Counting Palindromes in Substrings

Mikhail Rubinchik and Arseny M. Shur(✉)

Department of Algebra and Fundamental Informatics, Ural Federal University,
pr. Lenina, 51, Ekaterinburg, Russia
mikhail.rubinchik@gmail.com, arseny.shur@urfu.ru

Abstract. We propose a data structure and an online algorithm to report the number of distinct palindromes in any substring of an input string. Assume that the string S of length n arrives symbol-by-symbol and every symbol is followed by zero or more queries of the form "report the number of distinct palindromes in $S[i..j]$". We use $O(n \log n)$ total time to process the string plus $O(\log n)$ time per query. The required space is $O(n \log n)$ in general and $O(n)$ in a natural particular case. As a simple application, we describe an algorithm reporting all palindromic rich substrings of an input string in $O(n \log n)$ time and $O(n)$ space.

Keywords: Palindrome · Counting palindromes · Eertree · String algorithm · Online algorithm

1 Introduction

Palindromes are one of the most important repetitive structures in strings. During the last decades they were actively studied in formal language theory, combinatorics on words and stringology. Recall that a palindrome is any string $S = a_1 a_2 \cdots a_n$ equal to its reversal $\overleftarrow{S} = a_n \cdots a_2 a_1$.

Active studies on palindrome algorithmics began with the problem of online recognition of palindromes by (multi-tape) Turing machines. Slisenko in a huge paper [20] presented a 6-tape machine recognizing palindromes in real time; using Slisenko's ideas, Galil described a much simpler construction [9]. In the more powerful RAM model, a big variety of fast algorithms was developed for palindrome-related problems since then. Manacher [18] came up with a linear-time algorithm capturing all palindromic substrings of a string. The problem of counting distinct palindromes in a string was solved offline in [12] and online in [16]. Knuth, Morris, and Pratt [15] gave a linear-time algorithm for checking whether a string is a product of even-length palindromes. Galil and Seiferas [10] asked for such an algorithm for the *k-factorization* problem: decide whether or not a given string can be factored into exactly k non-empty palindromes, where k is an arbitrary constant. They presented an online algorithm for $k = 1, 2$ and an offline one for $k = 3, 4$. An $O(kn)$ time online algorithm for the length n

This work was Partially supported by the grant 16-01-00795 of the Russian Foundation of Basic Research.

G. Fici et al. (Eds.): SPIRE 2017, LNCS 10508, pp. 290–303, 2017.
DOI: 10.1007/978-3-319-67428-5_25

string and any k was designed in [17]. Close to the k-factorization problem is the problem of finding the *palindromic length* of a string, which is the minimal k in its k-factorization. This problem was solved by Fici et al. and independently by Tomohiro et al. in $O(n \log n)$ time [7,14]. Simpler algorithms with the same asymptotics for counting and factorization problems were proposed in [19] on the base of a new data structure called "eertree". Note that hard palindromic-related problems also exist: thus, it is NP-complete to check whether a string can be factorized into distinct palindromes [1].

In this paper we study a generalization of the palindrome counting problem: *in a string arriving online, report at any moment the number of distinct palindromes in any given substring.* Thus, if for $S = aabcac\cdots$ a request for $S[2..6]$ comes, the algorithm should quickly report "4" (a, b, c, cac, the empty string does not count) no matter of whether the request comes after the 6th or the 1006th letter of S. Clearly, an algorithm precomputing the answers to all requests has at least quadratic complexity. As usual in the online setting, we seek for an algorithm working in $O(n\,\mathrm{polylog}(n))$ time. Our main result is the algorithm processing the length n input string in $O(n \log n)$ time and space into a data structure which returns the number of distinct palindromes in any substring in $O(\log n)$ time.

We also consider a restricted version of the above counting problem, in which the requested substring must be a suffix of the current input. In this case the processing and query time remain the same, while the linear space suffices. Finally, we apply the obtained solutions to the offline version of the counting problem: given a string and a set of pairs of indices, output the number of distinct palindromes in each substring specified by such a pair.

A length n string contains at most n distinct palindromes [5]. The "rich" strings with the maximum number of palindromes are studied in many papers, see, e.g., [2,11,13]. We apply our technique to the problem of finding all rich substrings of an input string and solve it in $O(n \log n)$ time and $O(n)$ space.

The main theorem, construction and data structure are presented in Sect. 2. Section 3 contains the proof of the main theorem. In Sect. 4, we consider the offline counting problem and the problem on rich substrings.

1.1 Definitions and Notation

We study finite strings, viewing them as arrays of symbols: $S = S[1..n]$. The notation σ stands for the number of distinct symbols of the processed string. We write ε for the empty string, $|S|$ for the length of a string S, and $S[i..j]$ for $S[i]S[i+1]\dots S[j]$, where $S[i..i-1] = \varepsilon$ for any i. The same $[i..j]$ notation is used for ranges in arrays and integers. A *period* of a string $S = S[1..n]$ is any p, $0 < p \le n$ such that $S[1..n-p] = S[p+1..n]$. The minimal period of S is denoted by $\mathsf{per}(S)$. A string T is a *substring* of S (or *contained in* S) if $T = S[i..j]$ for some i and j; we say that T *occurs in* S *at position* i and order the occurrences of T in S by their position (e.g., "last" occurrence is the one with maximum j). A substring $S[1..j]$ (resp., $S[i..n]$) is a *prefix* (resp. *suffix*) of S. If a substring

(prefix, suffix) of S is a palindrome, it is called a *subpalindrome* (resp. *prefix-palindrome*, *suffix-palindrome*). The number of distinct subpalindromes of S is denoted by $\#\mathsf{Pal}(S)$. A subpalindrome $S[l..r]$ has *center* $(l+r)/2$. Throughout the paper we do not count ε as a palindrome.

Fact 1 ([5]). *For a string S and a symbol b, the string Sb (resp., bS) contains at most one palindrome which is not contained in S; this palindrome is the longest suffix-palindrome of Sb (resp., the longest prefix-palindrome of bS).*

Problem SUB-PCOUNT
Input: a sequence of queries $\mathsf{append}(b)$ (b is a symbol) and $\mathsf{return}(i, j)$ ($i \le j$ are integers such that j is at most the number of append's in the preceding input).
Processing/output (online): let S be the current string (initially $S = \varepsilon$). For each $\mathsf{append}(b)$ query, assign Sb to S, output nothing; for each $\mathsf{return}(i, j)$ query, output $\#\mathsf{Pal}(S[i..j])$.

In the *Restricted* SUB-PCOUNT, j in the input is always equal to the current length of S; thus, all requests are about the suffixes of the current string.

2 Main Result, Construction, and Data Structure

Theorem 1. *For an input sequence consisting of n* append *queries and some* return *queries,*

(1) SUB-PCOUNT *can be solved in $O(n \log n)$ time for all* append *queries plus $O(\log n)$ time per* return *query using $O(n \log n)$ space;*
(2) restricted SUB-PCOUNT *can be solved in the same time using $O(n)$ space.*

For a string $S = S[1..n]$ consider the *difference array* $A_n = A_n[1..n]$ such that $A_n[k] = \#\mathsf{Pal}(S[k..n]) - \#\mathsf{Pal}(S[k{+}1..n])$. By Fact 1, $A_n[k] \in \{0, 1\}$. Note that $\#\mathsf{Pal}(S[k..n]) = \sum_{i=k}^{n} A_n[k]$. So if we can efficiently

- build the difference arrays A_j for the prefixes $S[1..j]$ of S online
- retrieve the sums of bits from A_j,

we obtain a solution to the restricted SUB-PCOUNT. If we further show how to store all arrays A_j simultaneously in a compact form, we solve SUB-PCOUNT. We store and update difference arrays using some versions of *segment trees*.

2.1 Segment Tree

Segment tree is a data structure of probably folklore origin, which is popular in the ACM-ICPC community and allows fast computation of different symmetric functions (sums, minima, etc.) on all ranges of an integer array (see e.g., http://wcipeg.com/wiki/Segment_tree)[1]. Segment tree is an alternative to Fenwick's tree [6] and admits both lazy and persistent versions. Below we describe the version for sums which allows the upgrade to a persistent structure.

[1] Unfortunately, there is one more data structure with this name, see, e.g., Wikipedia.

I. For an *ordinary segment tree* storing an array of length n, where $2^{d-1} < n \leq 2^d$, take a fully balanced binary tree with 2^d leaves and delete the $2^d - n$ rightmost leaves together with all internal vertices having no remaining leaves in their subtrees; see Fig. 1. Each vertex has weight; the weight of ith (from the left) leaf is the ith element of the array, and the weight of an internal vertex is the sum of weights of its children (and thus, the total weight of all leaves in its subtree).

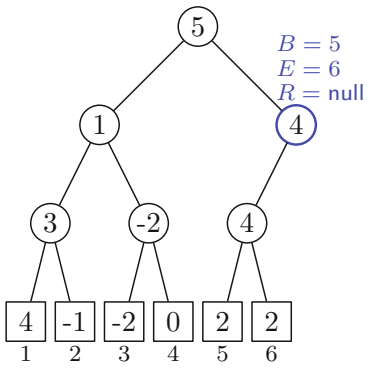

Fig. 1. Example of an ordinary segment tree. Numbers in vertices are weights. Some parameters of the blue vertex are shown. (Color figure online)

For each vertex v we store its children $L[v], R[v]$ ($L[v] = \mathsf{null}$ for leaves only; internal vertices may have $R[v] = \mathsf{null}$), the boundaries $B[v], E[v]$ of the range spanned by the leaves in its subtree, and the weight $W[v]$. We want to perform three types of queries to the stored array A, each in $O(\log n)$ time:

1) $\mathsf{add}(i, \Delta)$: adds Δ to the element $A[i]$;
2) $\mathsf{sum}(i, j)$: requests the sum of all elements of the subarray $A[i..j]$;
3) push: extends A to the right by a zero element.

To adjust weights, we descend from the root to the leaf i, adding Δ to the weight of each vertex. To push a new (nth) element, we descend from the root following the binary expansion of n, setting $E[v] = n$ for existing vertices and creating new vertices of zero weight on the way; if n is a power of 2, we start with creating a new root and making the existing tree its left subtree. For sums, the recursive function $SUM(i, j, v)$ is used:

1: **if** $[i..j] \cap [B[v]..E[v]] = \varnothing$ **then** return 0
2: **if** $[B[v]..E[v]] \subseteq [i..j]$ **then** return $W[v]$
3: **else** return $SUM(i, j, L[v]) + SUM(i, j, R[v])$

The query $\mathsf{sum}(i, j)$ is answered by $SUM(i, j, root)$.

For add and push the $O(\log n)$ time bound is obvious. For sum the bound follows from the observation that at most 4 vertices on each level are touched by the recursive procedure.

II. *Lazy segment tree* supports an extended form $\mathsf{add}(i, j, \Delta)$ of the update query: adding Δ to all $A[k]$ such that $i \leq k \leq j$. To perform such queries in $O(\log n)$ time, a new parameter δ is assigned to each vertex. For any v, the sum of the elements of $A[B[v]..E[v]]$ is $W[v] + (E[v] - B[v] + 1) \cdot \sum_{u \in pred[v]} \delta[u]$, where $pred[v]$ is the set of all predecessors of v including v itself. For an add query, the recursive operation $ADD(i, j, \Delta, v)$ is used:

1: **if** $[i..j] \cap [B[v]..E[v]] = \varnothing$ **then** stop
2: **if** $[B[v]..E[v]] \subseteq [i..j]$ **then** $\delta[v] \leftarrow \delta[v] + \Delta$; stop

3: **else** $W[v] \leftarrow W[v] + \Delta \cdot \#([i..j] \cap [B[v]..E[v]])$
4: $ADD(i, j, \Delta, L[v])$; $ADD(i, j, \Delta, R[v])$; stop

The update query $\mathsf{add}(i, j, \Delta)$ is performed by $ADD(i, j, \Delta, root)$.

Pushing a new element is similar to the ordinary version. Now consider the recursive function $SUM(i, j, \eta, v)$, where η accumulates δ's from $pred[v]$:

1: **if** $[i..j] \cap [B[v]..E[v]] = \varnothing$ **then** return 0
2: $\eta \leftarrow \eta + \delta[v]$
3: **if** $[B[v]..E[v]] \subseteq [i..j]$ **then** return $W[v] + \eta \cdot (E[v] - B[v] + 1)$
4: **else** return $SUM(i, j, \eta, L[v]) + SUM(i, j, \eta, R[v])$

The query $\mathsf{sum}(i, j)$ is answered by $SUM(i, j, 0, root)$. Clearly, both add and sum queries take the same time as the sum query in the ordinary version. Note that both ordinary and lazy versions use linear (in the size of A) space.

Maintaining the current difference array in either ordinary or lazy version of the segment tree suits as a part of solution of restricted SUB-PCOUNT. Indeed, each request $\mathsf{return}(i, j)$ to the difference array A_j is answered by the sum of the range $A_j[i..j]$ in $O(\log n)$ time, while the segment tree itself requires $O(n)$ space. However, to solve SUB-PCOUNT we need access to all arrays A_j simultaneously. Storing them in a naive way would mean an $\Omega(n^2)$ lower bound on the space and time used; to avoid this, we define a *persistent* segment tree. (For the basics on persistent structures see [4].)

III. *Persistent segment tree* stores a lazy segment tree with *all its previous versions* and answers the same queries as the lazy segment tree, but addressed to *any* version. In a persistent tree, we never change the parameters of an existing vertex; a clone of this vertex is created each time a change is needed, and the change is performed in the clone. Note that if we clone a vertex, all its predecessors must be cloned: they now have a new vertex as a child. In particular, a root is cloned on each change in the tree. Thus, the version of the tree is uniqely identified by the root; the array of pointers to the roots can be stored explicitly.

Consider the modification of the ADD procedure (pushing is done in a similar way, while SUM does not change the tree). Now $ADD(i, j, \Delta, v)$ returns a vertex instead of just stopping; in line 1 this is vertex v, and in remaining lines this is a clone v' of v, created if the condition in line 1 fails. In line 4 the results of recursive calls are assigned to $L[v']$ and $R[v']$.

Every update affects $O(\log n)$ vertices, so the memory used is $O((a+n) \log n)$ (for a add's and n push'es). For difference arrays, a push followed by a group of add's updates A_j to A_{j+1}; after this, a group of sum queries come. Thus, there is no need to store links to all versions of the tree; n links to the versions after each group of add's is enough, reducing the memory cost to $O(n \log n)$.

3 Combinatorics of Difference Arrays

In this section we describe how to process append queries, updating the difference array (stored as a segment tree) and auxiliary data structures. By "iteration"

we mean processing of one query (changing the input string from $S[1..j]$ to $S[1..j{+}1]$). Let $T, b, A[1..|T|]$ be a string, a symbol, and an integer array. Consider the function $UPDATE(Tb, A)$ which uses queries to the segment tree of A:[2]

1: push; $t \leftarrow$ number of suffix-palindromes of Tb
2: **for** $(i = 1;\ i \le t;\ i{+}{+})$ **do**
3: $\quad u_i \leftarrow i$th longest suffix-palindrome of Tb
4: $\quad k_i \leftarrow$ position of the suffix u_i in Tb
5: $\quad l_i \leftarrow$ position of last occurrence of u_i in T
6: $\quad \mathsf{add}(k_i, 1);\ \mathsf{add}(l_i, -1)$
7: return A

Lemma 2. *Let A_j, A_{j+1} be the difference arrays of the prefixes $S[1..j]$ and $S[1..j{+}1]$ of a string S. Then $A_{j+1} = UPDATE(S[1..j{+}1], A_j)$.*

Proof. First note that for each $i = 1, \ldots, t$ one has $A_{j+1}[k_i] = 1$ ($S[k_i..j{+}1]$ is a palindrome not contained in $S[k_i{+}1..j{+}1]$) and $A_j[l_i] = 1$ (the palindrome u_i is contained in $S[l_i..j]$ but not in $S[l_i{+}1..j]$). Now we check that $A_{j+1}[k]$ was computed correctly in all cases.

- $A_j[k] = A_{j+1}[k] = 0$: $k \notin \{k_i, l_i\}$ for any i, so there was no $\mathsf{add}(k, \Delta)$ query.
- $A_j[k] = 0$, $A_{j+1}[k] = 1$: $k \ne l_i$ for any i, so there was no $\mathsf{add}(k, -1)$ query. By definition of a difference array, all prefix-palindromes of $S[k..j]$ have other occurrences in $S[k..j]$, while some prefix-palindrome of $S[k..j{+}1]$ has not; hence $S[k..j{+}1]$ is a palindrome (say, u_i), $k = k_i$, and thus $\mathsf{add}(k, 1)$ was performed.
- $A_j[k] = 1$, $A_{j+1}[k] = 0$: $k \ne k_i$ for any i, no $\mathsf{add}(k, 1)$ query. Further, some prefix-palindrome u of $S[k..j]$ has no other occurrences in $S[k..j]$ but occurs in $S[k..j{+}1]$; hence u is a suffix-palindrome of $S[1..j{+}1]$ (say, u_i), $k = l_i$, and thus $\mathsf{add}(k, -1)$ was performed.
- $A_j[k] = 1$, $A_{j+1}[k] = 1$: let u (resp., v) be the prefix-palindrome of $S[k..j]$ (resp., $S[k..j{+}1]$) having no other occurences in this substring. Then either $v = u$ or $v = S[k..j{+}1]$. In the latter case, $k \in \{k_1, \ldots, k_t\}$. Since v is a palindrome, its prefix-palindrome u is its suffix-palindrome; so u is a suffix-palindrome of $S[1..j{+}1]$ and $k \in \{l_1, \ldots, l_t\}$. Hence both queries $\mathsf{add}(k, 1)$ and $\mathsf{add}(k, -1)$ were performed. In the former case, $S[k..j{+}1]$ is not a palindrome (otherwise v would occur as its suffix), so $k \notin \{k_1, \ldots, k_t\}$. Further, a palindrome at position k either is not a suffix of $S[1..j{+}1]$ or has a later occurrence in $S[k..j]$ as a suffix of v, so $k \notin \{l_1, \ldots, l_t\}$. Hence there was no $\mathsf{add}(k, \Delta)$ query.

Thus we proved that $UPDATE(S[1..j{+}1], A_j)$ correctly computes the number $A_{j+1}[k]$ for $k = 1, \ldots, j$. Finally, $k_t = j{+}1$, so $\mathsf{add}(k_t, 1)$ sets the correct value $A_{j+1}[j{+}1] = 1$. □

Due to Lemma 2, to update A_j to A_{j+1} one can iterate through suffix-palindromes of $S[1..j{+}1]$ preparing the lists $\{k_i\}_1^t$ and $\{l_i\}_1^t$ for add queries.

[2] By Fact 1, only l_1 can be undefined; if so, we assume that $\mathsf{add}(l_1, -1)$ is ignored.

To optimize this process, we rule out some of the positions which appear in two lists simultaneously. We need some special notions.

Let $u_1, \dots, u_t$ be all non-empty suffix-palindromes of a string S in the order of decreasing length. Since u_j is a suffix of u_i for any $i < j$, the sequence of minimal periods of $u_1, \dots, u_t$ is non-increasing. The sets of suffix-palindromes with the same minimal period are *series of palindromes* (for S):

$$\underbrace{u_1, \dots, u_{i_1}}_{p_1}, \underbrace{u_{i_1+1}, \dots, u_{i_2}}_{p_2}, \dots, \underbrace{u_{i_{r-1}+1}, \dots, u_t}_{p_r}.$$

We refer to the longest and the shortest palindrome in a series as its *head* and *baby* respectively (they coincide in the case of a 1-element series). A crucial observation [7,14,17] is that the length of a head is multiplicatively smaller than the length of the baby from the previous series, and thus every string of length n has $O(\log n)$ series. The following lemma on the structure of series is easily implied by [17, Lemmas 2,3,7].

Lemma 3. *Let U be a series of palindromes with period p for a string S. There exist $k \geq 1$ and unique palindromes u, v with $|uv| = p$, $v \neq \varepsilon$ such that U equals one of the following sets:*

(1) $U = \{(uv)^{k+1}u, (uv)^k u, \dots, (uv)^2 u\}$ and the next series' head is uvu,
(2) $U = \{(uv)^k u, (uv)^{k-1}u, \dots, uvu\}$ and the next series' head is u,
(3) $U = \{v^k, v^{k-1}, \dots, v\}$, $p = 1$, $|v| = 1$, $u = \varepsilon$, and U is the last series for S.

We need some further properties.

Lemma 4.

(1) If v is a baby in some series of palindromes, then it is a baby of any series containing v (in any string).
(2) If a palindrome v satisfies $\mathsf{per}(v) > |v|/2$, then v is a baby.
(3) Let subpalindromes u, v of S share the same center. If $|u| = |v| - 2$, u is a baby and v is not, then $\mathsf{per}(u) = \mathsf{per}(v)$.

Proof. Let v' be the longest suffix-palindrome of a palindrome v.

(1) "v is a baby" is equivalent to $\mathsf{per}(v) > \mathsf{per}(v')$ and thus depends on v only.
(2) We have $\mathsf{per}(v') \leq |v'| = |v| - \mathsf{per}(v) < \mathsf{per}(v)$, so v is a baby.
(3) Assume $\mathsf{per}(u) < \mathsf{per}(v)$. By statement 2, $\mathsf{per}(v) \leq |v|/2$. Then $\mathsf{per}(u) + \mathsf{per}(v) \leq |u| + 1$. Since both $\mathsf{per}(u)$, $\mathsf{per}(v)$ are periods of u, by the Fine–Wilf theorem [8] u has the period $\gcd(\mathsf{per}(u), \mathsf{per}(v))$. Since $\mathsf{per}(u)$ is the minimal period of u, it divides $\mathsf{per}(v)$. Lemma 3 implies $u = xyxyx$, $\mathsf{per}(u) = |xy|$, $\mathsf{per}(v) = |xyxy|$. Then $v = y[1]{\cdot}xyxyx{\cdot}y[1]$, implying that $\mathsf{per}(u)$ is a period of v. This contradiction proves statement 3. □

Now return to difference arrays.

Lemma 5. *Let $u_1, \ldots, u_r$ $(r > 1)$ be a series of suffix-palindromes of $S[1..j{+}1]$ occurring at positions $k_1 < \ldots < k_r$, respectively. Then for any $i = 1, \ldots, r-1$ the function $UPDATE(S[1..j{+}1], A_j)$ performs both* $\mathsf{add}(k_i, 1)$ *and* $\mathsf{add}(k_i, -1)$.

Proof. The suffix-palindrome u_i occurs at position k_i, so $\mathsf{add}(k_i, 1)$ is performed when processing u_i. Note that u_{i+1} occurs in $S[1..j{+}1]$ at position k_{i+1} as a suffix and also at position k_i as a prefix of u_i; by definition of a series, $|u_{i+1}| = |u_i| - p > p$, where p is the period of the series. Hence these two occurrences of u_{i+1} overlap. Then u_{i+1} cannot occur at position l such that $k_i < l < k_{i+1}$ (the occurrences of u_{i+1} at positions l and k_{i+1} would overlap, implying that $S[l..j{+}1]$ is a palindrome; but this is impossible, since u_i has no suffix-palindromes which are longer than u_{i+1}). Hence k_i is the position of the last occurrence of u_{i+1} in $S[1..j]$, and $\mathsf{add}(k_i, -1)$ is performed when processing u_{i+1}. □

According to Lemma 5, the statement of Lemma 2 remains true if we replace $UPDATE$ with the following function $UPDATE'(Tb, A)$:

1: push
2: Inc ← list of positions of babies in all series for Tb
3: Dec ← list of positions of last occurrences in T of heads in all series for Tb
4: **for** each $k \in$ Inc **do** $\mathsf{add}(k, 1)$
5: **for** each $l \in$ Dec **do** $\mathsf{add}(l, -1)$
6: return A

Lists Inc and Dec have length equal to the number of series, which is $O(\log n)$. Moreover, the following lemma holds.

Lemma 6. *The lists* Inc *and* Dec *can be built in $O(\log n)$ time.*

We store the information about palindromes in S in the *eertree* data structure [19]. Below we briefly describe the version of eertree used here.

Eertree is a tree-like data structure consisting of vertices, labeled edges, and two types of suffix links. Each vertex is identified with a unique subpalindrome of S; two special vertices correspond to ε and to "imaginary" palindrome -1. An edge with a label a leads from a vertex v to ava (from -1 to a). The suffix link $\mathsf{link}[v]$ points to the longest suffix-palindrome of v; the series link $\mathsf{serieslink}[v]$ points to the baby of the series to which v belongs; see Fig. 2 for an example. (In [19] the related version of eertree used series links to the head of the next series; the described version is built in almost the same way. We use it because it is convenient to store additional information in the babies.) Each vertex v contains: the length $\mathsf{len}[v]$ of the palindrome; the list $\mathsf{go}[v]$ of children (as a binary search tree); the

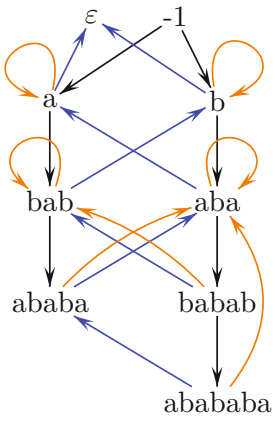

Fig. 2. Eertree for $S = abababa$ (with black edges, blue suffix links and orange series links; edge labels are omitted). No edges from ε means no even-length palindromes (Color figure online)

links $\mathsf{link}[v]$ and $\mathsf{serieslink}[v]$. The eertree requires $O(n)$ space and can be built online in $O(n \log \sigma)$ time with at most $O(\log n)$ time per symbol appended to the string. Each iteration either finds or creates the vertex corresponding to the longest suffix-palindrome maxPal of the updated string, so we have free access to maxPal. Note that we also have an $O(1)$ access to the smallest period of any palindrome v: $\mathsf{per}(v) = \mathsf{len}[v] - \mathsf{len}[\mathsf{link}[v]]$, and to the position of v as a suffix-palindrome: $\mathsf{pos}(v) = |S| - \mathsf{len}[v] + 1$.

Proof (of Lemma 6). Given the eertree of $S[1..j{+}1]$, it is easy to build the list Inc in $O(\log n)$ time. Starting from maxPal, we visit all babies in the order of decreasing length (the longest baby is $v = \mathsf{serieslink}[\mathsf{maxPal}]$, the next is $\mathsf{serieslink}[\mathsf{link}[v]]$, and so on) and append their positions to Inc.

For Dec, first recall that a palindrome v is a baby iff $\mathsf{per}(v) > \mathsf{per}(\mathsf{link}[v])$; we refer to any series with the baby v as *v-series*. Creating a vertex v in the eertree, we will check whether v is a baby; if yes, we will create a new stack for the occurrences of all v-series and point to this stack from v. An element of a stack is a pair $(U.\mathsf{card}, U.\mathsf{pos})$ consisting of the cardinality of the v-series U and the position of its head. The maintenance of stacks and the array Dec is as follows (top2 is the second element of a stack, the function pop2 pops it):

1: $u \leftarrow$ head of the current series; $v \leftarrow \mathsf{serieslink}[u]$
2: **if** $u \neq v$ **then** $\mathsf{top.card} \leftarrow \mathsf{top.card} + 1$ **else** $\mathsf{push}(1, \mathsf{pos}(u))$
3: **if** $\mathsf{top2} \neq \mathsf{null}$ **then** add $(\mathsf{top2.pos} + (\mathsf{top2.card} - \mathsf{top.card}) \cdot \mathsf{per}(v))$ to Dec
4: **if** $\mathsf{top.card} = \mathsf{top2.card}$ **then** pop2
5: $u \leftarrow \mathsf{link}[v]$ ▷ proceed to the next series

Example 7. Let us follow the evolution of the stack of the baby *aba*:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
$S =$	a	b	a	b	a	b	a	c	a	b	a	d	a	b	a	b	a	b	a	$\cdots$
			↓		↓		↓				↓				↓		↓		↓	
			1,1		2,1		3,1				1,9				1,13		2,13		3,13	
											3,1				3,1		3,1			

The stack is created when processing $S[1..3]$; for $S[1..5]$, the head of the *aba*-series is $ababa \neq aba$, so we increase $\mathsf{top.card}$; the same applies to $S[1..7]$. For $S[1..11]$ *aba* is the head, so we push a new element to the stack; then we add the position of the last occurrence of *aba* in $S[1..10]$, which is $5 = 1 + (3-1) \cdot 2$, to Dec. Processing $S[1..15]$, we push a new series, add 9 to Dec, and pop the series $(1, 9)$. For $S[1..17]$ we increase $\mathsf{top.card}$ and add the position $3 = 1 + (3-2) \cdot 2$ of the occurrence of *ababa* to Dec. Finally, for $S[1..19]$ we increase $\mathsf{top.card}$, add 1 to Dec and pop the series $(3, 1)$.

Consider a baby $v = xyx$ with $\mathsf{per}(v) = p = |xy|$ (for a baby of the form $xyxyx$ the same argument works). If v ends in S at position j, its next occurrence ends at position $j{+}p$ or later; otherwise, v would have a smaller period. If it ends at $j{+}p$, the occurence ending at j extends to $xyxyx$ and all palindromes from this v-series also extend. Hence at $j{+}p$ we also have a v-series; compared to

the v-series at j, it has one more element and the same position of the head. If the next occurrence of v ends at some $j' > j+p$, it becomes a new v-series of length 1. Thus, the line 2 correctly maintains the last v-series.

The **if** in line 4 ensures that the lengths of the stored v-series are in strictly decreasing order and the previous occurrence of the head of the top v-series is in the series contained in top2. Since in a k-element v-series the head has the length $|v| + (k-1)p$, the **if** in line 3 correctly identifies the position of this previous occurrence. Overall, each head is processed in $O(1)$ time, so the time to build Dec is $O(\log n)$. □

Remark 8. All series stored simultaneously in stacks have different heads. Hence all stacks together contain $O(n)$ elements (which is the number of distinct palindromes in S).

Storing difference arrays in a standard segment tree, we update it $O(n \log n)$ times while processing the whole string S. Thus, we can solve the restricted SUB-PCOUNT in $O(n \log^2 n)$ time and linear space. To improve working time, we use the lazy segment tree and update its segments in a lazy manner also. Note that in the proof of Lemma 6 we process suffix-palindromes in the order of decreasing length; so both lists Inc and Dec are sorted in increasing order.

We use two lists of "postponed" updates, iInc and iDec; their elements are pairs of the form (l, r), $1 \le l \le r \le n$, sorted by the first coordinate in increasing order. After building Inc and Dec, we do not update the segment tree immediately; instead, we add new updates to iInc/iDec and perform some of the earlier postponed updates. We store all postponed increments in iInc. On each iteration, after building Inc we "merge" iInc and Inc: for each $k \in$ Inc, if iInc contains a pair of the form $(k+1, r)$, we replace it with (k, r); if no such pair exists, we add the pair (k, k) to iInc; all pairs which were not changed, are deleted; deleting a pair (l, r) from iInc, we apply add$(l, r, 1)$ to the lazy segment tree. Note that after merging the size of iInc equals the size of Inc (and hence is $O(\log n)$). Since both lists are sorted, the merging takes $O(\log n)$ time. The lists iDec and Dec are processed in the same way. Thus we have

Lemma 9. *On each iteration, the lists* iInc *and* iDec *are updated in* $O(\log n)$ *time.*

To retrieve the sum of a range $[i..j]$ of A_j, we request the sum of $[i..j]$ from the segment tree and correct it checking the intersections of $[i..j]$ with all intervals stored in iInc and iDec. Note that even if some leaves of the segment tree contain integers different from 0,1 (the order of updates may be violated due to laziness), the sum is computed correctly. Since these lists are of logarithmic size, we get

Lemma 10. *A lazily updated lazy segment tree returns the sum of any range of the stored array in* $O(\log n)$ *time.*

Now we prove the property which is key for our time bound.

Lemma 11. *The total number of pairs deleted from the lists* iInc *and* iDec *during* n *iterations is* $O(n)$.

Proof. Assume that a pair (l, r) is deleted from iInc at $(j{+}1)$th iteration. This means that l is in Inc after jth iteration and $l{-}1$ is not in Inc after $(j{+}1)$th iteration; so we need to estimate the number of pairs (l, j) with this property. The same applies to the lists iDec and Dec.

We begin with the study of Inc. By definition, $u = S[l..j]$ is a baby palindrome, while $v{=}S[l{-}1..j{+}1]$ is not. The center of u is $c = (j{-}l)/2$. There are two cases for v.

Case Inc.1: v is not a palindrome. Then $S[1..j{+}1]$ and all longer prefixes of S have no suffix-palindrome with the center c, while $S[1..j]$ has; we say that the center c *dies* at $(j{+}1)$th iteration. Clearly, each center dies at most once and the centers of suffix-palindromes of the same string are different, so the number of pairs (l, j) falling into this case is bounded by the number of possible centers, which is $2n$.

Case Inc.2: v is a palindrome, but not a baby. By Lemma 4(3), $\mathsf{per}(v) = \mathsf{per}(u)$. For the case $|u| = 1$, there are at most n pairs of the form (j, j), so we assume $|u| > 1$ below. Let $u' = \mathsf{link}[u] = S[l'..j]$, $v' = \mathsf{link}[v]$. Since $|v'| = |v| - \mathsf{per}[v] = |u| + 2 - \mathsf{per}[u] = |u'| + 2$, we have $v' = S[l'{-}1..j{+}1]$. Then u' and v' share the center $c' = (j{-}l')/2$. In addition, $\mathsf{per}(u') < \mathsf{per}(u)$ and $\mathsf{per}(v') = \mathsf{per}(v)$ since u is a baby and v is not. Hence $|\mathsf{link}[u']| = |u'| - \mathsf{per}(u')$ and $|\mathsf{link}[v']| = |v'| - \mathsf{per}(v) < |u'| - \mathsf{per}(u') + 2$. Thus, the center c'' of $\mathsf{link}[u']$ dies at $(j{+}1)$th iteration. Different suffix-palindromes u points to different centers c'' (including the case $u' = S[j]$, $u'' = S[j{+}1..j] = \varepsilon$; the center $c'' = j{+}1/2$ also dies at the $(j{+}1)$th iteration), so the number of pairs (l, j) falling into this case is at most $2n$, as in the previous paragraph. Overall, the number of pairs (l, j) for all the lists Inc is $O(n)$.

Consider the array Dec. The pair (l, j) is such that (a) for some i, k, u one has $u = S[l..i] = S[k..j]$, u is the head of some series at jth iteration, and $S[l..i]$ is the last occurrence of u in $S[1..j{-}1]$; (b) the property (a) fails for the pair $(l{-}1, j{+}1)$. Consider $v = S[k{-}1..j{+}1]$ and $w = S[l{-}1..i{+}1]$. If v is a palindrome and $\mathsf{per}(v) = \mathsf{per}(u)$, then v is the head of some series at $(j{+}1)$th iteration. Indeed, if v is not a head, then $S[k{-}1{-}\mathsf{per}[v]..j{+}1]$ is a palindrome from the same series as v; hence $S[k{-}\mathsf{per}(v)..j]$ is a palindrome from the same series as u, so u is not a head, contradicting (a). Further, if $v = w$, then $S[l{-}1..i{+}1]$ is the last occurrence of w in $S[1..j]$; hence the pair $(l{-}1, j{+}1)$ satisfies (a), thus contradicting (b). Therefore, either v is not a palindrome, or $\mathsf{per}(v) > \mathsf{per}(u)$, or w is not a palindrome, or $\mathsf{per}(w) > \mathsf{per}(u)$. Below we prove a linear upper bound on the number of pairs (l, j) falling into each case.

Case Dec.1: v is not a palindrome. Same as Case Inc.1.

Case Dec.2: v is a palindrome, $\mathsf{per}(v) > \mathsf{per}(u)$. This is a simplified version of Case Inc.2: the center c' of $u' = \mathsf{link}[u] = S[k'..j]$ dies at the $(j{+}1)$th iteration (including the case $u' = \varepsilon$), and we have the number $2n$ of centers as the upper bound.

Case Dec.3: w is not a palindrome. This means death of the center $d = (i - l)/2$ at the $(i{+}1)$th iteration. At each iteration, starting with the $(j{+}1)$th, building

Dec one does not refer to palindromes with the center d because they have later occurrences inside the suffix u of $S[1..j]$. Thus, no other pair falling into this case can be related to d; again the number $2n$ of centers is the upper bound.

Case Dec.4: w is a palindrome, $\mathsf{per}(w) > \mathsf{per}(u)$. Similar to Case Dec.2, the center d' of $u' = \mathsf{link}[u] = S[l'..i]$ dies at the $(i{+}1)$th iteration. As in Case Dec.3, subsequent iterations do not refer to palindromes with the center d, because they occur later inside the suffix u of $S[1..j]$. So the $2n$ upper bound works here as well. Thus, we get $O(n)$ pairs (l, j) for all lists Dec. □

Proof (of Theorem 1). Let us process each query $\mathsf{append}(b)$ as follows:

1: push ▷ into a lazy segment tree
2: update $eertree(S)$ to $eertree(Sb)$
3: **if** a vertex v is created and v is a baby **then** create a stack for v-series
4: build Inc; update stacks; build Dec ▷ using $eertree(Sb)$; see Lemma 6
5: merge Inc and iInc; **for** each deleted pair (l, r) **do** $\mathsf{add}(l, r, 1)$
6: merge Dec and iDec; **for** each deleted pair (l, r) **do** $\mathsf{add}(l, r, -1)$

The segment tree (by definition), the eertree (see [19]), and auxiliary stacks (Remark 8) require $O(n)$ space; other data uses less space. The working time is: $O(n \log \sigma)$ for eertree [19]; $O(\log n)$ per iteration for the Inc/Dec lists (Lemma 6) and for the iInc/iDec lists (Lemma 9); $O(\log n)$ per add query (Sect. 2.1). Since the number of add queries is $O(n)$ (Lemma 11), we spend $O(n \log n)$ time for n append's, as required.

The answer to $\mathsf{return}(i, j)$ is

$$\mathsf{sum}(i, j) + \sum_{(l,r)\in \mathsf{iInc}} \#([i..j] \cap [l..r]) - \sum_{(l,r)\in \mathsf{iDec}} \#([i..j] \cap [l..r]).$$

This query for restricted SUB-PCOUNT is answered in $O(\log n)$ time by Lemma 10. For SUB-PCOUNT, this query is answered within the same time bound using the versions of the segment tree and the lists iInc, iDec after appending of $S[j]$. All versions of the segment tree are available from the persistent segment tree (which uses $O(n \log n)$ space); all versions of iInc/iDec can be stored explicitly, in $O(n \log n)$ space as well. Thus, both statements of the theorem are proved. □

4 Some Applications

First we describe all rich substrings of a given string. Since substrings of rich strings are rich, it is enough to list all maximal (not extendable) rich substrings.

Theorem 12. *All maximal rich substrings of a length n string S can be found in $O(n \log n)$ time and $O(n)$ space.*

Proof. The substring $S[i..j]$ is the longest rich suffix of the prefix $S[1..j]$ of S iff the difference array A_j contains 1's in all positions from $[i..j]$ and 0 in position

$i-1$ (if $i > 1$). Let $S[i..j]$ and $S[i'..j+1]$ be the longest rich suffixes of $S[1..j]$ and $S[1..j+1]$ respectively; then $S[i..j]$ is a maximal rich substring of S iff $i' > i$.

We process S online using the eertree and build the lists Inc and Dec on each iteration. To keep track of the longest rich suffixes, we do not need to store difference arrays. The procedure is as follows. After the first iteration, the longest rich suffix is $S[1..1]$. Assume that $S[i..j]$ is such a suffix after the jth iteration. On the $(j+1)$th iteration we compare the new lists Inc and Dec to find the maximum k which belongs to Dec but not to Inc; then $A_{j+1}[k] = 0$ and $A_{j+1}[l] = 1$ for all $l > \max\{i, k\}$. So if $k \geq i$, the new longest rich suffix is $S[k+1..j+1]$ and $S[i..j]$ is reported as a maximal rich substring; otherwise, the longest rich suffix is $S[i..j+1]$ and nothing is reported.

Since Inc and Dec have $O(\log n)$ length and are ordered, k can be found in $O(\log n)$ time (e.g., during the mergesort of Inc and Dec). By Lemma 6, the lists are built in $O(\log n)$ time also. So the whole string is processed in $O(n \log n)$ time. The space usage is dominated by the linear space for the eertree. □

Now consider the offline analog of SUB-PCOUNT: *given a string* $S[1..n]$ *and a set of intervals from* $[1..n]$*, report the number of distinct palindromes in each interval*. Using Theorem 1, we suggest the following solution.

- If the total length L of intervals is $O(n \log n)$, build an eertree for each interval separately and obtain the number of distinct palindromes from the size of the eertree; an eertree can be built offline in linear time [19, Proposition 11]. So we need $O(L)$ time and $O(l)$ space, where l is the maximum length of an interval.
- If the number k of intervals is $O(n \log n)$, we sort them by the right end in $O(k)$ time and solve the problem as restricted SUB-PCOUNT in $O((n+k) \log n)$ time and $O(n)$ space.
- If k is not $O(n \log n)$, we solve the problem as SUB-PCOUNT in $O(k \log n)$ time and $O(n \log n)$ space.

Finally, an interesting open question is the optimality of the solutions presented in this paper. As was pointed by one of the referees, since we compute sums only for suffixes of difference arrays, we might use Dietz's construction [3] to report such sums in $O(\log n / \log \log n)$ time. However, this works only if we update individual elements of arrays, so we should spend $\Omega(n \log^2 n / \log \log n)$ time processing $\Omega(n \log n)$ updates. So it remains open whether one can improve the time bounds of Theorem 1.

References

1. Bannai, H., Gagie, T., Inenaga, S., Kärkkäinen, J., Kempa, D., Piatkowski, M., Puglisi, S.J., Sugimoto, S.: Diverse palindromic factorization is NP-complete. In: Potapov, I. (ed.) DLT 2015. LNCS, vol. 9168, pp. 85–96. Springer, Cham (2015). doi:10.1007/978-3-319-21500-6_6
2. Brlek, S., Hamel, S., Nivat, M., Reutenauer, C.: On the palindromic complexity of infinite words. Int. J. Found. Comput. Sci. **15**(2), 293–306 (2004)

3. Dietz, P.F.: Optimal algorithms for list indexing and subset rank. In: Dehne, F., Sack, J.-R., Santoro, N. (eds.) WADS 1989. LNCS, vol. 382, pp. 39–46. Springer, Heidelberg (1989). doi:10.1007/3-540-51542-9_5
4. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. J. Comput. Syst. Sci. **38**(1), 86–124 (1989)
5. Droubay, X., Justin, J., Pirillo, G.: Episturmian words and some constructions of de Luca and Rauzy. Theoret. Comput. Sci. **255**, 539–553 (2001)
6. Fenwick, P.M.: A new data structure for cumulative frequency tables. Soft. Pract. Experience **24**(3), 327–336 (1994)
7. Fici, G., Gagie, T., Kärkkäinen, J., Kempa, D.: A subquadratic algorithm for minimum palindromic factorization. J. Discrete Algorithms **28**, 41–48 (2014)
8. Fine, N.J., Wilf, H.S.: Uniqueness theorems for periodic functions. Proc. Am. Math. Soc. **16**, 109–114 (1965)
9. Galil, Z.: Real-time algorithms for string-matching and palindrome recognition. In: Proceedings of 8th Annual ACM Symposium on Theory of Computing (STOC 1976), pp. 161–173. ACM, New York, USA (1976)
10. Galil, Z., Seiferas, J.: A linear-time on-line recognition algorithm for "Palstar". J. ACM **25**, 102–111 (1978)
11. Glen, A., Justin, J., Widmer, S., Zamboni, L.: Palindromic richness. Eur. J. Comb. **30**(2), 510–531 (2009)
12. Groult, R., Prieur, E., Richomme, G.: Counting distinct palindromes in a word in linear time. Inform. Process. Lett. **110**, 908–912 (2010)
13. Guo, C., Shallit, J., Shur, A.M.: Palindromic rich words and run-length encodings. Inform. Process. Lett. **116**(12), 735–738 (2016)
14. Tomohiro, I., Sugimoto, S., Inenaga, S., Bannai, H., Takeda, M.: Computing palindromic factorizations and palindromic covers on-line. In: Kulikov, A.S., Kuznetsov, S.O., Pevzner, P. (eds.) CPM 2014. LNCS, vol. 8486, pp. 150–161. Springer, Cham (2014). doi:10.1007/978-3-319-07566-2_16
15. Knuth, D.E., Morris, J., Pratt, V.: Fast pattern matching in strings. SIAM J. Comput. **6**, 323–350 (1977)
16. Kosolobov, D., Rubinchik, M., Shur, A.M.: Finding distinct subpalindromes online. In: Proceedings of Prague Stringology Conference, PSC 2013, pp. 63–69. Czech Technical University in Prague (2013)
17. Kosolobov, D., Rubinchik, M., Shur, A.M.: Pal^k is linear recognizable online. In: Italiano, G.F., Margaria-Steffen, T., Pokorný, J., Quisquater, J.-J., Wattenhofer, R. (eds.) SOFSEM 2015. LNCS, vol. 8939, pp. 289–301. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46078-8_24
18. Manacher, G.: A new linear-time on-line algorithm finding the smallest initial palindrome of a string. J. ACM **22**(3), 346–351 (1975)
19. Rubinchik, M., Shur, A.M.: EERTREE: an efficient data structure for processing palindromes in strings. In: Lipták, Z., Smyth, W.F. (eds.) IWOCA 2015. LNCS, vol. 9538, pp. 321–333. Springer, Cham (2016). doi:10.1007/978-3-319-29516-9_27
20. Slisenko, A.: Recognition of palindromes by multihead Turing machines. In: Proceeding of the Steklov Institute of Mathematics, vol. 129, pp. 30–202 (1973). In Russian, English translation by Silverman, R.H., American Mathematical Society, Providence, R.I. (1976), 25–208