Theoretical Computer Science 979 (2023) 114183

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Finding top-k longest palindromes in substrings [☆]

Kazuki Mitani [a], Takuya Mieno [b,*], Kazuhisa Seto [c], Takashi Horiyama [c]

[a] Graduate School of Information Science and Technology, Hokkaido University, Kita 14, Nishi 9, Kita-ku, Sapporo, 060-0814, Japan
[b] Department of Computer and Network Engineering, University of Electro-Communications, 1-5-1 Chofugaoka, Chofu, 182-8585, Japan
[c] Faculty of Information Science and Technology, Hokkaido University, Kita 14, Nishi 9, Kita-ku, Sapporo, 060-0814, Japan

ARTICLE INFO

Article history:
Received 12 June 2023
Accepted 8 September 2023
Available online 18 September 2023

Keywords:
String algorithms
Palindromes
Internal queries
Top-k queries

ABSTRACT

Palindromes are strings that read the same forward and backward. Problems of computing palindromic structures in strings have been studied for many years with the motivation of their application to biology. The longest palindrome problem is one of the most important and classical problems regarding palindromic structures, that is, to compute the longest palindrome appearing in a string T of length n. The problem can be solved in $\mathcal{O}(n)$ time by the famous algorithm of Manacher (1975) [27]. This paper generalizes the longest palindrome problem to the problem of finding the top-k longest palindromes in an arbitrary substring, including the input string T itself. The internal top-k longest palindrome query is, given a substring $T[i..j]$ of T and a positive integer k as a query, to compute the top-k longest palindromes appearing in $T[i..j]$. This paper proposes a linear-size data structure that can answer internal top-k longest palindromes query in optimal $O(k)$ time. Also, given the input string T, our data structure can be constructed in $\mathcal{O}(n \log n)$ time. For $k = 1$, the construction time is reduced to $\mathcal{O}(n)$.

1. Introduction

A string which is the same as its reversal is called a palindrome. Palindromes have been widely studied with the motivation of their application to biology [23]. Computing and counting palindromes in a string are fundamental tasks. Manacher [27] proposed an $\mathcal{O}(n)$-time algorithm that computes all maximal palindromes in the string of length n. Droubay et al. [17] showed that any string of length n contains at most $n+1$ distinct palindromes (including the empty string). Then, Groult et al. [22] proposed an $\mathcal{O}(n)$-time algorithm to enumerate the number of distinct palindromes in a string. The above $\mathcal{O}(n)$-time algorithms are time-optimal since reading the input string of length n takes $\Omega(n)$ time.

Regarding the longest palindrome computation, Funakoshi et al. [19] considered the problem of computing the longest palindromic substring of the string T' after a single character insertion, deletion, or substitution is applied to the input string T of length n. Of course, using $\mathcal{O}(n)$ time, we can obtain the longest palindromic substring of T' from scratch. However, this idea is naïve and appears to be inefficient. To avoid such inefficiency, Funakoshi et al. [19] proposed an $\mathcal{O}(n)$-space data structure that can compute the solution for any editing operation given as a query in $\mathcal{O}(\log(\min\{\sigma, \log n\}))$ time where σ is the alphabet size. Amir et al. [7] considered the dynamic longest palindromic substring problem, which is an extension of Funakoshi et al.'s problem where up to $\mathcal{O}(n)$ sequential editing operations are allowed. They proposed an algorithm that

☆ This article belongs to Section A: Algorithms, automata, complexity and games, Edited by Paul Spirakis.

* Corresponding author.
E-mail addresses: kazukida199911204649@eis.hokudai.ac.jp (K. Mitani), tmieno@uec.ac.jp (T. Mieno), seto@ist.hokudai.ac.jp (K. Seto), horiyama@ist.hokudai.ac.jp (T. Horiyama).

https://doi.org/10.1016/j.tcs.2023.114183
0304-3975/

Table 1
Manacher's algorithm [27] can compute the longest palindrome in a string T in linear time. We study three generalized problems and give efficient data structures and algorithms. All the proposed data structures are of linear size.

		longest palindrome	top-k palindromes
String T	preprocessing	$\mathcal{O}(n)$ time [27]	$\mathcal{O}(n)$ time
	query	$\mathcal{O}(1)$ time	$\mathcal{O}(k)$ time
Query substring	preprocessing	$\mathcal{O}(n)$ time	$\mathcal{O}(n \log n)$ time
$T[i..j]$	query	$\mathcal{O}(1)$ time	$\mathcal{O}(k)$ time

solves this problem in $\mathcal{O}(\sqrt{n}\log^2 n)$ time per a single character edit with high probability with a data structure of size $\mathcal{O}(n \log n)$, which can be constructed in $\mathcal{O}(n \log^2 n)$ time. Furthermore, Amir and Boneh [6] proposed an algorithm running in poly-logarithmic time per a single character substitution.

Internal queries are queries about substrings of the input string T. Let us consider a situation where we solve a certain problem for each of k different substrings of T. If we run an $\mathcal{O}(|w|)$-time algorithm from scratch for each substring w, the total time complexity can be as large as $\mathcal{O}(kn)$. To be more efficient, by performing an appropriate preprocessing on T, we construct some data structure for the query to output each solution efficiently. Such efficient data structures for palindromic problems are known. Rubinchik and Shur [31] proposed an algorithm that computes the number of distinct palindromes in a given substring of an input string of length n. Their algorithm runs in $\mathcal{O}(\log n)$ time with a data structure of size $\mathcal{O}(n \log n)$, which can be constructed in $\mathcal{O}(n \log n)$ time. Amir et al. [7] considered a problem of computing the longest palindromic substring in a given substring of the input string of length n; it is called the internal longest palindrome query. Their algorithm runs in $\mathcal{O}(\log n)$ time with a data structure of size $\mathcal{O}(n \log n)$, which can be constructed in $\mathcal{O}(n \log^2 n)$ time.

This paper proposes a new algorithm for the internal longest palindrome query. The algorithm of Amir et al. [7] uses 2-dimensional orthogonal range maximum queries [3,4,12]; furthermore, the time and space complexities of their algorithm are dominated by this query. Instead of 2-dimensional orthogonal range maximum queries, by using palindromic trees [32], weighted ancestor queries [20], and range maximum queries [18], we obtain a time-optimal algorithm.

Theorem 1. *Given a string T of length n over a linearly sortable alphabet, we can construct a data structure of size $\mathcal{O}(n)$ in $\mathcal{O}(n)$ time that can answer any internal longest palindrome query in $\mathcal{O}(1)$ time.*

Here, an alphabet is said to be *linearly sortable* if any sequence of n characters from Σ can be sorted in $\mathcal{O}(n)$ time. For example, the integer alphabet $\{1, 2, \dots, n^c\}$ for some constant c is linearly sortable because we can sort a sequence from the alphabet in linear time by using a radix sort with base n. We also assume the word-RAM model with word size $\omega \geq \log n$ bits for input size n.

Furthermore, we consider a more general problem of finding palindromes, i.e., the problem of finding the *top-k* longest palindromes in a substring of T, rather than just the longest palindrome in T. Then, we finally prove the following proposition, which will be given as a corollary in Section 4.

Corollary 1. *Given a string T of length n, we can construct a data structure of size $\mathcal{O}(n)$ in $\mathcal{O}(n \log n)$ time that can answer any internal top-k longest palindrome query in $\mathcal{O}(k)$ time.*

Our results are summarized in Table 1.

Related work Internal queries have been studied on many problems, not only those related to palindromic structures. For instance, Kociumaka et al. [26] considered the internal pattern matching queries that are ones for computing the occurrences of a substring U of the input string T in another substring V of T. Besides, internal queries for string alignment [14,33–35], longest common prefix [1,5,21,29], and longest common substring [7] have been studied in the last two decades. See [25] for an overview of internal queries. We also refer to [2,10,11,15,16,24] and references therein.

Paper organization The rest of this paper is organized as follows. Section 2 gives some notations and definitions. Section 3 shows our data structure to solve the internal longest palindrome queries. Section 4 shows how to compute the top-k longest palindromes in (sub)strings. Finally, Section 5 concludes this paper.

2. Preliminaries

2.1. Strings and palindromes

Let Σ be an alphabet. An element of Σ is called a *character*, and an element of Σ^* is called a *string*. The empty string ε is the string of length 0. The length of a string T is denoted by $|T|$. For each i with $1 \leq i \leq |T|$, the i-th character of T is denoted by $T[i]$. For each i and j with $1 \leq i, j \leq |T|$, the string $T[i]T[i+1]\cdots T[j]$ is denoted by $T[i..j]$. For convenience, let

	s	dif		
ε	0	0	$suf_{1,1}$	suf_1
a	1	1	$suf_{2,1}$	suf_2
aba	3	2	$suf_{3,1}$	suf_3
ababa	5	2	$suf_{3,2}$	
abababa	7	2	$suf_{3,3}$	
abababaabababa	14	7	$suf_{4,1}$	suf_4
abababaabababaabababa	21	7	$suf_{4,2}$	
abababaabababaabababababababaabababaabababa	43	22	$suf_{5,1}$	suf_5

Fig. 1. Palindromic suffixes of a string and the partition $(suf_1, \ldots, suf_5)$ of their lengths. Three integers $s_3 = 3$, $s_4 = 5$, and $s_5 = 7$ are represented by a single arithmetic progression suf_3 since $dif_3 = dif_4 = dif_5 = 2$. Since s_4 is the second smallest term in suf_3, $suf_{3,2} = s_4$.

$T[i'..j'] = \varepsilon$ if $i' > j'$. If $T = xyz$, then x, y, and z are called a *prefix*, *substring*, and *suffix* of T, respectively. They are called a *proper prefix*, a *proper substring*, and a *proper suffix* of T if $x \neq T$, $y \neq T$, and $z \neq T$, respectively. The string y is called an *infix* of T if $x \neq \varepsilon$ and $z \neq \varepsilon$. The reversal of string T is denoted by T^R, i.e., $T^R = T[|T|] \cdots T[2]T[1]$. A string T is called a *palindrome* if $T = T^R$. Note that ε is also a palindrome. For a palindromic substring $T[i..j]$ of T, the center of $T[i..j]$ is $\frac{i+j}{2}$. A palindromic substring $T[i..j]$ is called a *maximal palindrome* in T if $i = 1$, $j = |T|$, or $T[i-1] \neq T[j+1]$. In what follows, we consider an arbitrary fixed string T of length $n > 0$. In this paper, we assume that the alphabet Σ is linearly sortable. We also assume the word-RAM model with word size $\omega \geq \log n$ bits.

Let q be the number of palindromic suffixes of T. Let $suf(T) = (s_1, s_2, \ldots, s_q)$ be the sequence of the lengths of palindromic suffixes of T sorted in increasing order. Further let $dif_i = s_i - s_{i-1}$ for each i with $2 \leq i \leq q$. For convenience, let $dif_1 = 0$. Then, the sequence $(dif_1, \ldots, dif_q)$ is monotonically non-decreasing (Lemma 7 in [28]). Let $(suf_1, suf_2, \ldots, suf_p)$ be the partition of $suf(T)$ such that for any two elements s_i, s_j in $suf(T)$, $s_i, s_j \in suf_k$ for some k iff $dif_i = dif_j$. By definition, each suf_k forms an arithmetic progression. It is known that the number p of arithmetic progressions satisfies $p \in \mathcal{O}(\log n)$ [9,28]. For $1 \leq k \leq p$ and $1 \leq \ell \leq |suf_k|$, $suf_{k,\ell}$ denote the ℓ-th term of suf_k. Fig. 1 shows an example of the above definitions.

2.2. Tools

In this section, we list some data structures used in our algorithm in Section 3.

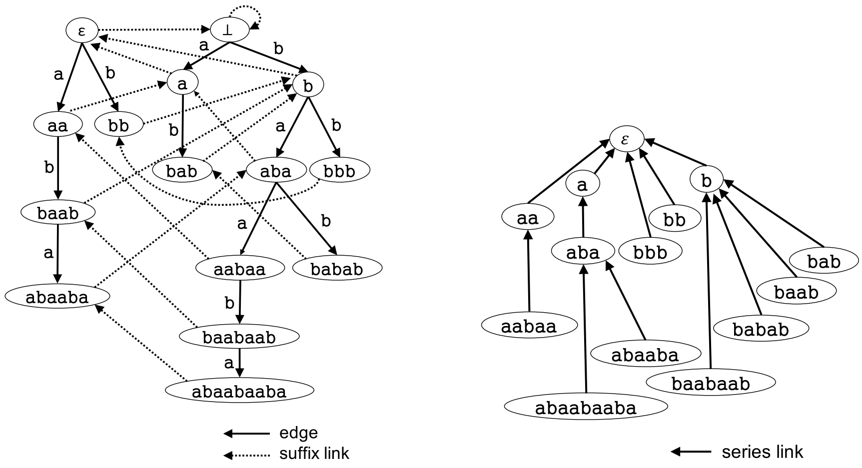
Palindromic trees and series trees The *palindromic tree* of T is a data structure that represents all distinct palindromes in T [32]. The palindromic tree of T, denoted by $\mathsf{paltree}(T)$, has d ordinary nodes and one auxiliary node $\perp$ where $d \leq n+1$ is the number of all distinct palindromes in T. Each ordinary node v corresponds to a palindromic substring of T (including the empty string ε) and stores its length as $\mathsf{weight}(v)$. For the auxiliary node $\perp$, we define $\mathsf{weight}(\perp) = -1$. For convenience, we identify each node with its corresponding palindrome. For an ordinary node v in $\mathsf{paltree}(T)$ and a character c, if nodes v and cvc exist, then an edge labeled c connects these nodes. The auxiliary node $\perp$ has edges to all nodes corresponding to length-1 palindromes. Each node v in $\mathsf{paltree}(T)$ has a *suffix link* that points to the longest palindromic proper suffix of v. Let $\mathsf{link}(v)$ be the string pointed to by the suffix link of v. We define $\mathsf{link}(\varepsilon) = \mathsf{link}(\perp) = \perp$. See Fig. 2(a) for example. For each node v corresponding to a non-empty palindrome in $\mathsf{paltree}(T)$, let $\delta_v = |v| - |\mathsf{link}(v)|$ be the difference between the lengths of v and its longest palindromic proper suffix. For convenience, let $\delta_\varepsilon = 0$. Each node v corresponding to a non-empty palindrome has a *series link* that points to the longest palindromic proper suffix u of v such that $\delta_u \neq \delta_v$. Let $\mathsf{serieslink}(v)$ be the string pointed to by the series link of v.

Let $\mathsf{LSufPal}$ be an array of length n such that $\mathsf{LSufPal}[j]$ stores a pointer to the node in $\mathsf{paltree}(T)$ corresponding to the longest palindromic suffix of $T[1..j]$ for each $1 \leq j \leq n$. The definition of $\mathsf{LSufPal}$ is identical to the array $\mathsf{node}[1]$ defined in [32], and it was shown that $\mathsf{node}[1]$ can be computed in $\mathcal{O}(n)$ time. Hence, $\mathsf{LSufPal}$ can be computed in $\mathcal{O}(n)$ time. Let $\mathsf{LPrePal}$ be an array of length n such that $\mathsf{LPrePal}[i]$ stores a pointer to the node in $\mathsf{paltree}(T)$ corresponding to the longest palindromic prefix of $T[i..n]$ for each $1 \leq i \leq n$. $\mathsf{LPrePal}$ can be computed in $\mathcal{O}(n)$ time as well as $\mathsf{LSufPal}$.

Theorem 2 *(Proposition 4.10 in [32]). Given a string T over a linearly sortable alphabet, the palindromic tree of T, including its suffix links and series links, can be constructed in $\mathcal{O}(n)$ time. Also,* $\mathsf{LSufPal}$ *and* $\mathsf{LPrePal}$ *can be computed in $\mathcal{O}(n)$ time.*[1]

Let us consider the subgraph $\mathcal{S}$ of $\mathsf{paltree}(T)$ that consists of all ordinary nodes and reversals of all series links. By definition, $\mathcal{S}$ has no cycle, and $\mathcal{S}$ is connected (any node is reachable from the node ε), i.e., it forms a tree. We call the tree

[1] Note that the definition of $\mathsf{LSufPal}$ is the same as that of $\mathsf{node}[1]$ in [32].



(a) paltree(T) (b) seriestree(T)

Fig. 2. Illustration for the palindromic tree and the series tree of string $T = \mathtt{abaabaabababbb}$. Since $\delta_{\mathtt{abaabaaba}} = |\mathtt{abaabaaba}| - |\mathtt{abaaba}| = 3$, $\delta_{\mathtt{abaaba}} = |\mathtt{abaaba}| - |\mathtt{aba}| = 3$, and $\delta_{\mathtt{aba}} = |\mathtt{aba}| - |\mathtt{a}| = 2$, then $\mathsf{serieslink}(\mathtt{abaabaaba}) = \mathtt{aba}$. Substring abaabaaba stores the arithmetic progression representing $\{6, 9\}$, abaaba stores the arithmetic progression representing $\{6\}$ and aba stores the arithmetic progression representing $\{3\}$.

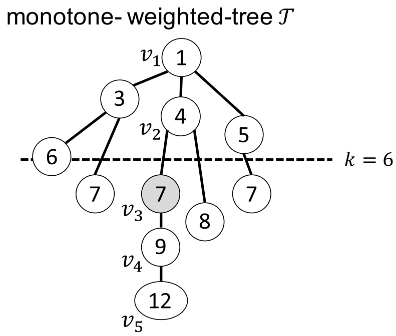


Fig. 3. Illustration for weighted ancestor query. Integers in nodes denote the weights. Given a node v_5 in a monotone-weighted tree $\mathcal{T}$ and an integer $k = 6$ for query, WAQ returns the node v_3 since v_3 is an ancestor of v_5, $\mathsf{weight}(v_3) > k = 6$, and the weight of the parent v_2 of v_3 is not greater than $k = 6$.

$\mathcal{S}$ the *series tree* of T and denote it by $\mathsf{seriestree}(T)$. By definition of series links, the set of lengths of palindromic suffixes of v that are longer than $|\mathsf{serieslink}(v)|$ can be represented by an arithmetic progression. Each node v stores the arithmetic progression, represented by a triple consisting of its first term, its common difference, and the number of terms. Arithmetic progressions for all nodes can be computed in linear time by traversing the palindromic tree. It is known that the length of a path consisting of series links is $\mathcal{O}(\log n)$ [32]. Hence, the height of $\mathsf{seriestree}(T)$ is $\mathcal{O}(\log n)$. See Fig. 2(b) for illustration.

Weighted ancestor query A rooted tree whose nodes are associated with integer weights is called a *monotone-weighted tree* if the weight of every non-root node is not smaller than the parent's weight. Given a monotone-weighted tree $\mathcal{T}$ for preprocessing and a node v and an integer k for query, a weighted ancestor query (WAQ) returns the ancestor u closest to the root of v such that the weight of u is greater than k. Let $\mathsf{WAQ}_{\mathcal{T}}(v, k)$ be the output of the weighted ancestor query for tree $\mathcal{T}$, node v, and integer k. See Fig. 3 for a concrete example. It is known that there is an $\mathcal{O}(N)$-space data structure that can answer any weighted ancestor query in $\mathcal{O}(\log\log N)$ time where N is the number of nodes in the tree [8]. In general, the query time $\mathcal{O}(\log\log N)$ is known to be optimal within $\mathcal{O}(N)$ space [30]. On the other hand, if the height of the input tree is low enough, the query time can be improved:

Theorem 3 *(Proposition 15 in [20]). Given a monotone-weighted tree with N nodes and height $\mathcal{O}(\omega)$, one can construct an $\mathcal{O}(N)$ space data structure in $\mathcal{O}(N)$ time that can answer any weighted ancestor query in constant time.*

In this paper, we use weighted ancestor queries only on the series tree of T whose height is $\mathcal{O}(\log n) \subseteq \mathcal{O}(\omega)$, where ω is the word size, thus we will apply Theorem 3. Note that we assume the word-RAM model with word size $\omega \geq \log n$ bits.

Range maximum query Given an integer array A of length m for preprocessing and two indices i and j with $1 \leq i \leq j \leq m$ for query, range maximum query returns the index of a maximum element in the sub-array $A[i..j]$. Let $\mathrm{RMQ}_A(i, j)$ be the output of the range maximum query for array A and indices i, j. In other words, $\mathrm{RMQ}_A(i, j) = \arg\max_k\{A[k] \mid i \le k \le j\}$. The following result is known:

Theorem 4 *(Theorem 5.8 in [18]). Let m be the size of the input array A. There is a data structure of size $2m + o(m)$ bits that can answer any range maximum query on A in constant time. The data structure can be constructed in $\mathcal{O}(m)$ time.*

3. Internal longest palindrome queries

In this section, we propose an efficient data structure for the internal longest palindrome query defined as follows:

Internal longest palindrome query
Preprocess: A string T of length n.
Query input: Two indices i and j with $1 \le i \le j \le n$.
Query output: The longest palindromic substring in $T[i..j]$.

Our data structure requires only $\mathcal{O}(n)$ words of space and can answer any internal longest palindrome query in constant time. To answer queries efficiently, we classify all palindromic substrings of T into palindromic prefixes, palindromic infixes, and palindromic suffixes. First, we compute the longest palindromic prefix and the longest palindromic suffix of $T[i..j]$. Second, we compute a palindromic infix that is a candidate for the answer. As we will discuss in a later subsection, this candidate may not be the longest palindromic infix of $T[i..j]$. Finally, we compare the three above palindromes and output the longest one.

3.1. Palindromic suffixes and prefixes

First, we compute the longest palindromic suffix of $T[i..j]$. In the preprocessing, we build $\mathsf{seriestree}(T)$ and a data structure for the weighted ancestor queries on $\mathsf{seriestree}(T)$, and compute $\mathsf{LSufPal}$ as well. The query algorithm consists of three steps:

Step 1: Obtain the longest palindromic suffix of $T[1..j]$.
We obtain the longest palindromic suffix v of $T[1..j]$ from $\mathsf{LSufPal}[j]$. If $|v| \le |T[i..j]|$, then v is the longest palindromic suffix of $T[i..j]$. Then, we return $T[j-|v|+1..j]$, and the algorithm terminates. Otherwise, we continue to Step 2.

Step 2: Determine the group to which the desired length belongs.
Let ℓ be the length of the longest palindromic suffix of $T[i..j]$ we want to know. We use the longest palindromic suffix v of $T[1..j]$ obtained in Step 1. First, we find the shortest palindrome u that is an ancestor of v in $\mathsf{seriestree}(T)$ and has a length at least $|T[i..j]|$. Such a palindrome (equivalently the node) u can be found by weighted ancestor query on the series tree, i.e., $u = \mathsf{WAQ}_{\mathsf{seriestree}(T)}(v, j-i)$. Then, $|u|$ is an upper bound of ℓ. Let suf_α be the group such that $|u| \in \mathit{suf}_\alpha$. If the smallest element $\mathit{suf}_{\alpha,1}$ in suf_α is at most $|T[i..j]|$, the length ℓ belongs to the same group suf_α as $|u|$. Otherwise, the length ℓ belongs to the previous group $\mathit{suf}_{\alpha-1}$.

Step 3: Calculate the desired length.
Let suf_β be the group to which the length ℓ belongs, which is determined in Step 2. Since suf_β is an arithmetic progression, i.e., $\mathit{suf}_{\beta,\gamma} = \mathit{suf}_{\beta,1} + (\gamma-1)\mathit{dif}_\beta$ for $1 \le \gamma \le |\mathit{suf}_\beta|$, the desired length ℓ can be computed by using a constant number of arithmetic operations. Then, we return $T[j-\ell+1..j]$.

See Fig. 4 for illustration. Now, we show the correctness of the algorithm and analyze time and space complexities.

Lemma 1. *We can compute the longest palindromic suffix and prefix of $T[i..j]$ in $\mathcal{O}(1)$ time with a data structure of size $\mathcal{O}(n)$ that can be constructed in $\mathcal{O}(n)$ time.*

Proof. In the preprocessing, we build $\mathsf{seriestree}(T)$, $\mathsf{LSufPal}$, $\mathsf{LPrePal}$ and a data structure of weighted ancestor query on $\mathsf{seriestree}(T)$ in $\mathcal{O}(n)$ time (Theorems 2 and 3). Recall that since the height of $\mathsf{seriestree}(T)$ is $\mathcal{O}(\log n) \subseteq \mathcal{O}(\omega)$, we can apply Theorem 3 to the series tree. Again, by Theorem 2 and 3, the space complexity is $\mathcal{O}(n)$ words of space.

In what follows, let ℓ be the length of the longest palindromic suffix of $T[i..j]$. In Step 1, we can obtain the longest palindromic suffix v of $T[1..j]$ by just referring to $\mathsf{LSufPal}[j]$. If $|v| \le |T[i..j]|$, v is also the longest palindromic suffix of $T[i..j]$, i.e., $\ell = |v|$. Otherwise, v is not a substring of $T[i..j]$. In Step 2, we first query $\mathsf{WAQ}_{\mathsf{seriestree}(T)}(v, j-i)$. The resulting node u corresponds to a palindromic suffix of $T[1..j]$, which is longer than $|T[i..j]|$. Let suf_α and suf_β be the groups to which $|u|$ and ℓ belong to, respectively. If the smallest element $\mathit{suf}_{\alpha,1}$ in suf_α is at most $j-i+1$, then the desired length ℓ satisfies $\mathit{suf}_{\alpha,1} \le \ell \le |u|$. Namely, $\beta = \alpha$. Otherwise, if s is greater than $j-i+1$, ℓ is not in suf_α but is in $\mathit{suf}_{\alpha-x}$ for some $x > 1$. If we assume that ℓ belongs to $\mathit{suf}_{\alpha-y}$ for some $y \ge 2$, the length of $\mathsf{serieslink}(u)$ belonging to $\mathit{suf}_{\alpha-1}$ is longer than

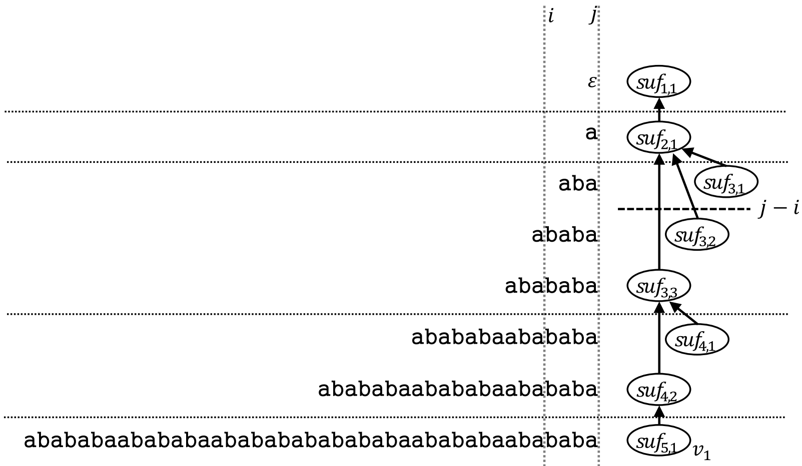


Fig. 4. Illustration for how to compute the longest palindromic suffix of $T[i..j]$, when $T[1..j] =$ abababaabababaababababababababaabababaababababa. The graph on the right hand depicts a part of the series tree of a string T, and the lengths of palindromes are written inside the nodes. In Step 1, we obtain the length $suf_{5,1}$ of the longest palindromic suffix v_1 of $T[1..j]$. In Step 2, we find $suf_{3,3}$ by $\mathsf{WAQ}_{\mathsf{seriestree}(T)}(v_1, j-i)$. Since $suf_{3,1} > j-i+1$, the desired length belongs to suf_3. In Step 3, since suf_3 is an arithmetic progression, we can find that $suf_{3,1}$ is the longest palindromic suffix of $T[i..j]$ in constant time.

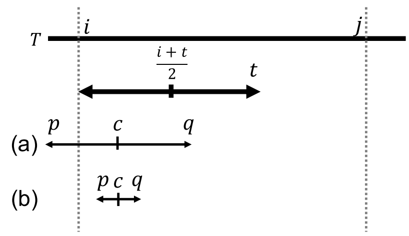


Fig. 5. Illustration for Lemma 2. Two-way arrows denote palindromic substrings of T. $T[i..t]$ is the longest palindromic prefix of $T[i..j]$. A palindrome whose center c is less than $\frac{i+t}{2}$ is either (a) not a substring of $T[i..j]$ or (b) shorter than the longest palindromic prefix of $T[i..j]$ as shown in this figure.

$T[i..j]$. However, it contradicts that u is the answer of $\mathsf{WAQ}_{\mathsf{seriestree}(T)}(v, j-i)$. Hence, if s is greater than $j-i+1$, then the length ℓ is in $suf_{\alpha-1}$. Namely, $\beta = \alpha - 1$. In Step 3, we can compute ℓ in constant time since we know the arithmetic progression suf_β to which ℓ belongs. More specifically, ℓ is the largest element that is in suf_β and is at most $j-i+1$.

Throughout the query algorithm, all operations, including WAQ and operations on arithmetic progressions, can be done in constant time. Thus, the query algorithm runs in constant time. □

We can compute the longest palindromic prefix of $T[i..j]$ in a symmetric way using LPrePal instead of LSufPal.

3.2. Palindromic infixes

Next, we compute the longest palindromic infix except for ones that cannot be the longest palindromic substring due to the longest palindromic prefix or the longest palindromic suffix of the query substring. We show that to find the desired palindromic infix, it suffices to consider maximal palindromes whose centers are between the centers of the longest palindromic prefix and the longest palindromic suffix of $T[i..j]$. Let t be the ending position of the longest palindromic prefix and s be the starting position of the longest palindromic suffix. Namely, $T[i..t]$ is the longest palindromic prefix and $T[s..j]$ is the longest palindromic suffix of $T[i..j]$.

Lemma 2. *Let w be a palindromic infix of $T[i..j]$ and c be the center of w. If $c < \frac{i+t}{2}$ or $c > \frac{s+j}{2}$, w cannot be the longest palindromic substring of $T[i..j]$.*

Proof. Palindrome w is a proper substring of $T[i..t]$ (resp. $T[s..j]$) if $c < \frac{i+t}{2}$ (resp. $c > \frac{s+j}{2}$). Then, w is shorter than $T[i..t]$ or $T[s..j]$ (see also Fig. 5). □

Then, we consider palindromes whose centers are between the centers of the longest palindromic prefix and the longest palindromic suffix of $T[i..j]$.

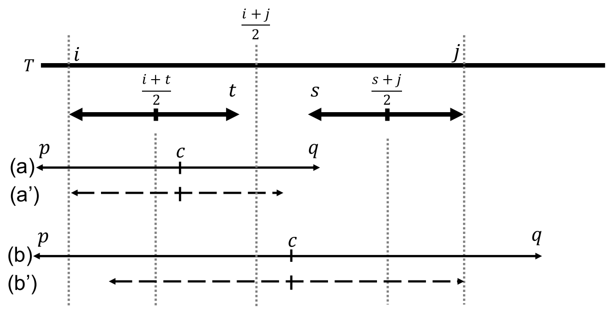


Fig. 6. Illustration for contradictions in the proof of Lemma 3. $T[i..t]$ is the longest palindromic prefix and $T[s..j]$ is the longest palindromic suffix of $T[i..j]$. If a palindrome as (a) exists, there exists a palindromic prefix (a') of $T[i..j]$ that is longer than $T[i..t]$, a contradiction. Similarly, the existence of a palindrome as (b) leads to a contradiction.

Lemma 3. *Let w be a palindromic substring of T and c be the center of w. If $\frac{i+t}{2} < c < \frac{s+j}{2}$, then w is a palindromic infix of $T[i..j]$.*

Proof. Let $w = T[p..q]$. Then, $c = \frac{p+q}{2}$. To prove that w is a palindromic infix, we show that $p > i$ and $q < j$. For the sake of contradiction, we assume $p \leq i$. If $\frac{i+t}{2} < c \leq \frac{i+j}{2}$, there exists a palindromic prefix w_1 whose center is c. This contradicts that $T[i..t]$ is the longest palindromic prefix of $T[i..j]$ since $T[i..t]$ is a substring of w_1 (see also Fig. 6). Otherwise, if $\frac{i+j}{2} < c < \frac{s+j}{2}$, there exists a palindromic suffix whose w_2 center is c. This contradicts that $T[s..j]$ is the longest palindromic suffix of $T[i..j]$ since $T[i..t]$ is a substring of w_2 (see also Fig. 6). Therefore, $p > i$. We can show $q < j$ in a symmetric way. □

By Lemmas 2 and 3, when a palindromic infix w of $T[i..j]$ is the longest palindromic substring of $T[i..j]$, the center of w must be located between $\frac{i+t}{2}$ and $\frac{s+j}{2}$. Furthermore, w is a maximal palindrome in T. In other words, w is the longest maximal palindrome in T whose center c satisfies $\frac{i+t}{2} < c < \frac{s+j}{2}$. To find such a (maximal) palindrome, we build a succinct RMQ data structure on the length-$(2n-1)$ array MP that stores the lengths of maximal palindromes in T. For each integer and half-integer $c \in \{1, 1.5, \ldots, n-0.5, n\}$, $\mathsf{MP}[2c-1]$ stores the length of the maximal palindrome whose center is c. By doing so, when the indices t and s are given, we can find a candidate for the longest palindromic substring which is an infix of $T[i..j]$ in constant time. More precisely, the length of the candidate is $\mathsf{MP}[\mathsf{RMQ}_{\mathsf{MP}}(i+t, s+j-2)]$ since the center c of the candidate satisfies $\frac{i+t}{2} < c < \frac{s+j}{2}$ $(i+t-1 < 2c-1 < s+j-1)$. By Manacher's algorithm [27], MP can be constructed in $\mathcal{O}(n)$ time. Then, we obtain the following lemma.

Lemma 4. *Given the longest palindromic prefix $T[i..t]$ and the longest palindromic suffix $T[s..j]$ of $T[i..j]$, we can compute the longest palindromic infix of $T[i..j]$ whose centers are between the centers of $T[i..t]$ and $T[s..j]$ in $\mathcal{O}(1)$ time with a data structure of size $\mathcal{O}(n)$ that can be constructed in $\mathcal{O}(n)$ time.*

By Lemmas 1 and 4, we have shown our main theorem:

Theorem 1. *Given a string T of length n over a linearly sortable alphabet, we can construct a data structure of size $\mathcal{O}(n)$ in $\mathcal{O}(n)$ time that can answer any internal longest palindrome query in $\mathcal{O}(1)$ time.*

4. Top-*k* longest palindromes

We denote by $\mathsf{TopLPal}_T([i,j],k)$ an array of occurrences of top-k longest palindromic substrings in $T[i..j]$ sorted in their lengths. In other words, $\mathsf{TopLPal}_T([i,j],k)[r] = [s,t]$ means that $T[s..t]$ is the r-th longest palindromic substring in $T[i..j]$. For simplicity, we denote $\mathsf{TopLPal}_T([1,|T|],k)$ as $\mathsf{TopLPal}_T(k)$.

4.1. Top-k longest palindromes in a string

First, we consider a problem to compute the top-k longest palindromes in the input string and propose an efficient algorithm.

Top-*k* longest palindromes problem
Input: A string T of length n and an integer k.
Output: An array $\mathsf{TopLPal}_T(k)$.

Firstly, we give an important observation of this problem. For a palindromic substring $P = T[\alpha..\beta]$, we call the substring $T[\alpha+1..\beta-1]$ the *shrink* of P. Note that the shrink of a palindrome is also a palindrome.

Observation 1. *The r-th longest palindrome in a string T is either*

(i) *a maximal palindrome in T or*
(ii) *the shrink of the q-th longest palindrome for some q with $1 \leq q \leq r-1$.*

In our algorithm, we precompute array $M[1..2n-1]$ and dynamically maintain array $R[1..n]$, where each $M[p]$ stores the p-th longest maximal palindrome in T, and each $R[\ell]$ stores the set of palindromes of length ℓ that are already returned and whose palindromic substring is not returned yet (if there is no such palindrome, let $R[\ell] = nil$).

Note that the sorted array M can be computed in $\mathcal{O}(n)$ time by using Manacher's algorithm and radix sorting. Also, since the longest palindrome is $M[1]$, we first return $M[1] = [s,t]$ and then update $R[t-s+1]$ to singleton $\{[s,t]\}$.

When we compute the r-th longest palindrome for some r with $2 \leq r \leq k$, we utilize Observation 1. Let ℓ_{r-1} be the length of the $(r-1)$-th longest palindrome. Then, the length ℓ_r of the r-th longest palindrome is in $\{\ell_{r-1}, \ell_{r-1}-1, \ell_{r-1}-2\}$ because the shrink of the $(r-1)$-th longest palindrome, whose length is $\ell_{r-1}-2$, is at least a palindrome. We pick up a longest palindrome Q within $R[\ell_{r-1}+2] \cup R[\ell_{r-1}+1] \cup R[\ell_{r-1}]$. Note that such Q always exists since $R[\ell_{r-1}] \neq nil$ at this step. Let Q^- be the shrink of Q. We compare its length $|Q^-|$ with the length of the longest maximal palindrome that has not returned yet. Then, the longer one is the r-th longest palindrome P_r. If P_r is equal to Q^-, we remove Q from $R[|Q|]$. At last, we append P_r to the set $R[|P_r|]$.

Since every operation in each r-th step can be done in constant time, the above algorithm runs in $\mathcal{O}(n+k)$ time. Also, since we remove Q from R when we return Q^-, most entries of R are *nil* except for at most three entries at each step. Thus, array R can be implemented within $\min\{3n, k\}$ words of space. In total, our algorithm requires $\mathcal{O}(n)$ working space. We have shown the next lemma.

Lemma 5. *Given a string T of length n and an integer k, we can compute $\mathsf{TopLPal}_T(k)$ in $\mathcal{O}(n+k)$ time with $\mathcal{O}(n)$ working space.*

We further show that the above algorithm can be applied to a query version of the top-k longest palindromes problem defined as below:

Top-k longest palindromes query
Input: A string T of length n.
Query input: An integer k.
Query output: An array $\mathsf{TopLPal}_T(k)$.

In the preprocessing phase, we compute and store the sorted top-n longest palindromes by using the algorithm of Lemma 5. If $k \leq n$, we just scan the pre-stored palindromes and return the top-k ones. Otherwise, we apply the aforementioned algorithm. For both cases, the query time is $\mathcal{O}(k)$, which is optimal. Thus, the following theorem holds.

Theorem 5. *After $\mathcal{O}(n)$-time preprocessing on an input string T of length n, we can compute $\mathsf{TopLPal}_T(k)$ in $\mathcal{O}(k)$ time for a given query integer k.*

4.2. Internal top-k longest palindromes query

This subsection considers a more general model; the internal query model.

Internal Top-k longest palindromes query
Input: A string T of length n.
Query input: An interval $[i, j]$ and an integer k.
Query output: An array $\mathsf{TopLPal}_T([i,j],k)$.

First, we give some observations and the idea of our algorithm. The following observation on palindromic symmetry is fundamental.

Observation 2. *Let $T[\alpha..\beta]$ be a palindrome and let c be an (half-) integer with $\alpha \leq c < \frac{\alpha+\beta}{2}$. There is a palindromic substring of $T[\alpha..\beta]$ whose center is c iff there is a palindromic substring of $T[\alpha..\beta]$ whose center is $\alpha+\beta-c$.*

Similar to Observation 1, we categorize the r-th longest palindrome. Let $LPP_{i,j}$ (resp. $LPS_{i,j}$) be the longest palindromic prefix (resp. suffix) of $T[i..j]$. Further let c_p (resp. c_s) be the center of $LPP_{i,j}$ (resp. $LPS_{i,j}$).

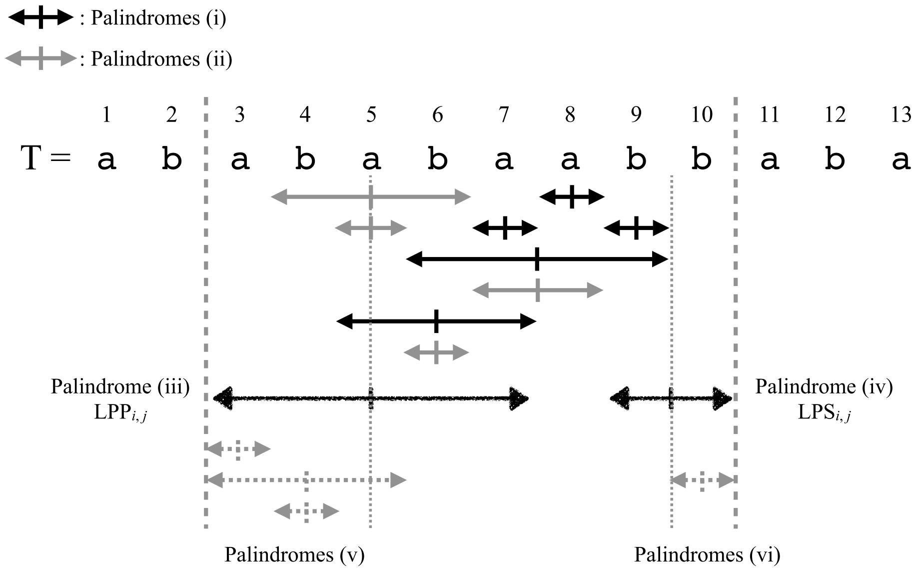


Fig. 7. Illustration for Observation 3. All non-empty palindromic substrings of $T[3..10]$ are depicted, and they are categorized into (i)–(vi).

Observation 3. *The r-th longest palindrome in a string $T[i..j]$ is one of the followings:*

(i) *a maximal palindrome of T whose center is between $c_p + 0.5$ and $c_s - 0.5$, inclusive,*
(ii) *the shrink of the q-th longest palindrome whose center is between c_p and c_s, inclusive, for some q with $1 \leq q \leq r-1$,*
(iii) *the longest palindromic prefix $LPP_{i,j}$ of $T[i..j]$,*
(iv) *the longest palindromic suffix $LPS_{i,j}$ of $T[i..j]$,*
(v) *a palindromic substring of $T[i..j]$ whose center is less than c_p, which is shorter than $LPP_{i,j}$, or*
(vi) *a palindromic substring of $T[i..j]$ whose center is greater than c_s, which is shorter than $LPS_{i,j}$.*

Note that the candidates (iii) and (iv) are not necessarily maximal palindromes of T, and thus, we cannot merge them with (i) in general. Also, the candidates (v) and (vi) are derived by Observation 2. See Fig. 7.

The first and the second candidates (i), (ii) are the same as those of Observation 1. To compute these candidates, we modify the algorithm of Lemma 5 and apply it to this problem. Instead of the sorted array $M[1..2n-1]$, we simply use array $\mathsf{MP}[1..2n-1]$ of the lengths of maximal palindromes in the positional order. In the preprocessing, we construct a *top-k Range Maximum Query* (top-k RMQ) data structure on array MP. The top-k RMQ (a.k.a. sorted range selection query) on an integer array $A[1..n]$ is, given an interval $[i, j] \subseteq [1, n]$ and a positive integer $k \leq j - i + 1$ as a query, to output a sorted list of the top-k largest elements in subarray $A[i..j]$. As for the top-k RMQ, the next result is known:

Theorem 6 *([13]). There is a data structure of size $\mathcal{O}(n)$ which can answer top-k RMQ in $\mathcal{O}(k)$ time for any k. Also, the data structure can be constructed in $\mathcal{O}(n \log n)$ time.*

After constructing a top-k RMQ data structure on array MP, we can enumerate the top-k longest maximal palindromes of T whose center is between $c_p + 0.5$ and $c_s - 0.5$ (i.e., candidates (i)). The second candidate (ii) can be maintained dynamically by using array $\tilde{R}$ of returned palindromes which is almost the same as R but contains only palindromes centered between c_p and c_s. The third and fourth candidates (iii) and (iv) are unique. Thus, we can easily treat them. The fifth and sixth candidates (v), (vi) can be found by palindromic symmetry of *LPP* and *LPS*.

Algorithm Now, we are ready to describe our algorithm. Given a query interval $[i, j]$ and a query integer k, we first run the algorithm of Theorem 1. Then, the longest palindromic substring P_1 in $T[i..j]$ and the longest palindromic prefix/suffix (equivalently, c_p and c_s) of $T[i..j]$ are obtained. We set $\tilde{R}[\ell] = \{[s, s+\ell-1]\}$ where s and ℓ are the starting position and the length of the longest palindromic substring of $T[i..j]$, respectively. The second-longest palindrome P_2 is the longest one in $(\{\mathsf{MP}[x] \mid 2c_p < x < 2c_s\} \cup \{T[\alpha+1..\beta-1] \mid [\alpha, \beta] \in \tilde{R}\} \cup \{LPP_{i,j}, LPS_{i,j}\}) \setminus \mathcal{R}$ where $\mathcal{R}$ is the set of palindromes that have been returned. Before detecting P_2, $\mathcal{R} = \{[s, s+\ell-1]\}$ holds, and after detecting P_2, we update $\mathcal{R} \leftarrow \mathcal{R} \cup \{P_2\}$. In addition, if P_2 is a substring of $LPP_{i,j}$ and the center of P_2 is greater than c_p, then there is the same palindrome as P_2 in the opposite position w.r.t. c_p. Thus, we add the (third-longest) palindrome into $\mathcal{R}$ and continue the procedure. In the case where P_2 is a substring of $LPS_{i,j}$ with a different center from $LPS_{i,j}$, the same symmetric procedure is applied. We iterate the above procedure until k-th longest palindrome is obtained.

Example 1. We give a running example using Fig. 7. Assume that we want to find the top-5 longest palindromes in substring $T[3..10] = \mathtt{ababaabb}$ of string $T = \mathtt{abababaabbaba}$. Palindromes in $T[3..10]$ are categorized as (i)–(vi). Here, the longest maximal palindrome whose center c is $5 < c < 9.5$ is $T[6..9]$. Also, $LPP_{3,10} = T[3..7]$ and $LPS_{3,10} = T[9..10]$ hold. Thus, the longest one in $\mathsf{TopLPal}_T([3, 10], 5)$ is $T[3..7]$. Then, $\tilde{R}[5] = \{[3, 7]\}$. Thus the second-longest palindrome is either $T[6..9]$ (a maximal palindrome), $T[9..10] = LPS_{3,10}$, or $T[4..6]$ (, which is the shrink of $T[3..7] \in \tilde{R}[5]$). Thus we return $T[6..9]$ and update $\tilde{R}[4] = \{[6, 9]\}$. The second-longest maximal palindrome whose center c is $5 < c < 9.5$ is $T[5..7]$. Thus the third-longest palindrome is either $T[5..7]$ (a maximal palindrome), $T[9..10] = LPS_{3,10}$, or $T[4..6]$. Since there are two longest ones $T[5..7]$ and $T[4..6]$ with the same length 3, we return them. Also, we remove $[3, 7]$ from $LPS_{i,j}$ since its shrink $T[4..6]$ has been returned. Finally, since $T[5..7]$ is a palindromic substring of $LPP_{3,10}$, there is a palindrome $T[3..5]$ of length 3 at the mirror position by Observation 2. So we also return $T[3..5]$ and update $\tilde{R}[3] = \{[5, 7], [4, 6], [3, 5]\}$. Then, we have returned the top-5 palindromes $\mathcal{R} = \{[3, 7], [6, 9], [5, 7], [4, 6], [3, 5]\}$, so the algorithm terminates.

Analyzing algorithm At each iteration, the longest palindrome in $\{\mathsf{MP}[x] \mid 2c_p < x < 2c_s\} \setminus \mathcal{R}$ can be computed in constant time by answering top-k RMQ in parallel. The longest palindrome in $\{T[\alpha + 1..\beta - 1] \mid [\alpha, \beta] \in \tilde{R}\} \setminus \mathcal{R}$ can be also computed in constant time since elements in $\tilde{R}$ are sorted by length (cf. the algorithm of Lemma 5). Trivially, the longest one in $\{LPP_{i,j}, LPS_{i,j}\}) \setminus \mathcal{R}$ can be found in constant time. By exploring the longest elements of the above three sets, the first four candidates in Observation 1 have been checked. By the remaining process, the existence of candidates (v) and (vi) in Observation 1 has also been checked. Therefore, the proposed algorithm runs correctly, and its time complexity is dominated by the query time for top-k RMQ. We obtain the next theorem.

Theorem 7. *Given a string T of length n over a linearly sortable alphabet, we can construct a data structure of size $\mathcal{O}(n + \pi_s(n))$ in $\mathcal{O}(n + \pi_p(n))$ time that can answer any internal top-k longest palindrome query in $\mathcal{O}(k + \pi_q(n, i, j, k))$ time where $\pi_p(n)$ is the preprocessing time for top-k RMQ, $\pi_s(n)$ is the size of top-k RMQ data structure, and $\pi_q(n, i, j, k)$ is the query time for top-k RMQ.*

We obtain the next corollary by plugging the result of Theorem 6 into Theorem 7.

Corollary 1. *Given a string T of length n, we can construct a data structure of size $\mathcal{O}(n)$ in $\mathcal{O}(n \log n)$ time that can answer any internal top-k longest palindrome query in $\mathcal{O}(k)$ time.*

5. Conclusions and open problems

In this paper, we considered three variants of the longest palindrome problem on the input string T of length n and proposed algorithms for them. The problems are the followings.

1. **The internal longest palindrome query**, which requires returning the longest palindrome appearing in substring $T[i..j]$.
2. **The top-k longest palindrome query**, which requires returning the top-k longest palindrome appearing in T.
3. **The internal top-k longest palindrome query**, which requires returning the top-k longest palindrome appearing in substring $T[i..j]$.

Note that every problem is a generalization of the longest palindrome problem, which can be solved in $\mathcal{O}(n)$ time [27]. Our proposed data structures are of size $\mathcal{O}(n)$ and can answer every query in optimal time, i.e., in $\mathcal{O}(1)$ time for the internal longest palindrome query and in $\mathcal{O}(k)$ time for the top-k queries. Construction time is $\mathcal{O}(n)$ for the first and the second problem, and $\mathcal{O}(n \log n)$ time for the third problem. Note that this $\mathcal{O}(n \log n)$ term is dominated by the preprocessing time for the top-k RMQ [13].

Open problems Our results achieved optimal time in terms of order notations. It will be an interesting open problem to develop a time-space tradeoff algorithm for variants of the longest palindrome problem. For example, for some parameter $\tau > 1$, can we design a data structure (except for the input string) of size $\mathcal{O}(n/\tau)$ which can answer the internal longest palindrome query in $\mathcal{O}(\tau)$ time? One of the other possible directions to reduce space is designing a data structure of size $\mathcal{O}(d)$ where d is the number of distinct palindromes occurring in T. It is known that d is at most $n + 1$ [17] and thus $\mathcal{O}(d) = \mathcal{O}(n)$ in the worst case. However, in most cases, d is much smaller than n. So, it is worthwhile to design such data structures. For example, the size of the palindromic tree is actually $\mathcal{O}(d)$ rather than $\mathcal{O}(n)$. Whether the space usage of our data structure can be reduced to $\mathcal{O}(d)$ is open.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was partially supported by JSPS KAKENHI Grant Numbers JP20H05964 (TH), JP21H05839 (KS), JP22K21273, and JP23H04381 (TM).

References

[1] Paniz Abedin, Arnab Ganguly, Wing-Kai Hon, Kotaro Matsuda, Yakov Nekrich, Kunihiko Sadakane, Rahul Shah, Sharma V. Thankachan, A linear-space data structure for range-LCP queries in poly-logarithmic time, Theor. Comput. Sci. 822 (2020) 15–22.
[2] Paniz Abedin, Arnab Ganguly, Solon P. Pissis, Sharma V. Thankachan, Efficient data structures for range shortest unique substring queries, Algorithms 13 (11) (2020) 1–9.
[3] Pankaj K. Agarwal, Range searching, in: Handbook of Discrete and Computational Geometry, Chapman and Hall/CRC, 2017, pp. 1057–1092.
[4] Stephen Alstrup, Gerth Stølting Brodal, Theis Rauhe, New data structures for orthogonal range searching, in: 2013 IEEE 54th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, 2000, p. 198.
[5] Amihood Amir, Alberto Apostolico, Gad M. Landau, Avivit Levy, Moshe Lewenstein, Ely Porat, Range LCP, J. Comput. Syst. Sci. 80 (7) (2014) 1245–1253.
[6] Amihood Amir, Itai Boneh, Dynamic palindrome detection, arXiv preprint, arXiv:1906.09732, 2019.
[7] Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, Jakub Radoszewski, Dynamic and internal longest common substring, Algorithmica 82 (12) (2020) 3707–3743.
[8] Amihood Amir, Gad M. Landau, Moshe Lewenstein, Dina Sokol, Dynamic text and static pattern matching, ACM Trans. Algorithms 3 (2) (2007) 19.
[9] Alberto Apostolico, Dany Breslauer, Zvi Galil, Parallel detection of all palindromes in a string, Theor. Comput. Sci. 141 (1) (1995) 163–173.
[10] Maxim Babenko, Paweł Gawrychowski, Tomasz Kociumaka, Ignat Kolesnichenko, Tatiana Starikovskaya, Computing minimal and maximal suffixes of a substring, Theor. Comput. Sci. 638 (2016) 112–121.
[11] Golnaz Badkobeh, Panagiotis Charalampopoulos, Dmitry Kosolobov, Solon P. Pissis, Internal shortest absent word queries in constant time and linear space, Theor. Comput. Sci. 922 (2022) 271–282.
[12] Jon Louis Bentley, Multidimensional divide-and-conquer, Commun. ACM 23 (4) (1980) 214–229.
[13] Gerth Stølting Brodal, Rolf Fagerberg, Mark Greve, Alejandro López-Ortiz, Online sorted range reporting, in: Algorithms and Computation: 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16–18, 2009, in: Proceedings, vol. 20, Springer, 2009, pp. 173–182.
[14] Panagiotis Charalampopoulos, Paweł Gawrychowski, Shay Mozes, Oren Weimann, An almost optimal edit distance oracle, in: 48th International Colloquium on Automata, Languages, and Programming (ICALP 2021), Schloss Dagstuhl-Leibniz-Zentrum Für Informatik, 2021.
[15] Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, Internal dictionary matching, Algorithmica 83 (7) (2021) 2142–2169.
[16] Panagiotis Charalampopoulos, Tomasz Kociumaka, Philip Wellnitz, Faster approximate pattern matching: a unified approach, in: 2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS), IEEE, 2020, pp. 978–989.
[17] Xavier Droubay, Jacques Justin, Giuseppe Pirillo, Episturmian words and some constructions of de Luca and Rauzy, Theor. Comput. Sci. 255 (1–2) (2001) 539–553.
[18] Johannes Fischer, Volker Heun, Space-efficient preprocessing schemes for range minimum queries on static arrays, SIAM J. Comput. 40 (2) (2011) 465–492.
[19] Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, Computing longest palindromic substring after single-character or block-wise edits, Theor. Comput. Sci. 859 (2021) 116–133.
[20] Moses Ganardi, Compression by contracting straight-line programs, in: Petra Mutzel, Rasmus Pagh, Grzegorz Herman (Eds.), 29th Annual European Symposium on Algorithms (ESA 2021), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 204, Schloss Dagstuhl – Leibniz-Zentrum Für Informatik, 2021, pp. 45:1–45:16.
[21] Arnab Ganguly, Manish Patil, Rahul Shah, Sharma V. Thankachan, A linear space data structure for range LCP queries, Fundam. Inform. 163 (3) (2018) 245–251.
[22] Richard Groult, Élise Prieur, Gwénaël Richomme, Counting distinct palindromes in a word in linear time, Inf. Process. Lett. 110 (20) (2010) 908–912.
[23] Dan Gusfield, Algorithms on stings, trees, and sequences: computer science and computational biology, ACM SIGACT News 28 (4) (1997) 41–60.
[24] Kociumaka Tomasz, Minimal suffix and rotation of a substring in optimal time, in: 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016), vol. 54, Schloss Dagstuhl–Leibniz-Zentrum Fuer Informatik, 2016, pp. 28:1–28:12.
[25] Kociumaka Tomasz, Efficient data structures for internal queries in texts, PhD thesis, University of Warsaw, 2018.
[26] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, Internal pattern matching queries in a text and applications, in: Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2014, pp. 532–551.
[27] Glenn Manacher, A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string, J. ACM 22 (3) (1975) 346–351.
[28] Wataru Matsubara, Shunsuke Inenaga, Akira Ishino, Ayumi Shinohara, Tomoyuki Nakamura, Kazuo Hashimoto, Efficient algorithms to compute compressed longest common substrings and compressed palindromes, Theor. Comput. Sci. 410 (8–10) (2009) 900–913.
[29] Kotaro Matsuda, Kunihiko Sadakane, Tatiana Starikovskaya, Masakazu Tateshita, Compressed orthogonal search on suffix arrays with applications to range LCP, in: 31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020), Schloss Dagstuhl-Leibniz-Zentrum Für Informatik, 2020.
[30] Mihai Pătraşcu, Mikkel Thorup, Time-space trade-offs for predecessor search, in: Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing, 2006, pp. 232–240.
[31] Mikhail Rubinchik, Arseny M. Shur, Counting palindromes in substrings, in: International Symposium on String Processing and Information Retrieval, Springer, 2017, pp. 290–303.
[32] Mikhail Rubinchik, Arseny M. Shur, EERTREE: an efficient data structure for processing palindromes in strings, Eur. J. Comb. 68 (2018) 249–265.
[33] Yoshifumi Sakai, A substring–substring LCS data structure, Theor. Comput. Sci. 753 (2019) 16–34.
[34] Yoshifumi Sakai, A data structure for substring-substring LCS length queries, Theor. Comput. Sci. 911 (2022) 41–54.
[35] Alexander Tiskin, Semi-local string comparison: algorithmic techniques and applications, Math. Comput. Sci. 1 (4) (2008) 571–603.