Palindrome Recognition In The Streaming Model

Petra Berenbrink[1], Funda Ergün[1,2], Frederik Mallmann-Trenn[1], and Erfan Sadeqi Azer[1]

1 Simon Fraser University, Burnaby, Canada
2 Indiana University, Bloomington, US

Abstract

A palindrome is defined as a string which reads forwards the same as backwards, like, for example, the string "racecar". In the *Palindrome Problem*, one tries to find all *palindromes* in a given string. In contrast, in the case of the *Longest Palindromic Substring Problem*, the goal is to find an arbitrary one of the longest palindromes in the string.

In this paper we present three algorithms in the streaming model for the the above problems, where at any point in time we are only allowed to use sublinear space. We first present a one-pass randomized algorithm that solves the *Palindrome Problem*. It has an *additive* error and uses $O(\sqrt{n})$ space. We also give two variants of the algorithm which solve related and practical problems. The second algorithm determines the exact locations of *all* longest palindromes using two passes and $O(\sqrt{n})$ space. The third algorithm is a one-pass randomized algorithm, which solves the *Longest Palindromic Substring Problem*. It has a *multiplicative* error using only $O(\log(n))$ space.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Palindromes, Streaming Model, Complementary Palindrome

Digital Object Identifier 10.4230/LIPIcs.STACS.2014.149

1 Introduction

A palindrome is defined as a string which reads forwards the same as backwards, e.g., the string "racecar". In the *Palindrome Problem* one tries to find all *palindromes* (palindromic substrings) in an input string. A related problem is the *Longest Palindromic Substring Problem* in which one tries to find any one of the longest palindromes in the input.

In this paper we regard the streaming version of both problems, where the input arrives over time (or, alternatively, is read as a stream) and the algorithms are allowed space sub linear in the size of the input. Our first contribution is a one-pass randomized algorithm that solves the *Palindrome Problem*. It has an *additive* error and uses $O(\sqrt{n})$ space. The second contribution is a two-pass algorithm which determines the exact locations of all longest palindromes. It uses the first algorithm as the first pass and uses $O(\sqrt{n})$ space. The third is a one-pass randomized algorithm for the *Longest Palindromic Substring Problem*. It has a *multiplicative* error using $O(\log(n))$ space. We also give two variants of the first algorithm which solve other related practical problems.[1]

Palindrome recognition is important in computational biology. Palindromic structures can frequently be found in proteins and identifying them gives researchers hints about the structure of nucleic acids. For example, in *nucleic acid secondary structure prediction*, one is interested in complementary palindromes which are considered in the full version.

[1] The full version of this paper can be accessed at arXiv:1308.3466.

31st Symposium on Theoretical Aspects of Computer Science (STACS'14).
Editors: Ernst W. Mayr and Natacha Portier; pp. 149–161
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

SYMPOSIUM ON THEORETICAL ASPECTS OF COMPUTER SCIENCE

Related work. While palindromes are well-studied, to the best of our knowledge there are no results for the streaming model. Manacher [5] presents a linear time online algorithm that reports at any time whether all symbols seen so far form a palindrome. The authors of [1] show how to modify this algorithm in order to find all palindromic substrings in linear time (using a parallel algorithm).

Some of the techniques used in this paper have their origin in the streaming pattern matching literature. In the *Pattern Matching Problem*, one tries to find all occurrences of a given pattern P in a text T. The first algorithm for pattern matching in the streaming model was shown in [6] and requires $O(\log(m))$ space. The authors of [3] give a simpler pattern matching algorithm with no preprocessing, as well as a related streaming algorithm for estimating a stream's Hamming distance to p-periodicity. Breslauer and Galil [2] provide an algorithm which does not report false negatives and can also be run in real-time. All of the above algorithms in the string model take advantage of Karp-Rabin fingerprints [4].

Our results. In this paper we present three algorithms, *ApproxSqrt*, *Exact*, and *ApproxLog* for finding palindromes and estimating their length in a given stream S of length n.

We assume that the workspace is bounded while the output space is unlimited.
Given an index m in stream S, $P[m]$ denotes the palindrome of maximal length cantered at index m of S. Our algorithms identify a palindrome $P[m]$ by its *midpoint* m and by its length $\ell(m)$. Our first algorithm outputs all palindromes in S and therefore solves the *Palindrome Problem.*

▶ **Theorem 1** (ApproxSqrt). *For any $\varepsilon \in [1/\sqrt{n}, 1]$* Algorithm ApproxSqrt$(S, \varepsilon)$ *reports for every palindrome $P[m]$ in S its midpoint m as well as an estimate $\tilde{\ell}(m)$ (of $\ell(m)$) such that w.h.p.*[2] *$\ell(m) - \varepsilon\sqrt{n} < \tilde{\ell}(m) \leq \ell(m)$. The algorithm makes one pass over S, uses $O(n/\varepsilon)$ time, and $O(\sqrt{n}/\varepsilon)$ space.*

The algorithm can easily be modified to report all palindromes $P[m]$ in S with $\ell(m) \geq t$ and no $P[m]$ with $\ell(m) < t - \varepsilon\sqrt{n}$ for some threshold $t \in \mathbb{N}$. For $t \leq \sqrt{n}$ one can modify the algorithm to report a palindrome $P[m]$ if and only if $\ell(m) \geq t$. Note, the algorithm is also $(1+\varepsilon)$-approximative.

Our next algorithm, *Exact*, uses two-passes to solve the *Longest Palindromic Substring Problem.* It uses *ApproxSqrt* as the first pass. In the second pass the algorithm finds the midpoints of all palindromes of length exactly $\ell_{\max}$ where $\ell_{\max}$ is the (initially unknown) length of the longest palindrome in S.

▶ **Theorem 2** (Exact). Algorithm Exact *reports w.h.p. ℓ_{max} and m for all palindromes $P[m]$ with a length of ℓ_{max}. The algorithm makes two passes over S, uses $O(n)$ time, and $O(\sqrt{n})$ space.*

Arguably the most significant contribution of this paper is an algorithm which requires only logarithmic space. In contrast to *ApproxSqrt* (Theorem 1) this algorithm has a multiplicative error and it reports only one of the longest palindromes (see *Longest Palindromic Substring Problem*) instead of all of them due to the limited space.

▶ **Theorem 3** (ApproxLog). *For any ε in $(0, 1]$,* Algorithm ApproxLog *reports w.h.p. an arbitrary palindrome $P[m]$ of length at least $\ell_{max}/(1+\varepsilon)$. The algorithm makes one pass over S, uses $O(\frac{n\log(n)}{\varepsilon\log(1+\varepsilon)})$ time, and $O(\frac{\log(n)}{\varepsilon\log(1+\varepsilon)})$ space.*

[2] We say an event happens *with high probability* (w.h.p.) if its probability is at least $1 - 1/n^c$ for $c \in \mathbb{N}$.

We also show two practical generalisations of our algorithms which can be run simultaneously. These results are presented in the next observation and the next lemma.

▶ Observation 4. For $\ell_{max} \geq \sqrt{n}$, there is an algorithm which reports w.h.p. the midpoints of all palindromes $P[m]$ with $\ell(m) > \ell_{max} - \varepsilon\sqrt{n}$. The algorithm makes one pass over S, uses $O(n/\varepsilon)$ time, and $O(\sqrt{n}/\varepsilon)$ space.

▶ **Lemma 5.** *For $\ell_{max} < \sqrt{n}$, there is an algorithm which reports w.h.p. ℓ_{max} and a $P[m]$ s.t. $\ell(m) = \ell_{max}$. The algorithm makes one pass over S, uses $O(n)$ time, and $O(\sqrt{n})$ space.*

In the full version of the paper we will show an almost matching bound for the additive error of *Algorithm ApproxSqrt*. In more detail, we will show that any randomized one-pass algorithm that approximates the length of the longest palindrome up to an additive error of $\varepsilon\sqrt{n}$ must use $\Omega(\sqrt{n}/\varepsilon)$ space.

2 Model and Definitions

Let $S \in \Sigma^n$ denote the input stream of length n over an alphabet Σ[3]. For simplicity we assume symbols to be positive integers, i.e., $\Sigma \subset \mathbb{N}$. We define $S[i]$ as the symbol at index i and $S[i,j] = S[i], S[i+1], \dots S[j]$. In this paper we use the streaming model: In one *pass* the algorithm goes over the whole input stream S, reading $S[i]$ in *iteration* i of the pass. In this paper we assume that the algorithm has a memory of size $o(n)$, but the output space is unlimited. We use the so-called word model where the space equals the number of $O(\log(n))$ registers (See [2]).

S contains an *odd palindrome* of length ℓ with midpoint $m \in \{\ell, \dots, n-\ell\}$ if $S[m-i] = S[m+i]$ for all $i \in \{1, \dots, \ell\}$. Similarly, S contains an *even palindrome* of length ℓ if $S[m-i+1] = S[m+i]$ for all $i \in \{1, \dots, \ell\}$. In other words, a palindrome is odd if and only if its length is odd. For simplicity, our algorithms assume palindromes to be even – it is easy to adjust our results for finding odd palindromes by apply the algorithm to $S[1]S[1]S[2]S[2]\cdots S[n]S[n]$ instead of $S[1,n]$.

The maximal palindrome (the palindrome of maximal length) in $S[1,i]$ with midpoint m is called $P[m,i]$ and the maximal palindrome in S with midpoint m is called $P[m]$ which equals $P[m,n]$. We define $\ell(m,i)$ as the maximum length of the palindrome with midpoint m in the substring $S[1,i]$. The maximal length of the palindrome in S with midpoint m is denoted by $\ell(m)$. Moreover, for $z \in \mathbb{Z} \setminus \{1, \dots, n\}$ we define $\ell(z) = 0$. Furthermore, for $\ell^* \in \mathbb{N}$ we define $P[m]$ to be an ℓ^*-palindrome if $\ell(m) \geq \ell^*$. Throughout this paper, $\tilde{\ell}()$ refers to an estimate of $\ell()$.

We use the KR-Fingerprint, which was first defined by Karp and Rabin [4] to compress strings and was later used in the streaming pattern matching problem (see [6], [3], and [2]). For a string S' we define the forward fingerprint (similar to [2]) and its reverse as follows. $\phi^F_{r,p}(S') = \left(\sum_{i=1}^{|S'|} S'[i] \cdot r^i\right) \mod p$ $\phi^R_{r,p}(S') = \left(\sum_{i=1}^{|S'|} S'[i] \cdot r^{l-i+1}\right) \mod p, <$ where p is an arbitrary prime number in $[n^4, n^5]$ and r is randomly chosen from $\{1, \dots, p\}$. We write ϕ^F (ϕ^R respectively) as opposed to $\phi^F_{r,p}$($\phi^R_{r,p}$ respectively) whenever r and p are fixed. We define for $1 \leq i \leq j \leq n$ the fingerprint $F^F(i,j)$ as the fingerprint of $S[i,j]$, i.e., $F^F(i,j) = \phi^F(S[i,j]) = r^{-(i-1)}(\phi^F(S[1,j]) - \phi^F(S[1,i-1])) \mod p$. Similarly, $F^R(i,j) = \phi^R(S[i,j]) = \phi^R(S[1,j]) - r^{j-i+1} \cdot \phi^R(S[1,i-1]) \mod p$. For every $1 \leq i \leq n - \sqrt{n}$ the

[3] All soundness, space, and time complexity analyses assumes $|\Sigma|$ to be polynomial. One can use a proper random hash function for bigger alphabets.

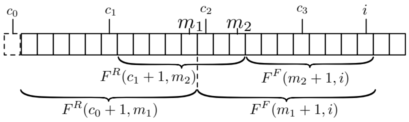

Figure 1 At iteration i two midpoints m_1 and m_2 are checked. Corresponding substrings are denoted by brackets. Note, the distance from c_0 to m_1 equals the distance from m_1 to i. Similarly, the distance from c_1 to m_2 equals the distance from m_2 to i.

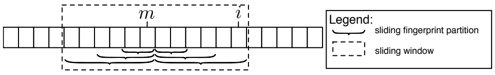

Figure 2 Illustration of the *Fingerprint Pairs* after iteration i of algorithm with $\sqrt{n} = 6$, $\varepsilon = 1/3$, and $m = i - \sqrt{n}$.

fingerprints $F^F(1, i-1-\sqrt{n})$ and $F^R(1, i-1-\sqrt{n})$ are called *Master Fingerprints*. Note that it is easy to obtain $F^F(i, j+1)$ by adding the term $S[j+1]r^{j+1}$ to $F^F(i, j)$. Similarly, we obtain $F^F(i+1, j)$ by subtracting $S[i]$ from $r^{-1} \cdot F^F(i, j)$. The authors of [2] observe useful properties which we state in the full version.

3 Algorithm Simple ApproxSqrt

In this section, we introduce a simple one-pass algorithm which reports all midpoints and length estimates of palindromes in S. Throughout this paper we use i to denote the current index which the algorithm reads. *Simple ApproxSqrt* keeps the last $2\sqrt{n}$ symbols of $S[1, i]$ in the memory.

It is easy to determine the exact length palindromes of length less than $\sqrt{n}$ since any such palindrome is fully contained in memory at some point. However, in order to achieve a better time bound the algorithm only *approximates* the length of short palindromes. It is more complicated to estimate the length of a palindrome with a length of at least $\sqrt{n}$. However, *Simple ApproxSqrt* detects that its length is at least $\sqrt{n}$ and stores it as an R_S-entry (introduced later) in a list L_i. The R_S-entry contains the midpoint as well as a length estimate of the palindrome, which is updated as i increases.

In order to estimate the lengths of the long palindromes the algorithm designates certain indices of S as *checkpoints*. For every checkpoint c the algorithm stores a fingerprint $F^R(1, c)$ enabling the algorithm to do the following. For every midpoint m of a long palindrome: Whenever the distance from a checkpoint c to m (c occurs before m) equals the distance from m to i, the algorithm compares the substring from c to m to the reverse of the substring from m to i by using fingerprints. We refer to this operation as *checking* $P[m]$ against checkpoint c. If $S[c+1, m]^R = S[m+1, i]$, then we say that $P[m]$ was *sucessfully checked* with c and the algorithm updates the length estimate for $P[m]$, $\tilde{\ell}(m)$. The next time the algorithm possibly updates $\tilde{\ell}(m)$ is after d iterations where d equals the distance between checkpoints. This distance d gives the additive approximation. See Figure 1 for an illustration.

We need the following definitions before we state the algorithm: For $k \in \mathbb{N}$ with $0 \leq k \leq \lfloor \frac{\sqrt{n}}{\varepsilon} \rfloor$ checkpoint c_k is the index at position $k \cdot \lfloor \varepsilon\sqrt{n} \rfloor$ thus checkpoints are $\lfloor \varepsilon\sqrt{n} \rfloor$ indices apart. Whenever we say that an algorithm stores a checkpoint, this means storing the data belonging to this checkpoint. Additionally, the algorithm stores *Fingerprint Pairs*, fingerprints of size $\lfloor \varepsilon\sqrt{n} \rfloor, 2\lfloor \varepsilon\sqrt{n} \rfloor, \ldots$ starting or ending in the middle of the sliding window. In the following, we first describe the data that the algorithm has in its memory after reading $S[1, i-1]$, then we describe the algorithm itself. Let $R_S(m, i)$ denote the representation of $P[m]$ which is stored at time i. As opposed to storing $P[m]$ directly, the algorithm stores m, $\tilde{\ell}(m, i)$, $F^F(1, m)$, and $F^R(1, m)$.

Memory invariants. Just before algorithm *Simple ApproxSqrt* reads $S[i]$ it has stored the following information. Note that, for ease of referencing, during an iteration i data structures are indexed with the iteration number i.
That is, for instance, L_{i-1} is called L_i after $S[i]$ is read.

1. The contents of the sliding window $S[i-2\sqrt{n}-1, i-1]$.
2. The two *Master Fingerprints* $F^F(1, i-1)$ and $F^R(1, i-1)$.
3. A list of *Fingerprint Pairs*: Let r be the maximum integer s.t. $r \cdot \lfloor \varepsilon\sqrt{n} \rfloor < \sqrt{n}$. For $j \in \{\lfloor \varepsilon\sqrt{n} \rfloor, 2 \cdot \lfloor \varepsilon\sqrt{n} \rfloor, \dots, r \cdot \lfloor \varepsilon\sqrt{n} \rfloor, \sqrt{n}\}$ the algorithm stores the pair $F^R((i-\sqrt{n})-j, (i-\sqrt{n})-1)$, and $F^F(i-\sqrt{n}, (i-\sqrt{n})+j-1)$. See Figure 2 for an illustration.
4. A list CL_{i-1} which consists of all fingerprints of prefixes of S ending at already seen checkpoints, i.e., $\mathrm{CL}_{i-1} = \left[F^R(1, c_1), F^R(1, c_2), \dots, F^R\left(1, c_{\lfloor (i-1)/\lfloor \varepsilon\sqrt{n} \rfloor \rfloor}\right)\right]$
5. A list L_{i-1} containing representation of all $\sqrt{n}$-palindromes with a midpoint located in $S[1, (i-1)-\sqrt{n}]$. The j^{th} entry of L_{i-1} has the form $R_S(m_j, i-1) = (m_j, \tilde{\ell}(m_j, i-1), F^F(1, m_j), F^R(1, m_j))$ where

   (a m_j is the midpoint of the j^{th} palindrome in $S[1, (i-1)-\sqrt{n}]$ with a length of at least $\sqrt{n}$. Therefore, $m_j < m_{j+1}$ for $1 \le j \le |L_{i-1}| - 1$.

   (b $\tilde{\ell}(m_j, i-1)$ is the current estimate of $\ell(m_j, i-1)$.

In the following, we explain how the algorithm maintains the above invariants.

Maintenance. At iteration i the algorithm performs the following steps. It is implicit that L_{i-1} and $CLi-1$ become L_i and CL_i respectively.

1. Read $S[i]$, set $m = i - \sqrt{n}$. Update the sliding window to $S[m-\sqrt{n}, i] = S[i-2\sqrt{n}, i]$
2. Update the *Master Fingerprints* to be $F^F(1, i)$ and $F^R(1, i)$.
3. If i is a checkpoint (i.e., a multiple of $\lfloor \varepsilon\sqrt{n} \rfloor$), then add $F^R(1, i)$ to CL_i.
4. Update all *Fingerprint Pairs*: For $j \in \{\lfloor \varepsilon\sqrt{n} \rfloor, 2 \cdot \lfloor \varepsilon\sqrt{n} \rfloor, , \dots, r \cdot \lfloor \varepsilon\sqrt{n} \rfloor, \sqrt{n}\}$
   - Update $F^R(m-j, m-1)$ to $F^R(m-j+1, m)$ and $F^F(m, m+j-1)$ to $F^F(m+1, m+j)$.
   - If $F^R(m-j+1, m) = F^F(m+1, m+j)$, then set $\tilde{\ell}(m, i) = j$.
   - If $\tilde{\ell}(m, i) < \sqrt{n}$, output m and $\tilde{\ell}(m, i)$.
5. If $\tilde{\ell}(m, i) \ge \sqrt{n}$, add *add* $R_S(m, i)$ to L_i:
   $L_i = L_i \ \circ \ (m, \tilde{\ell}(m, i), F^F(1, m), F^R(1, m))$.
6. For all c_k with $1 \le k \le \lfloor \frac{i}{\lfloor \varepsilon\sqrt{n} \rfloor} \rfloor$ and $R_S(m_j, i) \in L_i$ with $i - m_j = m_j - c_k$, check if $\tilde{\ell}(m_j, i)$ can be updated:
   - If the left side of m_j is the reverse of the right side of m_j (i.e., $F^R(c_k+1, m_j) = F^F(m_j+1, i)$) then update $R_S(m_j, i)$ by updating $\tilde{\ell}(m_j, i)$ to $i - m_j$.
7. If $i = n$, then report L_n.

In all proofs in this paper which hold w.h.p. we assume that fingerprints do not fail as we take less than n^2 fingerprints and by using the following Lemma, the probability that a fingerprint fails is at most $1/n^{c+2}$.

▶ **Lemma 6.** *(Theorem 1 of [2]) For two arbitrary strings s and s' with $s \neq s'$ the probability that $\phi^F(s) = \phi^F(s')$ is smaller than $1/n^{c+2}$ for some $c \in \mathbb{N}$.*

Thus, by applying the union bound the probability that no fingerprint fails is at least $1-n^{-c}$. The following lemma shows that the *Simple ApproxSqrt* finds all palindromes along with the estimates as stated in Theorem 1. *Simple ApproxSqrt* does not fulfill the time and space bounds of Theorem 1; we will later show how to improve its efficiency. The proof can be found in the full version.

▶ **Lemma 7.** *For any ε in $[1/\sqrt{n}, 1]$ $\text{ApproxSqrt}(S, \varepsilon)$ reports for every palindrome $P[m]$ in S its midpoint m as well as an estimate $\tilde{\ell}(m)$ such that w.h.p. $\ell(m) - \varepsilon\sqrt{n} < \tilde{\ell}(m) \leq \ell(m)$.*

4 A space-efficient version

In this section, we show how to modify *Simple ApproxSqrt* so that it matches the time and space requirements of Theorem 1. The main idea of the space improvement is to store the lists L_i in a compressed form.

Compression. It is possible in the simple algorithm for L_i to have linear length. In such cases S contains many overlapping palindromes which show a certain *periodic* pattern as shown in Corollary 12, which our algorithm exploits to compress the entries of L_i. This idea was first introduced in [6], and is used in [3], and [2]. More specifically, our technique is a modification of the compression in [2]. In the following, we give some definitions in order to show how to compress the list. First we define a *run* which is a sequence of midpoints of overlapping palindromes.

▶ **Definition 8** (ℓ^*−Run)**.** Let ℓ^* be an arbitrary integer and $h \geq 3$. Let $m_1, m_2, m_3, \ldots, m_h$ be consecutive midpoints of ℓ^*-palindromes in S. $m_1, \ldots, m_h$ form an *ℓ^*-run* if $m_{j+1} - m_j \leq \ell^*/2$ for all $j \in \{1, \ldots, h-1\}$.

In Corollary 12 we show that $m_2 - m_1 = m_3 - m_2 = \cdots = m_h - m_{h-1}$. We say that a run is maximal if the run cannot be extended by other palindromes. More formally:

▶ **Definition 9** (Maximal ℓ^*−Run)**.** An ℓ^*-run over $m_1, \ldots, m_h$ is *maximal* it satisfies both of the following: i) $\ell(m_1 - (m_2 - m_1)) < \ell^*$, ii) $\ell(m_h + (m_2 - m_1)) < \ell^*$.

Simple ApproxSqrt stores palindromes explicitly in L_i, i.e., $L_i = [R_S(m_1, i); \ldots; R_S(m_{|L_i|}, i)]$ where $R_S(m_j, i) = (m_j, \tilde{\ell}(m_j, i), F^F(1, m_j), F^R(1, m_j))$, for all $j \in \{1, 2, \ldots, h\}$. The improved *Algorithm ApproxSqrt* stores these midpoints in a compressed way in list $\hat{L}_i$. *ApproxSqrt* distinguishes among three cases: Those palindromes which

1. are not part of a $\sqrt{n}$-run are stored explicitly as before. We call them R_S-entries. Let $P[m, i]$ be such a palindrome. After iteration i the algorithm stores $R_S(m, i)$.
2. form a maximal $\sqrt{n}$-run are stored in a data structure called R_F-entry. Let $m_1, \ldots, m_h$ be the midpoints of a maximal $\sqrt{n}$-run. The data structure stores the following information.
   - m_1, $m_2 - m_1$, h, $\tilde{\ell}(m_1, i)$, $\tilde{\ell}(m_{\lfloor \frac{1+h}{2} \rfloor}, i)$, $\tilde{\ell}(m_{\lceil \frac{1+h}{2} \rceil}, i)$, $\tilde{\ell}(m_h, i)$,
   - $F^F(1, m_1)$, $F^R(1, m_1)$, $F^F(m_1 + 1, m_2)$, $F^R(m_1 + 1, m_2)$
3. form a $\sqrt{n}$-run which is not maximal (i.e., it can possibly be extended) in a data structure called R_{NF}-entry. The information stored in an R_{NF}-entry is the same as in an R_F-entry, but it does not contain the entries: $\tilde{\ell}(m_{\lfloor \frac{1+h}{2} \rfloor}, i)$, $\tilde{\ell}(m_{\lceil \frac{1+h}{2} \rceil}, i)$, and $\tilde{\ell}(m_h, i)$.

The algorithm stores only the estimate (of the length) and the midpoint of the following palindromes explicitly.

- $P[m]$ for an R_S-entry (Therefore all palindromes which are not part of a $\sqrt{n}$-run)
- $P[m_1]$, $P[m_{\lfloor (h+1)/2 \rfloor}]$, $P[m_{\lceil (h+1)/2 \rceil}]$, and $P[m_h]$ for an R_F-entry
- $P[m_1]$ for an R_{NF}-entry.

In what follows we refer to the above listed palindromes as *explicitly stored* palindromes. We argue in Observation 15 that in any interval of length $\sqrt{n}$ the number of *explicitly stored* palindromes is bounded by a constant.

4.1 Algorithm ApproxSqrt

In this subsection, we describe some modifications of *Simple ApproxSqrt* in order to obtain a space complexity of $O(\frac{\sqrt{n}}{\varepsilon})$ and a total running time of $O(\frac{n}{\varepsilon})$. *ApproxSqrt* is the same as *Simple ApproxSqrt*, but it compresses the stored palindromes. *ApproxSqrt* uses the same memory invariants as *Simple ApproxSqrt*, but it uses $\hat{L}_i$ as opposed to L_i.

ApproxSqrt uses the first four steps of *Simple ApproxSqrt*. Step 5, Step 6, and Step 7 are replaced. The modified Step 5 ensures that there are at most two R_S-entries per interval of length $\sqrt{n}$. Moreover, Step 6 is adjusted since *ApproxSqrt* stores only the length estimate of explicitly stored palindromes.

5. If $\tilde{\ell}(m,i) \geq \sqrt{n}$, obtain $\hat{L}_i$ by adding the palindrome with midpoint $m(= i - \sqrt{n})$ to $\hat{L}_{i-1}$ as follows:
   a. The last element in $\hat{L}_i$ is the following R_{NF}-entry $\big(m_1, m_2 - m_1, h, \tilde{\ell}(m_1,i), F^F(1,m_1), F^R(1,m_1), F^F(m_1+1,m_2), F^R(m_1+1,m_2)\big)$.
      i. If the palindrome can be added to this run, i.e., $m = m_1 + h(m_2 - m_1)$, then we increment the h in the R_{NF}-entry by 1.
      ii. If the palindrome cannot be added: Store $P[m,i]$ as an R_S-entry: $\hat{L}_i = \hat{L}_i \circ (m, \tilde{\ell}(m,i), F^F(1,m), F^R(1,m))$. Moreover, convert the R_{NF}-entry into the R_F-entry by adding $\tilde{\ell}(m_{\lfloor \frac{1+h}{2} \rfloor}, i)$, $\tilde{\ell}(m_{\lceil \frac{1+h}{2} \rceil}, i)$ and $\tilde{\ell}(m_h, i)$: First we calculate $m_{\lfloor \frac{1+h}{2} \rfloor} = m_1 + \left(\lfloor \frac{1+h}{2} - 1 \rfloor\right)(m_2 - m_1)$. One can calculate $m_{\lceil \frac{1+h}{2} \rceil}$ similarly. For $m' \in \{m_{\lfloor \frac{1+h}{2} \rfloor}, m_{\lceil \frac{1+h}{2} \rceil}, m_h\}$ calculate $\tilde{\ell}(m',i) = \max_{i-2\sqrt{n} \leq j \leq i} \{j - m' \mid \exists c_k \text{ with } j - m' = m' - c_k \text{ and } F^R(c_k+1, m') = F^F(m'+1, j)\}$.
   b. The last two entries in $\hat{L}_i$ are stored as R_S-entries and together with $P[m,i]$ form a $\sqrt{n}$-run. Then remove the entries of the two palindromes out of $\hat{L}_{i-1}$ and add a new R_{NF}-entry with all three palindromes to $\hat{L}_{i-1}$:
      $m_1, \tilde{\ell}(m_1,i), F^F(1,m_1), F^F(1,m_2), F^R(1,m_1), F^R(1,m_2), m_2 - m_1, h = 3$. Retrieve $F^F(m_1+1,m_2)$ and $F^R(m_1+1,m_2)$.
   c. Otherwise, store $P[m,i]$ as an R_S-entry: $\hat{L}_i = \hat{L}_i \circ (m, \tilde{\ell}(m,i), F^F(1,m), F^R(1,m))$
6. This step is similar to step 6 of *Simple ApproxSqrt* the only difference is that we check only for explicitly stored palindromes if they can be extended outwards. [4]
7. If $i = n$. If the last element in $\hat{L}_i$ is an R_{NF}-entry, then convert it into an R_F-entry as in 5(a)ii. Report L_n.

[4] This step is only important for the running time.

4.2 Structural Properties

In this subsection, we prove structural properties of palindromes. These properties allow us to compress (by using R_S-entries and R_F-entries) overlapping palindromes $P[m_1], \dots, P[m_h]$ in such a way that at any iteration i all the information stored $R_S(m_1, i), \dots, R_S(m_h, i)$ is available. The structural properties imply, informally speaking, that the palindromes are either far from each other, leading to a small number of them, or they are overlapping and it is possible to compress them. Lemma 11 shows this structure for short intervals containing at least three palindromes. Corollary 12 shows a similar structure for palindromes of a run which is used by *ApproxSqrt*. We first give the common definition of periodicity.

▶ **Definition 10** (*period*)**.** A string S' is said to have period p if it consists of repetitions of a block of p symbols. Formally, S' has period p if $S'[j] = S[j+p]$ for all $j = 1, \dots, |S'| - p$. [5]

▶ **Lemma 11.** *Let $m_1 < m_2 < m_3 < \cdots < m_h$ be indices in S that are consecutive midpoints of ℓ^*-palindromes for an arbitrary natural number ℓ^*. If $m_h - m_1 \leq \ell^*$, then*

(a) *$m_1, m_2, m_3, \dots, m_h$ are equally spaced in S, i.e., $|m_2 - m_1| = |m_{k+1} - m_k|$ $\forall k \in \{1, \dots, h-1\}$*

(b) $S[m_1+1, m_h] = \begin{cases} (ww^R)^{\frac{h-1}{2}} & h \textit{ is odd} \\ (ww^R)^{\frac{h-2}{2}} w & h \textit{ is even} \end{cases}$ *, where $w = S[m_1+1, m_2]$.*

Proof. Given $m_1, m_2, \dots, m_h$ and ℓ^* we prove the following stronger claim by induction over the midpoints $\{m_1, \dots, m_j\}$. (a') $m_1, m_2, \dots, m_j$ are equally spaced. (b') $S[m_1+1, m_j+\ell^*]$ is a prefix of ww^Rww^R... .

Base case $j = 2$: Since we assume m_1 is the midpoint of an ℓ^*-palindrome and $\ell^* \geq m_h - m_1 \geq m_2 - m_1 = |w|$, we have that $S[m_1 - |w| + 1, m_1] = w^R$. Recall that $\ell(m_2) \geq \ell^* \geq |w|$ and thus, $S[m_1+1, m_2+|w|] = ww^R$.

We can continue this argument and derive that $S[m_1+1, m_2+\ell^*]$ is a prefix of $ww^R \dots ww^R$. (a') for $j = 2$ holds trivially.

Inductive step $j-1 \to j$: Assume (a') and (b') hold up to m_{j-1}. We first argue that $|m_j - m_1|$ is a multiple of $|m_2 - m_1| = |w|$. Suppose $m_j = m_1 + |w| \cdot q + r$ for some integers $q \geq 0$ and $r \in \{1, \dots, |w|-1\}$. Since $m_j \leq m_{j-1} + \ell^*$, the interval $[m_1+1, m_{j-1}+\ell^*]$ contains m_j. Therefore, by inductive hypothesis, $m_j - r$ is an index where either w or w^R starts. This implies that the prefix of ww^R(or w^Rw) of size $2r$ is a palindrome and the string ww^R(or w^Rw) has period $2r$. On the other hand, by consecutiveness assumption, there is no midpoint of an ℓ^*-palindrome in the interval $[m_1+1, m_2-1]$. does not have a period of $2p$, a contradiction. We derive that $m_j - m_1$ is multiple of $|w|$.

Hence, we assume $m_j = m_{j-1} + q \cdot |w|$ for some q. The assumption that m_j is a midpoint of an ℓ^*-palindrome beside the inductive hypothesis implies (b') for j. The structure of $S[m_{j-1}+|w|-\ell^*+1, m_{j-1}+|w|+\ell^*]$ shows that $m_{j-1}+|w|$ is a midpoint of an ℓ^*-palindrome. This means that $m_j = m_{j-1} + |w|$. This gives (a') and yields the induction step. ◀

Corollary 12 shows the structure of overlapping palindromes and is essential for the compression. The main difference between Corollary 12 and Lemma 11 is the required distance between the midpoints of a run. Lemma 11 assumes that every palindrome in the run overlaps with all other palindromes. In contrast, Corollary 12 assumes that every palindrome $P[m_j]$ overlaps with $P[m_{j-2}]$, $P[m_{j-1}]$, $P[m_{j+1}]$, and $P[m_{j+2}]$. It can be proven by an induction over the midpoints and using Lemma 11. The proof is in the full version.

[5] Here, p is called a period for S' even if $p > |S'|/2$.

▶ **Corollary 12.** *If $m_1, m_2, \ldots, m_h$ form an ℓ^*-run for an arbitrary natural number ℓ^* then*

(a) *$m_1, m_2, m_3, \ldots, m_h$ are equally spaced in S, i.e., $|m_2 - m_1| = |m_{k+1} - m_k|$ $\forall k \in \{1, \ldots, h-1\}$*

(b) $S[m_1 + 1, m_h] = \begin{cases} (ww^R)^{\frac{h-1}{2}} & h \textit{ is odd} \\ (ww^R)^{\frac{h-2}{2}} w & h \textit{ is even} \end{cases}$ *, where $w = S[m_1 + 1, m_2]$.*

Lemma 13 shows the pattern for the lengths of the palindromes in each half of the run. This allows us to only store a constant number of length estimates per run. The proof can be found in the full version.

▶ **Lemma 13.** *At iteration i, let $m_1, m_2, m_3, ..., m_h$ be midpoints of a maximal ℓ^*-run in $S[1, i]$ for an arbitrary natural number ℓ^*. For any midpoint m_j, we have:*

$$\ell(m_j, i) = \begin{cases} \ell(m_1, i) + (j-1) \cdot (m_2 - m_1) & j < \frac{h+1}{2} \\ \ell(m_h, i) + (h-j) \cdot (m_2 - m_1) & j > \frac{h+1}{2} \end{cases}$$

4.3 Analysis of the Algorithm

We show that one can convert R_S-entries into a run and vice versa and *ApproxSqrt*'s maintenance of R_F-entries and R_{NF}-entries does not impair the length estimates. The following lemma shows that one can retrieve the length estimate of a palindrome as well as its fingerprint from an R_F-entry.

▶ **Lemma 14.** *At iteration i, the R_F-entry over $m_1, m_2, \ldots, m_h$ is a lossless compression of $[R_S(m_1, i); \ldots; R_S(m_h, i)]$*

Let *Compressed Run* be the general term for R_F-entry and R_{NF}-entry. We argue that in any interval of length $\sqrt{n}$ we only need to store at most two single palindromes and two *Compressed Runs*. Suppose there were three R_S-entries, then, by Corollary 12, they form a $\sqrt{n}$-run since they overlap each other. Therefore, the three R_S-entries would be stored in a *Compressed Run*. For a similar reason there cannot be more than two *Compressed Runs* in one interval of length $\sqrt{n}$. We derive the following observation.

▶ Observation 15. For any interval of length $\sqrt{n}$ there can be at most two R_S-entries and two *Compressed Runs* in L^*.

We now have what we need in order to prove Theorem 1; the proof is given in the full version.

5 Algorithm Exact

This section describes *Algorithm Exact* which determines the exact length of the longest palindrome in S using $O(\sqrt{n})$ space and two passes over S.
For the first pass this algorithm runs *ApproxSqrt* $(S, \frac{1}{2})$ (meaning that $\varepsilon = 1/2$) and the variant of *ApproxSqrt* described in Lemma 5 simultaneously. The first pass returns ℓ_{max} (Lemma 5) if $\ell_{max} < \sqrt{n}$. Otherwise, the first pass (Theorem 1) returns for every palindrome $P[m]$, with $\ell(m) \geq \sqrt{n}$, an estimate satisfying $\ell(m) - \sqrt{n}/2 < \tilde{\ell}(m) \leq \ell(m)$ w.h.p..

The algorithm for the second pass is determined by the outcome of the first pass. For the case $\ell_{max} < \sqrt{n}$, it uses the sliding window to find all $P[m]$ with $\ell(m) = \ell_{max}$. If $\ell_{max} \geq \sqrt{n}$, then the first pass only returns an additive $\sqrt{n}/2$-approximation of the palindrome lengths. We define the *uncertain intervals* of $P[m]$ to be: $I_1(m) = S[m - \tilde{\ell}(m) - \sqrt{n}/2 + 1, m - \tilde{\ell}(m)]$ and $I_2(m) = S[m + \tilde{\ell}(m) + 1, m + \tilde{\ell}(m) + \sqrt{n}/2]$.

The algorithm uses the length estimate calculated in the first pass to delete all R_S-entries (Step 3) which cannot be the longest palindromes. Similarly, the algorithm (Step 2) only keeps the middle entries of R_F-entries since these are the longest palindromes of their run (A proof can be found in the full version). In the second pass, *Algorithm Exact* stores $I_1(m)$ for a palindrome $P[m]$ if it was not deleted. *Algorithm Exact* compares the symbols of $I_1(m)$ symbol by symbol to $I_2(m)$ until the first mismatch is found. Then the algorithm knows the exact length $\ell(m)$ and discards $I_1(m)$. The analysis will show, at any time the number of stored uncertain intervals is bounded by a constant.

First Pass. Run the following two algorithms simultaneously:

1. *ApproxSqrt* $(S, {}^1\!/_2)$. Let L be the returned list.
2. Variant of *ApproxSqrt* (See Lemma 5) which reports ℓ_{max} if $\ell_{max} < \sqrt{n}$.

Second Pass

- $\ell_{max} < \sqrt{n}$: Use a sliding window of size $2\sqrt{n}$ and maintain two fingerprints $F^R[i - \sqrt{n} - \ell_{max} + 1, i - \sqrt{n}]$, and $F^F[i - \sqrt{n} + 1, i - \sqrt{n} + \ell_{max}]$. Whenever these fingerprints match, report $P[i - \sqrt{n}]$.
- $\ell_{max} \geq \sqrt{n}$: In this case, the algorithm uses a preprocessing phase first.

Preprocessing

1. Set $\tilde{\ell}_{max} = \max\{\tilde{\ell}(m) \mid P[m] \text{ is stored in } L \text{ as an } R_F \text{ or an } R_S \text{ entry}\}$.
2. For every R_F-entry R_F in L with midpoints $m_1, \ldots, m_h$ remove R_F from L and add $R_s(m,i) = (m, \tilde{\ell}(m), F^F(1,m), F^R(1,m))$ to L, for $m \in \{m_{\lfloor (h+1)/2 \rfloor}, m_{\lceil (h+1)/2 \rceil}\}$. To do this, calculate $m_{\lfloor \frac{1+h}{2} \rfloor} = m_1 + (\lfloor \frac{1+h}{2} \rfloor - 1)(m_2 - m_1)$ and $m_{\lceil \frac{1+h}{2} \rceil} = m_1 + (\lceil \frac{1+h}{2} \rceil - 1)(m_2 - m_1)$. Retrieve $F^F(1,m)$ and $F^R(1,m)$ for $m \in \{m_{\lfloor (h+1)/2 \rfloor}, m_{\lceil (h+1)/2 \rceil}\}$.
3. Delete all R_S-entries $(m_k, \tilde{\ell}(m_k), F^F(1,m_k), F^R(1,m_k))$ with $\tilde{\ell}(m_k) \leq \tilde{\ell}_{max} - {}^{\sqrt{n}}\!/_2$ from L.
4. For every palindrome $P[m] \in L$ set $I_1(m) := (m - \tilde{\ell}(m) - {}^1\!/_2\sqrt{n}, m - \tilde{\ell}(m)]$ and set $finished(m) :=$ false.

The resulting list is called L^*.

String processing. At iteration i the algorithm performs the following steps.

1. Read $S[i]$. If there is a palindrome $P[m]$ such that $i \in I_1(m)$, then store $S[i]$.
2. If there is a midpoint m such that $m + \tilde{\ell}(m) < i < m + \tilde{\ell}(m) + \frac{\sqrt{n}}{2}$, finished$(m)$ = false, and $S[m - (i - m) + 1] \neq S[i]$, then set finished$(m)$:= true and $\ell(m) = i - m - 1$.
3. If there is a palindrome $P[m]$ such that $i \geq \tilde{\ell}(m) + m + \frac{\sqrt{n}}{2}$, then discard $I_1(m)$.
4. If $i = n$, then output $\ell(m)$ and m of all $P[m]$ in L^* with $\ell(m) = \ell_{max}$.

We analyse *Exact* in the full version.

6 Algorithm ApproxLog

In this section, we present an algorithm which reports one of the longest palindromes and uses only logarithmic space. *ApproxLog* has a multiplicative error instead of an additive error term. Similar to *ApproxSqrt* we have special indices of S designated as checkpoints that we keep along with some constant size data in memory. The checkpoints are used to estimate the length of palindromes. However, this time checkpoints (and their data) are only stored for a limited time. Since we move from additive to multiplicative error we do not need checkpoints to be spread evenly in S. At iteration i, the number of checkpoints in any interval of fixed length decreases exponentially with distance to i. The algorithm stores a palindrome $P[m]$ (as an R_S-entry or R_{NF}-entry) until there is a checkpoint c such that $P[m]$ was checked unsuccessfully against c. A palindrome is stored in the lists belonging to the last checkpoint with which is was checked successfully. In what follows we set $\delta \triangleq \sqrt{1+\varepsilon} - 1$ for the ease of notation. Every checkpoint c has an attribute called $level(c)$. It is used to determine the number of iterations the checkpoint data remains in the memory.

Memory invariants. After algorithm *ApproxLog* has processed $S[1, i-1]$ and before reading $S[i]$ it contains the following information:

1. Two *Master Fingerprints* up to index $i-1$, i.e., $F^F(1, i-1)$ and $F^R(1, i-1)$.
2. A list of checkpoints CL_{i-1}. For every $c \in \mathrm{CL}_{i-1}$ we have
   - $level(c)$ such that c is in CL_{i-1} iff $c \geq (i-1) - 2(1+\delta)^{level(c)}$.
   - $fingerprint(c) = F^R(1, c)$
   - a list L_c. It contains all palindromes which were successfully checked with c, but with no other checkpoint $c' < c$. The palindromes in L_c are either R_S-entries or R_{NF}-entries (See *Algorithm ApproxSqrt*).
3. The midpoint m^*_{i-1} and the length estimate $\tilde{\ell}(m^*_{i-1}, i-1)$ of the longest palindrome found so far.

The algorithm maintains the following property. If $P[m, i]$ was successfully checked with checkpoint c but with no other checkpoint $c' < c$, then the palindrome is stored in L_c. The elements in L_c are ordered in increasing order of their midpoint. The algorithm stores palindromes as R_S-entries and R_{NF}-entries. This time however, the length estimates are not maintained. Adding a palindrome to a current run works exactly (the length estimate is not calculated) as described in *Algorithm ApproxSqrt*.

Maintenance. At iteration i the algorithm performs the following steps.

1. Read S[i]. Update the *Master Fingerprints* to be $F^F(1, i)$ and $F^R(1, i)$.
2. For all $k \geq k_0 = {}^{\log(1/\delta)}/_{\log(1+\delta)}$(The algorithm does not maintain intervals of size 0.)
   a. If i is a multiple of $\lfloor \delta(1+\delta)^{k-2} \rfloor$, then add the checkpoint $c = i$ (along with the checkpoint data) to CL_i. Set $level(c) = k$, $fingerprint(c) = F^R(1, i)$ and $L_c = \emptyset$.
   b. If there exists a checkpoint c with $level(c) = k$ and $c < i - 2(1+\delta)^k$, then prepend L_c to $L_{c'}$ where $c' = max\{c'' \mid c'' \in \mathrm{CL}_i \ and \ c'' > c\}$. Merge and create runs in L_c if necessary (Similar to step 5 of *ApproxSqrt*). Delete c and its data from CL_i.
3. For every checkpoint $c \in \mathrm{CL}_i$
   a. Let m_c be the midpoint of the first entry in L_c and $c' = max\{c'' \mid c'' \in \mathrm{CL}_i \ and \ c'' < c\}$. If $i - m_c = m_c - c'$, then we *check* $P[m]$ against c' by doing the following:
      i. If the left side of m_c is the reverse of the right side of m_c (i.e., $F^R(c', m_c) = F^F(m_c, i)$) then move $P[m_c]$ from L_c to $L_{c'}$ by adding $P[m_c]$ to $L_{c'}$:
         A. If $|L_{c'}| \leq 1$, store $P[m_c]$ as a R_S-entry.
         B. If $|L_{c'}| = 2$, create a run out of the R_S-entries stored in $L_{c'}$ and $P[m_c]$.
         C. Otherwise, add $P[m_c]$ to the R_{NF}-entry in $L_{c'}$.
      ii. If the left side of m_c is *not* the reverse of the right side of m_c, then remove m_c from L_c.
      iii. If $i - m_c > \tilde{\ell}(m^*_i)$, then set $m^*_i = m_c$ and set $\tilde{\ell}(m^*_i) = i - m_c$.
4. If $i = n$, then report m^*_i and $\tilde{\ell}(m^*_i)$.

6.1 Analysis

ApproxLog relies heavily on the interaction of the following two ideas. The pattern of the checkpointing and the compression which is possible due to the properties of overlapping palindromes (Lemma 11). On the one hand the checkpoints are close enough so that the length estimates are accurate (Lemma 19). The closeness of the checkpoints ensures that palindromes which are stored at a checkpoint form a run (Lemma 18) and therefore can be stored in constant space. On the other hand the checkpoints are far enough apart so that the number of checkpoints and therefore the required space is logarithmic in n.

We start off with an observation to characterise the checkpointing. Step 2 of the algorithm creates a checkpoint pattern: Recall that the level of a checkpoint is determined when the checkpoint and its data are added to the memory. The checkpoints of every level have the same distance. A checkpoint (along with its data) is removed if its distance to i exceeds a threshold which depends on the level of the checkpoint. Note that one index of S can belong to different levels and might therefore be stored several times. The following observation follows from Step 2 of the algorithm.

▶ Observation 16. At iteration i, $\forall k \geq k_0 = \left\lceil \frac{log(\frac{(1+\delta)^2}{\delta})}{log(1+\delta)} \right\rceil$. Let $C_{i,k} = \{c \in \mathrm{CL}_i \mid level(c) = k\}$.

1. $C_{i,k} \subseteq [i - 2(1+\delta)^k, i]$.
2. The distance between two consecutive checkpoints of $C_{i,k}$ is $\lfloor \delta(1+\delta)^{k-2} \rfloor$.
3. $|C_{i,k}| = \left\lceil \frac{2(1+\delta)^k}{\lfloor \delta(1+\delta)^{k-2} \rfloor} \right\rceil$.

This observation can be used to calculate the size of the checkpoint data which the algorithm stores at any time. The proof can also be found in the full version.

▶ **Lemma 17.** *At Iteration i of the algorithm the number of checkpoints is in* $O\left(\frac{\log(n)}{\varepsilon \log(1+\varepsilon)}\right)$.

The space bounds of Theorem 3 hold due to the following property of the checkpointing: If there are more than three palindromes stored in a list L_c for checkpoint c, then the palindromes form a run and can be stored in constant space as the following lemma shows.

▶ **Lemma 18.** *At iteration i, let $c \in CL_i$ be an arbitrary checkpoint. The list L_c can be stored in constant space.*

Proof. We fix an arbitrary $c \in \mathrm{CL}_i$. For the case that there are less than three palindromes belonging to L_c, they can be stored as R_S-entries in constant space. Therefore, we assume the case where there are at least three palindromes belonging to L_c and we show that they form a run. Let c' be the highest (index) checkpoint less than c, i.e., $c' = max\{c'' \mid c'' \in \mathrm{CL}_i \; and \; c'' < c\}$. We disregard the case that the index of c is 1. Let k be the minimum value such that $(1+\delta)^{k-1} < i - c \leq (1+\delta)^k$. Recall that L_c is the list of palindromes which the algorithm has successfully checked against c and not against c' yet. Let $P[m]$ be a palindrome in L_c. Since it was successfully checked against c we know that $i - m \geq m - c$. Similarly, since $P[m]$ was not checked against c' we have $i - m < m - c'$. Thus, for every $P[m]$ in L_c we have $\frac{i+c'}{2} < m \leq \frac{i+c}{2}$. Therefore, all palindromes stored in L_c are in an interval of length less than $\frac{i+c}{2} - \frac{i+c'}{2} = \frac{c-c'}{2}$. If we show that $\ell(m) \geq \frac{c-c'}{2}$ for all $P[m]$ in L_c, then applying Lemma 11 with $\ell^* = \frac{c-c'}{2}$ on the palindromes in L_c implies that they are forming a run. The run can be stored in constant space in an R_{NF}-entry. Therefore, it remains to show that $\ell(m) \geq \frac{c-c'}{2}$. We first argue the following: $c - c' \underset{Obs.\ 16}{\leq} \delta(1+\delta)^{k-2} \underset{\delta\leq 1}{\leq} \frac{(1+\delta)^{k-1}}{2} \underset{Def.\ of\ k}{<} \frac{i-c}{2}$. Since $P[m]$ was successfully checked against c and since $m > \frac{i+c'}{2}$ we derive that $\ell(m) > \frac{i+c'}{2} - c$. Therefore, $\ell(m) > \frac{i+c'}{2} - c = \frac{i-c}{2} + \frac{c'-c}{2} \underset{(6.1)}{>} c - c' + \frac{c'-c}{2} = \frac{c-c'}{2}$. ◀

The following lemma shows that the checkpoints are sufficiently close in order to satisfy the multiplicative approximation. The proof is in the full version.

▶ **Lemma 19.** ApproxLog *reports a midpoint* m^* *such that w.h.p.* $\frac{\ell_{max}}{(1+\varepsilon)} \leq \tilde{\ell}(m^*) \leq \ell_{max}$.

We are ready to prove Theorem 3. The correctness follows from Lemma 19. Lemma 17 and Lemma 18 yield the claimed space. In every iteration the algorithm processes every checkpoint in CL_i in constant time. The number of checkpoints is bounded by Lemma 17. A full proof can be found in the full version.

References

1 Alberto Apostolico, Dany Breslauer, and Zvi Galil. Parallel detection of all palindromes in a string. *Theoretical Computer Science*, 141(1–2):163–173, 1995.

2 Dany Breslauer and Zvi Galil. Real-time streaming string-matching. In Raffaele Giancarlo and Giovanni Manzini, editors, *Combinatorial Pattern Matching*, volume 6661 of *LNCS*, pages 162–172. Springer Berlin Heidelberg, 2011.

3 Funda Ergün, Hossein Jowhari, and Mert Sağlam. Periodicity in streams. In *Proceedings of the 13th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, and 14th International Workshop on Randomization and Computation (APPROX/RANDOM'10)*, volume 6302 of *LNCS*, pages 545–559, Berlin, Heidelberg, 2010. Springer-Verlag.

4 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, March 1987.

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

6 Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *Proceedings of the 2009 50th Annual IEEE Symposium on Foundations of Computer Science*, FOCS'09, pages 315–323, Washington, DC, USA, 2009. IEEE Computer Society.