Journal of Computer and System Sciences ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Fast algorithms for Abelian periods in words and greatest common divisor queries

T. Kociumaka, J. Radoszewski [*], W. Rytter

Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Banacha 2, 02-097 Warsaw, Poland

ARTICLE INFO

Article history:
Received 24 June 2014
Received in revised form 31 August 2016
Accepted 11 September 2016
Available online xxxx

Keywords:
Abelian period
Jumbled pattern matching
Greatest common divisor

ABSTRACT

We present efficient algorithms computing all Abelian periods of two types in a word. Regular Abelian periods are computed in $\mathcal{O}(n \log\log n)$ randomized time which improves over the best previously known algorithm by almost a factor of n. The other algorithm, for full Abelian periods, works in $\mathcal{O}(n)$ time. As a tool we develop an $\mathcal{O}(n)$-time construction of a data structure that allows $\mathcal{O}(1)$-time $\gcd(i, j)$ queries for all $1 \le i, j \le n$. This is a result of independent interest.

© 2016 Elsevier Inc. All rights reserved.

1. Introduction

The area of Abelian stringology was initiated by Erdős who posed a question about the smallest alphabet size for which there exists an infinite Abelian-square-free word; see [22]. The first example of such a word over a finite alphabet was given by Evdokimov [23]. An example over five-letter alphabet was given by Pleasants [39] and afterwards an optimal example over four-letter alphabet was shown by Keränen [33].

Several results related to Abelian stringology have been presented recently. The combinatorial results focus on Abelian complexity in words [3,13,19–21,24] and partial words [5–10]. A related model studied in the same context is k-Abelian equivalence [31]. Algorithms and data structures have been developed for Abelian pattern matching and indexing (also called jumbled pattern matching and indexing), most commonly for the binary alphabet [4,11,12,14,15,37,38] and for general alphabets as well [34,14]. Hardness results related to jumbled indexing have also been studied [1,29]. Abelian indexing has also been extended to trees [15] and graphs with bounded treewidth [28]. Another algorithmic focus is on Abelian periods in words, which were first defined and studied by Constantinescu and Ilie [16]. Abelian periods are a natural extension of the notion of a period from standard stringology [18] and are also related to Abelian squares; see [17].

We say that two words x, y are *commutatively equivalent* (denoted as $x \equiv y$) if one can be obtained from the other by permuting its symbols. Furthermore we say that x is an Abelian factor of y if there exists a word z such that $xz \equiv y$. We denote this relation as $x \subseteq y$.

Example 1. $00121001 \equiv 01010201$ and $00021 \subseteq 01010201$.

We consider words over an alphabet $\Sigma = \{0, \dots, \sigma - 1\}$. The *Parikh vector* $\mathcal{P}(w)$ of a word w shows frequency of each symbol of the alphabet in the word. More precisely, $\mathcal{P}(w)[c]$ equals to the number of occurrences of the symbol $c \in \Sigma$ in w.

* Corresponding author.
E-mail addresses: kociumaka@mimuw.edu.pl (T. Kociumaka), jrad@mimuw.edu.pl (J. Radoszewski), rytter@mimuw.edu.pl (W. Rytter).

http://dx.doi.org/10.1016/j.jcss.2016.09.003
0022-0000/© 2016 Elsevier Inc. All rights reserved.

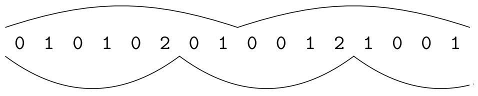

Fig. 1. The word 0101020100121001 together with its two Abelian periods: one is 6 (since $010102 \equiv 010012$ and $1001 \subseteq 010102$) and the other is 8 (since $01010201 \equiv 00121001$). The latter is also a full Abelian period. In total this word has two full Abelian periods (8 and 16), and ten Abelian periods (6, 8, 9, 10, 11, 12, 13, 14, 15, 16). Its shortest weak Abelian period is $(5, 3)$.

It turns out that x and y are commutatively equivalent if and only if the Parikh vectors $\mathcal{P}(x)$ and $\mathcal{P}(y)$ are equal. Moreover, x is an Abelian factor of y if and only if $\mathcal{P}(x) \le \mathcal{P}(y)$, i.e., if $\mathcal{P}(x)[c] \le \mathcal{P}(y)[c]$ for each coordinate c. Parikh vectors were introduced in the context of Abelian equivalence already in [16].

Example 2. For the words from Example 1 we have:

$$\mathcal{P}(00121001) = \mathcal{P}(01010201) = [4, 3, 1],$$
$$\mathcal{P}(00021) = [3, 1, 1] \le [4, 3, 1] = \mathcal{P}(01010201).$$

Let w be a word of length n. Let us denote by $w[i\,..\,j]$ the factor $w_i \dots w_j$ and by $\mathcal{P}_{i,j}$ the Parikh vector $\mathcal{P}(w[i\,..\,j])$. An integer q is called an *Abelian period* of w if for $k = \lfloor n/q \rfloor$:

$$\mathcal{P}_{1,q} = \mathcal{P}_{q+1,2q} = \cdots = \mathcal{P}_{(k-1)q+1,kq} \quad \text{and } \mathcal{P}_{kq+1,n} \le \mathcal{P}_{1,q}.$$

An Abelian period is called *full* if it is a divisor of n.

A pair (q, i) is called a *weak Abelian period* of w if q is an Abelian period of $w[i+1\,..\,n]$ and $\mathcal{P}_{1,i} \le \mathcal{P}_{i+1,i+q}$. Fig. 1 shows an example word together with its Abelian periods of various types.

Fici et al. [27] gave an $\mathcal{O}(n \log\log n)$-time algorithm finding all full Abelian periods and an $\mathcal{O}(n^2)$-time algorithm finding all Abelian periods. An $\mathcal{O}(n^2\sigma)$-time algorithm finding weak Abelian periods was developed in [25,26]. It was later improved to $\mathcal{O}(n^2)$-time in [17] and further improved to $\mathcal{O}(n^2/\sqrt{\log n})$-time for constant-sized alphabets in [36].

Our results We present an $\mathcal{O}(n)$-time deterministic algorithm finding all full Abelian periods. We also give an algorithm finding all Abelian periods, which we develop in two variants: an $\mathcal{O}(n \log\log n + n \log\sigma)$-time deterministic and an $\mathcal{O}(n \log\log n)$-time randomized. All our algorithms run on $\mathcal{O}(n)$ space in the standard word-RAM model with $\Omega(\log n)$ word size. The randomized algorithm is Monte Carlo and returns the correct answer with high probability, i.e. for each $c > 0$ the parameters can be set so that the probability of error is at most $\frac{1}{n^c}$. We assume that σ, the size of the alphabet, does not exceed n, the length of the word. However, it suffices that σ is polynomially bounded, i.e. $\sigma = n^{\mathcal{O}(1)}$; then the symbols of the word can be renumbered in $\mathcal{O}(n)$ time so that $\sigma \le n$.

As a tool we develop a data structure for gcd-queries. After $\mathcal{O}(n)$-time preprocessing, given any $i, j \in \{1, \dots, n\}$ the value $\gcd(i, j)$ can be computed in constant time. We are not aware of any solutions to this problem besides the folklore ones: preprocessing all answers ($\mathcal{O}(n^2)$ preprocessing, $\mathcal{O}(1)$ queries), using Euclid's algorithm (no preprocessing, $\mathcal{O}(\log n)$ queries) or prime factorization ($\mathcal{O}(n)$ preprocessing [30], queries in time proportional to the number of distinct prime factors, which is $\mathcal{O}(\frac{\log n}{\log\log n})$).

A preliminary version of this work appeared as [35].

Organization of the paper The auxiliary data structure for gcd-queries is presented in Section 2. In Section 3 we introduce the proportionality relation on Parikh vectors, which provides a convenient characterization of Abelian periods in a word. Afterwards in Sections 4 and 5 we present our main algorithms for full Abelian periods and Abelian periods, respectively. Each of the algorithms utilizes tools from number theory. The missing details of these algorithms related to the case of large alphabets are provided in the next two sections. In particular, in Section 6 we reduce efficient testing of the proportionality relation to a problem of equality of members of certain vector sequences, which potentially being of $\Theta(n\sigma)$ total size, admit an $\mathcal{O}(n)$-sized representation. Deterministic and randomized constructions of an efficient data structure for the vector equality problem (based on such representations) are left for Section 7. We end with a short section with some conclusions and open problems.

2. Greatest common divisor queries

The key idea behind our data structure for gcd is an observation that gcd-queries are easy when one of the arguments is prime or both arguments are small enough for the precomputed answers to be used. We exploit this fact by reducing each query to a constant number of such special-case queries.

In order to achieve this we define a *special decomposition* of an integer $k > 0$ as a triple (k_1, k_2, k_3) such that

$$k = k_1 \cdot k_2 \cdot k_3 \quad \text{and} \quad k_i \le \sqrt{k} \text{ or } k_i \in \textit{Primes} \text{ for } i = 1, 2, 3.$$

Example 3. $(2, 64, 64)$ is a special decomposition of 8192. $(1, 18, 479)$, $(2, 9, 479)$ and $(3, 6, 479)$ are, up to permutations, all special decompositions of 8622.

Let us introduce an operation $\otimes$ such that $(k_1, k_2, k_3) \otimes p$ results by multiplying the smallest of k_i's by p. For example, $(8, 2, 4) \otimes 7 = (8, 14, 4)$.

For an integer $k > 1$, let $MinDiv[k]$ denote the least prime divisor of k.

Fact 1. *Let $k > 1$ be an integer, $p = MinDiv[k]$ and $\ell = k/p$. If (ℓ_1, ℓ_2, ℓ_3) is a special decomposition of ℓ, then $(\ell_1, \ell_2, \ell_3) \otimes p$ is a special decomposition of k.*

Proof. Assume that $\ell_1 \le \ell_2 \le \ell_3$. If $\ell_1 = 1$, then $\ell_1 \cdot p = p$ is prime. Otherwise, ℓ_1 is a divisor of k and by the definition of p we have $p \le \ell_1$. Therefore:

$$(\ell_1 p)^2 = \ell_1^2 p^2 \le \ell_1^3 p \le \ell_1 \ell_2 \ell_3 p = k.$$

Consequently, $\ell_1 p \le \sqrt{k}$ and in both cases $(\ell_1 p, \ell_2, \ell_3)$ is a special decomposition of k. □

Fact 1 allows computing special decompositions provided that the values $MinDiv[k]$ can be computed efficiently. This is, however, a by-product of a linear-time prime number sieve of Gries and Misra [30].

Lemma 2. *([30], Section 5) The values $MinDiv[k]$ for all $k \in \{2, \dots, n\}$ can be computed in $\mathcal{O}(n)$ time.*

We proceed with the description of the data structure. In the preprocessing phase we compute in $\mathcal{O}(n)$ time two tables:

(a) a $Gcd\text{-}small[i, j]$ table such that $Gcd\text{-}small[i, j] = \gcd(i, j)$ for all $i, j \in \{1, \dots, \lfloor\sqrt{n}\rfloor\}$;
(b) a $Decomp[k]$ table such that $Decomp[k]$ is a special decomposition of k for each $k \le n$.

This phase is presented below in the algorithm Preprocessing(n). The *Gcd-small* table is filled using elementary steps in Euclid's subtraction algorithm and the *Decomp* table is computed according to Fact 1.

```
Algorithm Preprocessing($n$)
    for $i := 1$ to $\lfloor\sqrt{n}\rfloor$ do $Gcd\text{-}small[i, i] := i$;
    for $i := 2$ to $\lfloor\sqrt{n}\rfloor$ do
        for $j := 1$ to $i - 1$ do
            $Gcd\text{-}small[i, j] := Gcd\text{-}small[i - j, j]$;
            $Gcd\text{-}small[j, i] := Gcd\text{-}small[i - j, j]$;
    $Decomp[1] := (1, 1, 1)$;
    for $k := 2$ to $n$ do
        $p := MinDiv[k]$;
        $Decomp[k] := Decomp[k/p] \otimes p$;
    return $(Gcd\text{-}small, Decomp)$;
```

Fact 3. *If (x_1, x_2, x_3) is a special decomposition of $x \le n$, then for each $y \le n$ we can compute $\gcd(x_i, y)$ in constant time using the tables Gcd-small and Decomp.*

Proof. We compute $\gcd(x_i, y)$ as follows: if $x_i \le \sqrt{n}$, then

$$\gcd(x_i, y) = \gcd(x_i, y \bmod x_i) = Gcd\text{-}small[x_i, y \bmod x_i];$$

otherwise x_i is guaranteed to be prime, so the gcd can be greater than 1 only if $x_i \mid y$ and then $\gcd(x_i, y) = x_i$. □

The algorithm Query(x, y) computes $\gcd(x, y)$ for $x, y \le n$ using the special decomposition (x_1, x_2, x_3) of x as follows.

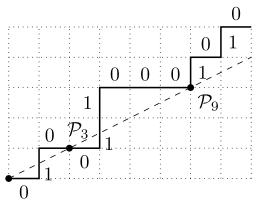

Fig. 2. A graph of the word $w = 0100110001010$, where each symbol 0 corresponds to a horizontal unit line segment and each symbol 1 corresponds to vertical unit line segment. Here $\mathcal{P}_3 = (2, 1)$ (the word 010) and $\mathcal{P}_9 = (6, 3)$ (the word 010011000). Hence, $3 \sim 9$. In other words, the points $\mathcal{P}_3$ and $\mathcal{P}_9$ lie on the same line originating from $(0, 0)$.

```
Algorithm Query(x, y)
    $(x_1, x_2, x_3) := Decomp[x]$;
    for $i := 1$ to 3 do
        $d_i := \gcd(x_i, y)$;                                {Fact 3}
        $y := y/d_i$;
    return $d_1 \cdot d_2 \cdot d_3$;
```

The correctness follows from the fact that

$$\gcd(x_1x_2x_3, y) = \gcd(x_1, y) \cdot \gcd(x_2x_3, y/\gcd(x_1, y))$$

and a similar equality for the latter factor. We obtain the following result.

Theorem 4. *After $\mathcal{O}(n)$-time preprocessing, given any $x, y \in \{1, \dots, n\}$ the value $\gcd(x, y)$ can be computed in constant time.*

Example 4. We show how the query works for $x = 7416$, $y = 8748$. Consider the following special decomposition of x: $(6, 12, 103)$. For $i = 1$ we have $d_1 = 6$ and y becomes 1458. For $i = 2$ we have $d_2 = 6$, therefore y becomes 243. In both cases, to compute d_i we used the table *Gcd-small*. For $i = 3$, x_i is a large prime number. We obtain that $d_3 = 1$ which yields $\gcd(x, y) = 6 \cdot 6 \cdot 1 = 36$.

3. Combinatorial characterization of Abelian periods

Let us fix a word w of length n. Let $\mathcal{P}_i = \mathcal{P}_{1,i}$. Two positions $i, j \in \{1, \dots, n\}$ are called *proportional*, which we denote $i \sim j$, if $\mathcal{P}_i[c] = D \cdot \mathcal{P}_j[c]$ for each $c \in \Sigma$, where D is a rational number independent of c. Note that if $i \sim j$ then the corresponding constant D equals i/j. Also note that $\sim$ is an equivalence relation; see also Fig. 2. In this section we exploit the connections between the proportionality relation and Abelian periods, which we conclude in Fact 5.

Definition 1. An integer k is called a *candidate* (a potential Abelian period) if and only if

$$k \sim 2k \sim 3k \sim \dots \sim \left\lfloor \frac{n}{k} \right\rfloor k.$$

By $Cand(n)$ we denote the set of all candidates.

We define the following *tail* table (assume $\min \emptyset = \infty$):

$$tail[i] = \min\{j : \mathcal{P}_{i,n} \le \mathcal{P}_{i-j,i-1}\}.$$

A similar table in the context of weak Abelian periods was introduced in [17].

Example 5. For the Fibonacci word

$$Fib_7 = \texttt{010010100100101001010}$$

of length 21, the first ten elements of the tail table are ∞, the remaining eleven elements are:

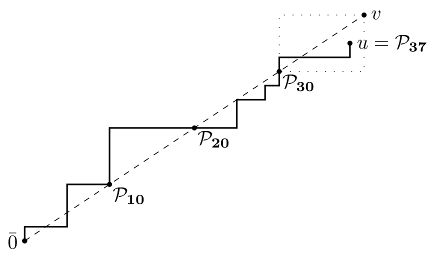

Fig. 3. Illustration of Fact 5b. The word $u = 1000111000111100000000011001011000001$ of length 37 has an Abelian period 10. We have $10 \sim 20 \sim 30$ and $\mathcal{P}_{31,37}$ is dominated by the period $\mathcal{P}_{10}$, i.e. the graph of the word ends within the rectangle marked in the figure (the point v dominates the point u).

i	11	12	13	14	15	16	17	18	19	20	21
$Fib_7[i]$	0	0	1	0	1	0	0	1	0	1	0
$tail[i]$	∞	10	11	8	8	7	5	5	3	2	2

The notion of a candidate and the tail table let us state a condition for an integer to be an Abelian period or a full Abelian period of w.

Fact 5. *Let w be a word of length n. A positive integer $q \le n$ is:*

(a) *a full Abelian period of w if and only if $q \in Cand(n)$ and $q \mid n$,*
(b) *an Abelian period of w if and only if*

$$q \in Cand(n) \quad \text{and} \quad tail[kq+1] \le q \quad \text{for} \quad k = \left\lfloor \frac{n}{q} \right\rfloor .$$

Proof. Let k be defined as in (b). Note that $q \in Cand(n)$ is equivalent to:

$$\mathcal{P}_{1,q} = \mathcal{P}_{q+1,2q} = \cdots = \mathcal{P}_{(k-1)q+1,kq},$$

which by definition means that q is a full Abelian period of $w[1..kq]$. This yields the conclusion part (a). To obtain part (b), it suffices to note that the condition $tail[kq+1] \le q$ is actually equivalent to

$$\mathcal{P}_{kq+1,n} \le \mathcal{P}_{(k-1)q+1,kq}.$$

See also Fig. 3 for an example. □

4. Algorithm for full Abelian periods

In this section we obtain a linear-time algorithm computing all full Abelian periods of a word. For integers $n, k > 0$ let $Mult(k,n)$ be the set of multiples of k not exceeding n. Recall that, by Fact 5a, q is a full Abelian period of w if and only if $q \mid n$ and $q \in Cand(n)$. This can be stated equivalently as:

$$q \mid n \quad \text{and} \quad Mult(q,n) \subseteq [n]_\sim,$$

where $[n]_\sim$ is the equivalence class of n under $\sim$. For a set $A \subseteq \{1,\dots,n\}$ we introduce the following auxiliary operation

$$FILTER1(A,n) = \{d \in \mathbb{Z}_{>0} : d \mid n \wedge Mult(d,n) \subseteq A\}$$

so that the following fact holds.

Observation 6. *$FILTER1([n]_\sim, n)$ is the set of all full Abelian periods.*

Thus the problem reduces to efficient computation of $[n]_\sim$ and implementation of the $FILTER1$ operation.

Example 6. $FILTER1(\{2,3,4,5,6,8,9,12\},12) = \{3,4,6,12\}$.

If $\sigma = \mathcal{O}(1)$, computing $[n]_\sim$ in linear time is easy, since one can store all vectors $\mathcal{P}_i$ and then test if $i \sim n$ in $\mathcal{O}(1)$ time. For larger alphabets, we defer the solution to Section 6, where we prove the following result.

Lemma 7. *For a word w of length n, $[n]_\sim$ can be computed in $\mathcal{O}(n)$ time.*

Lemma 8. *Let n be a positive integer and $A \subseteq \{1,\dots,n\}$. There exists an $\mathcal{O}(n)$-time algorithm that computes the set $FILTER1(A,n)$.*

Proof. Denote by $Div(n)$ the set of divisors of n. Let $A' = \{1,\dots,n\} \setminus A$. Observe that for $d \in Div(n)$

$$d \notin FILTER1(A,n) \quad \Longleftrightarrow \quad \exists_{j \in A'}\; d \mid j.$$

Moreover, for $d \in Div(n)$ and $j \in \{1,\dots,n\}$ we have

$$d \mid j \quad \Longleftrightarrow \quad d \mid d', \quad \text{where} \quad d' = \gcd(j,n).$$

These observations lead to the following algorithm.

```
Algorithm FILTER1$(A,n)$
    $D := X := Div(n)$;
    $A' := \{1,\dots,n\} \setminus A$;
    foreach $j \in A'$ do
        $D := D \setminus \{\gcd(j,n)\}$;
    foreach $d, d' \in Div(n)$ do
        if $d \mid d'$ and $d' \notin D$ then
            $X := X \setminus \{d\}$;
    return $X$;
```

We use $\mathcal{O}(1)$-time gcd-queries from Theorem 4. The number of pairs (d,d') is $o(n)$, since $|Div(n)| = o(n^{\varepsilon})$ for any $\varepsilon > 0$; see [2]. Consequently, the algorithm runs in $\mathcal{O}(n)$ time. □

Theorem 9. *Let w be a word of length n over the alphabet $\{0,\dots,\sigma-1\}$, where $\sigma \le n$. All full Abelian periods of w can be computed in $\mathcal{O}(n)$ time.*

Proof. By Observation 6, the computation of all full Abelian periods reduces to a single *FILTER1* operation, as shown in the following algorithm.

```
Algorithm Full Abelian periods
    Compute the data structure for answering gcd queries;     {Theorem 4}
    $A := [n]_{\sim}$;                                          {Lemma 7}
    $\mathcal{K} := FILTER1(A,n)$;                              {Lemma 8}
    return $\mathcal{K}$;
```

The algorithms from Lemmas 7 and 8 take $\mathcal{O}(n)$ time. Hence, the whole algorithm works in linear time. □

5. Algorithm for Abelian periods

In this section we develop an efficient algorithm computing all Abelian periods of a word. Recall that by Fact 5b, q is an Abelian period of w if and only if $q \in Cand(n)$ and $tail[kq+1] \le q$, where $k = \lfloor n/q \rfloor$. The following lemma allows linear-time computation of the tail table. It is implicitly shown in [17]; in the Appendix we provide a full proof for completeness.

Lemma 10. *Let w be a word of length n. The values $tail[i]$ for $1 \le i \le n$ can be computed in $\mathcal{O}(n)$ time.*

Thus the crucial part is computing $Cand(n)$. We define the following operation for an abstract $\approx$ relation on $\{k_0,\dots,n\}$:

$$FILTER2(\approx) = \{k \in \{k_0,\dots,n\} \,:\, \forall_{i \in Mult(k,n)}\; i \approx k\},$$

so that the following fact holds:

Observation 11. $FILTER2(\sim) = Cand(n)$. *(Here $k_0 = 1$.)*

It turns out that the set $FILTER2(\approx)$ can be efficiently computed provided that $\approx$ can be tested in $\mathcal{O}(1)$ time. As we have already noted in Section 4, if $\sigma = \mathcal{O}(1)$, then $\sim$ can be tested by definition in $\mathcal{O}(1)$ time. For larger alphabets, we develop an appropriate data structure in Section 6, where we prove the following lemma.

Lemma 12. *Let w be a word of length n over an alphabet of size σ and let q_0 be the smallest index q such that $w[1..q]$ contains each letter present in w. There exists a data structure of $\mathcal{O}(n)$ size which for given $i, j \in \{q_0, \dots, n\}$ decides whether $i \sim j$ in constant time. It can be constructed by an $\mathcal{O}(n\log\sigma)$-time deterministic or an $\mathcal{O}(n)$-time randomized algorithm (Monte Carlo, correct with high probability).*

Unfortunately, its usage is restricted to $i, j \geq q_0$. Nevertheless, if q is an Abelian period, then $q \geq q_0$, so it suffices to compute

$$FILTER2(\sim|_{\{q_0,\dots,n\}}) = Cand(n) \cap \{q_0, \dots, n\}.$$

Thus the problem reduces to efficient implementation of $FILTER2$ operation. This is the main objective of the following lemma.

Lemma 13. *Let $\approx$ be an arbitrary equivalence relation on $\{k_0, k_0+1, \dots, n\}$ which can be tested in constant time. Then, there exists an $\mathcal{O}(n\log\log n)$-time algorithm that computes the set $FILTER2(\approx)$.*

Before we proceed with the proof, let us introduce an equivalent characterization of the set $FILTER2(\approx)$.

Fact 14. *For $k \in \{k_0, \dots, n\}$*

$$k \in FILTER2(\approx) \iff \forall_{p\in Primes\,:\,kp\leq n}\ (k \approx kp\ \wedge\ kp \in FILTER2(\approx)).$$

Proof. First assume that $k \in FILTER2(\approx)$. Then, by definition, for each prime p such that $kp \leq n$, $kp \approx k$. Moreover, for each $i \in Mult(kp, n)$ we have $i \approx k \approx kp$, i.e. $i \approx kp$, which concludes that $kp \in FILTER2(\approx)$. This yields the $(\Rightarrow)$ part of the equivalence.

For a proof of the $(\Leftarrow)$ part, consider any k satisfying the right hand side of the equivalence and any integer $\ell \geq 2$ such that $k\ell \leq n$. We need to show that $k \approx k\ell$. Let p be a prime divisor of ℓ. By the right hand side, we have $k \approx kp$, and since $kp \in FILTER2(\approx)$, we get

$$kp \approx kp(\ell/p) = k\ell.$$

This concludes the proof of the equivalence. □

Proof of Lemma 13. The following algorithm uses Fact 14 for k decreasing from n to k_0 to compute $FILTER2(\approx)$. The condition:

$$Y = \{k_0, \dots, k\} \cup (FILTER2(\approx) \cap \{k+1, \dots, n\})$$

is an invariant of the algorithm.

```
Algorithm FILTER2($\approx$)
    $Y := \{k_0, \dots, n\}$;
    for $k := n$ downto $k_0$ do
        foreach $p \in Primes$, $kp \leq n$ do
      (⋆) if $kp \not\approx k$ or $kp \notin Y$ then
                $Y := Y \setminus \{k\}$;
    return $Y$;
```

In the algorithm we assume to have an ordered list of primes up to n. It can be computed in $\mathcal{O}(n)$ time; see [30]. For a fixed $p \in Primes$ the instruction (⋆) is called for at most $\frac{n}{p}$ values of k. The total number of operations performed by the algorithm is thus $\mathcal{O}(n\log\log n)$ due to the following well-known fact from number theory (see [2]):

$$\sum_{p\in Primes,\ p\leq n} \frac{1}{p} = \mathcal{O}(\log\log n). \quad \square$$

Theorem 15. *Let w be a word of length n over the alphabet $\{0,\dots,\sigma-1\}$, where $\sigma \le n$. There exist an $\mathcal{O}(n\log\log n + n\log\sigma)$-time deterministic and an $\mathcal{O}(n\log\log n)$-time randomized algorithm that compute all Abelian periods of w. Both algorithms require $\mathcal{O}(n)$ space.*

Proof. The pseudocode below uses the characterization of Fact 5b and Observation 11 restricted by $k_0 = q_0$ to compute all Abelian periods of a word.

```
Algorithm Abelian periods
    Build the data structure to test $\sim|_{\{q_0,\dots,n\}}$;          {Lemma 12}
    Compute table $tail$;                                              {Lemma 10}
    $Y := FILTER2(\sim|_{\{q_0,\dots,n\}})$;                            {Lemma 13}
    $\mathcal{K} := \emptyset$;
    foreach $q \in Y$ do
        $k := \lfloor n/q \rfloor$;
        if $tail[kq+1] \le q$ then $\mathcal{K} := \mathcal{K} \cup \{q\}$
    return $\mathcal{K}$;
```

The deterministic version of the algorithm from Lemma 12 runs in $\mathcal{O}(n\log\sigma)$ time and the randomized version runs in $\mathcal{O}(n)$ time. The algorithm from Lemma 13 runs in $\mathcal{O}(n\log\log n)$ time, and the algorithm from Lemma 10 runs in linear time. This implies the required complexity of the Abelian periods' computation. □

6. Implementation of the proportionality relation for large alphabets

The main goal of this section is to present data structures for efficient testing of the proportionality relation $\sim$ for large alphabets. Let q_0 be the smallest index q such that $w[1..q]$ contains each letter $c \in \Sigma$ present in w. Denote by $s = LeastFreq(w)$ a least frequent symbol of w. For $i \in \{q_0,\dots,n\}$ let $\gamma_i = \mathcal{P}_i/\mathcal{P}_i[s]$. We use vectors γ_i in order to deal with vector equality instead of vector proportionality; see the following lemma.

Lemma 16. *If $i, j \in \{q_0,\dots,n\}$, then $i \sim j$ is equivalent to $\gamma_i = \gamma_j$.*

Proof. ($\Rightarrow$) If $i \sim j$, then the vectors $\mathcal{P}_i$ and $\mathcal{P}_j$ are proportional. Multiplying any of them by a constant only changes the proportionality ratio. Hence, $\mathcal{P}_i/\mathcal{P}_i[s]$ and $\mathcal{P}_j/\mathcal{P}_j[s]$ are proportional. The denominators of both fractions are positive since $i, j \ge q_0$. However, the s-th components of γ_i and γ_j are 1. Consequently, these vectors are equal.

($\Leftarrow$) $\mathcal{P}_i/\mathcal{P}_i[s] = \mathcal{P}_j/\mathcal{P}_j[s]$ means that $\mathcal{P}_i$ and $\mathcal{P}_j$ are proportional. Hence, $i \sim j$. □

Example 7. Consider the word $w = 021001020021$ over an alphabet of size 3. Here $LeastFreq(w) = 1$ and $q_0 = 3$. We have:

$$\gamma_3 = (1,1,1),\quad \gamma_4 = (2,1,1),\quad \gamma_5 = (3,1,1),\quad \gamma_6 = (\tfrac{3}{2},1,\tfrac{1}{2}),\quad \gamma_7 = (2,1,\tfrac{1}{2}),$$
$$\gamma_8 = (2,1,1),\quad \gamma_9 = (\tfrac{5}{2},1,1),\quad \gamma_{10} = (3,1,1),\quad \gamma_{11} = (3,1,\tfrac{3}{2}),\quad \gamma_{12} = (2,1,1).$$

We see that

$$\gamma_4 = \gamma_8 = \gamma_{12} \quad \text{and} \quad \gamma_5 = \gamma_{10}$$

and consequently

$$4 \sim 8 \sim 12 \quad \text{and} \quad 5 \sim 10.$$

We define a natural way to store a sequence of vectors of the same length with a small total Hamming distance between consecutive elements. Recall that the Hamming distance between two vectors of the same length is the number of positions where these two vectors differ. The sequence $\mathcal{P}_i$ is a clear example of such a vector sequence. As we prove in Lemma 17, γ_i also is.

Definition 2. Given a vector v, consider an *elementary operation* of the form "$v[j] := x$" that changes the j-th component of v to x. Let $\bar{u}_1,\dots,\bar{u}_k$ be a sequence of vectors of the same dimension, and let $\xi = (\sigma_1,\dots,\sigma_r)$ be a sequence of elementary operations.

We say that ξ is a *diff-representation* of $\bar{u}_1, \dots, \bar{u}_k$ if $(\bar{u}_i)_{i=1}^k$ is a subsequence of the sequence $(\bar{v}_j)_{j=0}^r$, where

$$\bar{v}_j = \sigma_j(\dots(\sigma_2(\sigma_1(\bar{0})))\dots).$$

We denote $|\xi| = r$.

Example 8. Let ξ be the sequence:

$$v[1] := 1,\ v[2] := 2,\ v[1] := 4,\ v[3] := 1,\ v[4] := 3,\ v[3] := 0,\ v[1] := 1,$$
$$v[2] := 0,\ v[4] := 0,\ v[1] := 3,\ v[2] := 2,\ v[1] := 2,\ v[4] := 1.$$

The sequence of vectors produced by the sequence ξ, starting from $\bar{0}$, is:

$$(0,0,0,0), (1,0,0,0), (1,2,0,0), (4,2,0,0), (4,2,1,0), (4,2,1,3), (4,2,0,3),$$
$$(1,2,0,3), (1,0,0,3), (1,0,0,0), (3,0,0,0), (3,2,0,0), (2,2,0,0), (2,2,0,1).$$

Hence ξ is a diff-representation of the above vector sequence as well as all its subsequences.

Lemma 17.
(a) *$\sum_{i=q_0}^{n-1} dist_H(\gamma_i, \gamma_{i+1}) \le 2n$, where $dist_H$ is the Hamming distance.*
(b) *An $\mathcal{O}(n)$-sized diff-representation of $(\gamma_i)_{i=q_0}^n$ can be computed in $\mathcal{O}(n)$ time.*

Proof. Recall that $s = LeastFreq(w)$ and q_0 is the position of the first occurrence of s in w. To prove (a), observe that $\mathcal{P}_i$ differs from $\mathcal{P}_{i-1}$ only at the coordinate corresponding to $w[i]$. If $w[i] \ne s$, the same holds for γ_i and γ_{i-1}. If $w[i] = s$, the vectors γ_i and γ_{i-1} may differ on all σ coordinates. However, s occurs in w at most $\frac{n}{\sigma}$ times. This shows part (a).

As a consequence of (a), the sequence $(\gamma_i)_{i=q_0}^n$ admits a diff-representation with at most $2n + \sigma$ elementary operations in total. It can be computed by an $\mathcal{O}(n)$-time algorithm that apart from γ_i maintains $\mathcal{P}_i$ in order to compute the new values of the changing coordinates of γ_i. $\square$

Below we use Lemma 17 to develop data structures for efficient testing of the proportionality relation $\sim$. Recall that we apply the simpler Lemma 7 for computation of full Abelian periods and Lemma 12 for computation of regular Abelian periods.

Lemma 7. For a word w of length n, $[n]_\sim$ can be computed in $\mathcal{O}(n)$ time.

Proof. Observe that if $k \sim n$ then $k \ge q_0$. Indeed, if $k \sim n$, then $\mathcal{P}_k$ is proportional to $\mathcal{P}_n$, so all symbols occurring in w also occur in $w[1\dots k]$. Due to Lemma 16 this means that $\gamma_k = \gamma_n$. Using a diff-representation of (γ_i) provided by Lemma 17 we reduce the task to the following problem with $\delta_i = \gamma_i - \gamma_n$.

Claim 18. *Given a diff-representation of the vector sequence $\delta_{q_0}, \dots, \delta_n$ we can decide for which i vector δ_i is equal to $\bar{0}$ in $\mathcal{O}(\sigma + r)$ time, where σ is the size of the vectors and r is the size of the representation.*

The solution simply maintains δ_i and the number of non-zero coordinates of δ_i. This completes the proof of the lemma. $\square$

As the main tool for general proportionality queries we use a data structure for the following problem.

Integer vector equality queries
Input: A diff-representation ξ of a sequence $(\bar{u}_i)_{i=1}^k$ of integer vectors of dimension m, such that $r = |\xi|$ and the components of vectors are of absolute value $(m+r)^{\mathcal{O}(1)}$.
Queries: "Is $\bar{u}_i = \bar{u}_j$?" for $i, j \in \{1, \dots, k\}$.

In Section 7 we prove the following result.

Theorem 19. *The integer vector equality queries can be answered in $\mathcal{O}(1)$ time by a data structure of $\mathcal{O}(n)$ size, which can be constructed:*
(a) *in $\mathcal{O}(m + r\log m)$ time deterministically or*
(b) *in $\mathcal{O}(m + r)$ time using a Monte Carlo algorithm (with one-sided error, correct with high probability).*

Lemma 12. Let w be a word of length n over an alphabet of size σ and let q_0 be the smallest index q such that $w[1..q]$ contains each letter present in w. There exists a data structure of $\mathcal{O}(n)$ size which for given $i, j \in \{q_0, \dots, n\}$ decides whether $i \sim j$ in constant time. It can be constructed by an $\mathcal{O}(n \log \sigma)$-time deterministic or an $\mathcal{O}(n)$-time randomized algorithm (Monte Carlo, correct with high probability).

Proof. By Lemma 16, to answer the proportionality-queries it suffices to efficiently compare the vectors γ_i, which, by Lemma 17, admit a diff-representation of size $\mathcal{O}(n)$.

The Integer Vector Equality Problem requires integer values so we split (γ_i) into two sequences (α_i) and (β_i) of numerators and denominators, respectively. We need to store the fractions in a reduced form so that comparing numerators and denominators can be used to compare fractions. Thus we set

$$\alpha_i[j] = \mathcal{P}_i[j]/d \quad \text{and} \quad \beta_i[j] = \mathcal{P}_i[s]/d,$$

where $d = \gcd(\mathcal{P}_i[j], \mathcal{P}_i[s])$ can be computed in $\mathcal{O}(1)$ time using a single gcd-query of Theorem 4 since the values of $\mathcal{P}_i$ are non-negative integers up to n.

Consequently, the elements of (α_i) and (β_i) are also positive integers not exceeding n. This allows using Theorem 19, so that the whole algorithm runs in the desired $\mathcal{O}(n \log \sigma)$ and $\mathcal{O}(n)$ time, respectively, using $\mathcal{O}(n)$ space. □

7. Solution of Integer Vector Equality Problem

In this section we provide the missing proof of Theorem 19. Recall that in the Integer Vector Equality Problem we are given a diff-representation of a vector sequence $(\bar{u}_i)_{i=1}^k$, i.e. a sequence ξ of elementary operations $\sigma_1, \sigma_2, \dots, \sigma_r$ on a vector of dimension m. Each σ_i is of the form: set the j-th component to some value x. We assume that x is an integer of magnitude $(m+r)^{\mathcal{O}(1)}$. Let $\bar{v}_0 = \bar{0}$ and for $1 \le i \le r$ let $\bar{v}_i$ be the vector obtained from $\bar{v}_{i-1}$ by performing σ_i. Our task is answering queries of the form “Is $\bar{u}_i = \bar{u}_j$?” but it reduces to answering equality queries of the form “Is $\bar{v}_i = \bar{v}_j$?”, since $(\bar{u}_i)_{i=1}^k$ is a subsequence of $(\bar{v}_i)_{i=0}^r$ by definition of the diff-representation.

Definition 3. A function $H : \{0, \dots, r\} \to \{0, \dots, \ell\}$ is called an *ℓ-naming* for ξ if $H(i) = H(j)$ holds if and only if $\bar{v}_i = \bar{v}_j$.

In order to answer equality queries, we construct an ℓ-naming with $\ell = (m+r)^{\mathcal{O}(1)}$. Integers of this magnitude can be stored in $\mathcal{O}(1)$ space so this suffices to answer the equality queries in constant time.

7.1. Deterministic solution

Let $A = \{1, \dots, m\}$ be the set of coordinates. For any $B \subseteq A$, let $select_B[i]$ be the index of the ith operation concerning a coordinate from B in ξ. Moreover, let $rank_B[i]$, where $i \in \{0, \dots, r\}$, be the number of operations concerning coordinates in B among $\sigma_1, \dots, \sigma_i$ and let $r_B = rank_B[r]$.

Definition 4. Let ξ be a sequence of operations, A be the set of coordinates and $B \subseteq A$. Let $h : \{0, \dots, r_B\} \to \mathbb{Z}$ be a function. Then we define:

$$\begin{aligned} Squeeze(\xi, B) &= \xi_B \ \text{ where } \xi_B[i] = \xi[select_B[i]], \\ Expand(\xi, B, h) &= \eta_B \ \text{ where } \eta_B[i] = h(rank_B[i]). \end{aligned}$$

In other words, the squeeze operation produces a subsequence ξ_B of ξ consisting of operations concerning B. The expand operation is in some sense an inverse of the squeeze operation; it propagates the values of h from the domain B to the full domain A.

Example 9. Let ξ be the sequence from Example 8. Here $A = \{1, 2, 3, 4\}$. Let $B = \{1, 2\}$ and assume $H_B[0..8] = [0, 1, 2, 6, 2, 1, 4, 5, 3]$. Then:

$$\begin{aligned} select_B[1..8] &= (1, 2, 3, 7, 8, 10, 11, 12), \\ rank_B[0..13] &= (0, 1, 2, 3, 3, 3, 3, 4, 5, 5, 6, 7, 8, 8), \\ Expand(\xi, B, H_B) &= (0, 1, 2, 6, 6, 6, 6, 2, 1, 1, 4, 5, 3, 3). \end{aligned}$$

See also Fig. 4.

For a pair of sequences η', η'', denote by $Align(\eta', \eta'')$ the sequence of pairs η such that $\eta[i] = (\eta'[i], \eta''[i])$ for each i.

Moreover, for a sequence η of $r+1$ pairs of integers, denote by $Renumber(\eta)$ a sequence H of $r+1$ integers in the range $\{0, \dots, r\}$ such that $\eta[i] < \eta[j]$ if and only if $H[i] < H[j]$ for any $i, j \in \{0, \dots, r\}$.

The recursive construction of a naming function for ξ is based on the following fact.

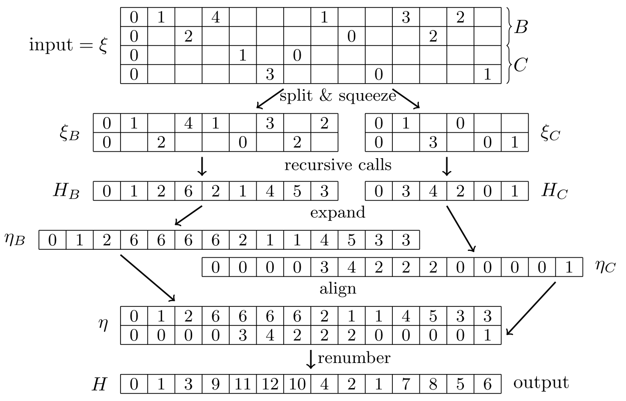

Fig. 4. A schematic diagram of performance of algorithm *ComputeH* for the sequence of elementary operations from Example 8. The columns correspond to elementary operations and the rows correspond to coordinates of the vectors.

Fact 20. *Let ξ be a sequence of elementary operations, $A = B \cup C$ ($B \cap C = \emptyset$) be the set of coordinates, H_B be an r_B-naming function for ξ_B and H_C an r_C-naming function for ξ_C. Additionally, let*

$$\eta_B = Expand(\xi, B, H_B), \quad \eta_C = Expand(\xi, C, H_C),$$
$$H = Renumber(Align(\eta_B, \eta_C)).$$

Then H is an r-naming function for ξ.

The algorithm makes an additional assumption about the sequence ξ.

Definition 5. We say that a sequence of operations ξ is *normalized* if for each operation $v[j] := x$ we have $x \in \{0, \dots, r_{\{j\}}\}$, where (as defined above) $r_{\{j\}}$ is the number of operations in ξ concerning the jth coordinate.

If for each operation $v[j] := x$ the value x is of magnitude $(m+r)^{\mathcal{O}(1)}$, then normalizing the sequence ξ, i.e., constructing a normalized sequence with the same answers to all equality queries, takes $\mathcal{O}(m+r)$ time. This is done using a radix sort of triples (j, x, i) and by mapping the values x corresponding to the same coordinate j to consecutive integers.

Lemma 21. *Let ξ be a normalized sequence of r operations on a vector of dimension m. An r-naming for ξ can be deterministically constructed in $\mathcal{O}(r \log m)$ time.*

Proof. If the dimension of vectors is 1 (that is, $|A| = 1$), the single components of the vectors $\bar{v}_i$ already constitute an r-naming. This is due to the fact that ξ is normalized.

For larger $|A|$, the algorithm uses Fact 20, see the pseudocode below and Fig. 4.

```
Algorithm ComputeH($\xi$)
    if $\xi$ is empty then return $\bar{0}$;
    if $|A| = 1$ then return $H$ computed naively;
    else
        Split $A$ into two halves $B$, $C$;
        $\xi_B := Squeeze(\xi, B)$; $\xi_C := Squeeze(\xi, C)$;
        $H_B := ComputeH(\xi_B)$; $H_C := ComputeH(\xi_C)$;
        $\eta_B := Expand(\xi, B, H_B)$; $\eta_C := Expand(\xi, C, H_C)$;
        return $Renumber(Align(\eta_B, \eta_C))$;
```

Let us analyze the complexity of a single recursive step of the algorithm. Tables *rank* and *select* are computed in $\mathcal{O}(r)$ time so both squeezing and expanding are performed in $\mathcal{O}(r)$ time. Renumbering, implemented using radix sort and bucket sort, also runs in $\mathcal{O}(r)$ time, since the values of H_B and H_C are positive integers bounded by r. Hence, the recursive step takes $\mathcal{O}(r)$ time.

We obtain the following recursive formula for $T(r,m)$, an upper bound on the execution time of the algorithm for a sequence of r operations on a vector of length m:

$$T(r,1) = \mathcal{O}(r), \quad T(0,m) = \mathcal{O}(1)$$
$$T(r,m) = T(r_1, \lfloor m/2 \rfloor) + T(r_2, \lceil m/2 \rceil) + \mathcal{O}(r) \quad \text{where } r_1 + r_2 = r.$$

A solution to this recurrence yields $T(r,m) = \mathcal{O}(r \log m)$. □

Lemma 21 yields part (a) of Theorem 19.

7.2. Randomized solution

Our randomized construction is based on fingerprints; see [32]. Let us fix a prime number p. For a vector $\bar{v} = (v_1, v_2, \ldots, v_m)$ we introduce a polynomial over the field $\mathbb{Z}_p$:

$$Q_{\bar{v}}(x) = v_1 + v_2 x + v_3 x^2 + \ldots + v_m x^{m-1} \in \mathbb{Z}_p[x].$$

Let us choose $x_0 \in \mathbb{Z}_p$ uniformly at random. Clearly, if $\bar{v} = \bar{v}'$ then

$$Q_{\bar{v}}(x_0) = Q_{\bar{v}'}(x_0).$$

The following lemma states that the converse is true with high probability.

Lemma 22. *Let $\bar{v} \neq \bar{v}'$ be vectors in $\{0,\ldots,n\}^m$. Let $p > n$ be a prime number and let $x_0 \in \mathbb{Z}_p$ be chosen uniformly at random. Then*

$$\mathbb{P}\left(Q_{\bar{v}}(x_0) = Q_{\bar{v}'}(x_0)\right) \le \tfrac{m}{p}.$$

Proof. Note that, since $p > n$,

$$R(x) = Q_{\bar{v}}(x) - Q_{\bar{v}'}(x) \in \mathbb{Z}_p[x]$$

is a non-zero polynomial of degree $\le m$, hence it has at most m roots. Consequently, x_0 is a root of R with probability bounded by m/p. □

Lemma 23. *Let $\bar{v}_1, \ldots, \bar{v}_r$ be vectors in $\{0,\ldots,n\}^m$. Let $p > \max(n, (m+r)^{c+3})$ be a prime number, where c is a positive constant, and let $x_0 \in \mathbb{Z}_p$ be chosen uniformly at random. Then $H(i) = Q_{\bar{v}_i}(x_0)$ is a naming function with probability at least $1 - \frac{1}{(m+r)^c}$.*

Proof. Assume that H is not a naming function. This means that there exist i, j such that $H(i) = H(j)$ despite $\bar{v}_i \neq \bar{v}_j$. Hence, by the union bound and Lemma 22 we obtain the conclusion of the lemma:

$$\mathbb{P}(H \text{ is not a naming}) \le \sum_{i,j\,:\,\bar{v}_i \neq \bar{v}_j} \mathbb{P}\left(H(i) = H(j)\right) \le \sum_{i,j\,:\,\bar{v}_i \neq \bar{v}_j} \tfrac{m}{p} \le \tfrac{mr^2}{p} \le \tfrac{1}{(m+r)^c}. \quad \square$$

Using Lemma 23 we obtain the following Lemma 24. It yields part (b) of Theorem 19 and thus completes the proof of the theorem.

Lemma 24. *Let ξ be a sequence of r operations on a vector of dimension m with values of magnitude $n = (m+r)^{\mathcal{O}(1)}$. There exists a randomized $\mathcal{O}(m+r)$-time algorithm that constructs a function H which is a k-naming for ξ with high probability for $k = (m+r)^{\mathcal{O}(1)}$.*

Proof. Assume all values in ξ are bounded by $(m+r)^{c'}$. Let $c \ge c'$. Let us choose a prime p such that

$$(m+r)^{3+c} < p < 2(m+r)^{3+c}.$$

Moreover let $x_0 \in \mathbb{Z}_p$ be chosen uniformly at random.

Then we set $H(i) = Q_{\bar{v}_i}(x_0)$. By Lemma 23, this is a naming function with probability at least $1 - \frac{1}{(m+r)^c}$.

If we know all powers $x_0^j \bmod p$ for $j \in \{1,\ldots,m\}$, then we can compute $H(i)$ from $H(i-1)$ (a single operation) in constant time. Thus $H(i)$ for all $1 \le i \le r$ can be computed in $\mathcal{O}(m+r)$ time. □

With a naming function stored in an array, answering equality queries is straightforward. In the randomized version, there is a small chance that H is not a naming function, which makes the queries Monte Carlo (with one-sided error). Nevertheless, the answers are correct with high probability.

8. Conclusions and open problems

We presented efficient algorithms for computation of all full Abelian periods and all Abelian periods in a word that work in $\mathcal{O}(n)$ time and $\mathcal{O}(n \log\log n + n \log\sigma)$ time (deterministic) or $\mathcal{O}(n \log\log n)$ time (randomized), respectively. An interesting open problem is the existence of an $\mathcal{O}(n \log\log n)$-time deterministic algorithm or a linear-time algorithm for computation of Abelian periods. Another open problem is to provide an $\mathcal{O}(n^{2-\epsilon})$-time algorithm computing the shortest weak Abelian periods in a word, for any $\epsilon > 0$, or a hardness proof for this problem (e.g. based on 3SUM problem as in [1,29]).

As a by-product we obtained an $\mathcal{O}(n)$ preprocessing time and $\mathcal{O}(1)$ query time data structure for $\gcd(i, j)$ computation, for any $1 \le i, j \le n$. We believe that applications of this result in other areas (not necessarily in Abelian stringology) are yet to be discovered.

Acknowledgments

Tomasz Kociumaka is supported by Polish budget funds for science in 2013–2017 as a research project under the 'Diamond Grant' program, grant no DI2012 01794 of the Ministry of Science and Higher Education, Republic of Poland. Jakub Radoszewski is supported by the Ministry of Science and Higher Education, Republic of Poland under the 'Iuventus Plus' program in 2015–2016, grant no 0392/IP3/2015/73. He also receives financial support of Foundation for Polish Science. Wojciech Rytter is supported by the Polish National Science Centre, grant no 2014/13/B/ST6/00770.

Appendix. Computation of tail table

The following lemma is implicitly shown in [17]. We provide a full proof for completeness.

Lemma 10. Let w be a word of length n. The values $tail[i]$ for $1 \le i \le n$ can be computed in $\mathcal{O}(n)$ time.

Proof. Define $tail'[i] = i - tail[i]$. The algorithm computes this table in $\mathcal{O}(n)$ time using the fact that its values are non-decreasing, i.e.

$$\forall_{1 \le i < n}\; tail'[i] \le tail'[i+1].$$

In the algorithm we store the difference $\Delta_i = \mathcal{P}(x_i) - \mathcal{P}(y_i)$ of Parikh vectors of $y_i = w[i..n]$ and $x_i = w[k..i-1]$ where $k = tail'[i]$. Note that $\Delta_i[c] \ge 0$ for any $c \in \Sigma$.

```
Algorithm Compute-tail(w)
  $\Delta := (0, 0, \ldots, 0)$;   {$\sigma$ zeros}
  $\Delta[w[n]] := 1$;           {boundary condition}
  $k := n$;
  for $i := n$ downto 1 do
    $\Delta[w[i]] := \Delta[w[i]] - 2$;
    while $k > 1$ and $\Delta[w[i]] < 0$ do
      $k := k - 1$;
      $\Delta[w[k]] := \Delta[w[k]] + 1$;
    if $\Delta[w[i]] < 0$ then $k := -\infty$;
    $tail'[i] := k$;
    $tail[i] := i - tail'[i]$;
```

Assume we have computed $tail'[i+1]$ and Δ_{i+1}. When we proceed to i, we move the symbol $w[i]$ from x to y and update Δ accordingly. At most one element of Δ might have dropped below 0. If there is no such element, we conclude that $tail'[i] = tail'[i+1]$. Otherwise we keep extending x to the left with new symbols and updating Δ until all its elements become non-negative.

The total number of iterations of the while-loop is $\mathcal{O}(n)$ since in each iteration we decrease the variable k, which is always positive, and we never increase this variable. Consequently, the time complexity of the algorithm is $\mathcal{O}(n)$. $\square$

References

[1] A. Amir, T.M. Chan, M. Lewenstein, N. Lewenstein, On hardness of jumbled indexing, in: J. Esparza, P. Fraigniaud, T. Husfeldt, E. Koutsoupias (Eds.), Automata, Languages, and Programming—41st International Colloquium, ICALP 2014, Proceedings, Part I, in: Lecture Notes in Computer Science, vol. 8572, Springer, 2014, pp. 114–125.

[2] T.M. Apostol, Introduction to Analytic Number Theory, Undergraduate Texts in Mathematics, Springer, 1976.

[3] S.V. Avgustinovich, A. Glen, B.V. Halldórsson, S. Kitaev, On shortest crucial words avoiding Abelian powers, Discrete Appl. Math. 158 (6) (2010) 605–607.

[4] G. Badkobeh, G. Fici, S. Kroon, Z. Lipták, Binary jumbled string matching for highly run-length compressible texts, Inf. Process. Lett. 113 (17) (2013) 604–608.

[5] F. Blanchet-Sadri, N. Fox, On the asymptotic Abelian complexity of morphic words, in: M. Béal, O. Carton (Eds.), Developments in Language Theory—17th International Conference, DLT 2013, Proceedings, in: Lecture Notes in Computer Science, vol. 7907, Springer, 2013, pp. 94–105.

[6] F. Blanchet-Sadri, J.I. Kim, R. Mercas, W. Severa, S. Simmons, D. Xu, Avoiding Abelian squares in partial words, J. Comb. Theory, Ser. A 119 (1) (2012) 257–270.

[7] F. Blanchet-Sadri, D. Seita, D. Wise, Computing Abelian complexity of binary uniform morphic words, Theor. Comput. Sci. 640 (2016) 41–51.

[8] F. Blanchet-Sadri, S. Simmons, Avoiding Abelian powers in partial words, in: G. Mauri, A. Leporati (Eds.), Developments in Language Theory, DLT 2011, Proceedings, in: Lecture Notes in Computer Science, vol. 6795, Springer, 2011, pp. 70–81.

[9] F. Blanchet-Sadri, S. Simmons, Abelian pattern avoidance in partial words, in: B. Rovan, V. Sassone, P. Widmayer (Eds.), Mathematical Foundations of Computer Science 2012—37th International Symposium, MFCS 2012, Proceedings, in: Lecture Notes in Computer Science, vol. 7464, Springer, 2012, pp. 210–221.

[10] F. Blanchet-Sadri, B.D. Winkle, S. Simmons, Abelian pattern avoidance in partial words, RAIRO Theor. Inform. Appl. 48 (3) (2014) 315–339.

[11] P. Burcsi, F. Cicalese, G. Fici, Z. Lipták, On table arrangements, scrabble freaks, and jumbled pattern matching, in: P. Boldi, L. Gargano (Eds.), Fun with Algorithms, 5th International Conference, FUN 2010, Proceedings, in: Lecture Notes in Computer Science, vol. 6099, Springer, 2010, pp. 89–101.

[12] P. Burcsi, F. Cicalese, G. Fici, Z. Lipták, Algorithms for jumbled pattern matching in strings, Int. J. Found. Comput. Sci. 23 (2) (2012) 357–374.

[13] J. Cassaigne, G. Richomme, K. Saari, L.Q. Zamboni, Avoiding Abelian powers in binary words with bounded Abelian complexity, Int. J. Found. Comput. Sci. 22 (4) (2011) 905–920.

[14] T.M. Chan, M. Lewenstein, Clustered integer 3SUM via additive combinatorics, in: R.A. Servedio, R. Rubinfeld (Eds.), Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, ACM, 2015, pp. 31–40.

[15] F. Cicalese, T. Gagie, E. Giaquinta, E.S. Laber, Z. Lipták, R. Rizzi, A.I. Tomescu, Indexes for jumbled pattern matching in strings, trees and graphs, in: O. Kurland, M. Lewenstein, E. Porat (Eds.), String Processing and Information Retrieval—20th International Symposium, SPIRE 2013, Proceedings, in: Lecture Notes in Computer Science, vol. 8214, Springer, 2013, pp. 56–63.

[16] S. Constantinescu, L. Ilie, Fine and Wilf's theorem for Abelian periods, Bull. Eur. Assoc. Theor. Comput. Sci. 89 (2006) 167–170.

[17] M. Crochemore, C.S. Iliopoulos, T. Kociumaka, M. Kubica, J. Pachocki, J. Radoszewski, W. Rytter, W. Tyczyński, T. Waleń, A note on efficient computation of all Abelian periods in a string, Inf. Process. Lett. 113 (3) (2013) 74–77.

[18] M. Crochemore, W. Rytter, Jewels of Stringology, World Scientific, 2003.

[19] J.D. Currie, A. Aberkane, A cyclic binary morphism avoiding Abelian fourth powers, Theor. Comput. Sci. 410 (1) (2009) 44–52.

[20] J.D. Currie, T.I. Visentin, Long binary patterns are Abelian 2-avoidable, Theor. Comput. Sci. 409 (3) (2008) 432–437.

[21] M. Domaratzki, N. Rampersad, Abelian primitive words, Int. J. Found. Comput. Sci. 23 (5) (2012) 1021–1034.

[22] P. Erdős, Some unsolved problems, Hung. Acad. Sci. Mat. Kut. Intéz. Közl. 6 (1961) 221–254.

[23] A.A. Evdokimov, Strongly asymmetric sequences generated by a finite number of symbols, Dokl. Akad. Nauk SSSR 179 (6) (1968) 1268–1271.

[24] G. Fici, A. Langiu, T. Lecroq, A. Lefebvre, F. Mignosi, J. Peltomäki, É. Prieur-Gaston, Abelian powers and repetitions in Sturmian words, Theor. Comput. Sci. 635 (2016) 16–34.

[25] G. Fici, T. Lecroq, A. Lefebvre, É. Prieur-Gaston, Computing Abelian periods in words, in: J. Holub, J. Žd'árek (Eds.), Proceedings of the Prague Stringology Conference 2011, Czech Technical University in Prague, Czech Republic, 2011, pp. 184–196.

[26] G. Fici, T. Lecroq, A. Lefebvre, É. Prieur-Gaston, Algorithms for computing Abelian periods of words, Discrete Appl. Math. 163 (2014) 287–297.

[27] G. Fici, T. Lecroq, A. Lefebvre, É. Prieur-Gaston, W. Smyth, Quasi-linear time computation of the Abelian periods of a word, in: J. Holub, J. Žd'árek (Eds.), Proceedings of the Prague Stringology Conference 2012, Czech Technical University in Prague, Czech Republic, 2012, pp. 103–110.

[28] T. Gagie, D. Hermelin, G.M. Landau, O. Weimann, Binary jumbled pattern matching on trees and tree-like structures, Algorithmica 73 (3) (2015) 571–588.

[29] I. Goldstein, T. Kopelowitz, M. Lewenstein, E. Porat, How hard is it to find (honest) witnesses?, in: P. Sankowski, C.D. Zaroliagis (Eds.), 24th Annual European Symposium on Algorithms, ESA 2016, in: LIPIcs, vol. 57, Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, 2016, pp. 45:1–45:16.

[30] D. Gries, J. Misra, A linear sieve algorithm for finding prime numbers, Commun. ACM 21 (12) (Dec. 1978) 999–1003.

[31] M. Huova, J. Karhumäki, A. Saarela, Problems in between words and Abelian words: *k*-Abelian avoidability, Theor. Comput. Sci. 454 (2012) 172–177.

[32] R.M. Karp, M.O. Rabin, Efficient randomized pattern-matching algorithms, IBM J. Res. Dev. 31 (2) (1987) 249–260.

[33] V. Keränen, Abelian squares are avoidable on 4 letters, in: W. Kuich (Ed.), ICALP, in: Lecture Notes in Computer Science, vol. 623, Springer, 1992, pp. 41–52.

[34] T. Kociumaka, J. Radoszewski, W. Rytter, Efficient indexes for jumbled pattern matching with constant-sized alphabet, in: H.L. Bodlaender, G.F. Italiano (Eds.), Algorithms – ESA 2013 – 21st Annual European Symposium, Proceedings, in: Lecture Notes in Computer Science, vol. 8125, Springer, 2013, pp. 625–636.

[35] T. Kociumaka, J. Radoszewski, W. Rytter, Fast algorithms for Abelian periods in words and greatest common divisor queries, in: N. Portier, T. Wilke (Eds.), 30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, Proceedings, in: LIPIcs, vol. 20, Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, 2013, pp. 245–256.

[36] T. Kociumaka, J. Radoszewski, B. Wiśniewski, Subquadratic-time algorithms for Abelian stringology problems, in: I.S. Kotsireas, S.M. Rump, C.K. Yap (Eds.), Mathematical Aspects of Computer and Information Sciences, 6th International Conference, MACIS 2015, Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 9582, Springer, 2015, pp. 320–334.

[37] T.M. Moosa, M.S. Rahman, Indexing permutations for binary strings, Inf. Process. Lett. 110 (18–19) (2010) 795–798.

[38] T.M. Moosa, M.S. Rahman, Sub-quadratic time and linear space data structures for permutation matching in binary strings, J. Discret. Algorithms 10 (2012) 5–9.

[39] P.A. Pleasants, Non-repetitive sequences, Proc. Camb. Philol. Soc. 68 (1970) 267–274.