, and u, v are palindromes. By Lemma 4, we obtain palindromes x, y such that $z = xy$ and $y \neq \varepsilon$. Now $w = u_1v_1 = u_2v_2$, where $u_1 = x, v_1 = y(xy)^{k-1}, u_2 = xyx, v_2 = y(xy)^{k-2}$ are palindromes.

For the converse, consider $w = u_1v_1 = u_2v_2$, where u_1, u_2, v_1, v_2 are palindromes and $|u_1| < |u_2| < |w|$. We claim that w is not primitive. The proof is by induction on the length of w. For $|w| \leq 2$, there is nothing to prove. Suppose $|w| > 2$. It follows from Lemmas 2 and 3 that there exist palindromes x, y such that $u_2 = u_1yx = (xy)^*x$. In the same way we obtain palindromes x', y' such that $v_1 = y'x'v_2 = (y'x')^*y'$. Hence $yx = y'x'$. Let z be a primitive string such that $yx = z^k$ for some $k > 0$. By Lemma 4, we obtain palindromes $\tilde{x}, \tilde{y}$ such that $x = (\tilde{x}\tilde{y})^*\tilde{x}$, $y = (\tilde{y}\tilde{x})^*\tilde{y}$, and $z = \tilde{y}\tilde{x}$. Similarly, we have palindromes $\tilde{x}', \tilde{y}'$ such that $x' = (\tilde{x}'\tilde{y}')^*\tilde{x}'$, $y' = (\tilde{y}'\tilde{x}')^*\tilde{y}'$, and $z = \tilde{y}'\tilde{x}'$. By induction hypothesis, $\tilde{x} = \tilde{x}'$ and $\tilde{y} = \tilde{y}'$. Finally, $w = (\tilde{x}\tilde{y})^{k'}$ for some $k' > 1$.

Denote by p the minimal period of a palindrome w. By Lemma 2, we obtain palindromes u, v such that $w = (uv)^*u$, $v \neq \varepsilon$, and $|uv| = p$. The string uv is primitive. The representation $(uv)^*u$ is called *canonical decomposition* of w. Let $w[i..j]$ be a subpalindrome of the string w. The number $(i+j)/2$ is the *center* of $w[i..j]$. The center is integer [half-integer] if the subpalindrome has an odd [resp., even] length. For any integer n, $\mathrm{shl}\,(w, n)$ denotes the string $w[t{+}1..|w|]w[1..t]$, where $t = n \bmod |w|$.

Lemma 6 (see [Lot, Chapter 8]). *A string w is primitive iff for any integer n, the equality* $\mathrm{shl}\,(w, n) = w$ *implies* $n \bmod |w| = 0$.

Lemma 7. *Suppose $(xy)^*x$ is a canonical decomposition of w and u is a subpalindrome of w such that $|u| \geq |xy|-1$; then the center of u coincides with the center of some x or y from the decomposition.*

Proof (of Lemma 7). Consider $w = \alpha u \beta$. Since a palindrome without the first and the last letter is a palindrome with the same center, it suffices to consider the cases $|u| = |xy|-1$ and $|u| = |xy|$. Assume $|u| = |xy|-1$ (the other case is similar). Suppose there are strings η, θ and a letter a such that $x = \eta a \theta$, $\alpha = (xy)^n \eta a$ for some $n \geq 0$, and $u = \theta y \eta$. (If the first letter of u lies inside y, the proof is the same.) Then $x = \overleftarrow{x} = \overleftarrow{\theta} a \overleftarrow{\eta}$, $u = \overleftarrow{u} = \overleftarrow{\eta} y \overleftarrow{\theta}$. Further, $xy = \eta a \theta y = \mathrm{shl}\ (a\theta y \eta, -|\eta|) = \mathrm{shl}\ (a\overleftarrow{\eta} y \overleftarrow{\theta}, -|\eta|)$. But $a\overleftarrow{\eta} y \overleftarrow{\theta} = \mathrm{shl}\ (\overleftarrow{\theta} a \overleftarrow{\eta} y, |\overleftarrow{\theta}|) = \mathrm{shl}\ (xy, |\overleftarrow{\theta}|)$. Hence $xy = \mathrm{shl}\ (xy, |\overleftarrow{\theta}| - |\eta|)$. Since xy is primitive, it follows from Lemma 6 that $|\overleftarrow{\theta}| = |\eta|$. Thus, $\overleftarrow{\theta} = \eta$ and $u = \overleftarrow{\eta} y \eta$.

Example 1. Consider $x = aba$, $y = ababa$, and $u = abaaba$. Obviously, $xyxyx$ is a canonical decomposition of $aba \cdot ababa \cdot aba \cdot ababa \cdot aba$, u is a suffix-palindrome of $xyxyx$, and $|u| = |xy|-2$. The center of u is not equal to the center of x or y. Therefore, the bound in Lemma 7 is optimal.

Now we briefly discuss the approach used in [GS]. The algorithm of [GS] essentially relies on the following "cancelation" lemma.

Lemma 8 (see [GS, Lemma C4]). *Suppose w is a palindromic pair; then there exist palindromes x and y such that $w = xy$ and either x is the longest prefix-palindrome of w or y is the longest suffix-palindrome of w.*

Unfortunately, it seems that even for the case of three palindromes, there are no similar results. Indeed, one can expect that if the string s is a concatenation of three nonempty palindromes, then there are palindromes x, y, z such that $s = xyz$ and at least one of the following statements holds:

1. x is the longest proper prefix-palindrome of $s[1..|s|-1]$;
2. z is the longest proper suffix-palindrome of $s[2..|s|]$;
3. xy is the longest proper prefix that is a palindromic pair;
4. yz is the longest proper suffix that is a palindromic pair.

The following example shows that this hypothesis does not hold.

Example 2. Consider the following string that is a concatenation of three nonempty palindromes: (For convenience, some groups of letters are separated by spaces.)

$$\underbrace{aba\ aba}\underbrace{b\ aba\ \mathbf{c}\ aba\ b\ aba\ aba\ aba\ aba\ b\ aba\ \mathbf{c}\ aba\ b}\underbrace{aba\ b\ aba}\ .$$

It turns out that there are no palindromes x, y, z such that xyz is equal to this string and x, y, z satisfy the hypothesis. To prove it, let us emphasize subpalindromes corresponding to the points of the hypothesis:

1. $\underbrace{aba\ aba\ b\ aba\ \mathbf{c}\ aba\ b\ aba\ aba}\ aba\ aba\ b\ aba\ \mathbf{c}\ aba\ b\ aba\ b\ aba$,
2. $aba\ aba\ b\ aba\ \mathbf{c}\ aba\ b\ aba\ aba\ aba\ aba\ b\ aba\ \mathbf{c}\ \underbrace{aba\ b\ aba\ b\ aba}$,
3. $\underbrace{aba}\ \underbrace{aba\ b\ aba\ \mathbf{c}\ aba\ b\ aba\ aba\ aba\ aba\ b\ aba\ \mathbf{c}\ aba\ b\ aba}\ b\ aba$,
4. $aba\ a\underbrace{ba\ b\ aba\ \mathbf{c}\ aba\ b\ aba\ aba\ aba\ aba\ b\ aba\ \mathbf{c}\ aba\ b\ ab}\ \underbrace{ababa}\ .$

3 Palindromic iterator

Let $w[i..j]$ be a subpalindrome of a string w. The number $\lfloor (j-i+1)/2 \rfloor$ is the *radius* of $w[i..j]$. Let $\mathcal{C} = \{c > 0 \colon 2c \text{ is an integer}\}$ be the set of all possible centers for subpalindromes. *Palindromic iterator* is the data structure containing a string *text* and supporting the following operations on it:

1. $\mathsf{append}_i(a)$ appends the letter a to the end;
2. maxPal returns the center of the longest suffix-palindrome;
3. $\mathsf{rad}(x)$ returns the radius of the longest subpalindrome with the center x;
4. $\mathsf{nextPal}(x)$ returns the center of the longest proper suffix-palindrome of the suffix-palindrome with the center x.

Example 3. Let $text = aabacabaa$. Then $\mathsf{maxPal} = 5$. Values of rad and $\mathsf{nextPal}$ are listed in the following table (the symbol "$-$" means undefined value):

$text$		a		a		b		a		c		a		b		a		a	
x	0.5	1	1.5	2	2.5	3	3.5	4	4.5	5	5.5	6	6.5	7	7.5	8	8.5	9	9.5
$\mathsf{rad}(x)$	0	0	1	0	0	1	0	0	0	4	0	0	0	1	0	0	1	0	0
$\mathsf{nextPal}(x)$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	8.5	$-$	$-$	$-$	$-$	$-$	$-$	9	9.5	$-$

A *fractional array* of length n is an array with n elements indexed by the numbers $\{x \in \mathcal{C} \colon 0 < x \leq \frac{n}{2}\}$. Fractional arrays can be easily implemented using ordinary arrays of double size. Let $\mathsf{refl}(x, y) = y + (y - x)$ be the function returning the position symmetric to x with respect to y.

Proposition 1. *Palindromic iterator can be implemented such that* append_i *requires amortized $O(1)$ time and all other operations require $O(1)$ time.*

Proof. Our implementation uses a variable s containing the center of the longest suffix-palindrome of $text$, and a fractional array r of length $2s$ such that for each $i \in \mathcal{C}$, $0 < i \leq s$, the number $r[i]$ is the radius of the longest subpalindrome centered at i. Obviously, $\mathsf{maxPal} = s$. Let us describe $\mathsf{rad}(x)$. If $x \leq s$, $\mathsf{rad}(x) = r[x]$. If $x > s$, then each palindrome with the center x has a counterpart with the center $\mathsf{refl}(x, s)$. On the other hand, $\mathsf{rad}(x) \leq |text| - \lfloor x \rfloor$, implying $\mathsf{rad}(x) = \min\{r[\mathsf{refl}(x, s)], |text| - \lfloor x \rfloor\}$. To implement $\mathsf{nextPal}$ and append_i, we need additional structures.

We define an array $lend[0..|text|-1]$ and a fractional array $nodes[\frac{1}{2}..|text|+\frac{1}{2}]$ to store information about maximal subpalindromes of $text$. Thus, $lend[i]$ contains centers of some maximal subpalindromes of the form $text[i+1..j]$. Precisely, $lend[i] = \{x \in \mathcal{C} \colon x < s \text{ and } \lceil x \rceil - \mathsf{rad}(x) = i+1\}$. Each center x is also considered as an element of a biconnected list with the fields $x.\mathrm{next}$ and $x.\mathrm{prev}$ pointing at other centers. We call such elements nodes and store in the array $nodes$. The following *invariant of palindromic iterator* holds.

Let $c_0 < \ldots < c_k$ be the centers of all suffix-palindromes of $text$. For each $j \in \overline{0, k-1}$, $nodes[c_j].\mathrm{next} = c_{j+1}$ and $nodes[c_{j+1}].prev = c_j$.

Clearly, $c_0 = s$, $c_k = |text| + \frac{1}{2}$. Let $\mathsf{link}(x)$ and $\mathsf{unlink}(x)$ denote the operations of linking x to the end of the list and removing x from the list, respectively.

Obviously, $\mathsf{nextPal}(x) = nodes[x].\mathrm{next}$. The following pseudocode of append_i uses the three-operand **for** loop like in the C language.

```
1:  function append_i(a)
2:      for (s_0 ← s; s < |text| + 1; s ← s + 1/2) do
3:          r[s] ← min(r[refl(s, s_0)], |text| − ⌊s⌋);                          ▷ fill r
4:          if ⌊s⌋ + r[s] = |text| and text[⌈s⌉−r[s]−1] = a then
5:              r[s] ← r[s] + 1;      ▷ here s is the center of the longest suffix-pal.
6:              break;
7:          lend[⌈s⌉−r[s]−1] ← lend[⌈s⌉−r[s]−1] ∪ {s};                          ▷ fill lend
8:      text ← text · a;
9:      link(nodes[|text|]); link(nodes[|text|+1/2]);        ▷ adding trivial suffix-pals.
10:     for each x in lend[⌈s⌉ − rad(s)] do
11:         unlink(nodes[refl(x, s)]);          ▷ removing invalid centers from the list
```

The code in lines 2–8 is a version of the main loop of Manacher's algorithm [Man]; see also [CR, Chapter 8]. The array $lend$ is filled simultaneously with r. Let us show that the invariant is preserved.

Suppose that a symbol is added to $text$ and the value of s is updated. Denote by S the set of centers $x > s$ such that the longest subpalindrome centered at x has lost its status of suffix-palindrome on this iteration. Once we linked the one-letter and empty suffix-palindromes to the list, it remains to remove the elements of S from it. Let $t = \lceil s \rceil - \mathsf{rad}(s)$. Since $text[t..|text|]$ is a palindrome, we have $lend[t] = \{\mathsf{refl}(x, s) \colon x \in S\}$. Thus, lines 10–11 unlink S from the list.

Since append_i links exactly two nodes to the list, any sequence of n calls to append_i performs at most $2n$ unlinks in the loop 10–11. Further, any such sequence performs at most $2n$ iterations of the loop 2–8 because each iteration increases s by $\frac{1}{2}$ and $s \le |text|$. Thus, append_i works in the amortized $O(1)$ time.

Example 4. Let $text = aabacaba$. The list of centers of suffix-palindromes contains $5, 7, 8, 8.5$. Now we perform $\mathsf{append}_i(a)$ using the above implementation. We underline suffix-palindromes of the source string for convenience: $aab\underline{aca\underline{b\underline{\underline{a}}}}a$. The centers 9, 9.5 are linked to the list in the line 9. The set of centers to be removed from the list is $S = \{7, 8\}$. Let $t = \lceil s \rceil - \mathsf{rad}(s) = 5 - 4 = 1$. Since $lend[t] = \{2, 3\}$, the loop 10–11 unlinks $S = \{\mathsf{refl}(i, s) \colon i \in lend[t]\}$ from the list. So, the new list contains $5, 8.5, 9, 9.5$.

4 Palindromic Engine

Palindromic engine is the data structure containing a string $text$, bit arrays m and res of length $|text|{+}1$, and supporting a procedure $\mathsf{append}(a, b)$ such that

1. $\mathsf{append}(a, b)$ appends the letter a to $text$, sets $m[|text|]$ to b, and calculates $res[|text|]$;
2. m is filled by append except for the bit $m[0]$ which is set initially;
3. $res[i] = 1$ iff there is $j \in \mathbb{N}$ such that $0 \le j < i$, $m[j] = 1$, and $text[j{+}1..i] \in$ Pal (thus $res[0]$ is always zero).

The following lemma is an immediate consequence of the third condition.

Lemma 9. *Let L be a language. Suppose that for any $i \in \overline{0,|text|}$, $m[i] = 1$ iff $text[1..i] \in L$; then for any $i \in \overline{0,|text|}$, $res[i] = 1$ iff $text[1..i] \in L \cdot \mathrm{Pal}$.*

Let f, g be functions of integer argument. We say that a palindromic engine works in $f(n)$ time and $g(n)$ space if any sequence of n calls to append on empty engine requires at most $f(n)$ time and $g(n)$ space.

Proposition 2. *Suppose a palindromic engine works in $f(n)$ time and space, and a language L is online recognizable in $g(n)$ time and space; then the language $L \cdot \mathrm{Pal}$ is online recognizable in $f(n) + g(n) + O(n)$ time and space.*

Proof. Assume that in the palindromic engine $m[0] = 1$ iff $\varepsilon \in L$. We scan the input string w sequentially from left to right. To process the ith letter of w, we feed it to the algorithm recognizing L and then call $\mathsf{append}(w[i], 1)$ or $\mathsf{append}(w[i], 0)$ depending on whether $w[1..i]$ belongs to L or not. Thus, by Lemma 9, $res[i] = 1$ iff $w[1..i] \in L \cdot \mathrm{Pal}$. Time and space bounds are obvious.

We use the palindromic iterator in our implementation of palindromic engine. Let $\mathsf{len}(x)$ be the function returning the length of the longest subpalindrome with the center x, i.e., $\mathsf{len}(x) = 2 \cdot \mathsf{rad}(x) + \lfloor x \rfloor - \lfloor x - \frac{1}{2} \rfloor$. The operations of bitwise "or", "and", "shift" are denoted by or, and, shl respectively. Let $x \overset{\text{or}}{\leftarrow} y$ be short for $x \leftarrow (x \text{ or } y)$. The naive $O(n^2)$ time implementation is as follows:

function $\mathsf{append}(a, b)$
    $\mathsf{append}_i(a)$; $n \leftarrow |text|$; $res[n] \leftarrow 0$; $m[n] \leftarrow b$;
    for ($x \leftarrow \mathsf{maxPal}$; $x \neq n+\frac{1}{2}$; $x \leftarrow \mathsf{nextPal}(x)$) **do**
        $res[n] \overset{\text{or}}{\leftarrow} m[n-\mathsf{len}(x)]$; ▷ loop through all suffix-palindromes

To improve the naive implementation, we have to decrease the number of suffix-palindromes to loop through. This can be done using "leading" subpalindromes.

A nonempty string w is *cubic* if its minimal period p is at most $|w|/3$. A subpalindrome $u = w[i..j]$ is *leading* in w if any period p of any longer subpalindrome $w[i'..j]$ satisfies $2p > |u|$. Thus, all non-leading subpalindromes are suffixes of leading cubic subpalindromes. For example, the only cubic subpalindrome of $w = aabababa$ is $w[2..8] = abababa$, and the only non-leading subpalindrome is $w[4..8] = ababa$.

Lemma 10. *Let $s = w[i..j]$ be a leading subpalindrome of w, with the canonical decomposition $(uv)^*u$, and $t = w[i'..j]$ be the longest proper suffix-palindrome of s that is leading in w. Then $t = u$ if $s = uvu$, and $t = uvu$ otherwise.*

Proof. Let $s = uvu$. By Lemma 3, u is the longest proper suffix-palindrome of s. Clearly, u is leading in w. If $s \neq uvu$, the assertion follows from Lemma 7.

Lemma 11. *A string of length n has at most $\log_{\frac{3}{2}} n$ leading suffix-palindromes.*

Proof. Let u, v be leading suffix-palindromes such that $|u| > |v|$. By Lemma 3, $|u|-|v|$ is a period of u. Let p be the minimal period of u. Since $|v| < 2p$ and $p \le |u|-|v|$, we conclude $|u| > \frac{3}{2}|v|$, whence the result.

To obtain a faster implementation of the palindromic engine, we loop through leading suffix-palindromes only. To take into account other suffix-palindromes, we gather the corresponding bits of m into an additional bit array z described below.

For every $i \in \overline{0,|text|}$, let j_i be the maximal number j' such that $text[i{+}1..j']$ is a leading subpalindrome. Since any empty subpalindrome is leading, j_i is well defined. Let p_i be the minimal period of $text[i{+}1..j_i]$. Denote by d_i the length of the longest proper suffix-palindrome of $text[i{+}1..j_i]$ such that $text[j_i{-}d_i{+}1..j_i]$ is leading in $text$. By Lemma 10, $d_i = \min\{(j_i{-}i){-}p_i, p_i{+}((j_i{-}i) \bmod p_i)\}$. The array z is maintained to support the following invariant:

$$z[i] = m[i] \text{ or } m[i{+}p_i] \text{ or } \ldots \text{ or } m[j_i{-}d_i{-}2p_i] \text{ or } m[j_i{-}d_i{-}p_i] \text{ for all } i \in \overline{0,|text|}\,.$$

Proposition 3. *The palindromic engine can be implemented to work in $O(n \log n)$ time and $O(n)$ space.*

Proof. Consider the following implementation of the function append. An instance of its work is given below in Example 5.

1: **function** $\mathsf{append}(a, b)$
2: $\quad \mathsf{append}_i(a);\ n \leftarrow |text|;\ res[n] \leftarrow 0;\ m[n] \leftarrow b;\ z[n] \leftarrow b;\ d \leftarrow 0;$
3: $\quad$ **for** $(x \leftarrow \mathsf{maxPal};\ x \ne n{+}\frac{1}{2};\ x \leftarrow n{-}(d{-}1)/2)$ **do** $\ \triangleright$ for leading suf-pal
4: $\quad\quad p \leftarrow \mathsf{len}(x) - \mathsf{len}(\mathsf{nextPal}(x));$ $\quad \triangleright$ min period of processed suf-pal
5: $\quad\quad d \leftarrow \min(p{+}(\mathsf{len}(x) \bmod p), \mathsf{len}(x){-}p);$ $\ \triangleright$ length of next leading s-pal
6: $\quad\quad$ **if** $3p > \mathsf{len}(x)$ **then** $\quad \triangleright$ processed suf-pal is not cubic
7: $\quad\quad\quad z[n{-}\mathsf{len}(x)] \leftarrow m[n{-}\mathsf{len}(x)];$
8: $\quad\quad$ **else** $z[n{-}\mathsf{len}(x)] \overset{\text{or}}{\leftarrow} m[n{-}d{-}p];$ $\quad \triangleright$ processed suf-pal is cubic
9: $\quad\quad res[n] \overset{\text{or}}{\leftarrow} z[n{-}\mathsf{len}(x)];$

Let $w_0, \ldots, w_k$ be all leading suffix-palindromes of $text$ and $|w_0| > \ldots > |w_k|$. We show by induction that the values taken by x are the centers of $w_0, \ldots, w_k$ (in this order). In the first iteration $x = \mathsf{maxPal}$ is the center of w_0. Let x be the center of w_i. The minimal period p of w_i is calculated in line 4 according to Lemmas 2 and 3. By Lemma 10, the value assigned to d in line 5 is $|w_{i+1}|$. Thus, the third operand in line 3 sets x to the center of w_{i+1} for the next iteration.

Let x and $(uv)^*u$ be, respectively, the center and the canonical decomposition of w_i. Denote by w any suffix-palindrome such that $|w_i| \ge |w| > |w_{i+1}|$. By Lemma 7, $w = (uv)^*u$. If the invariant of z is preserved, the assignment in line 9 is equivalent to the sequence of assignments $res[n] \overset{\text{or}}{\leftarrow} m[n{-}|w|]$ for all such w. Since i runs from 0 to k, finally one gets $res[n] \overset{\text{or}}{\leftarrow} m[n{-}|w|]$ for all suffix-palindromes w, thus providing that the engine works correctly. To finish the proof, let us show that our implementation preserves the invariant on-the-fly, setting the correct value of $z[n{-}|w_i|]$ in lines 7, 8 just before it is used in line 9.

As in the pseudocode presented above, denote by n the length of $text$ with the letter c appended. For any $j \in \overline{0, n{-}1}$, the bit $z[j]$ is changed iff $text[j{+}1..n]$ is a leading suffix-palindrome. Assume that $w_i = text[j{+}1..n]$ is a leading suffix-palindrome and x is its center. If w_i is not cubic, line 7 gives the correct value of $z[j]$, because $n{-}d{-}p = j$. Suppose w_i is cubic. Let $(uv)^*u$ be a canonical decomposition of w_i. Then $w' = text[i{+}1..n{-}|vu|]$ is a leading subpalindrome. Indeed, $w' = (uv)^*u$ and $|w'| \ge |uvuvu|$. For some $i' \le i$, suppose that $text[i'{+}1..n{-}|uv|]$ is a leading subpalindrome, p is its minimal period, and $2p < |w'|$; then since $p \ge |uv|$, we have, by Lemmas 2 and 7, that either $2p > |w'|$ or $|uv|$ divides p. Hence $text[i'{+}1..n{-}|uv|] = (uv)^*u$. Thus $i' = i$ because $text[i{+}1..n]$ is leading. Since w' is leading, we restore the invariant for $z[n{-}|w_i|]$ in line 8.

Since the number of iterations of the **for** cycle equals the number of leading suffix-palindromes of $text$, it is $O(\log n)$ by Lemma 11. This gives us the required time bound; the space bound is obvious.

Example 5. Let $text = ababab$. For $i = 0, \dots, 6$, denote by j_i the maximal number j' such that $text[i{+}1..j']$ is a leading subpalindrome. Let $(u_iv_i)^*u_i$ be a canonical decomposition of $text[i{+}1..j_i]$. We have $z[i] = m[i]$ for all i. The following table describes $text[i{+}1..j_i]$.

i	0	1	2	3	4	5	6
$text[i{+}1..j_i]$	$ababa$	$babab$	aba	bab	a	b	ε
u_i, v_i	a, b	b, a	a, b	b, a	ε, a	ε, b	ε, ε

Assume that now we call $\mathsf{append}(a, m[7])$, using the $O(n \log n)$ implementation above (for simplicity, we suppose that the array m is known in advance). In line 2, we get $text = abababa$, $res[7] = 0$, $z[7] = m[7]$. The leading suffix-palindromes of $text$ are $w_0 = abababa$, $w_1 = aba$, $w_2 = a$, $w_3 = \varepsilon$. Then the **for** loop passes three iterations: 1) $p = 2$, $d = 3 = |aba|$; 2) $p = 2$, $d = 1 = |a|$; 3) $p = 1$, $d = 0 = |\varepsilon|$. On the first iteration, $z[0] \overset{\text{or}}{\leftarrow} m[2]$ is assigned. Thus, $z[0]$ takes care of the non-leading suffix-palindrome $ababa$. On the next two iterations, the condition in line 6 is true, so we (re)assign $z[4] \leftarrow m[4]$ and $z[6] \leftarrow m[6]$. Finally, we get

$$res[7] = z[0] \text{ or } z[4] \text{ or } z[6] = m[0] \text{ or } m[2] \text{ or } m[4] \text{ or } m[6].$$

In order to demonstrate other features of this implementation of palindromic engine, we consider a few further calls to append.

Suppose the next call is $\mathsf{append}(b, m[8])$, giving us $text = abababab$. This call is much alike the previous one, with three iterations of the **for** loop, corresponding to the suffix-palindromes $w_0 = bababab$, $w_1 = bab$, and $w_2 = b$, and the only non-trivial assignment $z[1] \overset{\text{or}}{\leftarrow} m[3]$.

Now let us call $\mathsf{append}(a, m[9])$. We again have three iterations, for $w_0 = text = ababababa$, $w_1 = aba$, $w_2 = a$. In the first iteration, $z[0] \overset{\text{or}}{\leftarrow} m[4]$ is assigned; this gives us $z[0] = m[0]$ or $m[2]$ or $m[4]$. Thus, $z[0]$ takes care of both non-leading suffix-palindromes, following w_0.

Finally, assume that after several steps we have $text = ababababacabababab$ and now call $\mathsf{append}(a, m[19])$. On all omitted calls, no suffix-palindromes started in the first position of text, so the bit $z[0]$ was not in use and remained unchanged. On the current call, $z[0]$ is used again, but $w_0 = \mathit{text}$ is a non-cubic palindrome; hence, $z[0]$ is reset to $m[0]$ in line 7.

5 Linear Algorithm

Consider the *word-RAM* model with $\beta{+}1$ bits in the machine word, where the bits are numbered starting with 0 (the least significant bit). A standard assumption is $\beta > \log|\mathit{text}|$. For a bit array $s[0..n]$ and integers i_0, i_1 such that $0 \le i_1 - i_0 \le \beta$, we write $x \leftarrow s[\overrightarrow{i_0..i_1}]$ to get the number x whose jth bit, for any $j \in \overline{0,\beta}$, equals $s[i_0{+}j]$ if $0 \le i_0{+}j \le \min\{n, i_1\}$ and 0 otherwise. Similarly, $x \leftarrow s[\overleftarrow{i_0..i_1}]$ defines x with a jth bit equal to $s[i_1{-}j]$ if $\max\{0, i_0\} \le i_1{-}j \le n$ and to 0 otherwise. We write $s[\overleftarrow{i_0..i_1}] \leftarrow x$ and $s[\overrightarrow{i_0..i_1}] \leftarrow x$ for the inverse operations. A bit array is called *forward* [*backward*] if each read/write operation for $s[\overrightarrow{i_0..i_1}]$ [resp. $s[\overleftarrow{i_0..i_1}]$] takes $O(1)$ time. Forward [backward] arrays can be implemented on arrays of machine words with the aid of bitwise shifts.

Processing a string of length n, we can read/write a group of $\log n$ elements of forward or backward array in a constant time. In this section we speed up palindromic engine using bitwise operations on groups of $\log n$ bits. This sort of optimization is often referred to as four Russians' trick (see [ADKF]). Note that there is a simpler algorithm recognizing Pal^k in $O(kn \log n)$ time, but it cannot be sped up in this fashion.

In the sequel n denotes $|\mathit{text}|$. As above, our palindromic engine contains a palindromic iterator; by Proposition 1, all computations inside the iterator take $O(n)$ total time, so we need no speed up for that.

In the implementation described below, the array $m[0..n]$ and the auxiliary array $z[0..n]$ are backward, while slightly extended array $\mathit{res}[0..n{+}\beta]$ is forward.

5.1. Idea of the algorithm We say that a call to append is *predictable* if it retains the value of maxPal (or, in other words, extends the longest suffix-palindrome). For a predictable call, we know from symmetry which suffix-palindromes will be extended. This crucial observation allows us to fill $\mathit{res}[n..n{+}\beta]$ in advance so that in the next β calls we need only few changes of res provided that these calls are predictable.

Let $\mathit{text} = vs$ at some point, where s is the longest suffix-palindrome. The number of subsequent calls preserving maxPal is at most $|v| = n{-}\mathsf{len}(\mathsf{maxPal})$: this is the case if we add $\overleftarrow{v}$. Consider those calls. Let $c_0 < \ldots < c_k$ be the list of centers of all suffix-palindromes of text. Let $i \in \overline{1,k}$. After some predictable call c_i can vanish from this list. Let p_i be the number of predictable calls that retain c_i on the list. Then $p_i = \mathsf{rad}(\mathsf{refl}(c_i, \mathsf{maxPal})) - \mathsf{rad}(c_i)$ (in Fig. 1 $p_1 = 5{-}3 = 2$).

Let $j_i = n{-}\mathsf{len}(c_i)$. If the operation $\mathit{res}[\overrightarrow{n..n{+}\beta}] \overset{\text{or}}{\leftarrow} m[\overleftarrow{j_i{-}p_i..j_i}]$ is performed for some $i \in \overline{1, k{-}1}$, we do not need to consider the suffix-palindrome with the center c_i during the next β predictable calls. Similarly, if $\mathit{res}[\overrightarrow{n..n{+}\beta}] \overset{\text{or}}{\leftarrow} m[\overleftarrow{j_0{-}\beta..j_0}]$ or $(m[\overleftarrow{j_k{-}p_k..j_k{-}1}]$ shl 1) is performed, we do not consider the

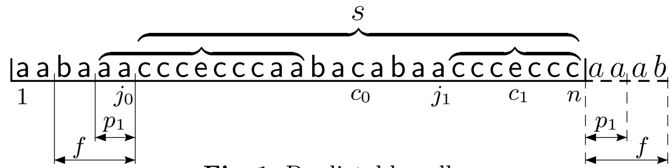

Fig. 1. Predictable calls.

centers c_0 and c_k (a shift appears because the empty suffix-palindrome is ignored). The algorithm is roughly as follows. When the assignments above are performed, each of the next β predictable calls just adds two suffix-palindromes (one-letter and empty) and performs the corresponding assignments for them. When an unpredictable call or the $(\beta{+}1)$st predictable call occurs, we make new assignments in the current position and use array z to reduce the number of suffix-palindromes to loop through. Let us consider details.

5.2. Algorithm We add to the engine an integer variable f such that $0 \le f \le \min\{\beta, n - \mathsf{len}(\mathsf{maxPal})\}$. The value of $res\overrightarrow{[n..n{+}f]}$ is called the *prediction*. Let us describe it. The centers c_i and the numbers p_i are defined in Sect. 5.1. Let $\mathrm{pr}\colon \{c_0, \dots, c_k\} \to \mathbb{N}_0$ be the mapping defined by $\mathrm{pr}(c_0) = f$ and $\mathrm{pr}(c_i) = \min\{p_i, f\}$ for $i > 0$. Obviously, $\mathrm{pr}(c_i)$ is computable in $O(1)$ time. According to Sect. 5.1, the following value, called *f-prediction*, takes care of the palindromes with the centers $c_0, \dots, c_k$ during all the time when they are suffix-palindromes:

$$m\overleftarrow{[j_0{-}\mathrm{pr}(c_0)..j_0]} \text{ or } \cdots \text{ or } m\overleftarrow{[j_{k-1}{-}\mathrm{pr}(c_{k-1})..j_{k-1}]} \text{ or } (m\overleftarrow{[j_k{-}\mathrm{pr}(c_k)..j_k{-}1]} \text{ shl } 1).$$

The prediction calculated by our algorithm will sometimes deviate from the f-prediction, but in a way that guarantees condition 3 of the definition of palindromic engine. Now we describe the nature of this deviation.

Let $c \in \mathcal{C}$ and $c > n{+}\frac{1}{2}$. Denote $c' = \mathsf{refl}(c, \mathsf{maxPal})$. Suppose $c' > 0$ and $\lceil c\rceil - \mathsf{rad}(c') \le n{+}1$ (see Fig. 2). Let r be a positive integer such that $r \le \mathsf{rad}(c'){+}1$ and $\lceil c\rceil - r \le n$. The values c and r are chosen so that after a number of predictable calls *text* will contain a suffix-palindrome with the center c and the radius $r{-}1$. Then $res[\lceil c\rceil{+}r{-}1] = 1$ if $m[\lceil c\rceil{-}r] = 1$. We call the value $g = m[\lceil c\rceil{-}r] \text{ shl } (\lceil c\rceil{+}r{-}1{-}n)]$ an *additional prediction*. The assignment $res[n..n{+}f] \overset{\text{or}}{\leftarrow} g$ performs disjunction of the bits $res[\lceil c\rceil{+}r{-}1]$ and $m[\lceil c\rceil - r]$ (we suppose $\lceil c\rceil{+}r{-}1 \le n{+}f$). Setting this bit to 1 is not harmful: if there will be no unpredictable calls before the position $\lceil c\rceil{+}r{-}1$, then this bit will be set to 1 when updating the f-prediction on the $\lfloor c\rfloor$th iteration. Additional predictions appear as a byproduct of the linear-time implementation of the engine.

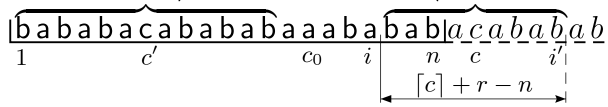

Fig. 2. Additional prediction; $c_0 = \mathsf{maxPal}$, $c' = \mathsf{refl}(c, c_0)$, $i = c - r$, $i' = c + r - 1$.

We define the prediction through the *main invariant of palindromic engine:* $res[\overrightarrow{n..n{+}f}]$ equals the bitwise "or" of the f-prediction and some additional predictions. Such a definition guarantees that $res[n] = 1$ iff $m[j] = 1$ and $text[j{+}1..n] \in \mathrm{Pal}$ for some j, $0 \le j < n$. Thus, the goal of $\mathsf{append}(a, b)$ is to preserve the main invariant. Our implementation of $\mathsf{append}(a, b)$ consists of three steps:

1. call $\mathsf{append}_i(a)$ to extend *text* (and increment n); then assign b to $m[n]$;
2. if maxPal remains the same and $f > 0$, decrement f and perform $res[\overrightarrow{n..n{+}f}] \stackrel{\text{or}}{\leftarrow} m[\overleftarrow{n{-}1{-}\mathrm{pr}(n)..n{-}1}]$ or $(m[\overleftarrow{n{-}\mathrm{pr}(n{+}\frac{1}{2})..n{-}1}]$ shl 1);
3. otherwise, assign $f \leftarrow \min\{\beta, n{-}\mathsf{len}(\mathsf{maxPal})\}$ and recalculate the prediction $res[\overrightarrow{n..n{+}f}]$.

The operations of step 2 correspond to a predictable call and obviously preserve the main invariant. In the sequel we only consider step 3; step 1 is supposed to be performed: a is appended to *text*, n is incremented, and $m[n] = b$.

5.3. Prediction recalculation Recall that $c_0 < \ldots < c_k$ are the centers of suffix-palindromes, $j_i = n{-}\mathsf{len}(c_i)$. First, clear the prediction: $res[\overrightarrow{n..n{+}f}] \leftarrow 0$. To get the f-prediction, it suffices to assign $res[\overrightarrow{n..n{+}f}] \stackrel{\text{or}}{\leftarrow} m[\overleftarrow{j_i{-}\mathrm{pr}(c_i)..j_i}]$ for $i = 0, \ldots, k{-}1$ and $res[\overrightarrow{n..n{+}f}] \stackrel{\text{or}}{\leftarrow} m[\overleftarrow{j_k{-}\mathrm{pr}(c_k)..j_k{-}1}]$ shl 1. But our algorithm processes leading suffix-palindromes only, and the bits of m that correspond to non-leading suffix-palindromes are accumulated in a certain fast accessible form in the array z. For simplicity, we process the empty suffix separately.

Let $i_0 < \ldots < i_h$ be integers such that $c_{i_0} < \ldots < c_{i_h}$ are the centers of all leading suffix-palindromes, $r \in \overline{0, h{-}1}$ and $s = i_{r+1} - i_r - 1 > 0$. Denote by w the suffix-palindrome centered at c_{i_r}. Let $(uv)^*u$ be the canonical decomposition of w. It follows from Lemma 7 that $c_{i_r+1}, \ldots, c_{i_r+s}$ are the centers of $(uv)^{s+1}u, \ldots, (uv)^2u$, $c_{i_r+s+1} = c_{i_{r+1}}$ is the center of uvu, and $w = (uv)^{s+2}u$. Then w is cubic. The converse is also true, i.e., if $w = (uv)^{s+2}u$ is a cubic suffix-palindrome, then $(uv)^{s+1}u, \ldots, (uv)^2u$ are non-leading suffix-palindromes, and uvu is a leading suffix-palindrome. So, non-leading suffix-palindromes are grouped into series following cubic leading suffix-palindromes.

Recall that the palindromic iterator allows one, in $O(1)$ time, to 1) get c_{i+1} from c_i; 2) find the minimal period of a suffix-palindrome; 3) using Lemma 10, get $c_{i_{r+1}}$ from c_{i_r}. The prediction recalculation involves the following steps:

1. accumulate some blocks of bits from m into z (see below);
2. for all $r \in \overline{0, h{-}1}$, assign $res[\overrightarrow{n..n{+}f}] \stackrel{\text{or}}{\leftarrow} m[\overleftarrow{j_{i_r}{-}\mathrm{pr}(c_{i_r})..j_{i_r}}]$;
3. for all $r \in \overline{1, h{-}1}$, if c_{i_r} is the center of a cubic suffix-palindrome and $\mathsf{len}(c_{i_r}) \le 2\beta$, assign $res[\overrightarrow{n..n{+}f}] \stackrel{\text{or}}{\leftarrow} m[\overleftarrow{j_{i_r+s}{-}\mathrm{pr}(c_{i_r+s})..j_{i_r+s}}]$ for $s = 1, 2, \ldots, i_{r+1}{-}i_r{-}1$;
4. for all $r \in \overline{0, h{-}1}$, if c_{i_r} is the center of a cubic suffix-palindrome and either $\mathsf{len}(c_{i_r}) > 2\beta$ or $c_{i_r} = c_0$, perform the assignments of step 3 in $O(1)$ time with the aid of the array z.

Thus, “short” and “long” non-leading suffix-palindromes are processed separately (resp., on step 3 and step 4). Steps 1 and 4 require further explanation.

5.4. Content of z and prediction of long suffix-palindromes Let w be a cubic leading suffix-palindrome such that $|w| > 2\beta$ or $|w| = \mathsf{len}(\mathsf{maxPal})$. Suppose $(uv)^*u$ is the canonical decomposition of w. Then $p = |uv|$ is the minimal period of w. Denote the centers of suffix-palindromes $w, \dots, uvu, u$ by $c_1, c_2, \dots, c_k$ respectively. Let us describe the behavior of those suffix-palindromes in predictable calls.

Let t be the longest suffix of $text$ with the period p (t is not necessarily a palindrome). Then $|t| = |w| + \mathsf{rad}(\mathsf{refl}(c_k, c_1)) - \mathsf{rad}(c_k)$ is computable in $O(1)$ time. Since w is leading and cubic, $|t| < |w| + p$. In a predictable call to append, the suffix t extends if $text[n] = text[n{-}p]$, and breaks otherwise. Suppose t extended to ta. The suffix-palindromes centered at $c_2, \dots, c_k$ also extended, while w extends iff $|w| < |t|$. Thus, in a series of such extensions of t the set of centers loses its first element during each p steps. Suppose t broke. Now the palindromes centered at $c_2, \dots, c_k$ broke, while w can extend provided that $w = t$.

Example 6. Let $text = baaaabaaa$. Then $\mathsf{maxPal} = 6$; $w = aaa$ is a leading cubic suffix-palindrome; $w = (uv)^*u$ for $u = \varepsilon$ and $v = a$; $t = w$. Suffix-palindromes aaa, aa, a, ε have the centers $c_1 = 8, c_2 = 8.5, c_3 = 9, c_4 = 9.5$ respectively. After the predictable call to append, $text = baaaabaaaa$, t is extended, and w (with the center c_1) broke. After the second predictable call, $text = baaaabaaaab$, t is broken, and only c_2 remains the center of a suffix-palindrome.

Consider the first f predictable calls. Let q be the maximal number such that the suffix t of period p extends over the first q of these calls. Since w is “long”, i.e., $|w| > 2\beta$ or w is the longest suffix-palindrome, and $f \le \beta$, one can be obtain q in $O(1)$ time: $q = \min\{f, \mathsf{rad}(\mathsf{refl}(c_k, \mathsf{maxPal})) - \mathsf{rad}(c_k)\}$. If $q < f$, the $(q{+}1)$st predictable call breaks the suffix of period p; as a result, at most one palindrome $w' = (uv)^*u$ extends to a suffix-palindrome at this moment (cf. Example 6). The length of w' in the initial text equals $|t|{-}q$, implying $(|t|{-}q{-}|u|) \bmod p = 0$. To process w', we perform $res[\overrightarrow{n..n{+}f}] \xleftarrow{\text{or}} m[\overleftarrow{j{-}\mathrm{pr}(c_i)..j}]$ for $j = n{-}|w'|$, $c_i = n{-}(|w'|{-}1)/2$. To process other palindromes $(uv)^*u$, we consider z.

Denote $j_t = n - |t|$, $j'_t = j_t + p - 1$, and $j_w = n - |w|$, see Fig. 3 a,b. We store the information about the series of palindromes $(uv)^*u$ in the block $z[j_t...j'_t]$ of length $p = |uv|$. For any $j \ge 0$, $i_j = j'_t - ((j + j'_t - j_w) \bmod p)$. Thus, $i_0 = j_w$, $i_1 = j_w{-}1$ if $j_w \ne j_t$, and $i_1 = j'_t$ otherwise. Hence while j increases, i_j cyclically shifts left inside the range $\overline{j_t, j'_t}$. We fill the block $z[j_t..j'_t]$ such that each of its bits is responsible for the whole series of suffix-palindromes with the period p.

$$\forall j \in \overline{0,\beta}\colon z[i_j] = m[i_j] \text{ or } m[i_j{+}p] \text{ or } \dots \text{ or } m[i_j{+}lp] \text{ for } l = \lfloor (n{-}i_j)/p \rfloor \ . \quad (1)$$

Let $r_w^0 = \min\{\beta, j_w{-}j_t\} + 1$, $r_w^1 = \min\{\beta{+}1{-}r_w^0, j'_t{-}j_w\}$. Clearly, $r_w^0 + r_w^1 = \min\{\beta{+}1, p\}$. Hence, i_j in (1) runs through the ranges $[j_w{-}r_w^0{+}1..j_w]$ and $[j'_t{-}r_w^1{+}1..j'_t]$. Let $d = (1 \text{ shl } (q{+}1)) - 1$; thus, d is the bit mask consisting of

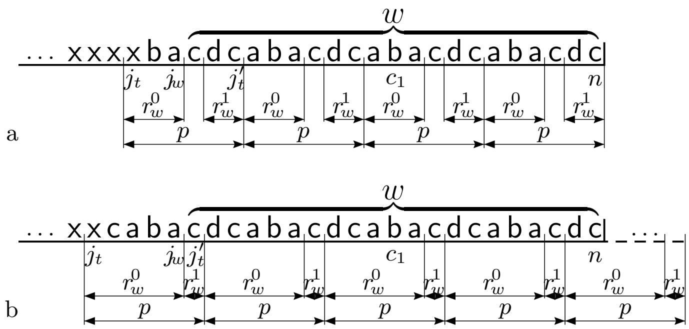

Fig. 3. Series of palindromes with a common period p. The cases presented are (a) $p > \beta{+}1$ $(= r_w^0 + r_w^1 = 5)$ and (b) $\beta{+}1 \geq p$ $(= r_w^0 + r_w^1 = 6)$.

$q{+}1$ ones. Suppose $\beta{+}1 < p$ (see Fig. 3,a). To recalculate the prediction, it suffices to assign $res[\overrightarrow{n..n{+}q}] \stackrel{\text{or}}{\leftarrow} d \,\text{and}(z[\overleftarrow{j_w{-}r_w^0{+}1..j_w}] \text{ or } (z[\overleftarrow{j_t'{-}r_w^1{+}1..j_t'}] \text{ shl } r_w^0))$. Suppose $\beta{+}1 \geq p$ (see Fig. 3,b). Let $k = \lceil q/p \rceil$. To recalculate the prediction, it suffices to perform the following:

$$\begin{aligned}
&res[\overrightarrow{n..n{+}q}] \stackrel{\text{or}}{\leftarrow} d \,\text{and}(z[\overleftarrow{j_t..j_w}] \text{ or } (z[\overleftarrow{j_w{+}1..j_t'}] \text{ shl } r_w^0)),\\
&res[\overrightarrow{n..n{+}q}] \stackrel{\text{or}}{\leftarrow} d \,\text{and}((z[\overleftarrow{j_t..j_w}] \text{ or } (z[\overleftarrow{j_w{+}1..j_t'}] \text{ shl } r_w^0)) \text{ shl } p),\\
&\dots\\
&res[\overrightarrow{n..n{+}q}] \stackrel{\text{or}}{\leftarrow} d \,\text{and}((z[\overleftarrow{j_t..j_w}] \text{ or } (z[\overleftarrow{j_w{+}1..j_t'}] \text{ shl } r_w^0)) \text{ shl } (kp))\ .
\end{aligned} \tag{2}$$

To perform these assignments in $O(1)$ time, we use a precomputed array g of length β such that $g[i] = \sum_{j=0}^{\lfloor \beta/i \rfloor} 2^{ij}$ is the bit mask containing ones separated by $i{-}1$ zeroes. Then the sequence of assignments (2) is equivalent to the operation $res[\overrightarrow{n..n{+}q}] \stackrel{\text{or}}{\leftarrow} d \,\text{and}((z[\overleftarrow{j_t..j_w}] \text{ or } (z[\overleftarrow{j_w{+}1..j_t'}] \text{ shl } r_w^0)) \cdot g[p])$.

Along with the f-prediction, the described method can produce additional predictions. Indeed, suppose we processed a cubic leading suffix-palindrome $w = (uv)^*u$. If $q > |v|$, the position $n{+}(|v|{+}1)/2$ is the center of the suffix-palindrome v after $|v|$ predictable calls. However, the corresponding assignment $res[n{+}|v|] \stackrel{\text{or}}{\leftarrow} m[n]$ is performed much earlier: calculating the prediction in the nth call of append, we accumulate the bit $m[n]$ in the array z (see (1)) and then use it in updating $res[n..n{+}q]$. The assignment $res[n{+}|v|{+}1] \stackrel{\text{or}}{\leftarrow} m[n{-}1]$ is performed at the same moment but corresponds to the $(|v|{+}1)$st predictable call, and so on. If $q > |vuv|$, we have the same situation with the suffix-palindrome vuv after $|vuv|$ calls. All these premature assignments are not necessary but bring no trouble.

Lemma 12. *Given the array z, the prediction recalculation requires $O(l + \min\{2\beta, s\})$ time, where l is the number of leading suffix-palindromes and s is the length of the second largest leading suffix-palindrome.*

Proof. The above analysis shows that each of steps 2, 4 takes $O(1)$ time per series of palindromes with a common period. Step 3 takes $O(\min\{2\beta, s\})$ time.

5.5. Recalculation of the array z and the time bounds

Lemma 13. *Recalculation of z requires $O(l + (n - n_0))$ time, where l is the number of leading suffix-palindromes and n_0 is the length of text at the moment of the previous recalculation.*

Proof. Given a cubic leading suffix-palindrome w with minimal period p, we set some bits in z according to (1). Recall that t is the longest suffix of *text* with the period p, $j_t = n - |t|$, $j'_t = j_t + p - 1$, $j_w = n - |w|$, $r^0_w = \min\{\beta, j_w - j_t\} + 1$, $r^1_w = \min\{\beta + 1 - r^0_w, j'_t - j_w\}$, $r^0_w + r^1_w = \min\{\beta{+}1, p\}$. Inside the range $\overline{j_t, j'_t}$ we have to fill the blocks $z[j_w{-}r^0_w{+}1..j_w]$ and $z[j'_t{-}r^1_w{+}1..j'_t]$. The main observation is that these blocks need only a little update after the previous recalculation.

1) *Suppose* $p > \beta{+}1$ (Fig. 3,a). If $|t| < 5p$, the assignment (1) requires an $O(1)$ time both for a single bit and for a block of length $\leq \beta$. Now consider the case $|t| \geq 5p$. For simplicity, the block $z[j_t..j'_t]$ is supposed to be cyclic. Then the blocks $z[j_w{-}r^0_w{+}1..j_w]$ and $z[j'_t{-}r^1_w{+}1..j'_t]$ form one segment of length β, denoted by S. Note that every sequence of $\beta{+}1$ calls to append contains a call that recalculates z. Therefore, the previous recalculation of the array z filled some segment S_1 of length $\beta{+}1$ in $z[j_t..j'_t]$, and S_1 either is adjacent to S from the right or overlaps S. Similarly, the second previous recalculation of z filled some segment S_2 which is either adjacent to S_1 or overlaps it, and so on. Since $|t| \geq 5p$, all recalculations during the last $2p$ iterations processed cubic suffix-palindromes with the period p. In these recalculations, all positions in S were filled (see Fig. 4). Thus, it suffices to perform $z[\overleftarrow{j_w{-}r^0_w{+}1..j_w}] \overset{\text{or}}{\leftarrow} m[\overleftarrow{j_w{-}r^0_w{+}1{+}kp..j_w{+}kp}]$, $z[\overleftarrow{j'_t{-}r^1_w{+}1..j'_t}] \overset{\text{or}}{\leftarrow} m[\overleftarrow{j'_t{-}r^1_w{+}1{+}kp..j'_t{+}kp}]$ for $k = \lfloor |t|/p \rfloor$ and $k = \lfloor |t|/p \rfloor - 1$, getting a $O(1)$ time bound again. Thus, the total time for all periods is $O(l)$.

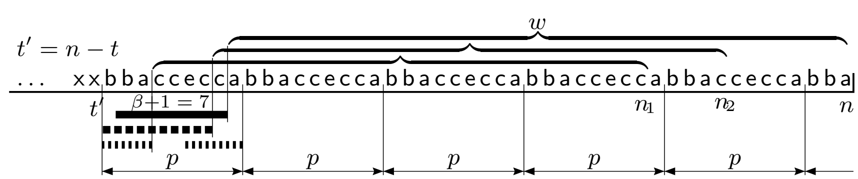

Fig. 4. A suffix of text at the moment of recalculation of z. The points n_1 and n_2 of some (not necessarily last!) previous recalculations are marked; the correspondent recalculated segments of $z[j_t..j'_t]$ are shown.

2) *Suppose* $p \leq \beta{+}1$ (Fig. 3,b). Then we must fill the whole range $z[j_t..j'_t]$. This case is similar to the above one but takes more than $O(1)$ time. We store the value n_0 in a variable inside the engine. Note that $n - n_0 \leq \beta{+}1$. Let $t_0 = |t| - (n{-}n_0)$. If $t_0 \geq 4p$, it follows, as above, that $z[j_t..j'_t]$ contains a lot of necessary values and to fix $z[j_t..j'_t]$, we perform $z[j_t..j'_t] \overset{\text{or}}{\leftarrow} m[j_t{+}kp..j'_t{+}kp]$ for every integer k such that $\lfloor t_0/p \rfloor \leq k \leq \lfloor |t|/p \rfloor$. If $t_0 < 4p$, we immediately perform $z[j_t..j'_t] \leftarrow m[j_t..j'_t]$ or $m[j_t{+}p..j'_t{+}p]$ or $\cdots$ or $m[j_t{+}kp..j'_t{+}kp]$ for $k = \lfloor |t|/p \rfloor$. Thus, the recalculation requires $O((n - n_0)/p)$ time. Summing up

these bounds for all p, we get $O(n{-}n_0)$, because the values of p are majorized by a geometric sequence.

Summing up the bounds for the cases 1), 2) finishes the proof.

Lemma 14. *After an unpredictable call to* append*, k successive predictable calls require $O(k)$ time in total.*

Proof. A predictable call without recalculation takes $O(1)$ time. The number of recalculations during these k calls is $\lfloor k/\beta \rfloor$. Since the number of leading suffix-palindromes is $O(\log n)$ by Lemma 11, it follows from Lemmas 12, 13 that the recalculation takes $O(\log n + \min\{2\beta, O(n)\}) + O(\log n + O(\beta)) = O(\beta)$ time, whence the result.

Lemma 15. *An unpredictable call requires $O(\mathsf{maxPal}{-}\mathsf{maxPal}_0 + n{-}n_0)$ time, where maxPal_0 is the center of the longest suffix-palindrome and n_0 is the length of text at the moment of the previous unpredictable call.*

Proof. Assume that $text = wr$ just before the current unpredictable call to append, where r is the longest suffix-palindrome. Note that r has the center maxPal_0. After this call one has $text = wrc = w't$, where t is the longest suffix-palindrome. If $|t| = 1$, the call takes $O(1)$ time. Suppose $t = cuc$ for some palindrome u. By Lemma 3, the number $p = |r| - |u|$ is a period of r. By Lemmas 12 and 13, the prediction recalculation takes $O(l + \min\{2\beta, s\}) + O(l + (n - n_0))$ time, where l is the number of leading suffix-palindromes and s is the length of the second longest leading suffix-palindrome. Since $l \le s$ and $p = 2(\mathsf{maxPal}{-}\mathsf{maxPal}_0)$, it suffices to prove that $s = O(p)$. If $|u| \le \frac{2}{3}|r|$, then $s < |u| \le 2p$. On the other hand, if $|u| > \frac{2}{3}|r|$ then p is a period of t by Lemma 7. Hence $s < 2p$ by the definition of a leading palindrome.

Proposition 4. *The palindromic engine can be implemented to work in $O(n)$ time and space.*

Proof. The correctness of the implementation described in Sect. 5.2, 5.3 was proved in Sect. 5.2–5.4. It remains to prove the time bound. Consider the sequence of n calls to append. Let $n_1 < n_2 < \ldots < n_k$ be the numbers of all unpredictable calls to append and $\mathsf{maxPal}_1 < \mathsf{maxPal}_2 < \ldots < \mathsf{maxPal}_k$ be the centers of the longest suffix-palindromes just before each of these calls. By Lemma 15, all these calls require $O(1 + (\mathsf{maxPal}_2{-}\mathsf{maxPal}_1) + (n_2{-}n_1) + (\mathsf{maxPal}_3{-}\mathsf{maxPal}_2) + (n_3{-}n_2) + \ldots + (\mathsf{maxPal}_k{-}\mathsf{maxPal}_{k-1}) + (n_k{-}n_{k-1})) = O(n)$ time. A reference to Lemma 14 ends the proof.

Proposition 4 together with Proposition 2 implies the main theorem.

6 Conclusion

In the RAM model considered in this paper all operations are supposed to be constant-time. This is the so called *unit-cost RAM.* Our algorithm heavily relies on multiplication and modulo operations, and we do not know whether it can be modified to use only addition, subtraction, and bitwise operations.

It was conjectured that there exists a context-free language that can not be recognized in linear time by a unit-cost RAM machine. This paper shows that a popular idea to use palindromes in the construction of such a language is quite likely to fail. For some discussion on this problem, see [Lee].

References

ADKF. V. Arlazarov, E. Dinic, M. Kronrod, I. Faradzev. On economical construction of the transitive closure of a directed graph, Dokl. Akad. Nauk. (in Russian) 194 (11) (1970) 1209–1210.

BG. D. Breslauer, L. Gąsieniec. Efficient String Matching on Coded Texts. CPM 1995. 27–40.

CR. M. Crochemore, W. Rytter. Jewels of Stringology. World Scientific Publ. (2002).

CR2. M. Crochemore, W. Rytter. Squares, cubes, and time-space efficient string searching, Algorithmica 13 (5) (1995) 405–425.

FGKK. G. Fici, T. Gagie, J. Kärkkäinen, D. Kempa. A Subquadratic Algorithm for Minimum Palindromic Factorization. arXiv:1403.2431 (2014).

GS. Z. Galil, J. Seiferas. A Linear-Time On-Line Recognition Algorithm for “Palstar”, J. ACM 25 (1) (1978) 102–111.

GJWZ. A. Glen, J. Justin, S. Widmer, L.Q. Zamboni. Palindromic richness. European J. Combinatorics 30(2): (2009) 510–531

KMP. D. E. Knuth, J. H. Morris, V. R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (1977) 323–350.

KRS. D. Kosolobov, M. Rubinchik, A. M. Shur, Finding Distinct Subpalindromes Online. In: Proc. Prague Stringology Conference (2013) 63–69.

Lee. L. Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication, J. ACM 49 (1) (2002) 1–15.

Lot. M. Lothaire. Algebraic combinatorics on words. Cambridge Univ. Press, 2002.

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

Val. L. G. Valiant, General context-free recognition in less than cubic time, Journal of Computer and System Sciences 10 (2) (1975) 308–314.