Algorithmica (2019) 81:3630–3654
https://doi.org/10.1007/s00453-019-00591-8

Tight Tradeoffs for Real-Time Approximation of Longest Palindromes in Streams

Paweł Gawrychowski[1] · Oleg Merkurev[2] · Arseny M. Shur[2] · Przemysław Uznański[3]

Received: 18 August 2017 / Accepted: 8 April 2019 / Published online: 4 June 2019

Abstract
We consider computing a longest palindrome in the streaming model, where the symbols arrive one-by-one and we do not have random access to the input. While computing the answer exactly using sublinear space is not possible in such a setting, one can still hope for a good approximation guarantee. Our contribution is twofold. First, we provide lower bounds on the space requirements for randomized approximation algorithms processing inputs of length n. We rule out Las Vegas algorithms, as they cannot achieve sublinear space complexity. For Monte Carlo algorithms, we prove a lower bound of $\Omega(M \log \min\{|\Sigma|, M\})$ bits of memory; here $M = n/E$ for approximating the answer with additive error E, and $M = \log n / \log(1+\varepsilon)$ for approximating the answer with multiplicative error $(1+\varepsilon)$. Second, we design four real-time algorithms for this problem. Three of them are Monte Carlo approximation algorithms for additive error, "small" and "big" multiplicative errors, respectively. Each algorithm uses $\mathcal{O}(M)$ words of memory. Thus the obtained lower bounds are asymptotically tight up to a logarithmic factor. The fourth algorithm is deterministic and finds a longest palindrome exactly if it is short. This algorithm can be run in parallel with a Monte Carlo algorithm to obtain better results in practice. Overall, both the time and space complexity of finding a longest palindrome in a stream are essentially settled.

Keywords Palindrome · Streaming · Lower bound · Real-time algorithm

Mathematics Subject Classification 68W32 · 68W25 · 68Q25

✉ Paweł Gawrychowski
gawry@cs.uni.wroc.pl

Extended author information available on the last page of the article

1 Introduction

In the streaming model of computation, a very long input arrives sequentially in small portions and cannot be stored in full due to space limitation. There is a variation of this model where several passes over the input stream are available, but in this paper we consider only the standard one-pass model. While well-studied in general, streaming is a rather recent trend in algorithms on strings. The main goals are minimizing the space complexity, i.e., avoiding storing the already seen prefix of the string explicitly, and designing real-time algorithm, i.e., processing each symbol in worst-case constant time. However, the algorithms are usually randomized and return the correct answer with high probability. The prime example of a problem on string considered in the streaming model is pattern matching, where we want to detect an occurrence of a pattern in a given text. It is somewhat surprising that one can actually solve it using polylogarithmic space in the streaming model, as proved by Porat and Porat [16]. A simpler solution was later given by Ergün et al. [6], while Breslauer and Galil designed a real-time algorithm [3]. Similar questions studied in such setting include multiple-pattern matching [4], approximate pattern matching [5], and parametrized pattern matching [11].

We consider computing a longest palindrome in the streaming model, where a palindrome is a fragment which reads the same in both directions. This is one of the basic questions concerning regularities in texts and it has been extensively studied in the classical non-streaming setting, see [1,8,13,15] and the references therein. The notion of palindromes, but with a slightly different meaning, is very important in computational biology, where one considers strings over $\{A, T, C, G\}$ and a palindrome is a sequence equal to its reverse complement (a reverse complement reverses the sequences and interchanges A with T and C with G); see [9] and the references therein for a discussion of their algorithmic aspects. Our results generalize to biological palindromes in a straightforward manner.

We denote by LPS the following problem: *given a string S, find the maximum length of a palindrome in S and a starting position of a palindrome of such length in S*. Solving LPS in the streaming model was first considered by Berenbrink et al. [2], who developed tradeoffs between the bound on the error and the space complexity for approximating the length of the longest palindrome with either additive or multiplicative error.[1] They presented the algorithms solving the LPS problem (i) in $\mathcal{O}\left(\frac{n\sqrt{n}}{E}\right)$ time and $\mathcal{O}\left(\frac{n}{E}\right)$ space with the additive error $E \in [1, \sqrt{n}]$; (ii) in $\mathcal{O}\left(\frac{n \log n}{\varepsilon \log(1+\varepsilon)}\right)$ time and $\mathcal{O}\left(\frac{\log n}{\varepsilon \log(1+\varepsilon)}\right)$ space with the multiplicative error $(1+\varepsilon)$, where $\varepsilon < 1$; (iii) in $\mathcal{O}(n)$ time and $\mathcal{O}(\sqrt{n})$ space exactly, *given the promise that the longest palindrome is shorter than $\sqrt{n}$*. All their algorithms are Monte Carlo, i.e., return the correct answer with high probability. They also proved that any Las Vegas algorithm achieving additive error E must necessarily use $\Omega\left(\frac{n}{E}\log|\Sigma|\right)$ bits of memory, which matches the space complexity of their Monte Carlo solution up to a logarithmic factor in the $E \in [1, \sqrt{n}]$ range. These results leave a number of open questions both on more efficient algorithms (e.g., only the algorithm

[1] If the maximum length of a palindrome in S is L, then the additive error E (resp., multiplicative error $(1+\varepsilon)$) means that the algorithm must return a number $\ell \geq L - E$ (resp., $\ell \geq L/(1+\varepsilon)$) and the position of a palindrome of length ℓ.

(iii) and a specific case of algorithm (i) are linear-time) and on tight lower bounds for the space complexity, in particular, for Monte Carlo algorithms. In the present paper we answer all these questions, essentially settling space and time complexity of LPS. As in [2], we use hashes and other common primitives of the streaming model, but otherwise our technique is different; in particular, we heavily rely on combinatorial lemmas on strings. This paper extends the early version [10] presented at CPM 2016.

Let us overview the obtained results. First, we show that Las Vegas algorithms cannot achieve sublinear space complexity at all; thus, in the streaming model LPS can be solved only with high probability. Second, we prove a lower bound of $\Omega(M \log \min\{|\Sigma|, M\})$ bits of memory for Monte Carlo algorithms; here $M = n/E$ for approximating the answer with additive error $E \in [1, n]$, and $M = \frac{\log n}{\log(1+\varepsilon)}$ for approximating the answer with multiplicative error $(1 + \varepsilon)$, where $\varepsilon > n^{-0.99}$. After this, we design three linear-time, and even real-time, Monte Carlo algorithms matching these lower bounds up to a logarithmic factor (moreover, the match is exact for wide ranges of involved parameters).

- Algorithm A for LPS with additive error $E \in [1, n]$ uses $\mathcal{O}(n/E)$ words of memory. Compared to (i), it uses the same space, works faster if $E = o(\sqrt{n})$, and lifts the restriction on E; another advantage is independence of the working time of the error. The space usage exactly matches the lower bound under reasonable assumptions $|\Sigma| > n^{0.01}$, $E < n^{0.99}$
- Algorithm M for LPS with multiplicative error $\varepsilon \in (0, 1]$ uses $\mathcal{O}\big(\frac{\log(n\varepsilon)}{\varepsilon}\big)$ words of memory. Compared to (ii), the space bound is lowered by at least the factor of ε^{-1} (since $\log(1 + \varepsilon)$ is equivalent to ε whenever $\varepsilon < 1$), the time bound is lowered by the factor of $\varepsilon^{-2} \cdot \log n$. The space usage matches the lower bound up to a logarithmic factor in the worst case (if ε is a constant and $|\Sigma| = \mathcal{O}(\log n)$); the match is exact if $\varepsilon < n^{-0.01}$, $|\Sigma| > n^{0.01}$
- Algorithm M' for LPS with multiplicative error $\varepsilon \in (1, n]$ uses $\mathcal{O}\big(\frac{\log n}{\log(1+\varepsilon)}\big)$ words of memory. It has no analogs in [2] and matches the space lower bound up to a logarithmic factor

Finally we present, for any m, a deterministic $\mathcal{O}(m)$-space real-time Algorithm E, solving LPS exactly if the answer is less than m and detecting a palindrome of length $\geq m$ otherwise. This is a significant improvement over (iii). Algorithm E shows that if the input stream is fully random, then with high probability its longest palindrome can be found exactly by a real-time algorithm within logarithmic space.

The paper is organized as follows: we study lower bounds in Sect. 2, and algorithms in Sect. 3. We also note that Monte Carlo algorithms compute hash values of certain substrings of the input string and use these values to check whether a substring is a palindrome; false positives correspond to hash collisions, false negatives are impossible.

1.1 Notation and Definitions

Let S denote a string of length n over an alphabet $\Sigma = \{1, \ldots, N\}$, where N is polynomial in n. We write $S[i]$ for the ith symbol of S and $S[i \ldots j]$ for its *substring* (or *factor*) $S[i]S[i+1]\cdots S[j]$; thus, $S[1\dots n] = S$. A *prefix* (resp. *suffix*) of S is a substring of the form $S[1\dots j]$ (resp., $S[j\dots n]$). A string S is a *palindrome* if it equals its *reversal* $S[n]S[n-1]\cdots S[1]$. By $L(S)$ we denote the length of a longest palindrome which is a factor of S. The symbol log stands for the binary logarithm.

We consider the *streaming model* of computation: the input string $S[1\dots n]$ (called the *stream*) is read left to right, one symbol at a time, and cannot be stored, because the available space is sublinear in n. The space is counted as the number of $\mathcal{O}(\log n)$-bit machine words. An algorithm is *real-time* if the number of operations between two reads is bounded by a constant. An approximation algorithm for a maximization problem has *additive error* E (resp., *multiplicative error* ε) if it finds a solution with the cost at least $OPT - E$ (resp., $\frac{OPT}{1+\varepsilon}$), where OPT is the cost of optimal solution; here both E and ε can be functions of the size of the input. For an instance $\mathsf{LPS}(S)$ of the LPS problem, $OPT = L(S)$.

A *Las Vegas algorithm* always returns a correct answer, but its working time and memory usage on the inputs of length n are random variables. A *Monte Carlo algorithm* gives a correct answer with high probability (greater than $1-1/n$) and has deterministic working time and space.

2 Lower Bounds

In this section we use Yao's minimax principle [18] to prove lower bounds on the space complexity of the LPS problem in the streaming model, where the length n and the alphabet Σ of the input stream are specified. We denote this problem by $\mathsf{LPS}_\Sigma[n]$.

Theorem 1 (Yao's minimax principle for randomized algorithms) *Let $\mathcal{X}$ be the set of inputs for a problem and $\mathcal{A}$ be the set of all deterministic algorithms solving it. For every $x \in \mathcal{X}$ and $a \in \mathcal{A}$, let $c(a, x) \geq 0$ be the cost of running a on x.*

Let $\mathcal{P}, \mathcal{Q}$ be probability distributions over $\mathcal{A}$ and $\mathcal{X}$, respectively. Then $\max_{x\in\mathcal{X}} \mathbf{E}_{A\sim\mathcal{P}}[c(A, x)] \geq \min_{a\in\mathcal{A}} \mathbf{E}_{X\sim\mathcal{Q}}[c(a, X)]$.

We use the above theorem for both Las Vegas and Monte Carlo algorithms. For Las Vegas algorithms, we consider only correct algorithms, and $c(x, a)$ is the memory usage. For Monte Carlo algorithms, we consider all algorithms (not necessarily correct) with memory usage not exceeding a certain threshold, and $c(x, a)$ is the correctness indicator function, i.e., $c(x, a) = 0$ if the algorithm is correct and $c(x, a) = 1$ otherwise.

Our proofs will be based on appropriately chosen padding. The padding requires a constant number of fresh characters. If Σ is twice as large as the number of required fresh characters, we can still use half of it to construct a difficult input instance, which does not affect the asymptotics. Otherwise, we construct a difficult input instance over Σ, then add enough new fresh characters to facilitate the padding, and finally reduce the resulting larger alphabet to binary at the expense of increasing the size of the input by a constant factor.

Lemma 1 *For any alphabet $\Sigma = \{1, 2, \dots, \sigma\}$ there exists a morphism $h : \Sigma^* \to \{0, 1\}^*$ such that, for any $c \in \Sigma$, $|h(c)| = 2\sigma + 6$ and, for any string w, w contains a palindrome of length ℓ if and only if $h(w)$ contains a palindrome of length $(2\sigma + 6)\cdot\ell$.*

Proof We set:

$$h(c) = 11^c01^{\sigma-c}10011^{\sigma-c}01^c1.$$

Clearly $|h(c)| = 2\sigma + 6$ and, because every $h(c)$ is a palindrome, if w contains a palindrome of length ℓ then $h(w)$ contains a palindrome of length $(2\sigma + 6) \cdot \ell$. Now assume that $h(w)$ contains a palindrome T of length $(2\sigma + 6) \cdot \ell$, where $\ell \geq 1$. If $\ell = 1$ then we obtain that w should contain a palindrome of length 1, which always holds. Otherwise, T contains 00 inside and

– either is centered inside some 00 and thus corresponds to a palindrome of odd length ℓ in w;
– or is centered in the middle between two consecutive occurrences of 00 and thus corresponds to a palindrome of even length ℓ in w.

In either case, the claim holds. □

For the padding we will often use an infinite string $\nu = 0^11^10^21^20^31^3\ldots$, or more precisely its prefixes of length d, denoted $\nu(d)$. Here 0 and 1 should be understood as two characters not belonging to the original alphabet. The longest palindrome in $\nu(d)$ has length $\mathcal{O}(\sqrt{d})$.

Theorem 2 (Las Vegas approximation) *Let A be a Las Vegas streaming algorithm solving* $\mathsf{LPS}_\Sigma[n]$ *with additive error* $E \leq 0.99n$ *or multiplicative error* $(1+\varepsilon) \leq 100$ *using* $s(n)$ *bits of memory. Then* $\mathbf{E}[s(n)] = \Omega(n \log|\Sigma|)$.

Proof By Theorem 1, it is enough to construct a probability distribution $\mathcal{Q}$ over Σ^n such that for any deterministic algorithm D, its expected memory usage on a string chosen according to Π is $\Omega(n \log|\Sigma|)$ in bits.

Consider solving $\mathsf{LPS}_\Sigma[n]$ with additive error E. We define $\mathcal{Q}$ as the uniform distribution over $\nu(\frac{E}{2})x\$\$y\nu(\frac{E}{2})^R$, where $x, y \in \Sigma^{n'}$, $n' = \frac{n}{2} - \frac{E}{2} - 1$, and \$ is a special character not in Σ. Let us look at the memory usage of D after having read $\nu(\frac{E}{2})x$. We say that x is "good" when the memory usage is at most $\frac{n'}{2}\log|\Sigma|$ and "bad" otherwise. Assume that $\frac{1}{2}|\Sigma|^{n'}$ of all x's are good, then there are two strings $x \neq x'$ such that the state of D after having read both $\nu(\frac{E}{2})x$ and $\nu(\frac{E}{2})x'$ is exactly the same. Hence the behavior of D on $\nu(\frac{E}{2})x\$\$x^R\nu(\frac{E}{2})^R$ and $\nu(\frac{E}{2})x'\$\$x^R\nu(\frac{E}{2})^R$ is exactly the same. The former is a palindrome of length $n = 2n' + E + 2$, so D must answer at least $2n' + 2$, and consequently the latter also must contain a palindrome of length at least $2n' + 2$. A palindrome inside $\nu(\frac{E}{2})x'\$\$x^R\nu(\frac{E}{2})^R$ is either fully contained within $\nu(\frac{E}{2})$, x', x^R or it is a middle palindrome. But the longest palindrome inside $\nu(\frac{E}{2})$ is of length $\mathcal{O}(\sqrt{E}) < 2n' + 2$ (for n large enough) and the longest palindrome inside x or x^R is of length $n' < 2n' + 2$, so since we have exluded other possibilities, $\nu(\frac{E}{2})x'\$\$x^R\nu(\frac{E}{2})^R$ contains a middle palindrome of length $2n' + 2$. This implies that $x = x'$, which is a contradiction. Therefore, at least $\frac{1}{2}|\Sigma|^{n'}$ of all x's are bad. But then the expected memory usage of D is at least $\frac{n'}{4}\log|\Sigma|$, which for $E \leq 0.99n$ is $\Omega(n \log|\Sigma|)$ as claimed.

Now consider solving $\mathsf{LPS}_\Sigma[n]$ with multiplicative error $(1+\varepsilon)$. An algorithm with multiplicative error $(1+\varepsilon)$ can also be considered as having additive error $E = n \cdot \frac{\varepsilon}{1+\varepsilon}$, so if the expected memory usage of such an algorithm is $o(n \log |\Sigma|)$ and $(1+\varepsilon) \le 100$ then we obtain an algorithm with additive error $E \le 0.99n$ and expected memory usage $o(n \log |\Sigma|)$, which we already know to be impossible. □

Now we move to Monte Carlo algorithms. We first consider exact algorithms solving $\mathsf{LPS}_\Sigma[n]$; lower bounds on approximation algorithms will be then obtained by padding the input appropriately. We introduce an auxiliary problem $\mathsf{midLPS}_\Sigma[n]$, which is to compute the length of the middle palindrome in a string of even length n over an alphabet Σ.

Lemma 2 *There exists a constant γ such that any randomized Monte Carlo streaming algorithm A solving $\mathsf{midLPS}_\Sigma[n]$ or $\mathsf{LPS}_\Sigma[n]$ exactly with probability $1 - \frac{1}{n}$ uses at least $\gamma \cdot n \log \min\{|\Sigma|, n\}$ bits of memory.*

Proof First we prove that if A is a Monte Carlo streaming algorithm solving $\mathsf{midLPS}_\Sigma[n]$ exactly using less than $\lfloor \frac{n}{2} \log |\Sigma| \rfloor$ bits of memory, then its error probability is at least $\frac{1}{n|\Sigma|}$.

By Theorem 1, it is enough to construct probability distribution $\mathcal{Q}$ over Σ^n such that for any deterministic algorithm D using less than $\lfloor \frac{n}{2} \log |\Sigma| \rfloor$ bits of memory, the expected probability of error on a string chosen according to $\mathcal{Q}$ is at least $\frac{1}{n|\Sigma|}$.

Let $n' = \frac{n}{2}$. For any $x \in \Sigma^{n'}$, $k \in \{1, 2, \ldots, n'\}$ and $c \in \Sigma$ we define

$$w(x, k, c) = x[1]x[2]x[3] \ldots x[n']x[n']x[n'-1]x[n'-2] \ldots x[k+1]c0^{k-1}.$$

Now $\mathcal{Q}$ is the uniform distribution over all such $w(x, k, c)$.

Choose an arbitrary maximal matching of strings from $\Sigma^{n'}$ into pairs (x, x') such that D is in the same state after reading either x or x'. At most one string per state of D is left unpaired, that is at most $2^{\lfloor \frac{n}{2} \log |\Sigma| \rfloor - 1}$ strings in total. Since there are $|\Sigma|^{n'} = 2^{n' \log |\Sigma|} \ge 2 \cdot 2^{\lfloor \frac{n}{2} \log |\Sigma| \rfloor - 1}$ possible strings of length n', at least half of the strings are paired. Let s be longest common suffix of x and x', so $x = vcs$ and $x' = v'c's$, where $c \ne c'$ are single characters. Then D returns the same answer on $w(x, n' - |s|, c)$ and $w(x', n' - |s|, c)$, even though the length of the middle palindrome is exactly $2|s|$ in one of them, and at least $2|s| + 2$ in the other one. Therefore, D errs on at least one of these two inputs. Similarly, it errs on either $w(x, n' - |s|, c')$ or $w(x, n' - |s|, c')$. Thus the error probability is at least $\frac{1}{2n'|\Sigma|} = \frac{1}{n|\Sigma|}$.

Now we can prove the lemma for $\mathsf{midLPS}_\Sigma[n]$ with a standard amplification trick. Say that we have a Monte Carlo streaming algorithm, which solves $\mathsf{midLPS}_\Sigma[n]$ exactly with error probability ε using $s(n)$ bits of memory. Then we can run its k instances simultaneously and return the most frequently reported answer. The new algorithm needs $\mathcal{O}(k \cdot s(n))$ bits of memory and its error probability ε_k satisfies the inequality

$$\varepsilon_k \le \sum_{2i<k} \binom{k}{i} (1-\varepsilon)^i \varepsilon^{k-i} \le 2^k \cdot \varepsilon^{k/2} = (4\varepsilon)^{k/2}.$$

Let $\kappa = \frac{1}{6}\frac{\log(4/n)}{\log(1/(n|\Sigma|))}$. We have

$$\kappa = \frac{1}{6}\frac{1-o(1)}{1+\log|\Sigma|/\log n} = \Theta\left(\frac{\log n}{\log n + \log|\Sigma|}\right) = \gamma \cdot \frac{1}{\log|\Sigma|}\log\min\{|\Sigma|, n\},$$

for some constant γ. Now we can prove the theorem. Assume that A uses less than $\kappa \cdot n\log|\Sigma| = \gamma \cdot n\log\min\{|\Sigma|, n\}$ bits of memory. Then running $\lfloor\frac{1}{2\kappa}\rfloor \geq \frac{3}{4}\frac{1}{2\kappa}$ (which holds since $\kappa < \frac{1}{6}$) instances of A in parallel requires less than $\lfloor\frac{n}{2}\log|\Sigma|\rfloor$ bits of memory. But then the error probability of the new algorithm is bounded from above by

$$\left(\frac{4}{n}\right)^{3/16\kappa} = \left(\frac{1}{n|\Sigma|}\right)^{18/16} \leq \frac{1}{n|\Sigma|}$$

which we have already shown to be impossible.

The lower bound for $\mathsf{midLPS}_\Sigma[n]$ can be translated into a lower bound for solving $\mathsf{LPS}_\Sigma[n]$ exactly by padding the input so that the longest palindrome is centered in the middle. Let $x = x[1]x[2]\ldots x[n]$ be the input for $\mathsf{midLPS}_\Sigma[n]$. We define

$$w(x) = x[1]x[2]x[3]\ldots x[n/2]1\underbrace{000\ldots0}_{n}1x[n/2+1]\ldots x[n].$$

Now if the length of the middle palindrome in x is k, then $w(x)$ contains a palindrome of length at least $n+k+2$. In the other direction, any palindrome inside $w(x)$ of length $\geq n$ must be centered somewhere in the middle block consisting of only zeroes and both ones are mapped to each other, so it must be the middle palindrome. Thus, the length of the longest palindrome inside $w(x)$ is exactly $n+k+2$, so we have reduced solving $\mathsf{midLPS}_\Sigma[n]$ to solving $\mathsf{LPS}_\Sigma[2n+2]$. We already know that solving $\mathsf{midLPS}_\Sigma[n]$ with probability $1-\frac{1}{n}$ requires $\gamma \cdot n\log\min\{|\Sigma|, n\}$ bits of memory, so solving $\mathsf{LPS}_\Sigma[2n+2]$ with probability $1-\frac{1}{2n+2} \geq 1-\frac{1}{n}$ requires $\gamma \cdot n\log\{|\Sigma|, n\} \geq \gamma' \cdot (2n+2)\log\min\{|\Sigma|, 2n+2\}$ bits of memory. Notice that the reduction needs $\mathcal{O}(\log n)$ additional bits of memory to count up to n, but for large n this is much smaller than the lower bound if we choose $\gamma' < \frac{\gamma}{4}$. □

To obtain a lower bound for Monte Carlo additive approximation, we observe that any algorithm solving $\mathsf{LPS}_\Sigma[n]$ with additive error E can be used to solve $\mathsf{LPS}_\Sigma[\frac{n-E}{E+1}]$ exactly by inserting $\frac{E}{2}$ zeroes between every two characters, in the very beginning, and in the very end. However, this reduction requires $\log(\frac{E}{2}) \leq \log n$ additional bits of memory for counting up to $\frac{E}{2}$ and cannot be used when the desired lower bound on the required number of bits $\Omega(\frac{n}{E}\log\min(|\Sigma|, \frac{n}{E})$ is significantly smaller than $\log n$. Therefore, we need a separate technical lemma which implies that both additive and multiplicative approximation with error probability $\frac{1}{n}$ require $\Omega(\log n)$ bits of space.

Lemma 3 *Let A be any randomized Monte Carlo streaming algorithm solving* $\mathsf{LPS}_\Sigma[n]$ *with additive error at most* $0.99n$ *or multiplicative error at most* $n^{0.49}$ *and error probability* $\frac{1}{n}$. *Then A uses* $\Omega(\log n)$ *bits of memory.*

Proof By Theorem 1, it is enough to construct a probability distribution $\mathcal{Q}$ over Σ^n such that for any deterministic algorithm D using at most $s(n) = o(\log n)$ bits of memory, the expected probability of error on a string chosen according to $\mathcal{Q}$ is $\frac{1}{2^{s(n)+2}} > \frac{1}{n}$.

Let $n' = s(n) + 1$. For any $x, y \in \Sigma^{n'}$, let $w(x, y) = \nu(\frac{n}{2} - n')^R x y^R \nu(\frac{n}{2} - n')$. Observe that if $x = y$ then $w(x, y)$ contains a palindrome of length n, and otherwise the longest palindrome there has length at most $2n' + \mathcal{O}(\sqrt{n}) = \mathcal{O}(\sqrt{n})$, thus any algorithm with additive error of at most $0.99n$ or with a multiplicative error at most $n^{0.49}$ must be able to distinguish between these two cases (for n large enough).

Let $S \subseteq \Sigma^{n'}$ be an arbitrary family of strings of length n' such that $|S| = 2 \cdot 2^{s(n)}$, and let $\mathcal{Q}$ be the uniform distribution on all strings of the form $w(x, y)$, where x and y are chosen uniformly and independently from S. By a counting argument, we can create at least $\frac{|S|}{4}$ pairs (x, x') of elements from S such that the state of D is the same after having read $\nu(\frac{n}{2} - n')^R x$ and $\nu(\frac{n}{2} - n')^R x'$. (If we create the pairs greedily, at most one such x per state of memory can be left unpaired, so at least $|S| - 2^{s(n)} = \frac{|S|}{2}$ elements are paired.) Thus, D cannot distinguish between $w(x, x')$ and $w(x, x)$, and between $w(x', x')$ and $w(x', x)$, so its error probability must be at least $\frac{|S|/2}{|S|^2} = \frac{1}{4 \cdot 2^{s(n)}}$. Thus if $s(n) = o(\log n)$, the error rate is at least $\frac{1}{n}$ for n large enough, a contradiction. □

Combining the reduction with the technical lemma and taking into account that we are reducing to a problem with string length of $\Theta(\frac{n}{E})$, we obtain the following.

Theorem 3 (Monte Carlo additive approximation) *Let A be any randomized Monte Carlo streaming algorithm solving* $\mathsf{LPS}_\Sigma[n]$ *with additive error E with probability* $1 - \frac{1}{n}$. *If* $E \leq 0.99n$ *then A uses* $\Omega(\frac{n}{E} \log \min\{|\Sigma|, \frac{n}{E}\})$ *bits of memory.*

Proof Define $\sigma = \min\{|\Sigma|, \frac{n}{E}\}$. Because of Lemma 3 it is enough to prove that $\Omega(\frac{n}{E} \log \sigma)$ is a lower bound when

$$E \leq \frac{\gamma}{2} \cdot \frac{n}{\log n} \log \sigma. \tag{1}$$

Assume that there is a Monte Carlo streaming algorithm A solving $\mathsf{LPS}_\Sigma[n]$ with additive error E with probability $1 - \frac{1}{n}$, using $o(\frac{n}{E} \log \sigma)$ bits of memory. Let $n' = \frac{n - E/2}{E/2+1} \geq \frac{n}{E}$ (the last inequality, equivalent to $n \geq E \cdot \frac{E}{E-2}$, holds because $E \leq 0.99n$ and because we can assume $E \geq 200$). Given a string $x[1]x[2] \ldots x[n']$, we run A on $0^E x[1] 0^{E/2} x[2] 0^{E/2} x[3] \ldots 0^{E/2} x[n'] 0^{E/2}$, using $\log(E/2) \leq \log n$ additional bits of memory, get some answer R, and then return the number $\left\lfloor \frac{R}{E/2+1} \right\rfloor$. We call this new Monte Carlo streaming algorithm A$'$. Recall that A reports the length of the longest palindrome with additive error E. Therefore, if the original string contains a palindrome of length r, the new string contains a palindrome of length $\frac{E}{2} \cdot (r+1) + r$, so $R \geq r(E/2+1)$ and A$'$ will return at least r. In the other direction, if A$'$ returns r, then the new string contains a palindrome of length $r(E/2+1)$. If such palindrome is centered so that $x[i]$ is matched with $x[i+1]$ for some i, then it clearly corresponds to a palindrome of length r in the original string. But otherwise every $x[i]$ within the palindrome is matched with 0, so in fact the whole palindrome corresponds to a streak of consecutive zeroes in the new string and can be extended to the left and to the right to start and end with 0^E, so again it corresponds to a palindrome of length r in the original string. Therefore, A' solves $\mathsf{LPS}_\Sigma[n']$ exactly with probability $1 - \frac{1}{(n'(E/2+1)+E/2)} \geq 1 - \frac{1}{n'}$ and uses $o(\frac{n'(E/2+1)+E/2}{E/2}\log\sigma) + \log n = o(n'\log\sigma) + \log n$ bits of memory. But this is smaller than the lower bound of Lemma 2:

$$\gamma \cdot n' \log\min\{|\Sigma|, n'\} \geq \frac{\gamma}{2}\cdot n'\log\sigma + \frac{\gamma}{2}\cdot\frac{n}{E}\log\sigma \geq \frac{\gamma}{2}\cdot n'\log\sigma + \log n$$

(the last inequality follows from (1)). This contradiction finishes the proof. □

Finally, we consider multiplicative approximation. The proof follows the same basic idea as of Theorem 3, however is more technically involved. The main difference is that due to uneven padding, we are reducing to $\mathsf{midLPS}_\Sigma[n']$ instead of $\mathsf{LPS}_\Sigma[n']$.

Theorem 4 (Monte Carlo multiplicative approximation) *Let A be any randomized Monte Carlo streaming algorithm solving $\mathsf{LPS}_\Sigma[n]$ with multiplicative error $(1+\varepsilon)$ with probability $1-\frac{1}{n}$. If $n^{-0.98} \leq \varepsilon \leq n^{0.49}$ then A uses $\Omega(\frac{\log n}{\log(1+\varepsilon)}\log\min\{|\Sigma|, \frac{\log n}{\log(1+\varepsilon)}\})$ bits of memory.*

Proof For $\varepsilon \geq n^{0.001}$ the claimed lower bound reduces to $\Omega(1)$ bits, which obviously holds. Thus we can assume that $\varepsilon < n^{0.001}$. Define

$$\sigma = \min\{|\Sigma|, \frac{1}{50}\frac{\log n}{\log(1+2\varepsilon)} - 2\}.$$

First we argue that it is enough to prove that $\mathcal{A}$ uses $\Omega(\frac{\log n}{\log(1+\varepsilon)}\log\sigma)$ bits of memory. Since $\log(1+2\varepsilon) \leq 0.001\log n + o(\log n)$, we have

$$\frac{1}{50}\frac{\log n}{\log(1+2\varepsilon)} - 2 \geq 18 - o(1) \tag{2}$$

and consequently

$$\frac{1}{50}\frac{\log n}{\log(1+2\varepsilon)} - 2 = \Theta\left(\frac{\log n}{\log(1+2\varepsilon)}\right). \tag{3}$$

Finally, observe that

$$\log(1+2\varepsilon) = \Theta(\log(1+\varepsilon)) \tag{4}$$

because $\log 2(1+\varepsilon) = \Theta(\log(1+\varepsilon))$ for $\varepsilon \geq 1$, and $\log(1+\varepsilon) = \Theta(\varepsilon)$ for $\varepsilon < 1$. From (3) and (4) we conclude that

$$\log\sigma = \Theta\left(\log\min\{|\Sigma|, \frac{\log n}{\log(1+\varepsilon)}\}\right). \tag{5}$$

Because of Lemma 3 and Eqs. (4) and (5), it is enough to prove that $\Omega(\frac{\log n}{\log(1+\varepsilon)}\log\sigma)$ is a lower bound when

$$\log(1+2\varepsilon) \le \gamma \cdot \frac{\log\sigma}{100}, \tag{6}$$

as otherwise $\Omega(\frac{\log n}{\log(1+\varepsilon)}\log\sigma) = \Omega(\frac{\log n}{\log(1+2\varepsilon)}\log\sigma) = \Omega(\log n)$.

Assume that there is a Monte Carlo streaming algorithm A solving $\mathsf{LPS}_\Sigma[n]$ with multiplicative error $(1+\varepsilon)$ with probability $1-\frac{1}{n}$ using $o(\frac{\log n}{\log(1+\varepsilon)}\log\sigma)$ bits of memory. Let $x = x[1]x[2]\ldots x[n']x[n'+1]\ldots x[2n']$ be an input for $\mathsf{midLPS}_\Sigma[2n']$. We choose n' so that $n = (1+2\varepsilon)^{n'+1}\cdot n^{0.99}$. Then $n' = \log_{(1+2\varepsilon)}(n^{0.01}) - 1 = \frac{1}{100}\frac{\log n}{\log(1+2\varepsilon)} - 1$. We choose $i_0, i_1, i_2, i_3, \ldots, i_{n'}$ so that $i_0 + \ldots + i_d = \lceil (1+2\varepsilon)^{d+1}\cdot n^{0.99}\rceil$ for any $0 \le d \le n'$. (Observe that for $\varepsilon = \Omega(n^{-0.98})$ we have $i_0 > n^{0.99}$ and $i_1, \ldots, i_d > 2n^{0.01} - 1$.) Finally we define

$$w(x) = \nu(i_{n'})^R x[1]\nu(i_{n'-1})^R \ldots x[n']\nu(i_0)^R \nu(i_0) x[n'+1]\nu(i_1)\ldots x[2n']\nu(i_{n'}).$$

If x contains a middle palindrome of length exactly $2k$, then $w(x)$ contains a middle palindrome of length $2(1+2\varepsilon)^{k+1}\cdot n^{0.99}$. Also, based on the properties of ν, any non-middle centered palindrome in $w(x)$ has length at most $\mathcal{O}(\sqrt{n})$, which is less than $n^{0.99}$ for n large enough. Since $\lceil 2(1+2\varepsilon)^k\cdot n^{0.99}\rceil\cdot(1+\varepsilon) < (2(1+2\varepsilon)^k\cdot n^{0.99}+1)\cdot(1+\varepsilon) < 2(1+2\varepsilon)^{k+1}\cdot n^{0.99}$, value of k can be extracted from the answer of A. Thus, if A approximates the middle palindrome in $w(x)$ with multiplicative error $(1+\varepsilon)$ with probability $1-\frac{1}{n}$ using $o(\frac{\log n}{\log(1+\varepsilon)}\log\sigma)$ bits of memory, we can construct a new algorithm A' solving $\mathsf{midLPS}_\Sigma[2n']$ exactly with probability $1-\frac{1}{n} > 1-\frac{1}{2n'}$ using

$$o\left(\frac{\log n}{\log(1+\varepsilon)}\log\sigma\right) + \log n \tag{7}$$

bits of memory. By Lemma 2 we get a lower bound

$$\begin{aligned}\gamma\cdot 2n'\log\min\{|\Sigma|, 2n'\} &= \frac{\gamma}{50}\cdot\frac{\log n}{\log(1+2\varepsilon)}\log\sigma - 2\gamma\log\sigma \\ &\ge \frac{\gamma}{100}\cdot\frac{\log n}{\log(1+2\varepsilon)}\log\sigma + \log n - 2\gamma\log\sigma, \end{aligned}\tag{8}$$

where the last inequality holds because of (6). On the other hand, for large n

$$\begin{aligned}&\frac{\gamma}{100}\cdot\frac{\log n}{\log(1+2\varepsilon)}\log\sigma - 2\gamma\log\sigma + \log n \\ &= \left(\frac{1}{100}\frac{\log n}{\log(1+2\varepsilon)} - 2\right)\gamma\log\sigma + \log n \\ &= \Theta\left(\frac{\log n}{\log(1+\varepsilon)}\log\sigma\right) + \log n\end{aligned}$$

so (8) exceeds (7), a contradiction. □

3 Real-Time Algorithms

In this section we design real-time Monte Carlo algorithms within the space bounds matching the lower bounds from Sect. 2 up to a factor bounded by $\log n$. The algorithms make use of the hash function known as the *Karp–Rabin fingerprint* [12]. We first describe this function and its properties and then provide an overview of our algorithms.

Let p be a fixed prime from the range $[n^{3+\alpha}, n^{4+\alpha}]$ for some $\alpha > 0$, and r be a fixed integer randomly chosen from $\{1, \ldots, p-1\}$. For a string S, its forward hash and reversed hash are defined, respectively, as

$$\phi^F(S) = \Big(\sum_{i=1}^{n} S[i] \cdot r^i\Big) \bmod p \;\text{ and }\; \phi^R(S) = \Big(\sum_{i=1}^{n} S[i] \cdot r^{n-i+1}\Big) \bmod p\,.$$

Clearly, the forward hash of a string coincides with the reversed hash of its reversal. Thus, if u is a palindrome, then $\phi^F(u) = \phi^R(u)$. The converse is also true modulo the (improbable) collisions of hashes, because for two strings $u \neq v$ of length m, the probability that $\phi^F(u) = \phi^F(v)$ is at most m/p. This property allows one to detect palindromes with high probability by comparing hashes. (This approach is somewhat simpler than the one of [2]; in particular, we do not need "fingerprint pairs" used there.) In particular, a real-time algorithm makes $\mathcal{O}(n)$ comparisons and thus faces a collision with probability $\mathcal{O}(n^{-1-\alpha})$ by the choice of p. All further considerations assume that no collisions happen. For an input stream S, we denote $F^F(i, j) = \phi^F(S[i \ldots j])$ and $F^R(i, j) = \phi^R(S[i \ldots j])$. Hashes of substrings can be extracted in constant time from the hashes of prefixes, as the next observation shows.

Proposition 1 [3] *The following equalities hold:*

$$\begin{aligned} F^F(i, j) &= r^{-(i-1)}\Big(F^F(1, j) - F^F(1, i-1)\Big) \bmod p\,, \\ F^R(i, j) &= F^R(1, j) - r^{j-i+1} F^R(1, i-1) \bmod p\,. \end{aligned}$$

Definition 1 For an input stream S, its *i-th frame* $I(i)$ is defined as the tuple $(i, F^F(1, i-1), F^R(1, i-1), r^{-(i-1)} \bmod p, r^i \bmod p)$.

The proposition below is immediate from definitions and Proposition 1.

Proposition 2 (1) *Given $I(i)$ and $S[i]$, the tuple $I(i+1)$ can be computed in $\mathcal{O}(1)$ time.*
(2) *Given $I(i)$ and $I(j+1)$, the string $S[i \ldots j]$ can be checked for being a palindrome in $\mathcal{O}(1)$ time.*

All algorithms in this section follow the same scheme. The outer cycle works in $n = |S|$ iterations; on ith iteration, the symbol $S[i]$ is read and processed, and the $(i+1)$th frame is computed from the ith frame. Each algorithm computes all frames but stores only a fraction of them, based on the available space. After reading $S[i]$, each algorithm checks whether some suffix of $S[1 \dots i]$ is a palindrome of bigger length than the longest previously found palindrome, and updates the longest palindrome respectively. The suffixes available for this check depend on stored frames; each check takes $\mathcal{O}(1)$ time by Proposition 2 (2). Several combinatorial lemmas are proved to show that on each iteration it is sufficient to check a constant number of suffixes.

Assume that $S[i \dots j]$ is the longest palindrome in S. It is quite probable that the ith frame will be unavailable after reading $S[j]$. However, we will be able to show that in all cases some "close enough" $(i+k)$th frame will be available after reading $S[j-k]$. Then the palindrome $S[i+k \dots j-k]$ will be found at the $(j-k)$th iteration, providing the approximation of the longest palindrome within the required error bound.

The technical part of the algorithms is the maintenance of the list of stored frames in a way that (a) guarantees the approximation error; (b) provides a constant-time access to the frames needed on each particular iteration; (c) allows a constant-time update at each iteration.

3.1 Additive Error

Theorem 5 *There is a real-time Monte Carlo algorithm solving each instance* LPS(S) *with the additive error $E = E(n)$ using $\mathcal{O}(n/E)$ space, where $n = |S|$.*

First we present a simple (and slow) algorithm which solves the posed problem, i.e., finds in S a palindrome of length $\ell(S) \geq L(S) - E$, where $L(S)$ is the length of the longest palindrome in S. Later this algorithm will be converted into a real-time one. We store the frames $I(j)$ for some values of j in a doubly-linked list SP in the decreasing order of j's. The longest palindrome currently found is stored as a pair $answer = (pos, len)$, where pos is its initial position and len is its length. Let $t_E = \lfloor \frac{E}{2} \rfloor$.

In Algorithm ABasic we add $I(j)$ to the list SP for each j divisible by t_E. This allows us to check, at ith iteration, any factor of the form $S[kt_E \dots i]$ for being a palindrome. We assume throughout the section that at the beginning of ith iteration the frame $I(i)$ is stored in a variable I.

Algorithm 1 : Algorithm ABasic, ith iteration

```
1: if $i$ mod $t_E = 0$ then
2:     add $I$ to the beginning of $SP$
3: read $S[i]$; compute $I(i+1)$ from $I$; $I \leftarrow I(i+1)$
4: for all elements $v$ of $SP$ do
5:     if $S[v \cdot i \dots i]$ is a palindrome and $answer.len < i - v \cdot i + 1$ then
6:         $answer \leftarrow (v.i, i - v.i + 1)$
```

Proposition 3 *Algorithm ABasic finds in S a palindrome of length $\ell(S) \geq L(S) - E$ using $\mathcal{O}(n/E)$ time per iteration and $\mathcal{O}(n/E)$ space.*

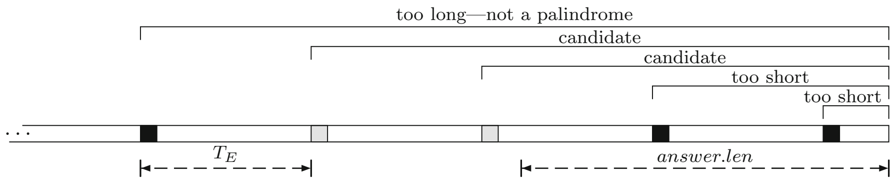

Fig. 1 Seeking for a longer palindrome. Squares indicate the numbers j such that the frame $I(j)$ is stored; brackets show substrings that can be checked for being palindromes. By Lemma 4, only substrings-"candidates" can be palindromes of length $> answer.len$

Proof Both the time and space bounds arise from the size of the list SP, which is bounded by $n/t_E = \mathcal{O}(n/E)$; the number of operations per iteration is proportional to this size due to Proposition 2. Now let $S[i \dots j]$ be a longest palindrome in S and let $j-i \geq E$ (otherwise there is nothing to prove). Let $k = \lceil \frac{i}{t_E} \rceil t_E$. Then $i \leq k < i+t_E$, and $S[[k \dots j-(k-i)]$ is a palindrome obtained from $S[i \dots j]$ by deleting $(k-i)$ letters from each end. At the kth iteration, $I(k)$ was added to SP; then the palindrome $S[k \dots j-(k-i)]$ was considered at the $(j-(k-i))$th iteration. Its length is

$$j-(k-i)-k+1 = j-i+1-2(k-i) > L(S) - 2t_E \geq L(S) - E,$$

so the longest palindrome found by Algortihm ABasic is at least this long. □

The resource to speed up Algorithm ABasic stems from the following

Lemma 4 *During one iteration, the length $answer.len$ is increased by at most $2 \cdot t_E$.*

Proof Let $S[j \dots i]$ be the longest palindrome found at the ith iteration. If $i-j+1 \leq 2t_E$ then the statement is obviously true. Otherwise the palindrome $S[j+t_E \dots i-t_E]$ of length $i-j+1-2t_E$ was considered before (at the $(i-t_E)$th iteration), and the statement holds again. □

Lemma 4 implies that at each iteration SP contains only two frames that can increase $answer.len$ (see Fig. 1). Hence we get the following Algorithm A.

Algorithm 2 : Algorithm A, ith iteration

```
1:  if $i$ mod $t_E = 0$ then
2:      add $I$ to the beginning of $SP$
3:      if $i = t_E$ then
4:          $sp \leftarrow first(SP)$
5:  read $S[i]$; compute $I(i+1)$ from $I$; $I \leftarrow I(i+1)$
6:  $sp \leftarrow previous(sp)$                                    ▷ if exists
7:  while $i - sp.i + 1 \leq answer.len$ and $(sp \neq last(SP))$ do
8:      $sp \leftarrow next(sp)$
9:  for all existing $v$ in $\{sp, next(sp)\}$ do
10:     if $S[v.i \dots i]$ is a palindrome and $answer.len < i-v.i+1$ then
11:         $answer \leftarrow (v.i, i-v.i+1)$
```

Due to Lemma 4, the cycle at lines 9–11 of Algorithm A computes the same sequence of values of $answer$ as the cycle at lines 4–6 of Algorithm ABasic. Hence

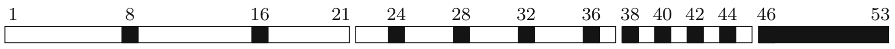

Fig. 2 The state of the list SP after the iteration $i = 53$ ($q_\varepsilon = 1$ is assumed). Black squares indicate the numbers j such that the frame $I(j)$ is currently stored. For example, (9) implies $ttl(28) = 2^{1+2+2} = 32$, so $I(28)$ will stay in SP until the iteration $28 + 32 = 60$

it finds a palindrome of required length by Proposition 3. Clearly, the space used by the two algorithms differs by a constant. To prove that an iteration of Algorithm A takes $\mathcal{O}(1)$ time, it suffices to note that the cycle in lines 7–8 performs at most two iterations. Theorem 5 is proved.

3.2 Multiplicative Error for $\varepsilon \leq 1$

Theorem 6 *There is a real-time Monte Carlo algorithm solving each instance* LPS(S) *with multiplicative error* $\varepsilon = \varepsilon(n) \in (0, 1]$ *using* $\mathcal{O}\left(\frac{\log(n\varepsilon)}{\varepsilon}\right)$ *space, where* $n = |S|$.

As in the previous section, we first present a simpler algorithm MBasic with non-linear working time and then upgrade it to a real-time algorithm. The algorithm must find a palindrome of length $\ell(S) \geq \frac{L(S)}{1+\varepsilon}$. The next lemma is straightforward.

Lemma 5 *If* $\varepsilon \in (0, 1]$, *the condition* $\ell(S) \geq L(S)(1 - \frac{\varepsilon}{2})$ *implies* $\ell(S) \geq \frac{L(S)}{1+\varepsilon}$.

We set $q_\varepsilon = \lceil \log \frac{2}{\varepsilon} \rceil$. The main difference in the construction of algorithms with the multiplicative and additive error is that here *all* frames are added to the list SP, but then, after a certain number of iterations, are deleted from it. The number of iterations the frame $I(i)$ is stored in SP is determined by the time-to-live function $ttl(i)$ defined below. This function is responsible for both the correctness of the algorithm and the space bound.

Algorithm 3 : Algorithm MBasic, ith iteration

```
1: add $I$ to the beginning of $SP$
2: for all $v$ in $SP$ do
3:     if $v.i + ttl(v.i) = i$ then
4:         delete $v$ from $SP$
5: read $S[i]$; compute $I(i+1)$ from $I$; $I \leftarrow I(i+1)$
6: for all $v$ in $SP$ do
7:     if $S[v.i \dots i]$ is a palindrome and $answer.len < i - v.i + 1$ then
8:         $answer \leftarrow (v.i, i - v.i + 1)$
```

Let $\beta(i)$ be the position of the rightmost 1 in the binary representation of i (the position 0 corresponds to the least significant bit). We define

$$ttl(i) = 2^{q_\varepsilon + 2 + \beta(i)}. \tag{9}$$

The definition is illustrated by Fig. 2. Next we state a few properties of the list SP.

Lemma 6 *For any integers $a \geq 1$ and $b \geq 0$, there exists a unique integer $j \in [a, a+2^b)$ such that $ttl(j) \geq 2^{q_\varepsilon+2+b}$.*

Proof By (9), $ttl(j) \geq 2^{q_\varepsilon+2+b}$ if and only if $\beta(j) \geq b$, i.e., j is divisible by 2^b by the definition of β. Among any 2^b consecutive integers, exactly one has this property. □

Figure 2 shows the partition of the range $(0, i]$ into intervals having lengths that are powers of 2 (except for the leftmost interval). In general, this partition consists of $m - q_\varepsilon$ intervals, which are, right to left,

$$(i - 2^{q_\varepsilon+2}, i], (i - 2^{q_\varepsilon+3}, i - 2^{q_\varepsilon+2}], \ldots, (i - 2^m, i - 2^{m-1}], (0, i - 2^m], \quad (10)$$

where $m = \lceil \log i \rceil - 1$ (if $m \leq q_\varepsilon$, there is a single interval). Lemma 6 and (9) imply the following lemma on the distribution of the elements of SP.

Lemma 7 *After each iteration, the first interval (resp., the last interval; each of the remaining intervals) in (10) contains $2^{q_\varepsilon+2}$ (resp., at most $2^{q_\varepsilon+1}$; exactly $2^{q_\varepsilon+1}$) positions for which the frames are stored in the list SP.*

The number of the intervals in (10) is $\mathcal{O}(\log(n\varepsilon))$, so from Lemma 7 and the definition of q_ε we have the following.

Lemma 8 *After each iteration, the size of the list SP is $\mathcal{O}\big(\frac{\log(n\varepsilon)}{\varepsilon}\big)$.*

Proposition 4 *Algorithm MBasic finds a palindrome of length $\ell(S) \geq \frac{L(S)}{1+\varepsilon}$ using $\mathcal{O}\big(\frac{\log(n\varepsilon)}{\varepsilon}\big)$ time per iteration and $\mathcal{O}\big(\frac{\log(n\varepsilon)}{\varepsilon}\big)$ space.*

Proof Both the time per iteration and the space are dominated by the size of the list SP. Hence the required complexity bounds follow from Lemma 8. For the proof of correctness, let $S[i \ldots j]$ be a palindrome of length $L(S)$. Further, let $d = \lfloor \log L(S) \rfloor$.

If $d < q_\varepsilon + 2$, the palindrome $S[i \ldots j]$ will be found exactly, because $I(i)$ is in SP at the jth iteration:

$$i + ttl(i) \geq i + 2^{q_\varepsilon+2} \geq i + 2^{d+1} > i + L(S) > j\,.$$

Otherwise, by Lemma 6 there exists a unique $k \in [i, i + 2^{d-q_\varepsilon-1})$ such that $ttl(k) \geq 2^{d+1}$. Hence, the palindrome $S[i+(k-i) \ldots j-(k-i)]$ will be found at the iteration $j - (k - i)$, because $I(k)$ is in SP at this iteration:

$$k + ttl(k) \geq i + ttl(k) \geq i + 2^{d+1} > j \geq j - (k - i)\,.$$

The length of this palindrome satisfies the requirement of the proposition:

$$\begin{aligned} j - (k - i) - (i + (k - i)) + 1 &= L(S) - 2(k - i) \geq L(S) - 2^{d-q_\varepsilon} \\ &\geq L(S) - L(S)/2^{q_\varepsilon} \geq L(S)(1 - \varepsilon/2)\,. \end{aligned}$$

The reference to Lemma 5 finishes the proof. □

Now we speed up Algorithm MBasic. It has two slow parts: deletions from the list SP and checks for palindromes. Lemmas 9 and 10 show that, similar to Sect. 3.1, $\mathcal{O}(1)$ checks are enough at each iteration.

Lemma 9 *Suppose that at some iteration the list SP contains consecutive elements $I(d), I(c), I(b), I(a)$. Then $b - a \le d - b$.*

Proof Let j be the number of the considered iteration. Note that $a < b < c < d$. Consider the interval in (10) containing a. If $a \in (j - 2^{q_\varepsilon+2}, j]$, then $b - a = 1$ and $d - b = 2$, so the required inequality holds. Otherwise, let $a \in (j - 2^{q_\varepsilon+2+x}, j - 2^{q_\varepsilon+2+x-1}]$. Then by (9) $\beta(a) \ge x$; moreover, any frame $I(k)$ such that $a < k \le j$ and $\beta(k) \ge x$ is in SP. Hence, $b - a \le 2^x$. By Lemma 7 each interval, except for the leftmost one, contains at least $2^{q_\varepsilon+1} \ge 4$ elements. Thus each of the numbers b, c, d belongs either to the same interval as a or to the previous interval $(j - 2^{q_\varepsilon+2+x-1}, j - 2^{q_\varepsilon+2+x-2}]$. Again by (9) we have $\beta(b), \beta(c), \beta(d) \ge x - 1$. So $c-b, d-c \ge 2^{x-1}$, implying the result. □

We want to avoid checking all frames from the list SP at line 7 of Algorithm MBasic. As in Sect. 3.1, we can save comparisons for the frames $I(a)$ where a is too big (even if $S[a \ldots n]$ is a palindrome, its length is at most len) or too small ($S[a \ldots n]$ is not a palindrome, since otherwise its "central" subpalindrome of length greater than len have been considered at one of the previous iterations). We call an element $I(a)$ of *SP valuable at ith iteration* if a is neither too big nor too small in the sense above. Thus it is enough to check the condition in line 7 only for the frames valuable at the current iteration.

Lemma 10 *At each iteration, SP contains at most three valuable frames. Moreover, if $I(d'), I(d)$ are consecutive elements of SP such that $i - d' < answer.len \le i - d$, where i is the number of the current iteration, then the valuable frames are consecutive in SP, starting with $I(d)$.*

Proof Let d be as in the condition of the lemma. If $I(d)$ is followed in SP by at most two frames, we are done. If it is not the case, let the next three frames be $I(c)$, $I(b)$, and $I(a)$, respectively. If $S[a \ldots i]$ is a palindrome then $S[a+(b-a) \ldots i-(b-a)]$ is also a palindrome. At the iteration $i-(b-a)$ the frame $I(b)$ was in SP, so this palindrome was considered by the algorithm. Hence, at the ith iteration the value $answer.len$ is at least the length of this palindrome, which is $i - a + 1 - 2(b - a)$. By Lemma 9, $b - a \le d - b$, implying $answer.len \ge i - a + 1 - (b - a) - (d - b) = i - d + 1$. This inequality contradicts the definition of d; hence, $S[a \ldots i]$ is not a palindrome. By the same argument, the frames following $I(a)$ in SP do not produce palindromes as well. Thus, only the frames $I(d)$, $I(c)$, $I(b)$ are valuable. □

Lemma 10 tells us that it is sufficient to execute lines 7–8 of Algorithm MBasic for at most three consecutive elements of SP (the picture is as in Fig. 1, but with up to three "candidates"). Now we turn to deletions. The function $ttl(x)$ has the following nice property.

Lemma 11 *The function $x \to x + ttl(x)$ is injective.*

Proof Note that $\beta(x + ttl(x)) = \beta(x)$ from the definition of *ttl*. Hence the equality $x + ttl(x) = y + ttl(y)$ implies $\beta(x) = \beta(y)$, then $ttl(x) = ttl(y)$ by (9), and finally $x = y$. $\square$

Lemma 11 implies that at most one element is deleted from SP at each iteration. To perform this deletion in $\mathcal{O}(1)$ time, we need an additional data structure. By $BS(x)$ we denote a linked list of maximal segments of 1's in the binary representation of x. For example, the binary representation of $x = 12345$ and $BS(x)$ are as follows:

13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	1	1	1	0	0	1

$BS(12345) = \{[0, 0], [3, 5], [10, 10], [12, 13]\}$

Clearly, $BS(x)$ uses $\mathcal{O}(\log x)$ space.

Lemma 12 *Both $\beta(x)$ and $BS(x + 1)$ can be obtained from $BS(x)$ in $\mathcal{O}(1)$ time.*

Proof The first number in $BS(x)$ is $\beta(x)$. Let us construct $BS(x + 1)$. Let $[a, b]$ be the first segment in $BS(x)$. If $a > 2$, then $BS(x + 1) = [0, 0] \cup BS(x)$. If $a = 1$, then $BS(x + 1) = [0, b] \cup (BS(x)\backslash[1, b])$. Now let $a = 0$. If $BS(x) = \{[0, b]\}$ then $BS(x + 1) = \{[b+1, b+1]\}$. Otherwise let the second segment in $BS(x)$ be $[c, d]$. If $c > b + 2$, then $BS(x + 1) = [b+1, b+1] \cup (BS(x)\backslash[0, b])$. Finally, if $c = b + 2$, then $BS(x + 1) = [b+1, d] \cup (BS(x)\backslash\{[0, b], [c, d]\})$. $\square$

Thus, if we support one list BS which is equal to $BS(i)$ at the end of the ith iteration, we have $\beta(i)$. If $I(a)$ should be deleted from SP at this iteration, then $i = a + ttl(a)$ and hence $\beta(a) = \beta(i)$ (see Lemma 11). The following lemma is trivial.

Lemma 13 *If $a < b$ and $ttl(a) = ttl(b)$, then $I(a)$ is deleted from SP before $I(b)$.*

By Lemma 13, the information about the positions with the same *ttl* (in other words, with the same β) is added to and deleted from SP in the same order. Hence it is possible to keep a queue $QU(x)$ of the pointers to all elements of SP corresponding to the positions j with $\beta(j) = x$. Such queues for each $x \in \{0, \dots, \lfloor\log n\rfloor\}$ constitute the last ingredient of the real-time Algorithm M presented below.

Proof of Theorem 6 After every iteration, Algorithm M has the same list SP (see Fig. 2) as Algorithm MBasic, because these algorithms add and delete the same elements. Due to Lemma 10, Algorithm M returns the same answer as Algorithm MBasic. Hence by Proposition 4 Algorithm M finds a palindrome of required length. Further, Algorithm M supports the list BS of size $\mathcal{O}(\log n)$ and the array QU containing $\mathcal{O}(\log n)$ queues of total size equal to the size of SP. Hence, it uses $\mathcal{O}(\frac{\log(n\varepsilon)}{\varepsilon})$ space in total by Lemma 8. The cycle in lines 13–14 performs at most three iterations. Indeed, let z be the value of sp after the previous iteration. Then this cycle starts with $sp = previous(z)$ (or with $sp = z$ if z is the first element of SP) and ends with $sp = next(next(z))$ at the latest. By Lemma 12, both $BS(i)$ and $\beta(i)$ can be computed in $\mathcal{O}(1)$ time. Therefore, each iteration takes $\mathcal{O}(1)$ time. $\square$

Remark Since for $n^{-0.99} \le \varepsilon \le 1$ the classes $\mathcal{O}\big(\frac{\log n}{\log(1+\varepsilon)}\big)$ and $\mathcal{O}\big(\frac{\log(n\varepsilon)}{\varepsilon}\big)$ coincide, Algorithm M uses space within a $\log n$ factor from the lower bound of Theorem 4. Furthermore, for an arbitrarily slowly growing function φ Algorithm M uses $o(n)$ space whenever $\varepsilon = \frac{\varphi(n)}{n}$.

Algorithm 4 : Algorithm M, ith iteration

```
1: add $I$ to the beginning of $SP$
2: if $i = 1$ then
3:     $sp \leftarrow first(SP)$
4: compute $BS[i]$ from $BS$; $BS \leftarrow BS[i]$; compute $\beta(i)$ from $BS$
5: if $QU(\beta(i))$ is not empty then
6:     $v \leftarrow$ element of $SP$ pointed by $first(QU(\beta(i)))$
7:     if $v = sp$ then
8:         $sp \leftarrow next(sp)$
9:     delete $v$; delete $first(QU(\beta(i)))$
10: add pointer to $first(SP)$ to $QU(\beta(i))$
11: read $S[i]$; compute $I(i+1)$ from $I$; $I \leftarrow I(i+1)$
12: $sp \leftarrow previous(sp)$                                    ▷ if exists
13: while $i - sp.i + 1 \leq answer.len$ and $sp \neq last(SP)$ do
14:     $sp \leftarrow next(sp)$
15: for all existing $v$ in $\{sp, next(sp), next(next(sp))\}$ do
16:     if $S[v.i \dots i]$ is a palindrome and $answer.len < i - v.i + 1$ then
17:         $answer \leftarrow (v.i, i - v.i + 1)$
```

3.3 Multiplicative Error for $\varepsilon > 1$

Theorem 7 *There is a real-time Monte Carlo algorithm solving each instance* LPS(S) *with multiplicative error* $\varepsilon = \varepsilon(n) \in (1, n]$ *using* $\mathcal{O}\big(\frac{\log(n)}{\log(1+\varepsilon)}\big)$ *space, where* $n = |S|$.

Our aim is to transform Algorithm M into real-time Algorithm M' which solves LPS(S) with the multiplicative error $\varepsilon > 1$ using $\mathcal{O}\big(\frac{\log n}{\log(1+\varepsilon)}\big)$ space. The basic idea of transformation is to replace all binary representations with those in base proportional to $1+\varepsilon$, and thus shrink the size of the lists SP and BS. To implement this idea, we define below the analogs of the functions $\beta(i)$ and $ttl(i)$, the lists SP and BS, and the queue QU. To distinguish analogs from their originals, we add $'$ to all notation.

First, we assume without loss of generality that $\varepsilon \geq 7$, as otherwise we can set $\varepsilon = 1$ and apply Algorithm M. Fix $k \leq \frac{1}{2}(1+\varepsilon)$ as the largest such even integer (in particular, $k \geq 4$). Let $\beta'(i)$ be the position of the rightmost non-zero digit in the k-ary representation of i. We define

$$ttl'(i) = \begin{cases} \frac{9}{2} \cdot k^{\beta'(i)} & \text{if } \beta'(i) > 0, \\ 4 & \text{otherwise.} \end{cases} \tag{11}$$

We define SP' as the list containing, after ith iteration, the frames $I(j)$ for all positions $j \leq i$ such that $j + ttl'(j) > i$. Similar to (10), we partition the range $(0; i]$ into intervals and then count the indices of frames from SP' in these intervals. The intervals are, right to left,

$$\begin{aligned} &(i-4, i], \big(i - \tfrac{9}{2}k, i-4\big], \big(i - \tfrac{9}{2}k^2, i - \tfrac{9}{2}k\big], \dots, \\ &\big(i - \tfrac{9}{2}k^m, i - \tfrac{9}{2}k^{m-1}\big], \big(0, i - \tfrac{9}{2}k^m\big], \end{aligned} \tag{12}$$

where $m = \lceil \log_k \frac{2i}{9} \rceil - 1$. We enumerate them from 0 to $m+1$.

Lemma 14 *Each interval in* (12) *contains at most 5 numbers of frames stored in* SP'. *Each of the intervals* $0, \ldots, m$ *contains at least 3 such numbers.*

Proof All 4 frames with the numbers from the 0th interval are in SP' by (11). For any $j = 1, \ldots, m+1$, a frame with the number in the jth interval is in SP' if and only if its position is divisible by k^j; see (11). The length of this interval is less than $\frac{9}{2}k^j$, giving us upper bound of $\lceil \frac{9}{2} \rceil = 5$ elements. Similarly, if $j \neq m+1$, the jth interval has the length $\frac{9}{2}k^j - \frac{9}{2}k^{j-1}$ and thus contains at least $\lfloor \frac{9}{2}\frac{k-1}{k} \rfloor$ numbers of frames in SP'. Since $k \geq 4$, the claim follows. □

Next we take Algorithm MBasic (see p. 14) and replace ttl and SP by their primed analogs. We refer to the resulting algorithm as Algorithm M'Basic.

Proposition 5 *Algorithm M'Basic finds a palindrome of length* $\ell(S) \geq \frac{L(S)}{1+\varepsilon}$ *using* $\mathcal{O}(\frac{\log n}{\log(1+\varepsilon)})$ *space.*

Proof Let $S[i \ldots j]$ be a palindrome of length $L(S)$. Let $d = \lfloor \log_k \frac{L(S)}{4} \rfloor$. Without loss of generality we assume $d \geq 0$, as otherwise $L(S) < 4 \leq ttl'(i)$ and the palindrome $S[i \ldots j]$ will be detected exactly. Since $L(S) \geq 4k^d$, let $a_1 < a_2 < a_3 < a_4 < a_5$ be consecutive positions which are multiples of k^d (i.e., $\beta'(a_1), \ldots, \beta'(a_5) \geq d$) such that $a_2 \leq \frac{i+j}{2} < a_3$. Then in particular $i < a_1$, and there is a palindrome $S[a_1 \ldots (i+j-a_1)]$ such that $a_3 \leq (i+j-a_1) < a_5$. Since $a_1 + ttl'(a_1) \geq a_5$, this particular palindrome will be detected by Algorithm M'Basic; thus $\ell(S) \geq a_3 - a_1 = 2k^d$. However, we have $L(S) < 4k^{d+1}$, hence $\frac{L(S)}{\ell(S)} < 2k \leq (1+\varepsilon)$.

Space complexity follows from bound on size of the list SP', which is at most $5\lceil \log_k \frac{2n}{9} \rceil + 1 = \mathcal{O}(\frac{\log n}{\log(1+\varepsilon)})$. □

We follow the framework of Sect. 3.2, providing an analogous speedup for Algorithm M'Basic. Consider the checks for palindromes. We adopt the same notion of a valuable frame as in Sect. 3.2: recall that a frame $I(a)$ is valuable at ith iteration if a is neither too big (making the substring $S[a \ldots i]$ too short to update the maximum) nor too small (such that $S[a \ldots i]$ cannot be a palindrome). First we need the following property, which is a more general analog of Lemma 9; an analog of Lemma 10 is then proved with its help.

Lemma 15 *Suppose that at some iteration the list* SP' *contains consecutive elements* $I(d), I(c)$, *and* $d \leq i - answer.len$, *where* i *is the number of the current iteration. Further, let* $I(a)$ *be another element of* SP' *at this iteration and* $a < c$. *If* c, d *belong to the same interval of* (12), *then* $I(a)$ *is not valuable.*

Proof Let c, d belong to the jth interval. Thus they are divisible by k^j and $d - c = k^j$. Since $a < c$, a is divisible by k^j as well. One of the numbers $\frac{d+a}{2}$, $\frac{c+a}{2}$ is divisible by k^j; take it as b. Let $\delta = b - a$. If $S[a \ldots i]$ is a palindrome, then $S[b \ldots i - \delta]$ is also a palindrome. Since at the ith iteration the left border of the jth interval was smaller than c, then at the $(i - \delta)$th iteration this border was smaller than $c - \delta \leq b$; hence, $I(b)$ was in SP' at that iteration, and the palindrome $S[b \ldots i - \delta]$ was considered there. Its length is

$$i - \delta - b + 1 = i + 1 + a - 2b \geq i + 1 + a - (d + a) \geq i - d + 1 > answer.len,$$

which is impossible by the definition of $answer.len$. So $S[a \dots i]$ is not a palindrome, and the claim follows. ◻

Lemma 16 *At each iteration, SP' contains at most three valuable elements. Moreover, if $I(d'), I(d)$ are consecutive elements of SP' such that $i - d' < answer.len \leq i - d$, where i is the number of the current iteration, then the valuable elements are consecutive in SP', starting with $I(d)$.*

Proof Let $a < b < c < d$ be such that the elements $I(d), I(c), I(b)$ are consecutive in SP' and $I(a)$ belongs to SP'. Then either b, c or c, d are in the same interval of (12), and thus a is not valuable by Lemma 15. ◻

Next we prove an analog of Lemma 11 to show that deletions from the list SP' can be performed in constant time.

Lemma 17 *The function $h(x) = x + ttl'(x)$ maps at most two different values of x to the same value. Moreover, if $h(x) = h(y)$ and $\beta'(x) \geq \beta'(y)$, then $\beta'(x) = \beta'(h(x)) + 1$ and $\beta'(y) = 0$.*

Proof Let $h(x) = h(y)$. If $\beta'(x) = \beta'(y)$ then $ttl'(x) = ttl'(y)$ by (11), implying $x = y$. Hence all preimages of $h(x)$ have different values of β'. Assume $\beta'(x) > \beta'(y)$. Then we have, for some integer j, $x = j \cdot k^{\beta'(x)}$ and $h(x) = (j+4)k^{\beta'(x)} + \frac{k}{2} \cdot k^{\beta'(x)-1}$ by (11). Since k is even, we get $\beta'(h(x)) = \beta'(x) - 1$. If $\beta'(y) > 0$, we repeat the same argument and obtain $\beta'(x) = \beta'(y)$, contradicting our assumption. Thus $\beta'(y) = 0$. The claim now follows. ◻

We also define a list $BS'(x)$, which maintains an RLE encoding of the k-ary representation of x. The list $BS'(x)$ has length $\mathcal{O}(\log_k n)$, can be updated to $BS'(x+1)$ in $\mathcal{O}(1)$ time, and provides the value $\beta'(x)$ in $\mathcal{O}(1)$ time also (we omit the proof since it is similar to Lemma 12). Further, Lemma 13 holds for the function ttl', so we introduce the queues $QU'(x)$ in the same way as the queues $QU(x)$ in Sect. 3.2. Having all these ingredients, we present Algorithm M' which speeds up Algorithm M'Basic using Lemmas 16, 17 and thus proves Theorem 7. The only significant difference between Algorithm M and Algorithm M' is in the deletion of tuples from the list (compare lines 5–9 of Algorithm M against lines 5–15 of Algorithm M').

3.4 The Case of Short Palindromes

A typical string contains only short palindromes, as Lemma 18 below shows (for more on palindromes in random strings, see [17]). Knowing this, it is quite useful to have a deterministic real-time algorithm which finds a longest palindrome exactly if it is "short", otherwise reporting that it is "long". The aim of this section is to describe such an algorithm (Theorem 8).

Lemma 18 *If an input stream $S \in \Sigma^*$ is picked up uniformly at random among all strings of length n, where $n \geq |\Sigma|$, then for any positive constant c the probability that S contains a palindrome of length greater than $\frac{2(c+1)\log n}{\log |\Sigma|}$ is $\mathcal{O}(n^{-c})$.*

Algorithm 5 : Algorithm M', ith iteration

```
1: add $I$ to the beginning of $SP'$
2: if $i = 1$ then
3:     $sp \leftarrow first(SP')$
4: compute $BS'[i]$ from $BS'$; $BS' \leftarrow BS'[i]$; compute $\beta'(i)$ from $BS'$
5: if $QU'(\beta'(i)+1)$ is not empty then
6:     $v \leftarrow$ element of $SP'$ pointed by $first(QU'(\beta'(i)+1))$
7:     if $v.i + ttl'(v.i) = i$ then
8:         if $v = sp$ then
9:             $sp \leftarrow next(sp)$
10:        delete $v$; delete $first(QU'(\beta'(i)+1))$
11: $v \leftarrow$ element of $SP'$ pointed by $first(QU'(0))$
12: if $v.i + ttl'(v.i) = i$ then
13:     if $v = sp$ then
14:         $sp \leftarrow next(sp)$
15:     delete $v$; delete $first(QU'(0))$
16: add pointer to $first(SP')$ to $QU'(\beta'(i))$
17: read $S[i]$; compute $I(i+1)$ from $I$; $I \leftarrow I(i+1)$
18: $sp \leftarrow previous(sp)$                                    ▷ if exists
19: while $i - sp.i + 1 \leq answer.len$ and $sp \neq last(SP')$ do
20:     $sp \leftarrow next(sp)$
21: for all existing $v$ in $\{sp, next(sp), next(next(sp))\}$ do
22:     if $S[v.i \dots i]$ is a palindrome and $answer.len < i-v.i+1$ then
23:         $answer \leftarrow (v.i, i-v.i+1)$
```

Proof A string S contains a palindrome of length greater than m if and only if S contains a palindrome of length $m+1$ or $m+2$. The probability P of containing such a palindrome is less than the expected number M of palindromes of length $m+1$ and $m+2$ in S. A factor of S of length l is a palindrome with probability $1/|\Sigma|^{\lfloor l/2\rfloor}$; by linearity of expectation, we have

$$M = \frac{n-m}{|\Sigma|^{\lfloor (m+1)/2\rfloor}} + \frac{n-m-1}{|\Sigma|^{\lfloor (m+2)/2\rfloor}}.$$

Substituting $m = \frac{2(c+1)\log n}{\log|\Sigma|}$, we get $M = \mathcal{O}(n^{-c})$, as required. □

Theorem 8 *Let m be a positive integer. There exists a deterministic real-time algorithm working in $\mathcal{O}(m)$ space, which, for each instance* LPS(S)*,*

- *solves* LPS(S) *exactly if $L(S) < m$;*
- *finds a palindrome of length m or $m+1$ as an approximated solution to* LPS(S) *if $L(S) \geq m$.*

To prove Theorem 8, we present an algorithm based on the Manacher algorithm [15]. We add two features: work with a sliding window instead of the whole string to satisfy the space requirements and lazy computation to achieve real time. (The fact that the original Manacher algorithm admits a real-time version was shown by Galil [7]; we adjusted Galil's approach to solve LPS.) The details follow.

We say that a palindromic factor $S[i \dots j]$ has *center* $\frac{i+j}{2}$ and *radius* $\frac{j-i}{2}$. Thus, odd-length (even-length) palindromes have integer (resp., half integer) centers and radiuses. This looks a bit weird, but allows one to avoid separate processing of these two types of palindromes. Manacher's algorithm computes, in an online fashion, an array of maximal radiuses of palindromes centered at every position of the input string S. A variation, which reports the longest palindrome in a string S as the pair $answer = (len, pos)$, is presented as Algorithm EBasic below. This variation is similar to the one of [14]. Here, n stays for the length of the input processed so far, c is the center of the longest suffix-palindrome of the processed string. The array of radiuses Rad has length $2n-1$ and its elements are indexed by all integers and half integers from the interval $[1, n]$. Initially, Rad is filled with zeroes. The left endmarker is added to the string for convenience. After each iteration, the following invariant holds: the element $Rad[i]$ has got its true value if $i < c$ and equals zero if $i > c$; the value $Rad[c] = n - c$ can increase at the next iteration. Note that the longest palindrome in S coincides with the longest suffix-palindrome of $S[1 \dots i]$ for some i. At the moment when the input stream ends, the algorithm has already found all such suffix-palindromes, so it can stop without filling the rest of the array Rad.

Algorithm 6 : Algorithm EBasic

```
1: $c \leftarrow 1$; $answer \leftarrow (1, 0)$; $n \leftarrow 1$; $S[0] \leftarrow$'#'
2: while not (end of input) do                                        ▷ iteration
3:     read($S[n+1]$)
4:     $s \leftarrow c$
5:     while $c < n+1$ do                                             ▷ main cycle
6:         $Rad[c] \leftarrow min(Rad[2*s-c], n-c)$
7:         if $c + Rad[c] = n$ and $S[c - Rad[c] - 1] = S[n+1]$ then
8:             $Rad[c] \leftarrow Rad[c] + 1$
9:             break                       ▷ longest suf-pal of $S[1 \dots n+1]$ is found
10:        $c \leftarrow c + 0.5$                            ▷ next candidate for the center
11:    $n \leftarrow n + 1$
12:    if $2 * Rad[c] + 1 > answer.len$ then
13:        $answer \leftarrow (2 * Rad[c] + 1, c - Rad[c])$
```

Remark During n iterations of Algorithm EBasic, the main cycle is executed at most $3n$ times. Indeed, each iteration executes the main cycle with the current value of c and increases c by 0.5 before each additional execution, if any. Since c never decreases, we get the mentioned bound. Each execution takes constant time, so Algorithm EBasic works in $\mathcal{O}(n)$ time but not in real time; for example, processing the last letter of the string $a^n b$ requires n executions of the main cycle.

By conditions of the theorem, we are not interested in palindromes of length $> m+1$. Thus, processing a suffix-palindrome of length m or $m+1$ we assume that the symbol comparison in line 7 fails. So Algorithm EBasic needs no access to $S[i]$ or $Rad[i]$ whenever $i < n-m$. Hence we store only recent values of S and Rad and use circular arrays CS and $CRad$ of size $\mathcal{O}(m)$ for this purpose. For example, the symbol $S[n-i]$ is stored in $CS[(n-i) \bmod (m+1)]$ during $m+1$ successive iterations and then is replaced by $S[n-i+m+1]$; the same scheme applies to the array Rad. In this way, all elements of S and Rad, needed by Algorithm EBasic, are accessible in constant time.

Further, we define a queue Q of size q for lazy computations; it contains symbols that are read from the input and await processing.

Now we describe real-time Algorithm E. It reads input symbols to Q and stops when the end of the input is reached. Each iteration consists of one read and the subsequent processing. For processing, Algorithm E runs Algorithm EBasic with Q as the input stream; a symbol is popped from Q when it is read. The processing is paused after three executions of the main cycle; the pause ends the iteration. If Algorithm EBasic stops earlier (trying to read from an empty queue), this also ends the iteration. On the next iteration, the processing resumes from the point it was stopped or paused. Note that the suffix of the input which remains in Q after the last iteration is left unprocessed. The high-level description of Algorithm E is as follows.

Algorithm 7 : Algorithm E, ith iteration

```
1: read $S[i]$ to $Q$
2: resume EBasic with $Q$ as the input stream
3: **if** EBasic stops or its main cycle has been executed 3 times **then**
4:     break
```

Proof of Theorem 8 Algorithm E is obviously real-time, so we have to check its time consumption and correctness. To analyze Algorithm E, consider the value $X = q + n - c$ after some iteration (clearly, this iteration has number $q+n$) and look at the evolution of X over time. Let Δf denote the variation of the quantity f at one iteration. Note that $\Delta(q+n) = 1$. Let us describe ΔX. First assume that the iteration contains three executions of the main cycle. Then $\Delta n = 0, 1, 2$ or 3 and, respectively, $\Delta c = 1.5, 1, 0.5$ or 0. Hence

$$\Delta X = 1 - \Delta c = 1 + (\Delta n - 3)/2 = 1 - (1 - \Delta q - 3)/2 = -(\Delta q)/2.$$

If the number of executions is one or two, then q becomes zero (and was 0 or 1 before this iteration); hence $\Delta n = 1 - \Delta q \geq 1$. Then $\Delta c \leq 0.5$ and finally $\Delta X > 0$. From these conditions on ΔX it follows that

($*$) if the current value of q is positive, then the current value of X is less than the value of X at the moment when q was zero for the last time.

Let X' be the previous value of X mentioned in ($*$). Since the difference $n - c$ does not exceed the radius of some palindrome, $X' \leq m/2$. Since $q \leq X < X'$, the queue Q uses $\mathcal{O}(m)$ space. Therefore the same space bound applies to Algorithm E.

It remains to prove that Algorithm E returns the same pair (len, pos) as Algorithm EBasic with a sliding window, in spite of the fact that Algorithm E stops earlier. Suppose that Algorithm E stops with $q > 0$ after n iterations. Then the longest palindrome that could be found by processing the symbols in Q has the radius $X = n + q - c$. Now consider the iteration mentioned in ($*$) and let n' and c' be the values of n and c after it; so $X' = n' - c'$. Since q was zero after that iteration, the processing phase on this iteration included reading the symbol $S[n']$ from Q and subsequent execution of the main cycle; during this execution Algorithm EBasic tried to extend a suffix-palindrome of $S[1...n'-1]$ with the center $c'' \leq c'$. If this extension was successful, then a palindrome of radius at least X' was found. If it was unsuccessful, then $c' \geq c'' + 1/2$ and hence $S[1...n'-1]$ has a suffix-palindrome of length at least $X' - 1/2$. Thus, a palindrome of length $X \leq X' - 1/2$ is not longer than a longest palindrome seen before, and processing the queue cannot change the pair (len, pos). Thus, Algorithm E is correct. The theorem is proved. □

Remark Lemma 18 and Theorem 8 show a practical way to solve LPS. Algorithm E is fast and lightweight ($2m$ machine words for the array Rad, m symbols in the sliding window and at most m symbols in the queue; compare to 17 machine words per one frame in the Monte Carlo algorithms). So it makes direct sense to run Algorithm M and Algorithm E, both in $\mathcal{O}(\log n)$ space, in parallel. Then either Algorithm E will give an exact answer (which happens with high probability if the input stream is a "typical" string) or both algorithms will produce approximations: one of fixed length and one with an approximation guarantee (modulo the hash collision).

References

1. Apostolico, A., Breslauer, D., Galil, Z.: Parallel detection of all palindromes in a string. Theor. Comput. Sci. **141**, 163–173 (1995)
2. Berenbrink, P., Ergün, F., Mallmann-Trenn, F., Azer, E.S.: Palindrome recognition in the streaming model. In: STACS 2014, LIPIcs, vol. 25, pp. 149–161. SchlossDagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl Publishing, Germany (2014)
3. Breslauer, D., Galil, Z.: Real-time streaming string-matching. In: Combinatorial Pattern Matching. LNCS, vol. 6661, pp. 162–172. Springer, Berlin (2011)
4. Clifford, R., Fontaine, A., Porat, E., Sach, B., Starikovskaya, T.A.: Dictionary matching in a stream. In: ESA 2015, LNCS, vol. 9294, pp. 361–372. Springer (2015)
5. Clifford, R., Fontaine, A., Porat, E., Sach, B., Starikovskaya, T.A.: The k-mismatch problem revisited. In: SODA 2016, pp. 2039–2052. SIAM (2016)
6. Ergün, F., Jowhari, H., Saglam, M.: Periodicity in streams. In: RANDOM 2010, LNCS, vol. 6302, pp. 545–559. Springer (2010)
7. Galil, Z.: Real-time algorithms for string-matching and palindrome recognition. In: Proceedings of 8th annual ACM symposium on theory of computing (STOC'76), pp. 161–173. ACM, New York (1976)
8. Galil, Z., Seiferas, J.: A linear-time on-line recognition algorithm for "Palstar". J. ACM **25**, 102–111 (1978)
9. Gawrychowski, P., Manea, F., Nowotka, D.: Testing generalised freeness of words. In: STACS 2014, LIPIcs, vol. 25, pp. 337–349. Dagstuhl Publishing (2014)
10. Gawrychowski, P., Merkurev, O., Shur, A.M., Uznanski, P.: Tight tradeoffs for real-time approximation of longest palindromes in streams. In: Proceedings CPM 2016, LIPIcs, vol. 54, pp. 18:1–18:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
11. Jalsenius, M., Porat, B., Sach, B.: Parameterized matching in the streaming model. In: STACS 2013, LIPIcs, vol. 20, pp. 400–411. Dagstuhl Publishing (2013)
12. Karp, R., Rabin, M.: Efficient randomized pattern matching algorithms. IBM J. Res. Dev. **31**, 249–260 (1987)
13. Knuth, D.E., Morris, J., Pratt, V.: Fast pattern matching in strings. SIAM J. Comput. **6**, 323–350 (1977)

14. Kosolobov, D., Rubinchik, M., Shur, A.M.: Finding distinct subpalindromes online. In: Proceedings of Prague Stringology conference. PSC 2013, pp. 63–69. Czech Technical University in Prague (2013)
15. Manacher, G.: A new linear-time on-line algorithm finding the smallest initial palindrome of a string. J. ACM **22**(3), 346–351 (1975)
16. Porat, B., Porat, E.: Exact and approximate pattern matching in the streaming model. In: FOCS 2009, pp. 315–323. IEEE Computer Society (2009)
17. Rubinchik, M., Shur, A.M.: The number of distinct subpalindromes in random words. Fundam. Inform. **145**, 371–384 (2016)
18. Yao, A.C.C.: Probabilistic computations: toward a unified measure of complexity (extended abstract). In: FOCS 1977, pp. 222–227. IEEE Computer Society (1977)

Affiliations

Paweł Gawrychowski[1] · **Oleg Merkurev**[2] · **Arseny M. Shur**[2] · **Przemysław Uznański**[3]

Oleg Merkurev
o.merkuryev@gmail.com

Arseny M. Shur
arseny.shur@urfu.ru

Przemysław Uznański
przemyslaw.uznanski@inf.ethz.ch

1 Institute of Computer Science, University of Wrocław, ul. Joliot-Curie 15, 50-383 Wrocław, Poland

2 Department of Algebra and Fundamental Informatics, Ural Federal University, pr. Lenina 51, Yekaterinburg, Russia 620000

3 Department of Computer Science, ETH Zürich, Universitätstrasse 6, 8092 Zürich, Switzerland