Algorithmica (2022) 84:742–756
https://doi.org/10.1007/s00453-021-00883-y

Computing The Maximum Exponent in a Stream

Oleg Merkurev[1] · Arseny M. Shur[1]

Received: 3 September 2020 / Accepted: 11 October 2021 / Published online: 3 November 2021

Abstract
We consider the streaming version of the following problem: given an input string s of length n, find the maximum exponent of a substring of s. We prove that any algorithm deciding, w.h.p., whether a string contains a square, uses memory of size $\Omega(n)$, and thus does not satisfy the limitations of the streaming model. Thus the considered problem has no exact solution in the streaming model. Our main result is a Monte Carlo algorithm which computes the maximum exponent up to an additive error $\varepsilon < 1/2$: it outputs a number α such that s has a substring of exponent α but no substrings of exponent $\alpha + \varepsilon$ or higher. The algorithm uses $\mathcal{O}(\frac{\log^2 n}{\varepsilon})$ words of memory and performs $\mathcal{O}(\log n)$ operations, including dictionary operations, per input symbol.

Keywords Stringology · Streaming algorithm · Periodic string · Exponent · Gapped repeat

Mathematics Subject Classification 68W32 · 68W27 · 68R15

1 Introduction

The exponent of a string, which is the ratio between its length and its minimum period, is a natural measure of global periodicity. At the same time, the maximum exponent of a substring, also known as *local* or *critical* exponent of a string, measures local periodicity. The study of exponents and local exponents of strings can be traced back to the seminal papers by Thue [19,20]. Here we focus on the algorithmic aspects. In the usual RAM model, the exponent of a string can be easily computed online in linear time; for example, the pattern preprocessing in the Knuth–Morris–Pratt string matching algorithm [11] gives the exponent of the pattern for free. Note that this

✉ Arseny M. Shur
arseny.shur@urfu.ru

Oleg Merkurev
o.merkuryev@gmail.com

[1] Ural Federal University, Ekaterinburg, Russia

result holds for any alphabet. For local exponents, the things are more complicated. The "easy" case occurs when a string contains *periodic* substrings (a string is periodic if its minimal period is at least twice smaller than its length). Assuming polynomial integer alphabet (which is a common assumption in stringology), all maximal periodic substrings, or *runs*, can be found in linear time [13]. Over a general alphabet, the asymptotically best known algorithm, proposed in [6], finds all runs in time $\mathcal{O}(n\cdot\alpha(n))$, where α is the inverse Ackermann function. In the opposite case, where the string contains no periodic substrings, only the case of a constant-size alphabet was analyzed. Here a linear-time algorithm was designed in [3]. This algorithm makes an explicit use of the lower bound on the local exponents of long strings ("repetition threshold"), which depends on the alphabet. We did not find any results for more general alphabets. A related problem of reporting all maximal substrings with exponents greater than a given threshold was considered, also over a constant-size alphabet, in [7,12]. The solution in optimal $\mathcal{O}(n/\varepsilon)$ time for the threshold $(1+\varepsilon)$ was presented in [7], together with a detailed survey of related results.

We consider the problem of computing the local exponent of a stream. The streaming model of computation became quite popular in stringology in the last decade, after impressive results on streaming string matching [5,16]. In the streaming model, the input string arrives online symbol by symbol; after reading a symbol, the algorithm must compute the answer for the string it has read. The main restriction is that the string cannot be stored after reading, because the available amount of memory is sublinear. This restriction is quite severe, and so most of the streaming algorithms are approximate, randomized, or both. Often, only Monte Carlo approximation algorithms can reach sublinear memory. This is the case for the computation of the local exponent as well as for other closely related problems: finding a longest repeated substring [14] and computing all runs [15]. The latter problem is closely related to finding the local exponent: it covers the case where the stream contains runs *that can be detected by a streaming algorithm.*

Our results are as follows. First we show that no streaming algorithm, even a Monte Carlo one, can compute the exact local exponent of a stream. More precisely, we show that it is impossible to correctly compare, using a sublinear amount of memory, the local exponent of a string with 2. Next we formulate the problem of approximating the local exponent with an additive error and then present the main result, which is the following theorem.

Theorem 1 *There exists a Monte Carlo streaming algorithm which*

- *given a length-n input stream s over a polynomial integer alphabet and an error parameter $\varepsilon \in (0; \frac{1}{2}]$, returns the number α such that s has a substring of exponent α but no substrings of exponent $\alpha+\varepsilon$ or higher;*
- *uses $\mathcal{O}(\frac{\log^2 n}{\varepsilon})$ words of memory and performs $\mathcal{O}(\log n)$ operations, including dictionary operations, per input symbol.*

To prove Theorem 1, both the results and the technique of [15] are heavily used.

2 Preliminaries

Let s denote a string of length n over an alphabet $\Sigma = \{1, \ldots, \sigma\}$, where σ is polynomial in n. We write $s[i]$ for the ith symbol of s and $s[i..j]$ for its *substring* (or *factor*) $s[i]s[i+1]\cdots s[j]$. Thus, $s[1..n] = s$. By convention, $s[i..j]$ is an empty string if $j < i$. A *prefix* (resp. *suffix*) of s is a substring of the form $s[1..j]$ (resp., $s[j..n]$). If $w = s[i..j]$, we say that w *occurs* (or *has an occurrence*) in s at position i. A *period* of s is a positive integer p such that $s[1..n-p] = s[p+1..n]$; $\mathsf{per}(s)$ denotes the minimum period of s. We often refer to the strings having period p as *p-periodic*. The *exponent* of s is the ratio $\exp(s) = |s|/\mathsf{per}(s)$, where $|s|$ denotes the length of s. The number $\mathsf{lexp}(s) = \max\{\exp(s[i..j]) \mid 1 \le i < j \le n\}$ is the *local* (or *critical*) exponent of s.

A *repetition of period p* is a string s such that $p = \mathsf{per}(s) \le |s|/2$. Thus, s is a repetition iff $\exp(s) \ge 2$; repetitions of exponent 2 are called *squares*. A string containing no repetitions (or, equivalently, no squares) is called *square-free*. A repetition $s[i..j]$ of period p is called a *run* (in s) if both $s[i-1..j]$ and $s[i..j+1]$, whenever exist, have no period p. Following [7], we use the term *subrepetition* for any string s satisfying $1 < \exp(s) < 2$. Such a string can be factorized as $s = uvu$, where $|uv| = \mathsf{per}(s)$; u is its *border*.

We work in the *streaming model* of computation: the input string $s[1..n]$ (the *stream*) is read from 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 (in this paper, log stands for the binary logarithm).

A *Monte Carlo algorithm* gives a correct answer with high probability (at least $1 - \frac{1}{n}$ on a length n input) and has deterministic working time and space. For the related streaming problems of finding a longest repeat and computing all runs [14,15], Monte Carlo approximation algorithms were used, and proofs that other types of algorithms cannot work in the streaming model were presented. The present paper follows the same pattern.

The main result of [15] is heavily related to the computation of local exponents, so we give its precise formulation. Let $\mathsf{approxRuns}$ denote the following approximate version of the problem of computing all runs in a stream:

- given an input string s and an error parameter $\varepsilon = \varepsilon(n) \in (0, \frac{1}{2}]$, report a set of substrings of s such that
  - (i) for each run of exponent $\alpha \ge 2+\varepsilon$ in s a single substring of this run is reported, having the same period as the run itself and the exponent at least $\alpha - \varepsilon$;
  - (ii) for runs of smaller exponent, zero or one substring of each run is reported; if a substring is reported, it has the same period as the run and the exponent at least 2.

In the algorithm which solved $\mathsf{approxRuns}$ we extended the set of elementary operations with dictionary operations (insert, delete, lookup). The optimal choice of dictionary depends on ε. The following theorem is the main result of [15].

Theorem 2 *There is a Monte Carlo streaming algorithm that solves $\mathsf{approxRuns}$ performing $\mathcal{O}(\log n)$ operations per read and using $\mathcal{O}(\frac{\log^2 n}{\varepsilon})$ words of memory.*

The algorithm from Theorem 2 can be used to approximate $\mathsf{lexp}(s)$ whenever $\mathsf{lexp}(s) \geq 2+\varepsilon$ (and sometimes this algorithm will be lucky to approximate $\mathsf{lexp}(s)$ in the case $2 \leq \mathsf{lexp}(s) < 2+\varepsilon$). The details are given in Sect. 4.

3 The Square-Freeness Problem

As it is known since Thue [19], there exist arbitrarily long square-free strings over three or more letters. Let $\mathsf{SqFree}(\Sigma, n)$ be the following decision problem:

– given a length-n stream over the alphabet Σ of size $\sigma > 3$, decide whether the stream is square-free.

The following theorem shows that sublinear memory is insufficient to solve this problem with high probability.

Theorem 3 *There is a constant γ such that every algorithm solving the problem SqFree(Σ, n) with probability at least $1-\frac{1}{n}$ uses at least $\gamma n \log \sigma$ bits of memory.*

Proof The scheme of proof is rather standard for this sort of results. We first use Yao's minimax principle [21] and then finalize the intermediary result exploiting the so-called amplification trick.
Step 1. Assume that some Monte Carlo streaming algorithm solves $\mathsf{SqFree}(\Sigma, n)$ exactly using less than $\lfloor \log F \rfloor$ bits of memory, where F is the number of square-free strings of length $n' = \frac{n}{2} - 1$ over $\sigma - 1$ letters. Let us prove that its error probability is at least $\frac{1}{n\sigma}$. According to Yao's minimax principle, it is sufficient to construct a probability distribution $\mathcal{Q}$ over Σ^n such that for any deterministic algorithm D using less than $\lfloor \log F \rfloor$ bits of memory, the expected probability of error on a string chosen according to $\mathcal{Q}$ is at least $\frac{1}{n\sigma}$.

We define a length-n string $w(x, k, c)$ as follows. We fix $\$ \in \Sigma$ and let $\Sigma_1 = \Sigma - \{\$\}$. Next we fix two arbitrary square-free strings u_1 and u_2 of length n' over Σ_1; the only restriction is that their first letters are distinct. Let x be a square-free string of length n' over Σ_1, $c \in \Sigma_1$, and $k \in \{1, \dots, n'\}$. Then

$$w(x, k, c) = x[1..n']\,\$c\,x[n'-k+2..n']\,\$\,u_i[1..n'-k],$$

where $i = 1$ if $u_1[1] \neq c$ and $i = 2$ otherwise. Due to the separators \$, the string $w(x, k, c)$ contains a unique square $(x[n'-k+1..n']\,\$)^2$ if $c = x[n'-k+1]$; in the case $c \neq x[n'-k+1]$, $w(x, k, c)$ is square free iff $cx[n'-k+2..n']$ is square free. Let $\mathcal{Q}$ be the uniform distribution over all strings $w(x, k, c)$.

Since the available memory is insufficient to distinguish between any two square-free strings from $\Sigma_1^{n'}$, there exists an "indistinguishable" pair (x, x') of such strings; that is, D is in the same state after reading either x or x'. Let $x = vcs$, $x' = v'c's$, where $v, v', s \in \Sigma_1^*$, $c, c' \in \Sigma_1$, and $c \neq c'$. Then D returns the same answer on $w(x, |s|+1, c)$ and $w(x', |s|+1, c)$, because the right halves of these two strings coincide. However, $w(x, |s|+1, c)$ contains the square $(cs\$)^2$, while $w(x', |s|+1, c)$ is square free. Therefore, D errs on one of the analysed inputs; similarly, it errs on either $w(x, |s|+1, c')$ or $w(x', |s|+1, c')$.

Consider an arbitrary maximal set of disjoint pairs (x, x') of square-free strings from $\Sigma_1^{n'}$, where the strings in each pair are indistinguishable by D. The memory of D has no more than $2^{\lfloor \log F \rfloor - 1} \leq F/2$ states. Since at most one string per state is left unpaired, the number of pairs is at least $F/4$. As was shown above, each pair causes two errors by D, to the total of at least $F/2$ errors. The number of strings in the distribution $\mathcal{Q}$ is $F \cdot n' \cdot (\sigma - 1)$, which implies that the probability of error is greater than $\frac{1}{n\sigma}$.
Step 2. To use amplification, we relate the parameter F to n and σ. Namely, we show that there exists a positive constant δ such that $\lfloor \log F \rfloor \geq \delta n \log \sigma$. Let $C_k(n)$ be the number of k-ary square-free strings of length n. Then $C_k(n)$ is an exponentially-growing function of n [4]. As was shown in [17], the base α_k of this exponential function satisfies, as a function of k, the condition $\alpha_k = (k-1) - 1/(k-1) - 1/(k-1)^3 + O(1/k^5)$. By definition of F we have $F = C_{\sigma-1}(n')$; so we can write $F > d(\sigma - 3)^{n/2-1}$ for some positive constant d and thus $\log F > (\frac{n}{2} - 1)\log(\sigma - 3) + \log d$. If $\sigma \geq 5$, this inequality implies the announced lower bound. For the remaining case $\sigma = 4$ we use a more precise bound $\alpha_3 > 1.3$; see [18, Theorem 4] for the method of obtaining lower bounds and [18, Table A.2] for numerical results. Hence in this case $F > d' {\cdot} 1.3^{n/2-1}$ for some positive constant d', and then $\log F > (\frac{n}{2} - 1)\log 1.3 + \log d' = \Omega(n) = \Omega(n \log \sigma)$. Thus we proved the existence of a positive δ such that $\lfloor \log F \rfloor \geq \delta n \log \sigma$.

Now assume that some Monte Carlo streaming algorithm A solves SqFree exactly with error probability $\varepsilon \leq \frac{1}{n}$ using $s(n)$ bits of memory. Then we can run its k instances simultaneously and return the most frequent answer. The new algorithm A_k uses $\mathcal{O}(k \cdot s(n))$ bits of memory and its error probability ε_k satisfies the inequality $\varepsilon_k \leq \sum_{2i<k} \binom{k}{i} (1-\varepsilon)^i \varepsilon^{k-i} \leq 2^k \cdot \varepsilon^{k/2} = (4\varepsilon)^{k/2}$. Recall that $\sigma = \mathcal{O}(n^p)$ for some constant p. Let $k = 2p + 3$ and take any positive $\gamma \leq \frac{\delta}{k}$. If $s(n) < \gamma n \log \sigma$, then algorithm A_k uses less than $\delta n \log \sigma \leq \lfloor \log F \rfloor$ bits of memory. On the other hand, the error probability of A_k is $\varepsilon_k \leq (4\varepsilon)^{k/2} \leq (\frac{4}{n})^{p+3/2}$, which is less than $\frac{1}{n\sigma}$ for n big enough because $\sigma = \mathcal{O}(n^p)$. From step 1 we know that this is impossible, so the theorem holds for the chosen value of γ. □

Theorem 3 shows that there is no hope to compute local exponents of streams exactly, because we cannot even correctly compare this exponent to 2 without an access to linear-size memory. Hence we come up with a natural approximation version $\mathsf{approxExp}$ of this problem:

- given a stream s and an error parameter $\varepsilon \in (0, \frac{1}{2}]$, find a number α such that $\alpha \leq \mathsf{lexp}(s) < \alpha + \varepsilon$.

The rest of the paper describes the algorithm solving $\mathsf{approxExp}$ within the resource limitations listed in Theorem 1. In fact, the algorithm does more: it is able to output a position of a substring of exponent α.

4 Tools

The algorithm solving $\mathsf{approxRuns}$ (Theorem 2) outputs substrings of s as triples (l, p, r) such that $s[l..r]$ is a repetition of period p, so it can keep track of the maximum exponent among these triples without spending additional time or space. This version, returning the maximum exponent found, is below referred to as Algorithm Rep. Note that if Algorithm Rep finds at least one repetition, then it solves approxExp and, by Theorem 2, satisfies the conditions of Theorem 1. So the problem is how to process the streams in which Algorithm Rep finds nothing.

We explore an obvious idea: design an approximation algorithm (Algorithm Sub) for square-free or "nearly square-free" streams and run it in parallel with Algorithm Rep. Both algorithms update the current maximum exponent. If a repetition is detected, we abort Algorithm Sub and continue running Algorithm Rep. In the description of Algorithm Sub we thus assume that no repetition was detected until the current iteration.

We make use of particular data structures and general organization of data introduced in [15] for Algorithm Rep. All necessary details are reproduced in this section.

4.1 Fingerprints, Frames, Checkpoints

We use Karp–Rabin fingerprints [10], which is a hash function ubiquitous in streaming string algorithms. Let p be a fixed prime from the range $[n^4, n^5]$, and r be a fixed integer randomly chosen from $\{1, \dots, p-1\}$. For a string s, its hash is defined as $\phi(s) = \left(\sum_{i=1}^{n} s[i]{\cdot}r^i\right) \bmod p$. The probability of hash collision for two strings of length m is at most m/p. Our algorithm compares hashes of strings having equal lengths of the form 2^j. The probability that a pair of such strings collide is less than n^3/p and thus less than the allowed error probability for Monte Carlo algorithms. Hence all further considerations assume that no collisions happen. For a string s, its *frame* is the tuple $(|s|, \phi(s), r^{|s|} \bmod p, r^{-|s|} \bmod p)$. The crucial property of frames is the following.

Lemma 1 *([5]) If the frames of any two of the strings A, B, AB are known, the frame of the third string can be computed in $\mathcal{O}(1)$ time.*

All definitions below refer to the input stream s. For any i, the ith iteration of a streaming algorithm processing s begins with reading $s[i]$ and ends just before reading $s[i{+}1]$. We write $I(i)$ for the frame of $s[1..i{-}1]$. Lemma 1 implies that one can compute $I(i{+}1)$ in $\mathcal{O}(1)$ time from $I(i)$ and $s[i]$.

All information currently stored by our algorithm is associated with *checkpoints*, which form a subset of all positions. Each position k becomes a checkpoint at the kth iteration and "lives" during $\mathsf{ttl}(k)$ iterations, where the time-to-live function is defined by $\mathsf{ttl}(k) = 2^{t_\varepsilon+2+\beta(k)}$ with $t_\varepsilon = \left\lceil \log \frac{2}{\varepsilon} \right\rceil$ and $\beta(k)$ being the maximum power of 2 dividing k. If $k + \mathsf{ttl}(k) = i$, then at the start of the ith iteration k "dies" (loses the status of checkpoint) and all associated information is deleted. See the example in Fig. 1.

Lemma 2 *([15])*

1) *The number of checkpoints at i'th iteration is $\mathcal{O}(\frac{\log i}{\varepsilon})$.*
2) *If $i - \mathsf{ttl}(i) > 0$, then the checkpoint $i - \mathsf{ttl}(i)$ dies at the i'th iteration. Otherwise, no checkpoint dies at this iteration.*

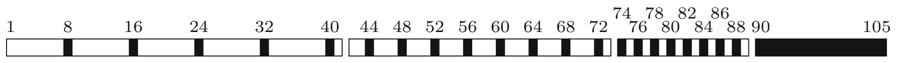


Fig. 1 The checkpoints (black) after the iteration $i = 105$ ($t_\varepsilon = 2$). For example, $\mathsf{ttl}(52) = 2^{2+2+2} = 64$, so the position 52 is a checkpoint until the iteration 116

Remark 1 We should say a few words about the functions $\beta(k)$ and $\lceil \log k \rceil$, both extensively used in our algorithm. Note that $\lceil \log k \rceil = \mathsf{msb}(k-1)+1$, where the function $\mathsf{msb}(x)$ returns the position of the most significant 1 in the binary representation of x. With the aid of fusion trees, $\mathsf{msb}(x)$ can be computed in constant number of operations with machine words; see [9]. A practical way to compute $\lceil \log k \rceil$ is to subtract the value $\mathsf{clz}(k)$ from the size of the machine word. The $\mathsf{clz}(x)$ function counts leading zeroes in the binary representation of an unsigned integer x in a machine word and is a standard CPU instruction. Further, $\beta(k) = \mathsf{ctz}(k)$, where the function $\mathsf{ctz}(k)$ counts trailing zeroes in the binary representation of k. Some CPU architectures contain $\mathsf{ctz}(k)$ as an instruction, but we can operate without it: it was shown in [15, Lemma 10] that all calls to $\beta()$ during one iteration can be performed in $\mathcal{O}(\log n)$ total time; this bound fits into Theorem 1.

4.2 Blocks, Groups, Navigation

Both Algorithm Rep and Algorithm Sub work with substrings of length 2^j, $j = 0, \ldots, \lfloor \log n \rfloor$. The notions introduced below are illustrated by Fig. 2. For each iteration i and each j we distinguish a subclass of substrings of length 2^j called *j-blocks*. A substring u of length 2^j of the stream s is a j-block (at the ith iteration) if it occurs in $s[1..i]$ at some checkpoint; i.e., if $u = s[k..k{+}2^j{-}1]$ where k is a checkpoint at the ith iteration. Note that u will lose the status of j-block at some future iteration if no checkpoints corresponding to the occurrences of u will remain alive. Below we say "checkpoint occurrence" instead of "occurrence at a checkpoint". For each j-block T we maintain a basic structure B_T called *group* and consisting of

- Frame of T
- Doubly-connected list *list* of all checkpoints, in increasing order, that are positions of occurrences of T
- Position *fresh* (described below) and its frame *fframe* $= I(\textit{fresh})$
- Number *ext* (described below)

Apart from the checkpoint occurrences of j-blocks, we are interested in their *fresh* occurrences (whether at checkpoints or not). An occurrence at position k is fresh at ith iteration, if $k > i - 2^{j+1} + 1$. This condition means that a fresh occurrence of a j-block was a suffix of the stream less than 2^j iterations below. So we immediately have

Observation 1 *Any two fresh occurrences of the same j-block overlap.*

By definitions of periods and repetitions, overlapping occurrences of the same string create a repetition. Hence Observation 1 implies

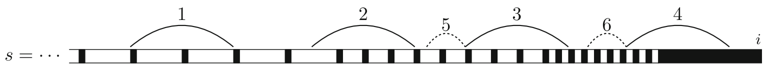


Fig. 2 Illustrating j-blocks and groups. Black positions are checkpoints; the arcs 1, 2, 3, 4 represent all occurrences of some substring T of length 2^j in the stream at the ith iteration. The occurrences 1 and 3 are at checkpoints (so T is a j-block), while the occurrences 2 and 4 are not. The group B_T contains the information about the occurrences 1, 3, and 4: the positions of 1 and 3 are in $B_T.list$, the position of 4 is $B_T.fresh$. The occurrence 2 is "forgotten": no information is stored about it. Finally, the arcs 5 and 6 represent equal substrings of length $B_T.ext$

Observation 2 *If the suffix of length* 2^j *of the current stream* $s[1..i]$ *is a* j*-block which already has a fresh occurrence at position* f*, then* $s[f..i]$ *is a repetition. Since* $s[f..f+2^j-1] = s[i-2^j+1..i]$*, this repetition has the period* $i - f + 1 - 2^j$ *by definition. Then* $\exp(s[f..i]) = \frac{i-f+1}{i-f+1-2^j}$.

Observation 3 *Algorithm Rep needs to memorize, in some compressed form,* all *fresh occurrences of each* j*-block. For Algorithm Sub, we simplified the structure of a group and store just one fresh occurrence. If a second fresh occurrence is detected, we use Observation 2, update the answer with the exponent* $\alpha > 2$ *of the repetition found, and stop the algorithm. The rest of the stream is then processed solely by Algorithm Rep.*

Finally, the extension number *ext* is used to memorize an additional information about the subrepetition uvu, where the right u is the fresh occurrence of a j-block and the left u is the previous checkpoint occurrence of the same block (depending on whether the fresh occurrence is at a checkpoint or not, the left u is either last or second last element of *list*). The number *ext* shows that the subrepetition uvu can be extended in s by *ext* symbols to the left, preserving the period.

Remark 2 A group is a constant-size structure (two frames, links to the beginning and the end of *list*, the numbers *fresh* and *ext*) plus a set of constant-size nodes (position, links to next and previous elements of the list). We store all groups in an array of constant-size cells endowed with a stack of empty cells. This way, creating a new group/node and deleting an existing group/node requires $\mathcal{O}(1)$ time. The size of the array is proportional to the number of groups plus the number of occurrences of j-blocks; both numbers are $\mathcal{O}(\frac{\log^2 n}{\varepsilon})$, as follows from Lemma 2(1).

For navigation we use five dictionaries described in the following table. The values in the first four dictionaries are stored as links.

Id	Key	Value
H_1	j, hash F	group of the j-block with hash F
H_2	j, checkpoint k	group of the j-block occurring at k
H_3	j, position k	group of the j-block having the fresh occurrence at k
HH	j, checkpoint k	node for k in the group of the j-block occurring at k
HC	checkpoint k	frame $I(k)$

5 Algorithm Sub

Let $w = s[t..i]$ be a substring, $p = \mathsf{per}(w)$. Let f be the smallest position satisfying $f \geq t$ and $\mathsf{ttl}(f) \geq 2p$, and let g be the largest position such that $g \leq i$ and $\mathsf{ttl}(g - p + 1) \geq 2p$. We call the substring $s[f..g]$ of w the *core* of w.

Lemma 3 *Every substring* $s[t..i]$ *with* $\exp(s[t..i]) \geq 1 + \varepsilon$ *has a nonempty core. The exponent of the core is greater than* $\exp(s[t..i]) - \varepsilon$.

Proof Let $\mathsf{ttl}(x) \geq 2p$ be the minimum time-to-live satisfying this inequality. Then $\mathsf{ttl}(x) = 2^{\lceil \log p \rceil + 1}$, so $\beta(x) = \lceil \log p \rceil - 1 - t_\varepsilon$ by the definition of ttl. Hence the distance between two consecutive positions with $\mathsf{ttl} \geq 2p$ is $2^{\beta(x)} < 2^{(\log p + 1) - 1 - \log \frac{2}{\varepsilon}} = \frac{\varepsilon p}{2}$. Therefore, the numbers f and g from the definition of the core satisfy $f - t, i - g < \frac{\varepsilon p}{2}$. Since $|s[t..i]| \geq (1 + \varepsilon)p$, the length of $s[f..g]$ is strictly greater than p. So $s[f..g]$ has period p as a substring of $s[t..i]$, and $\exp(s[f..g]) > \exp(s[t..i]) - \frac{\varepsilon p}{p} = \exp(s[t..i]) - \varepsilon$, as required. □

Lemma 3 shows that the core of a substring approximates its exponent with the desired precision. The idea of Algorithm Sub is to detect cores using the information about j-blocks, stored in groups. The detection becomes possible because the definition of a core implies that certain positions are checkpoints at the moment when the core is read (see Fig. 3). When a core is detected, its exponent is computed and used to update the answer. Some cores can be missed by the algorithm; the crucial fact, proved in Lemma 4 below, is that a core is missed only if some other substring of bigger exponent was detected before. This fact ensures the correctness of the answer found by Algorithm Sub.

One iteration of Algorithm Sub is presented below as Algorithm 1. As was already said, Algorithm Sub works in parallel with Algorithm Rep, which is responsible for "big" exponents. Moreover, the two algorithms share the auxiliary stages at each iteration, with some details simplified for Algorithm Sub. In line 1, we read a new symbol, compute the frame of the whole string and store it in the dictionary HC. Then three nontrivial stages follow. These stages are described in [15] as Algorithms 1–3, endowed with the proofs of correctness and time bounds. So here we just recall the performed operations in brief.

Stage 1 (line 2). Lemma 2 indicates the only checkpoint which can die at the current iteration. We process each of $\mathcal{O}(\log n)$ j-blocks at the checkpoint position, deleting the checkpoints from their groups and from dictionaries; groups without checkpoints are also deleted. The dictionaries H_2 and HH are used, an entry in HC is deleted.

Stage 2 (line 3). The j-blocks of the form $s[k..i]$, where k is a checkpoint, are processed. For each block we compute its hash, retrieving $I(k)$ from the dictionary HC and using Lemma 1. Then we extract the group B of this block from the table H_1; if B does not exist, it is created. A node for k is added to $B.list$, and an element is added to HH. The occurrence at k is fresh, so $B.fresh$ is set to k, $B.fframe$ is set to $I(k)$, and a new element is added to the dictionary H_3. If a fresh occurrence existed before, a repetition is detected, which implies the abortion of the algorithm; the rest of the stream will be processed solely by Algorithm Rep. The stage includes one more loop over j: using H_2, the expired fresh *checkpoint* occurrences are deleted.

Algorithm 1 (Algorithm Sub, ith iteration)

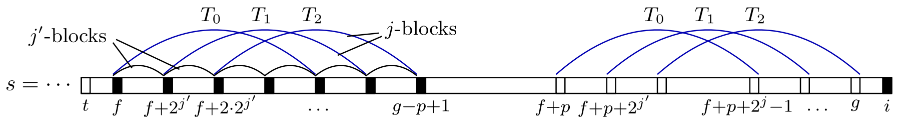
```
1: read $s[i]$; compute $I(i+1)$ from $I(i)$ and $s[i]$; add $I(i+1)$ to $HC$
2: if $i - \mathsf{ttl}(i) > 0$ then delete the checkpoint $i - \mathsf{ttl}(i)$    ▷ [15] Algorithm 1
3: update groups    ▷ [15] Algorithm 2
4: for $\{j \leftarrow 0; k = i - 2^j + 1$ && $k \geq 1; j$++$\}$ do
5:     find the group $B$ of the suffix $T = s[k..i]$    ▷ [15] Algorithm 3
6:     if $B = null$ then    ▷ $T$ has no checkpoint occurrences
7:         continue
8:     if $B.fresh < k$ then    ▷ two fresh occurrences: a repetition
9:         update $answer$ with $1 + \frac{2^j}{k - B.fresh}$; stop
10:    if $B.fresh = k$ then    ▷ $k$ is a checkpoint, added to $B$ in line 2
11:        $f \leftarrow B.list.top.next$    ▷ rightmost non-fresh checkpoint occurrence of $T$
12:    if $B.fresh = null$ then
13:        $f \leftarrow B.list.top$    ▷ rightmost checkpoint occurrence of $T$
14:        $B.fresh \leftarrow k$; $B.fframe \leftarrow I(k)$    ▷ setting the position of fresh occurrence of $T$
15:    $p \leftarrow k - f$; $j' \leftarrow \max\{0, \lceil \log p \rceil - 1 - t_\varepsilon\}$
16:    $B.ext = 0$
17:    $B' \leftarrow H_2(j, f - 2^{j'})$
18:    if $B'.fresh = k - 2^{j'}$ then
19:        $B.ext \leftarrow 2^{j'}$; $b_1 \leftarrow B'.list.top$; $b_2 \leftarrow B'.list.top.next$
20:        if $b_1 = f - 2^{j'}$ or ($b_1 = k - 2^{j'}$ and $b_2 = f - 2^{j'}$)) then
21:            $B.ext \leftarrow B.ext + B'.ext$
22:    update $answer$ with $1 + \frac{2^j + B.ext}{p}$    ▷ $p$-periodic subrepetition $s[f - B.ext..i]$ is found
```

Stage 3 (line 5). A trick is used here to find the hash of the suffix $T = s[k..i]$ of length 2^j in the case if k is not a checkpoint. If T has no checkpoint occurrences, it is useless in the search of (sub)repetitions, and we skip it. But if T has such an occurrence, then its prefix of length 2^{j-1} is a $(j-1)$-block occurring at the same checkpoint. We check at H_3 whether there is a $(j-1)$-block T' with the fresh occurrence at k. If yes, we extract the frame $I(k)$ as *fframe* of T' and get the hash of T by Lemma 1. The fresh occurrence of T' at k is then deleted as expired (it was not deleted at Stage 2 because k is not a checkpoint). After computing the hash of T, the group of T is retrieved from the dictionary H_1. If no such group exists (T' has checkpoint occurrences but T has not), we skip T (lines 6–7).

In lines 8–9, the algorithm stops according to Observation 3 if two fresh occurrences of T (at $B.fresh$ and at k) are found. Lines 10–22 are related to the main procedure: core detection.

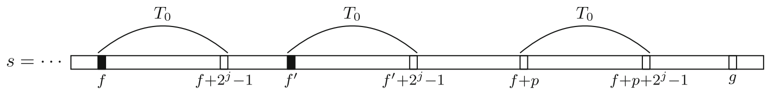
Lemma 4 *Suppose that Algorithm Sub processed a stream s and no repetitions were detected. Let $s[t..i]$ be a substring of s such that $\exp(s[t..i]) \geq 1+\varepsilon$, $p = per(s[t..i])$ and $s[t-1..i]$ is not p-periodic. Then Algorithm Sub detected either the core of $s[t..i]$ or some other substring of exponent greater than the exponent of this core.*

Proof Let $s[f..g]$ be the core of $s[t..i]$ (see Fig. 3 for an illustration). Since it has period p, $s[f..g-p] = s[f+p..g]$ by definition. Let us denote this repeated part by u. One has $|u| = g - f - p + 1$. Let $j = \lfloor \log |u| \rfloor$. Then u is strictly shorter than a $(j+1)$-block and (non-strictly) longer than a j-block. We prove the lemma by showing that Algorithm Sub followed one of two scenarios:



T_0 T_1 T_2
j'-blocks
j-blocks
T_0 T_1 T_2
$s = \cdots$
t f $f{+}2^{j'}$ $f{+}2{\cdot}2^{j'}$ $\dots$ $g{-}p{+}1$ $f{+}p$ $f{+}p{+}2^{j'}$ $f{+}p{+}2^j{-}1$ $\dots$ g i

Fig. 3 Finding the core $s[f..g]$ of a p-periodic substring $s[t..i]$ (Lemma 4). Black positions are checkpoints, white positions can have any status. In the picture, $j = j' + 2$ and $k = 6$



T_0 T_0 T_0
$s = \cdots$
f $f{+}2^j{-}1$ f' $f'{+}2^j{-}1$ $f{+}p$ $f{+}p{+}2^j{-}1$ g

Fig. 4 Illustrating the proof of Lemma 4: an additional occurrence of T_0 at position f'

(i) The core was detected at the gth iteration: for the j-block which is the suffix of $s[1..g]$ (T_2 in Fig. 3) the previous occurrence at a checkpoint was at distance p and $ext = |u| - 2^j$; these two conditions imply $s[f..g{-}p] = s[f{+}p..g]$;

(ii) A substring of exponent bigger than the exponent of the core was detected no later than at the gth iteration.

Let $j' = \max\{0, \lceil \log p \rceil - 1 - t_\varepsilon\}$. As shown in the proof of Lemma 3, consecutive positions with $\mathsf{ttl} \ge 2p$ are at distance $2^{j'}$. Since f and $g - p + 1$ have $\mathsf{ttl} \ge 2p$ by the definition of a core, $(g - p + 1) - f = |u| = h \cdot 2^{j'}$ for some $h \ge 1$. Hence each of the positions $f, f + 2^{j'}, \dots, f + (h{-}1) \cdot 2^j$ remains a checkpoint for at least $2^{\lceil \log p \rceil + 1}$ iterations and $u = s[f..g{-}p]$ is a concatenation of h j'-blocks at these checkpoints. These j'-blocks can be combined into overlapping j-blocks, denoted by $T_0, T_1, \dots, T_d$ as in Fig. 3. Note that $d = h - 2^{j-j'}$; in particular, if h is a power of 2, then $s[f..g{-}p]$ is a single j-block.

Now we are going to show that Algorithm Sub followed either scenario (i) or scenario (ii) with respect to the core $s[f..g]$. To do this, we analyze the iterations $i_0, \dots, i_d$ in which the right (in Fig. 3) occurrences of the j-blocks $T_0, T_1, \dots, T_d$ were suffixes of the stream; that is $i_r = f + p + 2^j + r2^{j'} - 1$ for $r = 0, \dots, d$ (thus $i_d = g$). First consider the iteration i_0. At this iteration, f was a checkpoint and the stream had the j-block T_0 at position $f{+}p$ as a suffix. Hence, Algorithm Sub found, in line 11 or line 13, the rightmost previous checkpoint occurrence of T_0. The checkpoint is either f or some $f' > f$. In the latter case, consider the three occurrences of T_0: at f, f', and $f{+}p$ (Fig. 4). These occurrences neither overlap nor touch: otherwise, T_0 had two fresh occurrences at some moment and a repetition was detected, which is impossible by the conditions of the lemma. Thus these occurrences form two subrepetitions $s[f..f'{+}2^j{-}1]$ and $s[f'..i_0]$ overlapping by the middle occurrence. One of these subrepetitions was detected at the iteration i_0 and the other one had been found earlier at the iteration $f' + 2^j - 1$. The sum of their periods is p. Then one of the periods is at most $p/2$, and the exponent of the corresponding substring is at least $1 + \frac{2^{j+1}}{p}$. By definition of j, $2^{j+1} > |u|$, so this exponent is greater than the exponent of the core, which is $1 + \frac{|u|}{p}$. Therefore, scenario (ii) was realized.

Now consider the case where the rightmost previous checkpoint occurrence of T_0 is at f. The two rightmost occurrences of T_0 do not overlap or touch because no repetition was detected. Then, in particular, $2^j < p$ (see Fig. 3). The algorithm computed f,

p, and j' in line 15 and set the extension of T_0 to 0 in line 16. Next, in line 17 the algorithm retrieved the j-block at position $f - 2^{j'}$. This position was a checkpoint at the considered iteration. Indeed, $\mathsf{ttl}(f - 2^{j'}) \geq 2p$ (as mentioned above, consecutive positions with $\mathsf{ttl} \geq 2p$ are at distance $2^{j'}$). Then $\mathsf{ttl}(f - 2^{j'}) \geq 2^{j+2}$ because $2^j < p$. It remains to note that 2^{j+2} is greater than the difference between the current position $i_0 = f + p + 2^j - 1$ and $f - 2^{j'}$. The retrieved j-block differs from the j-block at the position $f + p - 2^{j'}$ because $f - 2^{j'} < t$ by the definition of a core and $s[t-1] \neq s[t+p-1]$ be the conditions of the lemma. Hence the condition in line 18 is false (note that $k = p + f$); so the algorithm detected the subrepetition $s[f..i_0]$ and updated the answer with its exponent$1 + \frac{2^j}{p}$ in line 22.

Next consider the iteration i_1, in which the stream had the suffix T_1. At this iteration, the algorithm found the previous checkpoint occurrence of T_1. If the checkpoint is greater than $f + 2^{j'}$, then we repeat the above argument about three occurrences of T_0 for the occurrences of T_1 and conclude that scenario (ii) took place. Assume that the occurrence was found at $f + 2^{j'}$. In line 17, the j-block at the checkpoint f was retrieved; this block is T_0 and its fresh occurrence satisfies the condition in line 18. The extension of T_1 is set in line 19 to 2^j. The value $B'.ext$ had been set to 0 during the iteration i_0 we considered above; thus the iteration i_1 ended by updating the answer in line 22 with the exponent $1 + \frac{2^j + 2^{j'}}{p}$ of the subrepetition $s[f..i_1]$.

Finally we prove by induction the following claim: for any $r \leq d$ either scenario (ii) was detected no later than at the iteration i_r or the iteration i_r ended by updating the answer with the exponent $1 + \frac{2^j + r2^{j'}}{p}$ of the subrepetition $s[f..i_r]$. The base case is proved above; we proceed with the step case.

Consider the iteration i_r for some $r \geq 2$ and suppose that scenario (ii) was not detected till its end. At the iteration i_r, the algorithm found the previous checkpoint occurrence of T_r. If this checkpoint is greater than $f + r{\cdot}2^{j'}$, scenario (ii) is detected repeating the above argument for T_0 (Fig. 4). Since we assumed it was not detected, the previous checkpoint occurrence of T_r was at $f + r{\cdot}2^{j'}$. By the same argument, the previous checkpoint occurrence of T_{r-1}, found at the iteration i_{r-1}, was at $f + (r-1)2^{j'}$. Further, the iteration i_{r-1} ended with updating the answer with the exponent $1 + \frac{2^j + (r-1)2^{j'}}{p}$ of the string $s[f..i_{r-1}]$ by the inductive hypothesis. This means (see line 22) that the extension of the block T_{r-1} was set to $(r-1)2^{j'}$.

Consider the rest of the iteration i_r after the checkpoint $f + r{\cdot}2^{j'}$ was found. In line 17, the block T_{r-1} was retrieved. The condition in line 18 is true: it refers to the occurrence of T_{r-1} we analysed at the iteration i_{r-1}. In addition, the condition in line 20 holds: it says that when the algorithm processed the suffix T_{r-1} at that iteration, it found the checkpoint occurrence at $f + (r-1)2^{j'}$. Hence the algorithm set the extension of T_r to $2^{j'}$ at line 19 and updated it to $2^{j'} + (r-1)2^{j'} = r{\cdot}2^{j'}$ in line 21. Then in line 22 the answer is updated with the required exponent $1 + \frac{2^j + r2^{j'}}{p}$. The claim is proved.

For $r = d$, the claim says that if scenario (ii) was not detected, then at gth iteration the answer was updated with the exponent $1 + \frac{2^j + d2^{j'}}{p}$. As $2^j + d2^{j'} = h2^{j'} = |u|$, this is exactly the exponent of the core $s[f..g]$. This means that scenario (i) took place. The lemma is proved. □

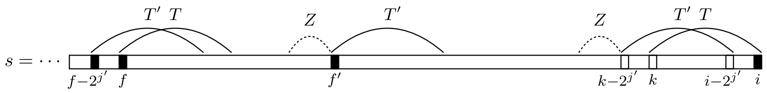


Fig. 5 Illustrating special case of Algorithm 1: the extensions of the j-blocks T and T' refer to subrepetitions with different periods. At the ith iteration, the condition in line 20 fails, preventing the algorithm from adding the extension of T' to the extension of T

Remark 3 The condition in line 20 prevents Algorithm Sub from an error in a tricky situation illustrated by Fig. 5. At the current iteration i, for a suffix T of length 2^j the rightmost checkpoint occurrence was found at position f. The period p and the number j' were computed, the j-block T' at the position $f - 2^{j'}$ was retrieved, and this block appeared to have a fresh occurrence at $k - 2^{j'}$. Then the extension of T is set to $2^{j'}$ (the subrepetition $s[f-2^{j'}..i]$ is p-periodic). However, it would be an error to further increase the extension of T by the extension of T', because they are related to different subrepetitions: the extension of T' reflects the equality of two substrings Z shown in the picture. The condition in line 20 fails (if $k - 2^{j'}$ is a checkpoint, then $b_1 = k - 2^{j'}$ and $b_2 = f'$; if not, then $b_1 = f'$), so no error happens.

Proof of Theorem 1 We run Algorithm Sub and Algorithm Rep in parallel (that is, after reading $s[i]$ the ith iteration of each algorithm is performed; the order does not matter). If s contains a run of exponent at least $2 + \varepsilon$, Algorithm Rep finds $\mathsf{lexp}(s)$ with the error less than ε. If any of the algorithms detects a repetition x (Algorithm Sub detects repetitions with the condition in line 8), then $\mathsf{lexp}(s) \geq \exp(x) \geq 2$. So either $\exp(x)$ is a valid answer to $\mathsf{approxExp}$, or $\mathsf{lexp}(s) \geq 2+\varepsilon$ and the answer can be found solely by Algorithm Rep. Hence, on detecting a repetition we stop Algorithm Sub and run Algorithm Rep for the rest of the stream.

Now suppose that no repetitions were found. Let x be a substring of s satisfying $\mathsf{lexp}(s) = \exp(x) \geq 1+\varepsilon$. By Lemma 4, the answer obtained by Algorithm Sub cannot be smaller than $\exp(z)$, where z is the core of x. By Lemma 3, $\exp(z) > \exp(x) - \varepsilon$. Hence Algorithm Sub solved $\mathsf{approxExp}$ correctly.

Algorithm Rep obliges the required space and time limitations by Theorem 2. It remains to estimate space and time consumed by Algorithm Sub. The space usage of Algorithm Sub is dominated by groups, which occupy $\mathcal{O}(\frac{\log^2 n}{\varepsilon})$ words of space by Remark 2. The logarithmic time bounds for lines 2, 3, and 5 were proved in [15]; the remaining part of the algorithm uses $\mathcal{O}(1)$ operations per value of j (for $\lceil \log p \rceil$ see Remark 1), which is $\mathcal{O}(\log n)$ per iteration. □

6 Conclusion and Open Questions

In this paper we presented the first streaming algorithm to compute the exponent of the input string. As often happens for streaming problems, the solution belongs to the class of Monte Carlo approximation algorithms because neither deterministic algorithms nor exact Monte Carlo algorithms can solve the problem in sublinear memory. Our algorithm has competitive parameters: its space usage is polynomial in $\log n$ and linear in $1/\varepsilon$, where n is the length of the input and ε is the additive error parameter. The $\mathcal{O}(\log n)$ update time is also close to the minimum possible. In addition, our algorithm uses only practical data structures and the $\mathcal{O}$-bounds hide no big constants.

Thus the natural question is "can one do better?" More specifically,

- Are there any space or time lower bounds for the approxExp problem?
- Is it possible to improve space usage?
- Is it possible to improve the update time to "pure" $\mathcal{O}(\log n)$, without dictionary operations?

Concerning the last question, we reproduce the remark from [15] on the choice of dictionaries, which shows what is the current "pure" time bound.

Remark 4 If ε is small (inverse polynomial), it makes sense to use dynamic perfect hash tables [2,8] as dictionaries. Both cited versions guarantee that with probability $1-m^{-c}$, where m is the dictionary size and c is an arbitrary constant, all dictionary operations will take $\mathcal{O}(1)$ time. Thus the total probability of a failed run of an algorithm can still be kept below $1/n$ with $\mathcal{O}(\log n)$ elementary operations between reads. However, this is not the case for big (such as constant or inverse polylog) values of ε. So in this case we suggest to use deterministic dictionaries by Anderson and Thorup [1] which give us $\mathcal{O}\Big(\sqrt{\frac{\log\log n}{\log\log\log n}}\cdot\log n\Big)$ elementary operations between reads.

References

1. Andersson, A., Thorup, M.: Dynamic ordered sets with exponential search trees. J. ACM **54**(3), 13 (2007)
2. Arbitman, Y., Naor, M., Segev, G.: De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In: 36th International Colloquium Automata, Languages and Programming, ICALP 2009, *Lecture Notes in Computer Science*, vol. 5555, pp. 107–118. Springer (2009)
3. Badkobeh, G., Crochemore, M., Toopsuwan, C.: Computing the maximal-exponent repeats of an overlap-free string in linear time. In: Proceedings 19th International Symposium String Processing and Information Retrieval, SPIRE 2012, *Lecture Notes in Computer Science*, vol. 7608, pp. 61–72. Springer (2012)
4. Brandenburg, F.J.: Uniformly growing k-th power-free homomorphisms. Theoret. Comput. Sci. **23**, 69–82 (1983)
5. Breslauer, D., Galil, Z.: Real-time streaming string-matching. In: Combinatorial pattern matching. LNCS, vol. 6661, pp. 162–172. Springer, Berlin (2011)
6. Crochemore, M., Iliopoulos, C.S., Kociumaka, T., Kundu, R., Pissis, S.P., Radoszewski, J., Rytter, W., Walen, T.: Near-optimal computation of runs over general alphabet via non-crossing LCE queries. In: 23rd International Symposium String Processing and Information Retrieval, SPIRE 2016, *Lecture Notes in Computer Science*, vol. 9954, pp. 22–34 (2016)
7. Crochemore, M., Kolpakov, R., Kucherov, G.: Optimal bounds for computing α-gapped repeats. Inf. Comput. **268**, 104434 (2019)
8. Dietzfelbinger, M., auf der Heide, F.M.: Dynamic hashing in real time. In: Informatik, pp. 95–119. Springer (1992)
9. Fredman, M.L., Willard, D.E.: Surpassing the information theoretic bound with fusion trees. J. Comput. Syst. Sci. **47**(3), 424–436 (1993)
10. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. **31**(2), 249–260 (1987)
11. Knuth, D.E., Morris, J., Pratt, V.: Fast pattern matching in strings. SIAM J. Comput. **6**, 323–350 (1977)
12. Kolpakov, R., Podolskiy, M., Posypkin, M., Khrapov, N.: Searching of gapped repeats and subrepetitions in a word. In: Proceedings 25th Annual Symposium Combinatorial Pattern Matching, CPM

2014, Moscow, Russia, June 16-18, 2014, *Lecture Notes in Computer Science*, vol. 8486, pp. 212–221. Springer (2014)
13. Kolpakov, R.M., Kucherov, G.: Finding maximal repetitions in a word in linear time. IEEE Comput. Soc. **99**, 596–604 (1999)
14. Merkurev, O., Shur, A.M.: Searching long repeats in streams. In: 30th Annual Symposium on Combinatorial Pattern Matching CPM 2019, *LIPIcs*, vol. 128, pp. 1–14 (2019)
15. Merkurev, O., Shur, A.M.: Searching runs in streams. In: Proceedings 26th International Symposium String Processing and Information Retrieval, SPIRE 2019, *Lecture Notes in Computer Science*, vol. 11811, pp. 203–220. Springer (2019)
16. Porat, B., Porat, E.: Exact and approximate pattern matching in the streaming model. In: 50th Annual IEEE Symposium on Foundations of Computer Science, 2009. FOCS'09. pp. 315–323. IEEE (2009)
17. Shur, A.M.: Growth of power-free languages over large alphabets, vol. 6072, pp. 350–361. Springer, Berlin (2010)
18. Shur, A.M.: Growth properties of power-free languages. Comput. Sci. Rev. **6**, 187–208 (2012)
19. Thue, A.: Über unendliche Zeichenreihen. Norske vid. Selsk. Skr. Mat. Nat. Kl. **7**, 1–22 (1906)
20. Thue, A.: Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen. Norske vid. Selsk. Skr. Mat. Nat. Kl. **1**, 1–67 (1912)
21. Yao, A.: Probabilistic computations: toward a unified measure of complexity. In: Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS), pp. 222–227 (1977)