Tight tradeoffs for approximating palindromes in streams

Paweł Gawrychowski and Przemysław Uznanski

1Max-Planck-Institut für Informatik, Saarbrücken, Germany
2LIF, CNRS – Aix Marseille University

October 24, 2014

Abstract

We consider the question of finding the longest palindrome in a text of length \( n \) in the streaming model, where the characters arrive one-by-one, and we cannot go back and retrieve a previously seen character. While it is easy to see that one cannot compute the answer exactly using sublinear memory in such a setting, one can hope for a good approximation guarantee.

We focus on the two most natural variants, where we aim for either additive or multiplicative approximation of the length of the longest palindrome. We first show, that there is no point in considering either deterministic or Las Vegas algorithms in such a setting, as they cannot achieve sublinear space complexity. For Monte Carlo algorithms, we provide a lowerbound of \( \Omega\left(\frac{n}{E}\right) \) bits for approximating the answer with an additive error \( E \), and \( \Omega\left(\frac{\log n}{\varepsilon}\right) \) bits for approximating the answer within a multiplicative factor \( (1 + \varepsilon) \). Then we construct a generic Monte Carlo algorithm, which by choosing the parameters appropriately achieves space complexity matching up to a logarithmic factor for both variants. This substantially improves the previous results [Berenbrink et al., STACS 2014] and essentially settles the space complexity of the problem.

1 Introduction

A recent trend in algorithms on strings is to develop efficient solutions in the streaming model, where the input arrives character-by-character, and we cannot go back to retrieve a previously seen character. We want to avoid storing the already seen part of the input explicitly, and aim to achieve small space complexity. On the other hand, we are perfectly happy with randomized solutions, i.e., the answer should be correct with high probability.

*up to a logarithmic factor.
The most classical problem in stringology is pattern matching, where we are given a pattern and a text, and want to detect all occurrences of the former in the latter. While it is well-known that it can be solved in linear time (and even just constant additional space) in the classic setting, where we process the text character-by-character, but can access any letter of the pattern whenever we want, it is somewhat surprising that one can actually solve it using just a polylogarithmic space in the streaming model. This was first proved by Porat and Porat [9]. A simpler solution was later given by Ergün et al. [4], and finally Breslauer and Galil [3] constructed a real-time algorithm in such setting. Similar question were also studied for different approximate version of pattern matching.

Pattern matching is very closely related to detecting periodicities, and in fact Ergün et al. [4] also show an efficient algorithm for computing the period, where \( p \) is a period of \( T[1..n] \) if \( T[i] = T[i + p] \) for all \( i = 1, 2, \ldots, n - p \), and the period is the smallest period. Periods are the most basic notion of regularities. Another basic and well-studied notion is the one of palindromes, defined as fragments which are the same when read in both directions. Here the most natural question is the one of finding the longest palindrome inside the text. As proved by Manacher [8], this can be done in linear time. Later the question has been extensively studied in the parallel setting, see Apostolico et al. [1] and the references therein.

Hence the natural question is whether we can compute the longest palindrome in the streaming model. This has been recently studied by Berenbrink et al. [2], who developed tradeoffs between the bound on the error and the space complexity for the additive and multiplicative variant of the problem, that is, for approximating the length of the longest palindrome with either an additive or multiplicative error. Their algorithms were Monte Carlo, i.e., returned correct answer with high probability. They also proved that any Las Vegas algorithm achieving an additive error \( E \) must necessarily use \( \Omega \left( \frac{n}{E} \log |\Sigma| \right) \) bits of memory, which matches the space complexity of their solution up to a logarithmic factor in the \( E \in [1, \sqrt{n}] \) range, but leaves at least two questions. Firstly, does the lowerbound still hold for Monte Carlo algorithm? Secondly, what is the best possible space complexity when \( E \in (\sqrt{n}, n] \) range in the additive variant, and what about the multiplicative version? We answer all these questions, essentially settling the space complexity of approximating the length of the longest palindrome in the streaming model.

**Model.** We work in the streaming model and consider two variants of the problem, called additive and multiplicative, respectively. The model works as follows: we are first given the length of the text \( n \) and the bound on the desired error \( E \) (in the additive variant) or \( \varepsilon \) (in the multiplicative variant), then the characters \( T[1], T[2], \ldots, T[n] \in \Sigma \) arrive one-by-one, i.e., in the \( h \)-th step we get \( T[h] \). After having seen the whole input we are required to output a number \( \ell \), such that the length of the longest palindrome in \( T[1..n] \) is either between \( \ell \) and \( \ell + E \) (in the additive variant) or \( \ell \) and \( (1 + \varepsilon) \cdot \ell \) (in the multiplicative variant).
variant). We have \( s(n) \) bits of memory available, where we can store an arbitrary data. It is important to remember that the whole procedure operates in steps corresponding to the characters, i.e., in the \( h \)-th step we can only access \( T[h] \) and the memory.

Now we are interested in the possible tradeoffs between \( s(n) \) and the bound on the error. Intuitively, the larger the allowed error is, the less space we need, but what is the exact relation? We consider Las Vegas and Monte Carlo algorithms. A Las Vegas algorithm always returns a correct answer, but its memory usage \( s(n) \) is a random variable with a hopefully small expected value. Finally, a Monte Carlo algorithm returns a correct answer with high probability, and its memory usage \( s(n) \) does not depend on the random choices, where high probability means \( 1 - \frac{1}{n^c} \).

We assume that \( |\Sigma| \) is at most polynomial in \( n \), and when analyzing the space (and time) complexity of our algorithms assume that the memory consists of words of size \( \Theta(\log n) \). All basic arithmetical operations, including addition, subtraction, multiplication, integer division, and counting trailing zeroes, take \( O(1) \) time on such words.

**Previous work.** Berenbrink et al. \cite{2} constructed a streaming algorithm achieving additive error \( E \) using \( \mathcal{O}(\frac{n}{E}) \) space and \( \mathcal{O}(\frac{n^{1.5}}{E}) \) total time for any \( E \in [1, \sqrt{n}] \), and a streaming algorithm guaranteeing multiplicative error \( (1 + \varepsilon) \) using \( \mathcal{O}(\frac{\log n}{\varepsilon^2}) \) space and \( \mathcal{O}(\frac{n \log n}{\varepsilon^2}) \) total time for any \( \varepsilon \in (0, 1] \). Both algorithms were Monte Carlo, meaning that the approximation guarantee holds with probability \( 1 - \frac{1}{n^c} \). They also proved that additive error \( E \) cannot be guaranteed in smaller space using a Las Vegas algorithm in the \( E \in [1, \sqrt{n}] \) range.

**Our results.** We significantly improve on the previous results as follows and essentially settle the space complexity of the problem in both variants.

Firstly, we prove that any Las Vegas algorithm approximating (in either variant) the length of the longest palindrome inside a text of length \( n \) over an alphabet \( \Sigma \) must necessarily use \( \Omega(n \log |\Sigma|) \) bits of memory (see Theorem 6.2). Hence Las Vegas randomization is simply not the right model for this particular problem. Then we move to Monte Carlo algorithms, and prove the following lowerbounds on their space complexity:

- \( \Omega(\log n) \) bits to achieve additive error \( E \) with high probability if \( E \leq \frac{n}{8} \) (see Theorem 6.8),
- \( \Omega(\frac{n}{E} \log |\Sigma|) \) bits to achieve additive error \( E \) with high probability if \( E = \mathcal{O}(\frac{n}{\log n}) \) (see Theorem 6.8),
- \( \Omega(\frac{n}{E} \log |\Sigma|) \) bits to achieve multiplicative error \( 1 + \varepsilon \) with high probability if \( \varepsilon = \Omega(n^{-0.99}) \) (see Theorem 6.9).

*Formally, \( s(n) = \mathcal{O}(f(n)) \) with high probability if for any \( c \) there exists \( \gamma(c) \) such that \( s(n) \leq \gamma(c) f(n) \).

†They state their result as \( \mathcal{O}(\frac{\log n}{\varepsilon \log(1+\varepsilon)}) \), but for \( \varepsilon \in (0, 1] \) we have \( \log(1 + \varepsilon) = \Theta(\varepsilon) \).
Secondly, we construct a generic Monte Carlo approximation algorithm, which by adjusting the parameters appropriately matches our lowerbound up to a logarithmic multiplicative factor. In more detail, our algorithm uses $O\left(\frac{n}{r} + \log n\right)$ space in the additive variant (see Theorem 3.1 and Theorem 4.7) and $O\left(\frac{\log(n\varepsilon^2)}{\varepsilon^2}\right)$ space in the multiplicative variant (see Theorem 3.2 and Theorem 4.7), where the space is measured in words consisting of $\Theta(\log n)$ bits. This essentially settles the space complexity of the problem, as it can be seen that our lower- and upperbounds differ by at most a logarithmic factor. Moreover, the time complexity of our algorithm is always $O(n \log n)$, irrespectively of the choice of the parameters (see Theorem 5.8).

Overview of the methods. As usual in the streaming model, we apply hashing, more specifically Karp-Rabin fingerprints. We store such fingerprints for some carefully chosen prefixes of the already seen part of the text. Informally, these chosen prefixes become more and more sparse as we move closer to the beginning, with the details depending on the variant. This is formalized in Section 2. Then, in Section 3, we present a generic algorithm, which maintains for every potential palindrome a separate process, which uses the fingerprints to compute its radius. We show how to choose the parameters of that generic algorithm as to guarantee good bound on the error in both variants, and then move to actually implementing it in small space in Section 4. The intuition is that multiple sufficiently long palindromes appearing close to each other imply a certain periodicity of the corresponding fragment of the text, which can be exploited to speed up the algorithms. This insight (dating way back to Apostolico, Breslauer and Galil [1], who used it to parallelize detecting palindromes) allows us to simulate the generic algorithm using just logarithmic additional space on the top of the space used by the landmarks. To make the simulation time-efficient, we need a number of additional ideas presented in Section 5. Finally, in Section 6 we apply Yao’s minimax principle to derive the lowerbounds. For Las Vegas algorithms, this is straightforward, but requires more work for Monte Carlo algorithms.

2 Preliminaries

For a word $w$, we denote its length by $|w|$, and its $i$-th letter by $w[i]$ for any $i = 1, 2, \ldots, |w|$. Similarly, $w[i..j]$ denotes its fragment starting at the $i$-th and ending at the $j$-th character, and $w^R$ denotes its reversal, i.e., $w[i]w[i+1]w[1]$.. $w[1]$. The period $\text{per}(w)$ of $w$ is the smallest natural number such that $w[i] = w[i + \text{per}(w)]$ for all $i = 1, 2, \ldots, |w| - \text{per}(w)$. The well-known periodicity lemma states that if $p$ and $q$ are periods of $w$ and $p + q \leq |w|$, then $\gcd(p, q)$ is also a period of $w$.

We focus on detecting even palindromes. All solutions can be extended to deal with odd palindromes with the standard trick of duplicating every letter, as done in [1]. The palindromic radius at $c$ is the largest $R(c)$ such that $T[c..(c+}$
that a fingerprint \( \Phi( \cdot ) \) which takes \( O(1) \) space assuming that \( |w| \leq n \). One can perform the following operations on such fingerprints in \( O(1) \) time:

- **concatenation** given \( \Phi(w) \) and \( \Phi(v) \), find \( \Phi(wv) \),
- **erasing a prefix** given \( \Phi(wv) \) and \( \Phi(w) \), find \( \Phi(v) \),
- **erasing a suffix** given \( \Phi(wv) \) and \( \Phi(v) \), find \( \Phi(w) \),
- **reversal** given \( \Phi(w) \), find \( \Phi(w^R) \).

Additionally, given \( \Phi(w^k) \) and \( k \) we can compute \( \Phi(w) \) in \( O(\log n) \) time.

**Lemma 2.1.** Given \( \Phi(w^k) \) and \( k \), we can compute \( \Phi(w) \) in \( O(\log n) \) time.

**Proof.** First we need to compute \( x^{|w|} \) given \( x^{|w|} \). We calculate \( \alpha = k^{-1} \mod (p - 1) \) using the extended Euclidean algorithm in \( O(\log k) \) time, then from the Fermat’s little theorem \( x^{\alpha k^{|w|}} = x^{|w|} \), and \( x^{\alpha k^{|w|}} \) can be computed from \( x^{|w|} \) in \( O(\log \alpha) = O(\log n) \) time. Computing \( x^{-|w|} \) is done similarly, so the remaining part is to show how to calculate \( f_x(w) \) given \( f_x(w^k) \). From the definition:

\[
f_x(w^k) = f_x(w)(1 + x^{|w|} + x^{2|w|} + \ldots + x^{(k-1)|w|}) = f_x(w)x^{|w|} - 1 \mod p,
\]

so we can calculate (again using the extended Euclidean algorithm) \( (x^{|w|} - 1)^{-1} \mod p \) in \( O(\log n) \) time and then finally extract \( f_x(w) \) from \( f_x(w^k) \) with two multiplications. \qed

The fingerprints allow us to check if two strings are the same. Formally, to check if \( u = v \) we compare \( \Phi(u) \) and \( \Phi(v) \). If \( u = v \) then \( \Phi(u) = \Phi(v) \), and if \( u \neq v \) while \( |u|, |v| \leq n \) then \( \Phi(u) = \Phi(v) \) with probability at most \( \frac{1}{n^2} \). The latter situation is called a false positive. Because the running time of our algorithms will be always polynomial in \( n \), and we will be operating on strings of length at most \( n \), by the union bound the probability of a false positive can be made \( \frac{1}{n^c} \) for any \( c \) by choosing \( m \) large enough.

**Landmarks.** Our algorithms need to store some values of \( \Phi(i) \), where \( \Phi(i) = \Phi(T[1..i]) \). The intuition is that after reading \( T[h] \) we calculate the hash of the currently seen prefix \( \Phi(h) \) and keep it available for some time. Formally, we define \( \mathcal{Y} = \bigcup_{\lambda} Y_\lambda \) as the set of all landmarks, where \( Y_\lambda \) is the set of landmarks on level \( \lambda \), meaning that they are of the form \( 2^\lambda \cdot j \). We also define the set of
landmarks strictly on level $\lambda$, meaning that they are on level $\lambda$, and not on level $\lambda + 1$. We maintain the value of $\Phi(t)$ for all $t \in \mathcal{Y}$, thus allowing us to find $\Phi(T[t+1..t'])$ for any $t, t' \in \mathcal{Y}$ in $O(1)$ time. Technically, $\mathcal{Y}$ depends on the current value of $h$, so we will be saying that $t$ is a landmark at $h$, meaning that $t \in \mathcal{Y}$ just after reading $T[h]$.

For each level of landmarks we fix its size $b_\lambda$, and keep only $b_\lambda$ most recently seen landmarks on that level. Thus, after reading $T[h]$, the $\lambda$-th level consists of positions $2^\lambda(\lfloor \frac{h}{2^\lambda} \rfloor - i) = h - (h \mod 2^\lambda) - 2^\lambda \cdot i$ for $i = 0, 1, \ldots, b_\lambda - 1$. The values $b_\lambda$ are chosen depending on the whether we aim for additive or multiplicative approximation. In both cases, we require that either $\lambda$ is the last level and we have all possible landmarks there, or the leftmost landmark on level $\lambda + 1$ is strictly on the left of all landmarks on level $\lambda$. We also require that there is an $L$ such that $b_0 = b_1 = \ldots = b_{L-1}$, and $L$ is the last level, where possibly $b_L$ is larger.

Under such definition of landmarks, after increasing $h$ by one, we need to add and remove at most one landmark per every level, which is $O(\log h)$ in total.

The landmarks on each level are kept in a random access array with cyclic addressing, thus using $O(b_\lambda + 1)$ space while allowing accesses and updates in $O(1)$ time.

### 3 Basic algorithm

We start with the basic algorithm. A proper choice of all $b_\lambda$ will guarantee a small additive or multiplicative error, even though the time and space complexity will be very high. Nevertheless, the basic algorithm will serve as a good starting point for developing first the space efficient version, and then finally the time and space efficient solution.

The idea of the algorithm is that for every possible center $c$ we create a process $P(c)$, which keeps on computing the corresponding radius $R(c)$. We call a process alive if it has not found $T[c + \Delta]$ such that $T[c - 1 - \Delta] \neq T[c + \Delta]$ yet, and dead otherwise. The process starts with $R(c) = 0$ and then uses the landmarks to update the value of $R(c)$ (and also the final answer) whenever possible. To verify if $R(c) \geq h - c + 1$ we need to compare $\Phi(T[c..h])$ and $\Phi(T[(2c-h-1)..(c-1)])$. This requires accessing $\Phi(h)$, $\Phi(c-1)$ and $\Phi(2c-h-2)$, see Fig. 1. We can simply maintain the current value of $\Phi(h)$ and store $\Phi(c-1)$ when the process is created, but retrieving $\Phi(2c-h-2)$ is only possible when $2c-h-2$ is a landmark. Therefore, the process can update its $R(c)$ only when $2c-h-2$ is a landmark. Allowing $P(c)$ to update its current value of $R(c)$ when $2c-h-2$ is a landmark will be referred to as running $P(c)$ using $2c-h-2$. If $T[(2c-h-1)..h]$ is a palindrome, we say that $P(c)$ succeeds, and otherwise we say that it fails.

We would like to guarantee that running a process is a $O(1)$ time procedure, so we need to quickly check if $2c-h-2$ is currently a landmark (and if so, access the stored $\Phi(2c-h-2)$). This can be easily done by iterating through all possible levels, but we want a faster method. We consider the last $L$-th level
separately in \( O(1) \) time. For all lower levels, the values \( b_\lambda \) are all the same. We compute the largest power of 2 dividing \( 2c - h - 2 \), call it \( 2^\lambda \), then \( 2c - h - 2 \) cannot be a landmark on level larger than \( \lambda \). On the other hand, if \( 2c - h - 2 \) is a landmark on level \( \lambda' < \lambda \), then it is also a landmark on level \( \lambda \). Therefore, we only need to consider the \( \lambda \)-th level. This allows us to run any process in \( O(1) \) time, and furthermore the state of any \( P(c) \) can be fully described by specifying \( \Phi(c - 1) \) (\( R(c) \) is not stored explicitly, unless mentioned otherwise). Even if a process is dead already, there is no harm in running it anyway, as it will fail. Such situation will be called running a zombie.

Scheduling scheme. The basic algorithm simply runs all processes after reading the next \( T[h] \). Of course this is very slow and space consuming, as we need to store all \( \Phi(c - 1) \). In the next sections we will modify the basic algorithm as to run just a few processes. Such optimized version will be referred to as a scheduling scheme. A scheduling scheme should guarantee that any alive \( P(c) \) such that \( 2c - h - 2 \) is a landmark is run, unless we can either be sure that it would fail anyway, or there is another \( P(c') \) such that \( c' < c \) and \( 2c' - h - 2 \) is also a landmark, and we can be sure that it succeeds (in particular, \( P(c') \) is still alive).

**Theorem 3.1.** Any scheduling scheme with \( b_L = \infty \) approximates the longest palindrome with an additive error \( 2^L \).

**Proof.** Consider an arbitrary palindrome \( T[(c - x)\ldots(c + x - 1)] \). We will show that any scheduling scheme with \( b_L = \infty \) returns at least \( x - 2^L \). We can assume \( x \geq 2^L \), then there exists \( y \in (x - 2^L, x) \) such that \( 2^L \mid c - y - 1 \). Therefore, \( c - y - 1 \) is permanently a landmark on level \( L \). \( P(c) \) will be alive at \( c + y - 1 \), so by the properties of a scheduling scheme we will run a process detecting a palindrome with radius at least \( y > x - 2^L \). Thus any scheduling scheme with \( b_L = \infty \) approximates the longest palindrome with an additive error \( 2^L \). \( \square \)

**Theorem 3.2.** Any scheduling scheme with \( b_0 = b_1 = \ldots, b_{\log(n/D)} = D \) for \( D \geq 6 \) approximates the longest palindrome with a multiplicative error \( 1 + \mathcal{O}(1/D) \).

**Proof.** Consider an arbitrary palindrome \( T[(c - x)\ldots(c + x - 1)] \). We will show that any such scheduling scheme returns at least \( x/(1 + \mathcal{O}(1/D)) \). Let \( \lambda \) be the smallest integer such that \((D - 1) \cdot 2^\lambda \geq 2 \cdot x \). We have two cases.

\( \lambda = 0 \) After reading \( T[c + x - 1] \), all \( c + x - 1, c + x - 2, \ldots, c + x - D \) are landmarks on level 0, and \( c - x - 1 \) is one of them because \( 2x < D \), so \( T[(c - x)\ldots(c + x - 1)] \) or a longer palindrome is detected.
\[ \lambda > 0 \text{ In the interval } [c-x-1, c+x-1] \text{ there are at most } \left\lceil \frac{2x}{x} \right\rceil + 1 \leq D \] numbers divisible by \( 2^\lambda \), thus there exists \( y \in (x - 2^\lambda, x] \) such that \( c-y-1 \) was a landmark on level \( \lambda \) after reading \( T[c+x-1] \). As \( P(c) \) is still alive at \( c+x-1 \), we will detect a palindrome of radius at least \( y \). In other words, we will approximate \( x \) with an additive error \( 2^\lambda \). However, since \( \lambda \) was chosen to be minimal, we have that \( 2 \cdot x > (D-1) \cdot 2^\lambda - 1 \), so we can bound the multiplicative error from above by \( \frac{x}{x-2^\lambda} \), which is at most \( 1 + \Omega \left( \frac{1}{D} \right) \) for \( D \geq 6 \).

Therefore, any such scheduling scheme approximates the longest palindrome with multiplicative error \( 1 + \Omega(1/D) \).

\[ \text{Lemma 3.3. Assuming that } b_\lambda \geq 12 \text{ for all } \lambda \leq L, \text{ for any } \Delta = 2^\ell - 1 \text{ and for any } c \text{ there is at least one } h \in [c+5\Delta, c+6\Delta] \text{ such that } 2c-h-2 \text{ is a landmark after reading } T[h]. \]

\[ \text{Proof. Observe that there exists unique } h \in [c+5\Delta, c+6\Delta] \text{ such that } 2^\ell \mid 2c-h-2. \text{ Because } h-(2c-h-2) \leq 2+12\cdot\Delta < 12 \cdot 2^\ell, \text{ after reading } T[h] \text{ there are two possibilities. If } \ell \leq L \text{ then } 2c-h-2 \text{ is among the 12 last seen positions divisible by } 2^\ell. \text{ If } \ell > L \text{ then } 2c-h-2 \text{ is definitely a landmark anyway.} \]

\section{Space-efficient algorithm}

The disadvantages of the basic algorithm from the previous section is that, in the worst case, there might be \( \Theta(n) \) alive process which need to be maintained. However, our implementation of landmarks was already space efficient, using \( O(\frac{n}{\varepsilon}) \) and \( O\left( \frac{\log(n\varepsilon)}{\varepsilon} \right) \) space in the additive and the multiplicative version, respectively. Thus we only need to store the information about the alive processes in a compressed form.

\textbf{Partition scheme.} We maintain a partition of \( T[1..h] \) into a number of disjoint segments stored in a linked list. The length of every segment is a power of 2, their lengths are nonincreasing as one moves to the right, and there is \( M \) such that we have between \( A \) and \( B \) segments of length \( 2^\ell \) for every \( \ell = 0, 1, \ldots, M-1 \), and between 1 and \( B \) segments of length \( 2^{4\ell} \), where \( A \) and \( B \) are constants to be specified later. Whenever we increase \( h \) by one, a new segment of length \( 2^0 \) appears, and then we possibly take two adjacent segments of length \( 2^\ell \) such that the segment on their left (if any) is longer, and merge them into one segment of length \( 2^{\ell+1} \). We call this a partition scheme, as there is some flexibility as to when the merging happens.

\textbf{Lemma 4.1.} There is a partition scheme with \( A = 3 \) and \( B = 5 \), which guarantees that after adding a new segment of length \( 2^0 \) we can merge in \( O(1) \) time at most one pair of adjacent segments of length \( 2^\ell \), such that there are 3 segments of the same length \( 2^\ell \) on their right.
**Proof.** This can be seen as a simple example of the recursive slow-down method of Kaplan and Tarjan [6]. Let $2^{a_1}, 2^{a_2}, 2^{a_3}, \ldots$ be the lengths of the segments in the current partition, where $a_1 \leq a_2 \leq a_3 \leq \ldots$. We group together all segments with the same length, and denote the number of segments of length $2^\ell$ by $c_\ell$. We will show how to maintain $c_\ell \in \{3, 4, 5\}$ for every $\ell = 0, 1, 2, \ldots, M - 1$ and $c_M \in \{1, 2, 3, 4, 5\}$, where $2^M$ is the maximum length of a segment in the current partition. To this end, we will keep the following invariant: if $c_\ell = 5$ then there exists $j \in \{0, 1, \ldots, i - 1\}$ such that $c_j = 3$ and $c_{j+1} = \ldots = c_{i-2} = c_{i-1} = 4$. We call such partition valid.

We must show that, given a valid partition of $T[1..h]$, we can construct in $O(1)$ time a valid partition of $T[1..h+1]$. We start with creating a new segment of length $2^0$ and adding it to the previous partition, which increases $c_0$ by one. Now there are two cases.

$c_0 = 5$ We merge two (leftmost) segments of length $2^0$ into a segment of length $2^1$, or in other words we decrease $c_0$ by two ($c_0$ is now equal to 3) and increase $c_1$ by one. Because the initial value of $c_0$ was 4, the only way the invariant could have been broken is that $c_1$ was 3, $c_2 = \ldots = c_{i-1} = 4$ and $c_i = 5$ for some $i \geq 3$. But then the new $c_1$ becomes 4, and all $c_2, c_3, \ldots, c_{i-1}$ are now 4, so the invariant holds.

$c_0 = 4$ If there is $i$ such that $c_1 = \ldots = c_{i-2} = c_{i-1} = 4$ and $c_i = 5$, then we merge the two (leftmost) segments of length $2^i$ into a segment of length $2^{i+1}$, which decreases $c_i$ by two and increases $c_{i+1}$ by one. As in the previous case, the only way the invariant could have been broken is that $c_{i+1}$ was 3, $c_{i+2} = c_{i+3} = \ldots = c_{j-1} = 4$ and $c_j = 5$ for some $j \geq i + 2$. Then $c_i$ becomes 3, and all $c_{i+1}, c_{i+2}, \ldots, c_{j-1}$ are now 4, so the invariant holds.

To implement the update, we group together all consecutive $i$’s with the same value of $c_i$. In other words, we store a list of lists of segments. This allows us to find $i$ from the second case in $O(1)$ time.

Instead of storing $\Phi(c-1)$ for every alive $P(c)$ separately, for every segment group together all alive processes $P(c)$ such that $c$ lies inside the segment. We need the following result, which follows from a definition of a palindrome, see [1].

**Lemma 4.2.** If $c < c’, c’ - c \leq 2^\ell$ and $R(c), R(c’) \geq 2^\ell$, then $2(c’ - c)$ is a period of $T[(c - 2^\ell)..(c’ + 2^\ell - 1)]$.

The intuition is that if there are at least 5 alive processes in a single segment of length $2^\ell$, then the whole segment is periodic with period at most $\frac{1}{2}2^\ell$. This suggest the following strategy to compress the information about alive processes. We distinguish between two types of segments, and store either a sparse or a dense description of each segment. How a particular segment is described will not change during the execution of our algorithm.
Figure 2: Alive processes inside $s$ with a dense description are of the form $P(c + \alpha |w|)$.

**Sparse description.** We explicitly store a list of all processes inside the segment which can be potentially still alive. We guarantee that there are at most 4 processes on that list, and that if a process is not on the list, it is surely dead. We do not guarantee that all processes on the list are still alive, but whenever we run one of them and the check fails, we declare it dead and remove from the list.

**Dense description.** We guarantee that there exists a word $w$ such that $|w| \leq \frac{1}{4}2^k$ for which the whole segment of length $2^k$ is a subword of $(ww^R)\infty$, see Fig. 2. Denoting the segment by $s$, this implies that $\text{per}(s) \leq \frac{1}{2}|s|$ and $s$ has a palindromic subword of length $\text{per}(s)$. In such a case we store a multiple of the period, denoted by $p = k \text{per}(s) \leq \frac{1}{2}2^k$, such that the only alive processes inside the segment are of the form $P(c + \alpha p)$ for $\alpha \geq 0$, where $T[c..(c+p-1)]$ is an even palindrome fully within the segment. (We do not require that all such processes are still alive.) We store $\Phi(c-1)$ and $\Phi(T[c..(c+p-1)])$. We also either guarantee that all processes of the form $P(c + \alpha p)$ with odd $\alpha$ inside the segment are dead, or we also store $\Phi(T[c..(c+p\frac{p}{2}-1)])$. Then we can construct any relevant process inside $s$ in $O(\log |s|)$ time using binary exponentiation, where relevant means either of the form $P(c + \alpha p)$ or $P(c + \alpha p)$, depending on whether we do or do not store $\Phi(T[c..(c+p\frac{p}{2}-1)])$. No other process can be alive inside $s$.

Notice that irrespectively of the type of a description of a segment, it requires just $O(1)$ space. Furthermore, given a partition of $T[1..h]$ into segments, and having a sparse or a dense description of each segment, we are able to run all alive processes. While this is obvious for sparse descriptions, it requires a proof for dense descriptions.

**Lemma 4.3.** Given a dense description of a segment, we can run all relevant processes in $O(1)$ time each.

**Proof.** From the definition of a dense description, it is enough to run all alive processes $P(c + \alpha p)$ for an integer $\alpha$ which are inside the segment. We cannot know which of them are still alive, so we will run all of them (thus possible running some zombies). To this end we need to compute any $\Phi(c + \alpha p - 1)$, preferably in $O(1)$ time. We know $\Phi(c-1)$, $p$ and $\Phi(T[c..(c+p-1)])$. This is enough to compute all $\Phi(c + \alpha p - 1)$ for even values of $\alpha$ in $O(1)$ time each by starting with $T[1..(c-1)]$ and then using $\Phi(T[c..(c+p-1)])$ to repeatedly simulate either appending or erasing $T[c..(c+p-1)]$ and updating the current
\(\Phi(c + \alpha \frac{p}{2} - 1)\). All \(\Phi(c + \alpha \frac{p}{2} - 1)\) with odd \(\alpha\) can be computed similarly using the stored \(\Phi(T[c..(c + \frac{p}{2} - 1)])\). It might be the case that we do not have \(\Phi(T[c..(c + \frac{p}{2} - 1)])\) available, though, but then we know that all processes \(P(c + \alpha \frac{p}{2})\) with odd \(\alpha\) are dead and can be ignored.

We use Lemma 4.1 to maintain a partition of \(T[1..h]\) into segments. After reading \(T[h]\) we create a sparse description of the new segment of length 2\(^0\) and then need to merge at most one pair of adjacent segments. After having updated the partition, we run all alive processes in all segments. Therefore, now we need to show how to merge a pair of adjacent segments \(s, s'\) of length 2\(^\ell\) as to obtain a new segment \(ss'\). There are two cases depending on how the segments \(s\) and \(s'\) are described.

Both descriptions are sparse. In such a case we can merge the lists of \(s\) and \(s'\) as to get a list of at most 8 processes inside \(ss'\) such that all other processes inside are dead. If the list consists of at most 4 processes, it is a valid sparse description of \(ss'\). Otherwise we have at least 5 processes \(P(c_1), P(c_2), \ldots, P(c_5)\). We need the following observation, which follows from the properties of our partition scheme.

Observation 4.4. When a segment of length 2\(^\ell\) is being created, the number of already seen characters on its right is at most 3 \cdot 2^{\ell - 1} + 5(2^{\ell - 1} - 1) = 2^{\ell + 2} - 5.

When it is being destroyed, the number of already seen characters on its right is at least 3(2^{\ell + 1} - 1).

By the above observation, any \(P(c_i)\) could have been run everywhere in the interval \([c_i + 2^{\ell} + 2^{\ell + 2} - 5, c_i + 3(2^{\ell + 1} - 1)]\). By Lemma 3.3 every \(P(c_i)\) had at least one landmark available in that interval, so its radius must be at least 2\(^{\ell - 1}\) + 2\(^{\ell + 2} - 4 \geq 2^{\ell + 1}\). Therefore, we have a list of 5 processes inside a segment of length 2\(^{\ell + 1}\), all of which have radii at least 2\(^{\ell + 1}\) (except when \(\ell = 0\), but then we cannot have 5 different processes inside). This is enough to construct a dense description of the segment by the following lemma.

Lemma 4.5. Given a list of \(m \geq 5\) processes \(P(c_1), P(c_2), \ldots, P(c_m)\) inside a segment of length 2\(^\ell\) such that their radii are all at least 2\(^\ell\) and no other process inside the segment is alive, we can construct in \(O(m + \log n)\) time a dense description of the segment.

Proof. We rearrange the processes so that \(c_1 < c_2 < \ldots < c_m\) and define \(\Delta_i = c_{i+1} - c_i\). Every 2\(\Delta_i\) is a period of the segment by Lemma 4.2. We claim that by the periodicity lemma also \(\gcd(2\Delta_1, 2\Delta_2, \ldots, 2\Delta_{m-1})\) is a period of the segment. This can be seen by the following reasoning: if the radii at \(c < c' < c''\) are all at least 2\(^\ell\), \(c'' - c \leq 2^{\ell}\), 2\(d\) \(\mid 2(c' - c)\) is a period of \(T[(c - 2^{\ell})..(c' + 2^{\ell} - 1)]\) and 2\(d\) \(\mid 2(c'' - c')\) is a period of \(T[(c' - 2^{\ell})..(c'' + 2^{\ell} - 1)]\), then by the periodicity lemma 2\(\gcd(d, d')\) is a period of the whole \(T[(c - 2^{\ell})..(c'' + 2^{\ell} - 1)]\). Then by induction \(p = 2\gcd(\Delta_1, \Delta_2, \ldots, \Delta_{m-1})\) is a period of \(T[(c_1 - 2^{\ell})..(c_k + 2^{\ell} - 1)]\), which contains the whole segment inside. Because \(\Delta_1 + \Delta_2 + \ldots + \Delta_m \leq 2^{\ell}\),
and \( m \geq 5 \), \( \Delta_i \leq \frac{1}{2}2^i \) for at least one \( i \), so \( p \leq \frac{1}{2}2^i \) and consequently \( p \) must be a multiple of \( \text{per}(s) \).

Now we can construct a dense description. We compute \( p \) in \( \mathcal{O}(m + \log n) \) with \( m - 1 \) applications of the Euclidean algorithm, and set \( c = c_1 \). Because \( p \mid 2\Delta_i \) for every \( i \), all \( c_i \) are of the form \( c + \alpha \frac{p}{2} \). Furthermore, because \( p \leq \min(\Delta_1, \Delta_2) \), \( c + p \leq c_2 \), so \( T[c..(c + p - 1)] \) is fully within the segment. Finally, we must argue that \( T[c..(c + p - 1)] \) is an even palindrome. First observe that \( T[(c_3 - p)..(c_3 + p)] \) lies fully within the segment, and consider two cases.

- If \( c_3 = c + \alpha p \), then \( T[(c_3 - p)..(c_3 + p)] = T[c..(p - 1)]^2 \). Because the palindromic radius at \( c_3 \) is at least \( 2^\ell \geq p \), \( T[c..(c + p - 1)] \) is a palindrome.

- If \( c_3 = c + \beta \frac{p}{2} + \alpha p \), then \( T[(c_3 - \frac{p}{2})..(c_3 + \frac{p}{2})] = T[c..(c + p - 1)] \). Because the palindromic radius at \( c_3 \) is at least \( 2^\ell \geq \frac{p}{2} \), \( T[c..(c + p - 1)] \) is a palindrome.

Apart from storing \( P(c) \), we must also compute and keep \( \Phi(T[c..(c + p - 1)]) \) and, possibly, \( \Phi(T[c..(c + \beta \frac{p}{2} - 1)]) \). Let \( T[c..(c + p - 1)] = ww^R \) and recall that \( c = c_1 \). We have two cases.

- \( T[c_1..(c_2 - 1)] = (ww^R)^\beta \). Then we can extract \( \Phi(ww^R) \) from \( \Phi(c_1 - 1) \) and \( \Phi(c_2 - 1) \).

- \( T[c_1..(c_2 - 1)] = (ww^R)^\beta w \). Then \( T[c_2..(c_3 - 1)] = w^R(ww^R)^\gamma \) or \( T[c_2..(c_3 - 1)] = w^{R^2}(ww^R)^\gamma w^R \). In the former case we can extract \( \Phi(ww^R) \) from \( \Phi(c_1 - 1) \) and \( \Phi(c_3 - 1) \). In the latter case we can extract \( \Phi(w^R w) \) from \( \Phi(c_2 - 1) \) and \( \Phi(c_3 - 1) \), then compute \( \Phi((ww^R)^\beta) \), use it to extract \( \Phi(w) \) from \( \Phi(c_1 - 1) \) and \( \Phi(c_2 - 1) \), and calculate \( \Phi(w^R) \).

So in all cases we are able to extract \( \Phi(ww^R) \) in \( \mathcal{O}(\log n) \) time. Then either all \( c_i \) are of the form \( c + \alpha \frac{p}{2} \) with even \( \alpha \), in which case we do not have to compute \( \Phi(w) \), as there are no other alive processes inside the segment. Otherwise, we can find \( c_i = c + \frac{p}{2} + \beta p \), where \( \beta \) is an integer. Using \( \Phi(ww^R) \), \( \Phi(c - 1) \) and \( \Phi(c_i - 1) \) we can finally extract \( \Phi(w) \).

This settles the situation when both descriptions are sparse. Before we move to the remaining case, we need an additional tool. If a description of a segment is dense, we maintain some additional information about the processes inside. Informally, we would like to know which of them are still alive, but of course we cannot afford to explicitly maintain such information. We can only afford to store a short buffer, where we keep information about a few most recently run processes. Formally, the buffer is a list of processes \( P(c) \) together with their corresponding values of \( R(c) \). We do not require that \( P(c) \) is still alive, so it might have happened that it has been run again after reading \( T[h'] \) with \( h < h' \), but the more recent run was unsuccessful. The buffer is updated whenever we successfully run a process \( P(c) \) inside the segment. There are two cases:

- \( P(c) \) was in the buffer, then we move it to the front and update its corresponding \( R(c) \),
• $P(c)$ was not in the buffer, then we prepend it to the buffer together with the current $R(c)$, and if the length of the buffer is now 6 we remove the last element from there.

This clearly guarantees that the length of the buffer is always at most 5. A less trivial consequence of the above implementation is as follows.

**Lemma 4.6.** If a segment with dense description of length $2^\ell$ is being destroyed while at most 4 processes in its buffer have radii at least $2^{\ell+1}$, then no other process inside the segment can be still alive.

**Proof.** By Observation 4.4 and how we process segments with dense descriptions, any $P(c)$ which might be still alive could have been run everywhere in the interval $[c + 2^\ell + 2^{\ell+2} - 5, c + 3(2^{\ell+1} - 1)]$, and by Lemma 3.3 it had at least one landmark available in that interval. Also, whenever we run any $P(c)$ inside the segment in the interval $[c + 2^\ell + 2^{\ell+2} - 5, \infty)$, and it succeeds, $R(c)$ is set to at least $2^{\ell+1}$ (except when $\ell = 0$, but then there is just one process inside the segment, so the buffer surely contains it). Therefore, if the buffer contains at most 4 processes with radii at least $2^{\ell+1}$, any $P(c)$ such that $R(c) \geq 2^{\ell+1}$ is stored in the buffer, and no other process can be still alive.

**One description is dense.** Applying Lemma 4.6 to $s$ (if its description is dense) or $s'$ (if its description is dense), we either get that one of these segments contains at least 5 processes with radii at least $2^{\ell+1}$ in its buffer, or we get a list of at most 4 potentially still alive processes inside each segment. In the latter case we concatenate the lists to get a list of at most 8 processes inside $ss'$ such that all other processes inside are dead. If the list contains at most 4 processes, we construct a sparse description of $ss'$, and otherwise we apply Lemma 4.5 to construct a dense description of $ss'$ in $O(\log n)$ time. In the former case we get a list of between 5 and 10 alive processes $P(c_1), P(c_2), \ldots, P(c_m)$ inside $ss'$. It might be the case that there are also some other processes inside the segment which are still alive, but they are not stored in the buffer of the corresponding segment. Nevertheless, proceeding as in the proof of Lemma 4.5 we can compute in $O(\log n)$ time $p$ and $c$ such that $T[c..(c + p - 1)]$ is an even palindrome fully within $ss'$, all $c_i$ are of the form $c + \alpha_i^2$, and $p \leq \frac{1}{2}|ss'|$ is a period of $ss'$. This is not a valid dense description yet, as $s$ or $s'$ (or both) might have dense descriptions, and we cannot guarantee that all alive processes there are of the form $c + \alpha_i^2$.

Consider the case when $s$ has a dense description, meaning that we have $p'$ and $c'$ such $T[c'..(c' + p' - 1)]$ is an even palindrome fully within $s$, all alive processes there are of the form $c' + \alpha_i^2$, and $p' \leq \frac{1}{2}|s|$ is a period of $s$. If $p'|p$ there is nothing to do. Otherwise, because the list $P(c_1), P(c_2), \ldots, P(c_m)$ contains at least 5 processes inside $s$ we have $p \leq \frac{1}{4}|s|$ and by the periodicity lemma $\gcd(p, p')$ is a period of $s$. Then $\gcd(p, p')$ must be actually a period of the whole $ss'$. Now we claim that $p$ can be, in fact, replaced by $\gcd(p, p')$. This is because if a power of a word is a palindrome, the word itself must be a palindrome, so $T[c'..(c' + \gcd(p, p') - 1)]$ is an even palindrome, and we can
compute its hash in $O(\ell)$ time. We might also need to compute $\Phi(T[c'(c' + \gcd(p,p') - 1)])$. This happens when:

- $c_i = c + \gcd(p,p') + \alpha \gcd(p,p')$ for some $i$. Then we can clearly compute $\Phi(T[c'(c' + \gcd(p,p') - 1)])$ from $\Phi(c_i - 1)$ and $\Phi(T[c'(c' + \gcd(p,p') - 1)])$ in $O(\log n)$ time.

- Alternatively, if we know $\Phi(T[c'(c' + \frac{p}{2} - 1)])$ and $\frac{p}{2}$ is not a multiple of $\gcd(p,p')$. Then $\Phi(T[c'(c' + \gcd(p,p') - 1)])$ can be computed from $\Phi(T[c'(c' + \frac{p}{2} - 1)])$ and $\Phi(T[c'(c' + \gcd(p,p') - 1)])$ in $O(\log n)$ time.

The case when $s'$ has a dense description, or both $s$ and $s'$ have dense descriptions, can be dealt with similarly. This finishes the description of how to maintain the descriptions of all segments in our partition of $T[1..h]$, and allows us to simulate the basic algorithm in $O(1)$ space for every segment, plus the space required to store the landmarks (which depends on whether we aim for additive or multiplicative approximation).

**Theorem 4.7.** Any scheduling scheme with $b_{\lambda} \geq 12$ for all $\lambda \leq L$ can be simulated using $O(\log h)$ additional space on the top of the space taken by the landmarks.

5 Time-efficient algorithm

The simulation from the previous section was space-efficient, but unfortunately not time-efficient. The reason for its potentially time complexity is that we might have one (or many) segments with dense descriptions and small periods, which in turn requires running many processes. Fortunately, this is the only reason the time to process a single $T[h]$ might exceed $O(\log h)$, as merging at most one pair of segments and running the processes in all segments with sparse descriptions takes just $O(\log h)$ time. In this section we will show how to simulate running all processes in a single segment with a dense description in a $O(1)$ time.

Consider a dense description of a segment $s$. Recall that it consists of a process $P(c)$ and a value of $p$, such that we want to run all $P(c')$ inside $s$ of the form $c' = c + \alpha \frac{p}{2}$, where $2c - h - 2$ is a landmark. Because we store $\Phi(T[c..c + p - 1])$, and either know also $\Phi(T[c..c + \frac{p}{2} - 1])$ or are guaranteed that only even values of $\alpha$ need to be considered, we can construct and run all relevant $P(c')$ in $O(1)$ time each, but there might be many of them. However, the crucial insight is that there are only two consequences of running such $P(c')$: we might update the final answer, and we might also store it in the buffer (or move it to the front there). Therefore, if we can guarantee that a particular $P(c')$ will fail anyway, we can avoid running it altogether. Similarly, if we can guarantee that many processes $P(c')$ will succeed, it is enough to run just the 5 leftmost of them.
We will build on these two simple observations to simulate running all processes of such form in a single segment with a dense description in $O(1)$ total time. We start with observing that, when considering such a segment, just a constant number of levels is relevant.

**Associated landmark levels.** Consider a segment $s$. If, for some $c$ inside $s$, $2c - h - 2$ is a landmark strictly on level $\lambda$ at $h$, we say that $\lambda$ is a landmark level associated to $s$.

**Lemma 5.1.** There are at most 4 landmark levels associated to a single segment, and they can be all determined $O(\log h)$ time.

**Proof.** Consider a segment $s$ of length $2^\ell$ and any $c$ inside. By Observation 4.4 when the segment is being created by merging two segments of length $2^{\ell-1}$ we have $h - c \geq 3(2^\ell - 1)$. Similarly, when the segment is being destroyed by merging with an adjacent segment of length $2^\ell$ to form a segment of length $2^{\ell+1}$ we have $h - c < 2^{\ell+1} + 2^{\ell+3} - 5 = 5(2^{\ell+1} - 1)$. Consequently, we can bound $2(h - c + 1)$, which is the number of already seen characters on the right of $2c - h - 2$, as follows:

$$2(h - c + 1) < 10 \cdot 2^{\ell+1} - 8$$
$$2(h - c + 1) \geq 6 \cdot 2^\ell - 4$$

If $2c - h - 2$ is a landmark strictly on level $\lambda < L$, then the number of already seen characters on its right belongs to $[b_0 \cdot 2^{\lambda-1}, b_0 \cdot 2^\lambda)$. Bounding the number of different landmark levels associated to $s$ requires counting $\lambda < L$ such that $[b_0 \cdot 2^{\lambda-1}, b_0 \cdot 2^\lambda) \cap [6 \cdot 2^\ell - 4, 10 \cdot 2^{\ell+1} - 8) \neq \emptyset$. The condition translates into:

$$b_0 \cdot 2^{\lambda-1} \leq 10 \cdot 2^{\ell+1} - 8 - 1$$
$$b_0 \cdot 2^\lambda - 1 \geq 6 \cdot 2^\ell - 4$$

which is equivalent to $2^\lambda \cdot b_0 \in [6 \cdot 2^\ell - 3, 40 \cdot 2^\ell - 18]$. If $\lambda = L$, the number of already seen characters on the right is at least $b_0 \cdot 2^{\lambda-1}$, so the condition becomes $2^\lambda \cdot b_0 \leq 40 \cdot 2^\ell - 18$. All in all, there are at most 4 different possible values of $\lambda$.

Generating the landmark levels associated with a given segment can be done in $O(\log h)$ time by performing the above calculation. $\square$

Due to Lemma 5.1 in order to achieve the claimed $O(\log n)$ time complexity for processing a single $T[h]$, we only need to show how to run all relevant processes inside a segment with a dense description using landmarks on a particular level $\lambda$ associated to that segment in $O(1)$ time. This is the most technical part of the paper, so we start with a weaker version, where processing a single $T[h]$ takes as much as $O(\log^2 h + \log n)$ time, which will be then improved to achieve the final result, and provide an overview of the remaining part of this section below.
Overview. We start with analyzing which relevant processes \( P(c') \) should be run because of a landmark on level \( \lambda \). After some basic arithmetical manipulation, we get a succinct description of all such values of \( c' \). To avoid considering all of them, which might be too costly, we apply two lemmas characterizing the structure of palindromes in a sufficiently periodic fragment of the text, described in Lemma 5.5 and Lemma 5.6 (these observations go back to [1], but we need a slightly different formulation). To apply them, we need to compute how far the periodicity of a segment with a dense description continues to the left and to the right. To this end, we relax the notion of landmarks, introducing the so-called ghost landmarks, which allow us to operate on a longer suffix of the already seen \( T[1..h] \). Then, using these ghost landmarks, we can binary search to compute how far the periodicity extends, and apply the structural results to isolate at most 5 relevant processes, which should be run as to guarantee the correctness. This allows us to achieve \( O(\log^2 h + \log n) \) time to process a single \( T[h] \). We further improve the complexity to \( O(\log n) \) by precomputing more data when a segment is being created, and using the landmarks more carefully.

Relevant processes. We need to consider all relevant processes \( P(c') \), such that \( 2c' - h - 2 \) is a landmark on level \( \lambda \), implying that \( 2\lambda \mid 2c' - h - 2 \). The condition is equivalent to:

\[
\alpha \cdot p = h + 2 - 2c \pmod{2^\lambda}
\]

which, denoting \( 2^\ell = \gcd(p, 2^\lambda) \), is in turn equivalent to:

\[
\frac{\alpha \cdot p}{2^\ell} = \frac{h + 2 - 2c}{2^\ell} \pmod{2^{\lambda-\ell}}
\]

(unless \( 2^\ell \) does not divide \( h + 2 - 2c \), when no \( c' \) needs to be considered), so by computing the multiplicative inverse we finally get a base solution to \( \ell \):

\[
ono = \frac{h + 2 - 2c}{2^\ell} \cdot \left( \frac{p}{2^\ell} \right)^{-1} \pmod{2^{\lambda-\ell}}
\]

and the general solution is:

\[
\alpha = \alpha_0 + t \cdot 2^{\lambda-\ell} \quad \text{for} \quad t \in \{-1, 0, 1, \ldots\}
\]

Thus we also get the solution to the original equation:

\[
c' = c'_0 + t \cdot \frac{p}{2} 2^{\lambda-\ell} = c'_0 + t \cdot \frac{1}{2} \lcm(2^\ell, p) \quad \text{where} \quad c'_0 = c + \alpha_0 \frac{p}{2} \tag{2}
\]

Therefore, with a simple calculation we get a succinct description of all values of \( c' \) which should be taken into the account. Before we proceed further, let us comment on the complexity of the calculation. Since \( \lambda \) is fixed, both values of

\[
2^\ell = \gcd(p, 2^\lambda) \quad \text{and} \quad \left( \frac{p}{2^\ell} \right)^{-1} \pmod{2^{\lambda-\ell}}
\]

are constant. Therefore, the complexity is linear in \( n \) and \( h \).

16
can be computed in $O(\log h)$ time when we create the segment and stored there. Thus we can access these values in $O(1)$ time when required.

The situation now is that we have a dense description of a segment, and want to run all processes $P(c')$ of the form (2) inside the segment. Additionally, because we do not necessarily have all possible landmarks on level $\lambda$, just a few most recent, we are interested only in sufficiently large $c'$. Observe, that we can analyze separately processes of the following two forms:

$$P(c'_0 + t \cdot \text{lcm}(2^\lambda, p))$$  \hspace{1cm} (3)

$$P(c''_0 + t \cdot \text{lcm}(2^\lambda, p)) \quad \text{where} \quad c''_0 = c'_0 + \frac{1}{2} \text{lcm}(2^\lambda, p)$$ \hspace{1cm} (4)

From now on we will only consider the former, as the whole reasoning still holds after replacing $c'_0$ by $c''_0$. We will also assume that only $t \geq 0$ need to be considered, which can be always ensured by decreasing $c'_0$ by an appropriate multiple of $\text{lcm}(2^\lambda, p)$.

The initial process $P(c'_0)$ can be constructed in $O(\log h)$ time. Now recall that $p$ is a period of the whole segment, so $\text{lcm}(2^\lambda, p)$ is its period as well, and furthermore we can assume that $\text{lcm}(2^\lambda, p) \leq \frac{1}{2}|s|$, as otherwise there are just at most two relevant processes to run, which can be constructed and stored together with the segment in $O(\log h)$ time. Intuitively, knowing how far the period extends to the left and to the right allows us to restrict the number of processes to run by an argument based on the combinatorial properties of palindromes. While computing how far the period extends exactly is not possible in our setting, it can be approximated quite well using the landmarks. First, we need to introduce the notion of ghost landmarks.

**Ghost landmarks.** For every level of landmarks $\lambda$, we store $f_\lambda = 4 \cdot b_\lambda$ most recently seen landmarks on level $\lambda$. All ghost landmarks can be maintained in the same manner as the regular landmarks, so storing them does not change the complexity of our algorithm.

**Lemma 5.2.** If $\lambda$ is a landmark level associated to a segment $s$, then for any $c$ inside $s$ there is at least one ghost landmark on level $\lambda$ in $T[1..(2c - h - 2)]$.

**Proof.** Consider a segment $s$ of length $2^\ell$. By Observation 4.4, the number of already seen characters on the right of $s$ when it is being created is at least $3(2^\ell - 1)$. Let $c'$ be any position inside $s$ causing $\lambda$ to be associated to $s$, i.e., $2^\lambda \mid 2c' - h' - 2$, and denote $x' = 2c' - h' - 2$. Notice that $h'$ might be either smaller or larger than the current $h$. Because $c'$ is a landmark on level $\lambda$ at $h'$, we have that $2^\lambda \cdot b_\lambda \geq h' - x' = 2(h' - c') + 2 \geq 2 + 6(2^\ell - 1)$.

Now consider any $c$ inside $s$ and denote $x = 2c - h - 2$. Since $c$ and $c'$ both belong to the same segment of length $2^\ell$, $c' - c \leq 2^\ell - 1$. Applying Observation 4.4 again, we also get that $h - h' \leq 5(2^\ell+1 - 1) - 1 - 3(2^\ell - 1) = 7 \cdot 2^\ell - 3$.

Thus the number of already seen characters on the right of $x$ can be bounded as follows:

$$h - x = 2h - 2c + 2 \leq (2h' - 2c' + 2) + 2(2^\ell - 1) + 14 \cdot 2^\ell - 6.$$  

17
Because $h' - x' = 2(h' - c') + 2 \geq 2 + 6(2^\ell - 1)$, we have $16 \cdot 2^\ell - \frac{32}{3} \leq \frac{8}{3} \cdot (h' - x')$, so the above bound can be rewritten as:

$$h - x \leq \frac{11}{3} \cdot (h' - x') + \frac{8}{3} \leq \frac{11}{3} b_\lambda \cdot 2^\lambda + \frac{8}{3} \leq (4b_\lambda - 1) \cdot 2^\lambda$$

where the last inequality holds because $b_\lambda \geq 12$. Since the leftmost ghost landmark on level $\lambda$ has at least $(4b_\lambda - 1) \cdot 2^\lambda$ already seen characters on its right, by the above calculation it must be on the left of $x = 2c - h - 2$ as claimed.

Now going back to approximating how far the period extends to the left and to the right, we proceed as follows. We choose $w$ of length $\text{lcm}(2^\lambda, p)$ starting at $T[c_0']$. Because we have adjusted $c_0'$ so that only $t \geq 0$ need to be considered and $|w| \leq \frac{1}{2}|s|$, we can assume that $w$ is fully within the segment. We know that the whole segment can be covered by repeating $w$ to the left and to the right (where, possibly, the last repetition is a suffix or a prefix of $w$, respectively), and would like to figure out how far we can continue that until we hit either a boundary of the already seen $T[1..h]$, or a subword of length $|w|$ which is different than $w$. This can be approximated quite well using the ghost landmarks, assuming that $w$ repeats at least twice.

**Lemma 5.3.** For any $w$ such that $T[i..(i + 2|w| - 1)] = w^2$, $2^\lambda \mid |w|$, and $T[1..(i - 1)]$ contains at least one ghost landmark on level $\lambda$, we can compute in $O(\log h)$ time $r \geq 2$ such that $T[i..(i + r|w| - 1)] = w^r$ and either $i + (r+2)|w| > h$ or $T[i..(i + (r+2)|w| - 1)] \neq w^{r+2}$.

**Proof.** By the assumption about ghost landmark on level $\lambda$, we can access any $\Phi(2^\lambda \cdot j)$ with $j \geq \left\lfloor \frac{|w|}{2^\lambda} \right\rfloor$ in $O(1)$ time. Hence if we are lucky and $i = 2^\lambda \cdot j + 1$, we can compute $\Phi(T[(i+\alpha|w|)..(i+\beta|w| - 1)])$ for any $0 \leq \alpha \leq \beta$ in $O(1)$ time, which allows us to binary search for $r$ in $O(\log h)$ time. In more detail, to check if $T[i..(i + r|w| - 1)] = w^r$ we check if $|w|$ is a period of $T[i..(i + r|w| - 1)]$, which can be done by comparing $\Phi(T[(i+|w|)..(i+r|w| - 1)])$ and $\Phi(T[i..(i + r - 1)|w| - 1)])$.

In the general case, let $i = 2^\lambda \cdot j + 1 + \Delta$, where $\Delta \in [0, 2^\lambda)$. If $T[i..(i + r|w| - 1)] = w^r$ and $r \geq 2$, then $|w|$ is a period of $T[(2^\lambda \cdot j + 2^\lambda + \alpha|w|)]$, see Fig. 3 where $\alpha = r - 1$. In the other direction, if $|w|$ is a period of $T[(2^\lambda \cdot j + 2^\lambda + 1)..(2^\lambda \cdot j + 2^\lambda + \alpha|w|)]$ and $r \geq 2$, then $r \geq \alpha$. (The assumption that $r \geq 2$ is crucial.) Hence we can determine the largest $\alpha$ such that $|w|$ is a period of $T[(2^\lambda \cdot j + 2^\lambda + 1)..(2^\lambda \cdot j + 2^\lambda + \alpha|w|)]$ in $O(\log h)$ time using ghost landmarks on level $\lambda$, and then simply return $\alpha$, which guarantees $r \in \{\alpha, \alpha + 1\}$. \qed
Lemma 5.4. For any \( w \) such that \( T[i..(i + 2|w| - 1)] = w^2 \), \( 2^α \mid |w| \), and \( T[1..(2i - h - 2)] \) contains at least one ghost landmark on level \( λ \), we can compute in \( O(\log h) \) time \( ℓ \geq 0 \) such that \( T[(i - ℓ)|w|.i - 1] = w^ℓ \) and either \( i - (ℓ + 2)|w| < 2i - h - 2 \) or \( T[(i - (ℓ + 2)|w|).i - 1] \neq w^{ℓ+2} \).

Proof. The proof will be very similar to the proof of Lemma 5.3 except that we have to take into the account the fact that while there might be many more repetitions of \( w \) to the left, we might not have enough ghost landmarks on level \( λ \) to detect them.

Let \( i = 2^λ \cdot j + 1 + Δ \), where \( Δ \in [0, 2^λ) \). If \( T[(i - ℓ)|w|.i - 1] = w^ℓ \), then \( |w| \) is a period of \( T[(2^λ \cdot j - α|w| + 2^λ + 1..(2^λ \cdot j + |w| + 2^λ)] \), where \( α = ℓ \). In the other direction, if \( |w| \) is a period of \( T[(2^λ \cdot j - α|w| + 2^λ + 1..(2^λ \cdot j + |w| + 2^λ)] \), then \( ℓ \geq α - 1 \). So we only need to binary search for the largest \( α \) such that \( |w| \) is a period of \( T[(2^λ \cdot j - α|w| + 2^λ + 1..(2^λ \cdot j + |w| + 2^λ)] \) and return \( \max(0, α - 1) \).

The remaining difficulty is that \( 2^λ \cdot j - α|w| + 2^λ \) might lie too far on the left to be a ghost landmark on level \( λ \), so the binary search needs to be slightly modified. We first choose the largest \( α_0 \) such that \( 2^λ \cdot j - α_0|w| + 2^λ \) is a ghost landmark on level \( λ \). There are two possibilities.

1. \( |w| \) is a period of \( T[(2^λ \cdot j - α_0|w| + 2^λ + 1..(2^λ \cdot j + |w| + 2^λ)] \), then the largest \( α \) might exceed \( α_0 \). But we can return \( ℓ = \max(0, α_0 - 1) \), because then \( i - (ℓ + 1)|w| ≤ i - α_0|w| \), and the choice of \( α_0 \) and the assumption, by Lemma 5.2 implies \( i - (α_0 + 1)|w| < 2i - h - 2 \), so \( i - (ℓ + 2)|w| < 2i - h - 2 \).

2. Otherwise, we binary search over all \( α \leq α_0 \), and return \( \max(0, α - 1) \).

We can binary search for \( α_0 \) in \( O(\log h) \) time, so the total time is \( O(\log h) \).

We apply Lemma 5.3 and Lemma 5.4 to approximate how many times \( w \) repeats on its right and on its left in \( T[c_0..(2c_0 - h - 2)\).c_0 - 1] \) with accuracy 1, assuming that \( w^2 \) occurs at \( T[c_0] \). Notice that there might be many more repetitions to the left in the whole \( T[1..(c_0 - 1)] \), but Lemma 5.4 does not allow us to detect all of them. Now the crucial insight is that even though we do now know the exact number of repetitions, we can iterate through the at most 4 possible combinations of the number of of repetitions to the left and to the right, and the additionally consider the possibility that there is only a single occurrence of \( w \) in the segment. Hence we need to iterate through 5 possibilities in total. For each such combination, we will restrict the number of processes which should be run, therefore by running the processes determined for each of these combinations we will not lose the correctness. Hence from now on we assume that we know the exact number of repetitions of \( w \) to the left and to the right.

We need the following two simple structural results, which allow us to bound the palindromic radius in a sufficiently periodic subword of the text. A similar (in spirit) argument appeared already in \( 11 \), but we need a slightly different formulation. We say that a palindrome centered at \( c \) reaches \( h \) if \( R(c) \geq h - c + 1 \).
Lemma 5.5. Consider $uw^kv$ starting at position $i$ in $T[1..h]$, where $|u| = |w| = |v|$, $w$ is a palindrome, and $u, v \neq w$. For any $\alpha \in \{1, 2, \ldots, k\}$, if the palindrome centered at $i + \alpha|u|$ reaches $h$ then $\alpha = \frac{k}{2} + 1$.

Proof. Take any $\alpha \in \{1, 2, \ldots, k\}$. For a palindrome centered at $i + \alpha|u|$ to reach $h$, $R(i + \alpha|u|)$ must be at least $\min(\alpha - 1, k + 1 - \alpha)|u|$. But then either $u = w^R$ or $v = w^R$, which is a contradiction.

Lemma 5.6. Consider $uw^kv$ starting at position $i$ in $T[1..h]$, where $|u| = |w| = |v|$, $w$ is a palindrome, $u \neq w$, and $h - i - (k + 1)|u| + 1 < |w|$. For any $\alpha \in \{1, 2, \ldots, \lceil \frac{k}{2} \rceil \}$, the palindrome centered at $i + \alpha|u|$ cannot reach $h$. Additionally, either all palindromes centered at $i + \alpha|u|$ with $\alpha \in \{\lceil \frac{k}{2} \rceil + 1, \ldots, k\}$ reach $h$, or none of them do.

Proof. Take any $\alpha \in \{1, 2, \ldots, k\}$. If $\alpha \leq \lceil \frac{k}{2} \rceil$, then because $u \neq w$ the radius at $i + \alpha|u|$ is too small for the palindrome centered at $i + \alpha|u|$ to reach $h$. Otherwise, let $T[i..h] = uw^kv$, where $|v| < |u|$ because $h - i - (k + 1)|u| + 1$, see Fig. 4. Now either $v$ is not a prefix of $w$, and we actually get the situation from Lemma 5.5, so only $\alpha = \frac{k}{2} + 1$ can possibly correspond to a palindrome reaching $h$, or $v$ is a prefix of $w$, and for all $\alpha \geq \lceil \frac{k}{2} + 1 \rceil$ the palindrome centered at $i + \alpha|u|$ reaches $h$.

Recall that we want to run all $P(c_0' + t|w|)$ inside the segment with $t \geq 0$, and our $w$ starts at $T[c_0']$. We know that $w$ repeats $\ell$ times to the left in $T[(2c_0' - h - 2)\ldots(c_0' - 1)]$ and $r$ times to the right till the end of the already seen $T[1..h]$. The actual number of repetitions of $w$ to the left in the whole $T1..(c_0' - 1)$, denoted $\ell'$, might be larger than $\ell$. By Lemma 5.5 and Lemma 5.6 either all processes of the form $P(c_0' + (\ell' + \alpha)|w|)$ with $\alpha \geq \lceil \frac{\ell + r}{2} + 1 \rceil$ will succeed, or just the one with $\alpha = \lceil \frac{\ell + r}{2} + 1 \rceil$ will succeed. Because the size of the buffer is 5, we only need to ensure that the 5 leftmost processes which will succeed are run. To guarantee this, we run all processes of the form $P(c_0' + (\max(-\ell + \lceil \frac{\ell + r}{2} + 1 \rceil, 0) + x)|w|)$ for $x = 0, 1, 2, 3, 4$ which are still inside the segment. This is correct, as following two cases show.

1. The process $P(c_0' + (-\ell' + \alpha)|w|)$ with $\alpha = \lceil \frac{\ell + r}{2} + 1 \rceil$ is on the left of the segment, so either all or none processes of such form in the segment are alive.
2. The process $P(c'_0 + (-\ell' + \alpha)|w|)$ with $\alpha = -\frac{\ell' + 1}{2}$ is inside the segment, so $\ell'$ cannot be too large. More precisely, $\ell' \leq r$, and consequently $\ell = \ell'$.

We run a constant number of processes, each of them in $O(\log h)$ time, as we need to construct the appropriate $P(c'_0 + t|w|)$ out of $P(c)$ and $\Phi(T[c..(c+p-1)])$. This allows us to process all segments with dense descriptions in $O(\log^2 h)$ total time.

**Theorem 5.7.** Any scheduling scheme with $b_\lambda \geq 12$ for all $\lambda \leq L$ can be simulated using $O(\log h)$ additional space on the top of the space taken by the landmarks and $O(\log^2 h + \log n)$ time to process a single $T[h]$.

We would like to speed up processing of a segment with a dense description to $O(1)$ time. There are two technical obstacles. Firstly, we need to remove the binary search used to approximate how many times $w$ can be repeated to the left and to the right. Secondly, we have to argue that the required $P(c'_0 + t|w|)$ can be constructed in $O(1)$ time.

**Computing how far $w$ can be repeated.** Recall that $|w| = \text{lcm}(2^\lambda, p)$, $w$ starts at $T[c'_0]$ and lies fully within a segment $s$, and furthermore $|w|$ is a period of the whole $s$. As mentioned before, we can also assume that $|w| \leq \frac{1}{2}|s|$, as otherwise there are at most two processes which might need to be run. We can compute how many times $w$ can be repeated to its left (or rather approximate this value as described in Lemma 5.4) when the segment is created, as the result does not depend on the current value of $h$. Similarly, we can compute how many times it can be repeated to the right when we create the segment, but here the important difference is that we might continue till the very end of the current $T[1..h]$, i.e., the next copy of $w$ might extend beyond the current prefix $T[1..h]$. It can be seen that in such a case the next time we need to deal with the same segment, at most one additional copy of $w$ fits inside $T[1..h]$. This happens because the segment is relevant when $2^\ell \mid h + 2 - 2c$, and $|w| \geq 2^\ell$. Therefore, the number of times $w$ repeats to the right can be maintained in $O(1)$ time.

**Constructing $P(c'_0 + t|w|)$.** Recall that $c'_0 = c + r't_3$, where $r' = \left(\frac{h + 2 - 2c}{2-r} \cdot r\right) \mod 2^{\lambda-\ell}$ for a previously precomputed $r$. Thus we will need to construct a particular $P(c'_0)$ whenever $2^\ell \mid h + 2 - 2c$. Therefore, when we are constructing the segment, we can find smallest $h' \geq h$ such that $2^\ell \mid h' + 2 - 2c$, and then construct $P(c'_0)$ in $O(\log h)$ time to have it available when we reach $h'$. We can also precompute and store both $\Phi(T[c..(c+p-1)])$ and $\Phi(T[c..(c+p-1)]^{2^\lambda-\ell})$ then, all in $O(\log h)$ time. To perform the update in $O(1)$ time, notice that:

- If the segment stores the value of $\Phi(T[c..(c+p-1)])$, we can use it together with precomputed values $\Phi(T[c..(c+p-1)]^{r'})$ and $\Phi(T[c..(c+p-1)]^{2^\lambda-\ell})$ to update the value of $\Phi(c'_0)$. The key observation is that the next time $2^\ell \mid h + 2 - 2c$, the value of $r'$ increases by $r$, and possibly decreases by $2^{\lambda-\ell}$. 

21
• Otherwise we are guaranteed that all relevant processes are of the form $P(c + \alpha p)$, thus we can perform the updates using just the precomputed values $\Phi(T[c..(c+p-1)]^t)$ and $\Phi(T[c..(c+p-1)]^{2^\lambda - t})$ as described above.

Now notice that as $|w| = \text{lcm}(2^\lambda, p)$ is a multiple of both $2^\lambda$ and $p$, we store all possible ghost landmarks on level $\lambda$ inside the current segment, and the segment is periodic with period $p$, any $P(c_0' + t|w|)$ with $t \geq 0$ inside the segment can be constructed in $O(1)$ time if we precompute $\Phi(T[c_0'..(c_0' + |w| - 1)])$ in $O(\log h)$ time when we create the segment. In more detail, either $t \in \{0, 1\}$ and we use the precomputed value to calculate the answer in $O(1)$ time, or the rightmost ghost landmark in $T[1..(c_0' + t|w| - 1)]$ is within the segment, call it $2^\lambda \cdot j$. Then the rightmost ghost landmark on level $\lambda$ in $T[1..(c_0' + t|w| - 1)]$ is also within the segment, call it $2^\lambda \cdot j'$. Now because of the periodicity, $T[(2^\lambda \cdot j + 1)\ldots(c_0' + |w| - 1)] = T[(2^\lambda \cdot j' + 1)\ldots(c_0' + t|w| - 1)]$, so having $\Phi(c_0' + |w| - 1)$ (which can be computed from $\Phi(c_0' - 1)$ and $\Phi(T[c_0'..(c_0' + |w| - 1)])$) and $\Phi(2^\lambda \cdot j')$ allows us to calculate $\Phi(c_0' + t|w| - 1)$ in $O(1)$ time.

This gives us the claimed the complexity for simulating any scheme in $O(\log n)$ time.

**Theorem 5.8.** Any scheduling scheme with $b_\lambda \geq 12$ for all $\lambda \leq L$ can be simulated using $O(\log h)$ additional space on the top of the space taken by the landmarks and $O(\log n)$ time to process a single $T[h]$.

### 6 Lower bounds

In this section we provide lower bounds for computing the largest radius of any palindrome in a word $n$ over an alphabet $\Sigma$, which we denote $\text{PALIN}_{\Sigma}[n]$. The lower bounds are based on the Yao’s minimax principle [10] formulated below.

**Theorem 6.1** (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. Then, for any $x \in \mathcal{X}$ and $A \in \mathcal{A}$, the cost of running $A$ on $x$ is denoted by $c(a, x) \geq 0$.

Let $p$ be the probability distribution over $\mathcal{A}$, and let $A$ be an algorithm chosen at random according to $p$. Let $q$ be the probability distribution over $\mathcal{X}$, and let $X$ be an input chosen at random according to $q$. Then the worst-case expected cost of the randomized algorithm is at least as large as the cost of the best deterministic algorithm against the chosen distribution on the inputs:

$$\max_{x \in \mathcal{X}} E[c(A, x)] \geq \min_{a \in \mathcal{A}} E[c(a, X)].$$

We use the above lemma 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.

22
We will first prove that trying to construct a Las Vegas approximation algorithm, either in the additive or the multiplicative error variant, is essentially pointless, as its memory usage must necessarily be $\Omega(n \log |\Sigma|)$ in expectation. The lower bound trivially transfers to deterministic approximation algorithms.

By the Yao’s minimax principle, it is enough to construct a distribution over the inputs, which is hard for any deterministic algorithm using a smaller amount of memory. We restrict the inputs to a large family of strings with symmetric padding in the middle, which guarantees that the longest palindrome is centered in the very middle, and a simple counting argument is then enough. Intuitively, deciding the equality between first half and reversed second half is hard, as there are more possible inputs in each half than there are states of memory for the deterministic algorithm. A bound on multiplicative approximation follows as a simple consequence of the fact that any multiplicative error algorithm is also an additive error algorithm, thus one can be reduced to another.

**Theorem 6.2** (Las Vegas additive approximation). Let $A$ be a Las Vegas streaming algorithms solving $\text{PALIN}_\Sigma[n]$ with an additive error $E \leq (\frac{1}{4} - \epsilon)n$ using $s(n)$ bits of memory. Then $E[s(n)] = \Omega(n \log |\Sigma|)$. 

**Proof.** By Theorem 6.1, it is enough to construct a probability distribution $P$ over $\Sigma^n$ such that for any deterministic algorithm $D$, its expected memory usage on a word chosen according to $P$ is $\Omega(n \log |\Sigma|)$ in bits.

We define $P$ as the uniform distribution over $x0^{2E}110^{2E}y$, where $x, y \in \Sigma^{n'}$, and $n' = \frac{n}{2} - 2E - 1$. Let us look at the memory usage of $D$ after having read (almost) the first half of a string, i.e., $x$. We say that $x$ is "good" when the memory usage is at most $\frac{n'}{2} \log |\Sigma|$ and "bad" otherwise. More than $\frac{1}{2}|\Sigma|n'$ of all $x$’s are good, otherwise expected memory usage would be already too big. Thus there are two strings $x \neq x'$ such that the state of $D$ after having read both $x$ and $x'$ is exactly the same. Hence the behavior of $D$ on $x0^{2E}110^{2E}xR$ and $x'0^{2E}110^{2E}xR$ is exactly the same. The former is a palindrome of radius $n' + 2E + 1$, so $D$ must answer at least $n + E + 1$, and consequently the latter must contain also a palindrome of radius at least $n + E + 1$ as a substring. Observe that the center of such a long palindrome inside $x'0^{2E}110^{2E}xR$ must be within a distance at most $E$ from the very center of the whole string. We consider two cases.

1. The distance is 0, then $x'0^{2E}110^{2E}xR$ is a palindrome, so $x = x'$, a contradiction.

2. The distance is strictly positive, but at most $E$. Because of the symmetry, we assume that the center of the palindrome is on the left of the center of the string. Then the left 1 in the middle 11 must have a corresponding 1 further on the left. More precisely, there must be other 1 within a distance of at most $2E$ on the left. But all $2E$ characters on the left are zeroes, 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 $\frac{1}{4}n \log |\Sigma|$, which for $E \leq (\frac{1}{4} - \epsilon)n$ is $\Omega(n \log |\Sigma|)$ as claimed.

**Theorem 6.3** (Las Vegas multiplicative approximation). Let $A$ be a Las Vegas streaming algorithm solving $\text{PALIN}_\Sigma[n]$ with a multiplicative error $(1 + \epsilon) \leq 2 - \epsilon$ using $s(n)$ bits of memory. Then $\mathbb{E}[s(n)] = \Omega(n \log |\Sigma|)$.

**Proof.** Let us assume on the contrary that there is actually such algorithm with expected memory usage $o(n \log |\Sigma|)$. Observe that any algorithm with multiplicative error $(1 + \epsilon)$ can also be considered as having an additive error $E = n \frac{2 - \epsilon}{1 + \epsilon}$. Thus our assumed algorithm reports with additive error $E \leq (\frac{1}{4} - \epsilon')n$, which is direct contradiction with Theorem 6.2.

Now we focus on Monte Carlo approximation algorithms. We will be working with an auxiliary problem $\text{MID-PALIN}_\Sigma[n]$, where we are interested in computing the radius of the middle palindrome, that is a palindrome with the center exactly in the half of a word of length $n$ over an alphabet $\Sigma$.

We start with a technical Lemma 6.4 showing than any algorithm solving $\text{MID-PALIN}_\Sigma[n]$ in small memory must have a large error probability. By the Yao’s minimax lemma, it is enough to construct a distribution over the inputs, such that any deterministic algorithm using small memory is not able to distinguish between two inputs with different answers reasonably often by a counting argument. Then we apply the lemma to prove that any algorithm solving $\text{MID-PALIN}_\Sigma[n]$ with small error probability must necessarily have a high memory usage in Lemma 6.5. This is possible because we can amplify the error probability by running multiple instances of the same algorithm in parallel.

**Lemma 6.4.** Let $A$ be a Monte Carlo streaming algorithm solving $\text{MID-PALIN}_\Sigma[n]$ exactly. If $A$ uses less than $\left\lfloor \frac{n}{2} \log |\Sigma| \right\rfloor$ bits of memory, then its error probability is $\Omega(\frac{1}{n|\Sigma|})$.

**Proof.** By Theorem 6.1, it is enough to construct probability distribution $\mathcal{P}$ over $\Sigma^n$ such that for any deterministic algorithm $D$ using less than $\left\lfloor \frac{n}{2} \log |\Sigma| \right\rfloor$ bits of memory, the expected probability of error on a word chosen according to $\mathcal{P}$ is $\Omega(\frac{1}{n|\Sigma|})$.

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

Since there are $|\Sigma|^{n'} = 2^{n' \log |\Sigma|} \geq 2 \cdot 2^{\frac{n}{2} \log |\Sigma|} - 1$ possible strings of length $n'$, we can partition at least half of them into pairs $(x, x')$, such that $D$ is in the same state after reading either $x$ or $x'$. (If we choose an arbitrary maximal matching of strings into pairs, at most half of possible strings will be left unpaired, that is one per each possible state of $D$. ) Let $s$ be longest common suffix of $x$ and $x'$, so $x = vcs$ and $x' = v'c's$, where $c \neq c'$ are single characters. Then
$D$ returns the same answer on $w(x, n' - |s|, c)$ and $w(x', n' - s, c)$, even though
the radius of the middle palindrome is exactly $|s|$ in one of them, and at least
$|s| + 1$ 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 $\frac{1}{2n^{|\Sigma|}} = \Omega(\frac{1}{n^{|\Sigma|}})$.

\begin{lemma}
Let $A$ be any randomized Monte Carlo streaming algorithm solving
MID-PALIN\(_\Sigma\)[\(n\)] exactly with probability $1 - \frac{1}{n^{|\Sigma|}}$. Then there exists a constant $\gamma(c)$ such that $A$ uses at least $\gamma(c) \cdot n \log |\Sigma|$ bits of memory.
\end{lemma}

\begin{proof}
We will use the standard amplification technique. Say that we have a Monte Carlo streaming algorithm, which solves MID-PALIN\(_\Sigma\)[\(n\)] exactly with error probability $\varepsilon$ 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 $O(ks(n))$ bits of memory and its error probability $\varepsilon_k$ satisfies:

$$
\varepsilon_k \leq \sum_{2i>k} \binom{k}{i} (1 - \varepsilon)^i \varepsilon^{k-i} \leq 2^k \cdot \varepsilon^k = (2\varepsilon)^k.
$$

Let us choose $\gamma(c) = \frac{1}{2} \max(\log(2/\varepsilon), 1) = \Theta(1)$, where the constant in $\Omega(1/(n|\Sigma|))$ is chosen as in Lemma 6.4. Now we can prove the theorem. Assume that $A$ uses less than $\gamma(c) \cdot n \log |\Sigma|$ bits of memory. Then running $\left \lfloor \frac{1}{2^{\gamma(c)}} \right \rfloor$ instances of $A$ in parallel requires less than $\left \lfloor \frac{1}{2^{\gamma(c)}} \right \rfloor$ bits of memory. But then the error probability of the new algorithm is bounded from above by:

$$
\left( \frac{2}{n^c} \right)^{\frac{1}{2^{\gamma(c)}}} = \frac{1}{n^{|\Sigma|}}
$$

which contradicts Lemma 6.4. \qed

Finally, we are ready to provide a lower bound on solving PALIN\(_\Sigma\)[\(n\)] exactly. We apply the already seen trick of padding the input in the middle, so that the longest palindrome is always the middle palindrome. Then, by reduction, the previously proven lower bound on the memory required to solve MID-PALIN\(_\Sigma\)[\(n\)] exactly applies.

\begin{theorem} (Monte Carlo exact). Let $A$ be any randomized Monte Carlo streaming algorithm solving PALIN\(_\Sigma\)[\(n\)] exactly with high probability. Then there exists a constant $\gamma'(c)$ such that $A$ uses at least $\gamma'(c) \cdot n \log |\Sigma|$ bits of memory.
\end{theorem}

\begin{proof}
We will reduce solving MID-PALIN\(_\Sigma\)[\(n\)] to solving PALIN\(_\Sigma\)[\(3n\)]. Let $n' = \frac{n}{2}$ and $x = x[1]x[2]\ldots x[n]$ be the input for MID-PALIN\(_\Sigma\)[\(n\)]. We define:

$$
w(x) = x[1]x[2]x[3]\ldots x[n']1000\ldots 01x[n'+1]\ldots 1x[n].
$$

Now if the radius of the middle palindrome in $x$ is $k$, then $w(x)$ contains a palindrome of radius at least $n + 2k$. In the other direction, any palindrome

\pagebreak
inside \( w(x) \) of radius at least \( n \) must be centered somewhere in the middle block consisting of only zeroes, and then it follows that it must be the middle palindrome there. Thus, the longest palindrome inside \( w(x) \) is of length exactly \( n + 2k \), so we have reduced solving \( \text{MID-PALIN}_\Sigma[n] \) to solving \( \text{PALIN}_\Sigma[3n] \), and by Lemma 6.5 solving \( \text{PALIN}_\Sigma[n] \) requires \( \gamma'(c) \cdot n \log |\Sigma| \) bits of memory for some constant \( \gamma'(c) \). Notice that the reduction needs \( \mathcal{O}(\log n) \) additional bits of memory to count up to \( n \), but this is much smaller than the lowerbound anyway.

Observe that although both \( \gamma(c) \) and \( \gamma'(c) \) are constant in terms of \( n \), they depend on the choice of the exponent \( c \) in the bound on the probability. To be specific, \( \lim_{c \to \infty} \gamma(c) = \infty \) and \( \lim_{c \to \infty} \gamma'(c) = \infty \).

Before we move to the additive and the multiplicative approximation, we need one more technical lemma. If a Monte Carlo algorithm is to recognize inputs containing long palindromes, the number of bits of memory it uses needs to be at least logarithmic with respect to the inverse of the error probability. This can be shown by a counting argument, where we look at the state of the algorithm after having read the half of the input, and pad the input so that it either contains very long palindrome, or only quite short palindromes.

**Lemma 6.7.** Let \( A \) be any randomized Monte Carlo streaming algorithm solving \( \text{PALIN}_\Sigma[n] \) with an additive error \( E \leq \frac{n}{8} \) using \( s(n) < \frac{n}{4} \) bits of memory. Then the error probability of \( A \) is \( \Omega(\frac{1}{|\Sigma|^n}) \).

**Proof.** By Theorem 6.1 it is enough to construct a probability distribution \( \mathcal{P} \) over \( \Sigma^n \), such that for any deterministic algorithm \( D \) using at most \( s(n) \) bits of memory, the expected probability of error on a word chosen according to \( \mathcal{P} \) is \( \Omega(\frac{1}{|\Sigma|^n}) \).

Let \( n' = \frac{3}{2} n \). For any \( x, y \in \Sigma^{n'} \), let \( w(x, y) = 0^{n'-1}1xy10^{n'-1} \). Observe that if \( x = y \) then \( w(x, y) \) contains a palindrome of radius \( \frac{n}{2} \), and otherwise the longest palindrome there has radius at most \( \frac{3}{2} n \), thus any algorithm with an additive error at most \( \frac{n}{8} \) must be able to distinguish between these two cases.

Let \( S \subseteq \Sigma^{n'} \) be an arbitrary family of words of length \( n' \) such that \( |S| = 2 \cdot 2^{\delta(n)} \), and let \( \mathcal{P} \) be the uniform distribution on all words 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 \( \frac{|S|}{2} \) pairs \( (x, x') \) of elements from \( S \) such that the state of \( D \) is the same after having read \( 0^{n'-1}1x \) and \( 0^{n'-1}1x' \). (If we create the pairs greedily, at most one such \( x \) per state of memory can be left unpaired, so at least \( |S| - 2^{\delta(n)} = \frac{|S|}{2} \) 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 \( \frac{1}{2} \cdot \frac{1}{|S|} = \Omega(\frac{1}{|\Sigma|^n}) \). \qed

Now we are ready to present a lower bounds on Monte Carlo additive and multiplicative approximations. The former follows from the lower bound for exact algorithms, as we can always pad the input with \( E \) zeroes between every two consecutive letters. The latter uses a similar argument, but there the
padding increases the length of the input exponentially. In both cases, we are really simulating running the algorithm for the exact version on the new input, which requires being able to construct the padding. Constructing the padding needs a logarithmic number of additional bits for storing a counter, and we need to be careful so that this is not too much.

**Theorem 6.8** (Monte Carlo additive approximation). Let \( \mathcal{A} \) be any randomized Monte Carlo streaming algorithm solving \( \text{PALIN}_{\Sigma} [n] \) with an additive error \( E \leq \frac{n}{\log n} \) with high probability. Then \( \mathcal{A} \) uses \( \Omega(\log n) \) bits of memory, and if \( E = \mathcal{O}\left(\frac{n}{\log n}\right) \), then \( \mathcal{A} \) uses \( \Omega\left(\frac{n}{E} \log |\Sigma|\right) \) bits of memory.

**Proof.** First, observe that the \( \Omega(\log n) \) part follows trivially from Lemma 6.7, as any algorithm with the error probability bounded by \( \mathcal{O}\left(\frac{1}{\log n}\right) \) have to use \( \Omega(\log n) \) bits of memory.

To prove the \( \Omega\left(\frac{n}{E} \log |\Sigma|\right) \) part, we will pad the input word with separating zeroes, which will allow us to use the lower bound for the exact version. Assume that there is a Monte Carlo streaming algorithm \( \mathcal{A} \) solving \( \text{PALIN}_{\Sigma} [n] \) with an additive error \( E \) using \( o\left(\frac{n}{E} \log |\Sigma|\right) \) bits of memory and returning the correct answer with probability \( 1 - \frac{1}{n^2} \). Let \( n' = \frac{n-E}{E+1} \). Given a word \( x[1]x[2] \ldots x[n'] \), we can simulate running \( \mathcal{A} \) on \( 0^E x[1]0^E x[2]0^E x[3] \ldots 0^E x[n']0^E \) to get \( R \) (using \( \mathcal{O}(\log E) \) additional bits of memory), and then return \( \left\lceil \frac{n}{E+1} \right\rceil \). We call this new Monte Carlo streaming algorithm \( \mathcal{A}' \). Recall that \( \mathcal{A} \) reports the length of the longest palindrome with an additive error \( E \). Therefore, if the original word contains a palindrome of length \( \ell \), the new word contains a palindrome of length \( E + \ell(E+1) \), so \( R \geq \ell(E+1) \) and \( \mathcal{A}' \) will return at least \( \ell \). In the other direction, if \( \mathcal{A}' \) returns \( \ell \), then the new word contains a palindrome of length \( \ell(E+1) \).

Such palindrome can be always extended to the left and to the right so that it starts and ends with \( 0^E \), and then it is clear that it corresponds to a palindrome of length at least \( \ell \) in the original word. Therefore, \( \mathcal{A}' \) solves \( \text{PALIN}_{\Sigma} [n'] \) exactly with probability \( 1 - \frac{1}{(n'(E+1)+E)^c} \geq 1 - \frac{1}{n'^c} \), that is with high probability, and uses \( o\left(n'(E+1)+E \right) \log |\Sigma| + \mathcal{O}(\log E) = o(n' \log |\Sigma|) + \mathcal{O}(\log n) \) bits of memory. Thus for \( c \) large enough, \( \gamma'(c) \cdot n' \log |\Sigma| \) from Theorem 6.6 dominates both \( \mathcal{O}(\log n) \) (which does not depend on \( c \)) and \( o(n' \log |\Sigma|) \), so we have a contradiction.

**Theorem 6.9** (Monte Carlo multiplicative approximation). Let \( \mathcal{A} \) be any randomized Monte Carlo streaming algorithm solving \( \text{PALIN}_{\Sigma} [n] \) with a multiplicative error \( (1 + \varepsilon) \) with high probability. If \( \varepsilon = \Omega(n^{-0.99}) \), then \( \mathcal{A} \) uses \( \Omega\left(\frac{\log n}{\varepsilon} \log |\Sigma|\right) \) bits of memory.

**Proof.** Assume that there is a Monte Carlo streaming algorithm \( \mathcal{A} \) solving \( \text{PALIN}_{\Sigma} [n] \) with multiplicative error \( (1 + \varepsilon) \) with probability \( 1 - \frac{1}{n^c} \) using \( o\left(\frac{\log n}{\varepsilon} \log |\Sigma|\right) \) bits of memory.

Let \( x = x[1]x[2] \ldots x[n']x[n'+1] \ldots x[2n'] \) be the input for \( \text{MID-PALIN}_{\Sigma} [2n'] \). We define an auxiliary word \( v_i = 0^i 1^i 0^i 1^i \ldots 0^i 1^i \) and choose \( n \) so that \( n = (1 + \varepsilon)n' \cdot n^{0.99} \). Then \( n' = \log_{(1+\varepsilon)} n^{0.01} = \Theta\left(\frac{\log n}{\varepsilon}\right) \). We choose \( i_1 \) so that
\[ |v_1| = (1 + \varepsilon)n^{0.99} \text{ and } i_2, i_3, \ldots, i_{n'} \text{ so that } |v_{i_j}| = (1 + \varepsilon)^{j-1} \cdot \varepsilon \cdot n^{0.99} \text{ for all } j \geq 1. \] Then, for any \( d \geq 0 \), \( |v_1| + \ldots + |v_d| = (1 + \varepsilon)^d \cdot n^{0.99} \) (Observe that for \( \varepsilon = \Omega(n^{-0.99}) \) and large \( n \), \( |v_1|, \ldots, |v_d| \geq 2 \)). Finally we define:

\[ w(x) = x[1]v_{i_1}^R, x[2]v_{i_2}^R, \ldots x[n']v_{i_n'}^R, x[n'+1]v_{i_1} \ldots v_{i_{n'}}x[2n']. \]

If \( x \) contains a middle palindrome of radius \( k \), then \( w(x) \) contains a middle palindrome of radius \( (1 + \varepsilon)^k \cdot n^{0.99} \). Also, any non-middle palindrome in \( w(x) \) has radius at most \( \sqrt{i_{n'}} = \sqrt{n \cdot \frac{\varepsilon}{1+\varepsilon}} \), which is smaller than \( n^{0.99} \) for \( n \) large enough. Thus, if \( A \) approximates the middle palindrome in \( w(x) \) with multiplicative error \( (1 + \varepsilon) \) with probability \( (1 - \frac{1}{n^c}) \), that is with high probability, using \( o(\frac{n^{0.99}}{\log |\Sigma|}) \) bits of memory, we can construct a new algorithm \( A' \) solving \textsc{MID-PALIN}_{\Sigma}[2n'] exactly with probability \( (1 - \frac{1}{n^c}) \), that is with high probability. But then \( A' \) uses \( o(n' \log |\Sigma|) \) bits of memory for running \( A \) and \( \Theta(\log n) \) additional memory. Now we can choose value of \( c \) large enough that the term \( \gamma(c) \cdot n' \log |\Sigma| \) from Lemma 6.5 is greater than \( \Theta(\log n) \), since the latter does not depend on \( c \). Also, \( \gamma(c) \cdot n' \log |\Sigma| = \Theta(n' \log |\Sigma|) \) for \( n' \) large enough dominates \( o(n' \log |\Sigma|) \). Thus we have a contradiction with Lemma 6.5. \( \square \)

References

[1] Alberto Apostolico, Dany Breslauer, and Zvi Galil. Parallel detection of all palindromes in a string. *Theor. Comput. Sci.*, 141(1&2):163–173, 1995.

[2] Petra Berenbrink, Funda Ergün, Frederik Mallmann-Trenn, and Erfan Sadeqi Azer. Palindrome Recognition In The Streaming Model. In Ernst W. Mayr and Natacha Portier, editors, *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, volume 25 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 149–161, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[3] Dany Breslauer and Zvi Galil. Real-time streaming string-matching. In Raffaele Giancarlo and Giovanni Manzini, editors, *CPM*, volume 6661 of *Lecture Notes in Computer Science*, pages 162–172. Springer, 2011.

[4] Funda Ergün, Hossein Jowhari, and Mert Saglam. Periodicity in streams. In Maria J. Serna, Ronen Shaltiel, Klaus Jansen, and José D. P. Rolim, editors, *APPROX-RANDOM*, volume 6302 of *Lecture Notes in Computer Science*, pages 545–559. Springer, 2010.

[5] N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the AMS*, 16:109–114, 1965.

[6] Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *Proceedings of the Twenty-seventh Annual ACM
[7] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[8] Glenn K. Manacher. A new linear-time “on-line” algorithm for finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, 1975.

[9] Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *FOCS*, pages 315–323. IEEE Computer Society, 2009.

[10] Andrew Chi-Chih Yao. Probabilistic computations: Toward a unified measure of complexity (extended abstract). In *FOCS*, pages 222–227. IEEE Computer Society, 1977.