Fast Algorithms for Abelian Periods in Words and Greatest Common Divisor Queries

Tomasz Kociumaka[1], Jakub Radoszewski[1], and Wojciech Rytter[1,2]

1 **Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland**
[kociumaka,jrad,rytter]@mimuw.edu.pl

2 **Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, ul. Chopina 12/18, 87-100 Toruń, Poland**

Abstract

We present efficient algorithms computing all Abelian periods of two types in a word. The algorithm computing regular Abelian periods works in $O(n \log\log n)$ time and improves over the best previously known algorithm by almost a factor of n. The second one is a linear time algorithm for full Abelian periods. As a tool we develop an $O(n)$ time construction of a data structure that allows constant time $\gcd(i, j)$ queries for all $i, j \leq n$, which is a result of independent interest.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Abelian period, greatest common divisor

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

The area of Abelian stringology was initiated by Erdös who posed a question about the smallest alphabet size for which there exists an infinite Abelian-square-free word, see [11]. An example of such a word over five-letter alphabet was given by Pleasants [19] and afterwards the optimal example over four-letter alphabet was shown by Keränen [16]. Quite recently there have been several results on Abelian complexity in words [2, 8, 9, 10] and partial words [3, 4] and on Abelian pattern matching [5, 17, 18]. Abelian periods were first defined and studied by Constantinescu and Ilie [6].

We say that two words are commutatively equivalent, if one can be obtained from the other by permuting its symbols. This relation can be conveniently described using *Parikh vectors*, which show frequency of each symbol of the alphabet in a word: x and y are commutatively equivalent if and only if the Parikh vectors $\mathcal{P}(x)$ and $\mathcal{P}(y)$ are equal. Parikh vectors were used in the context of Abelian periods already in [6]

Let w be a non-empty word of length n over alphabet Σ. Then $\mathcal{P}(w)$ is an array such that $\mathcal{P}(w)[c]$ equals to the number of occurrences of the symbol $c \in \Sigma$ in w. Let us denote by $w[i\,..\,j]$ the factor $w_i \ldots w_j$. and by $\mathcal{P}_{i,j}$ the Parikh vector $\mathcal{P}(w[i\,..\,j])$. For two vectors Q_1, Q_2 we write $Q_1 \leq Q_2$ if $Q_1[c] \leq Q_2[c]$ for each coordinate c.

An integer q is called an *Abelian period* of w if for $k = \left\lfloor \frac{n}{q} \right\rfloor$

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

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 $\mathcal{P}_{1,i} \leq \mathcal{P}_{i+1,i+q}$. For example, the word *ababacabaabcbaab* has full Abelian periods 8 and 16, Abelian periods $6, 8, 9, 10, 11, 12, 13, 14, 15, 16$ and its shortest weak period is $(5, 3)$.

Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Fici et al. [13] gave an $O(n \log\log n)$ time algorithm for full Abelian periods and an $O(n^2)$ time algorithm for Abelian periods. An $O(n^2 m)$ time algorithm for weak Abelian periods was developed in [12] and it was recently improved to $O(n^2)$ time [7].

Our results. We present an $O(n)$ time deterministic algorithm finding all full Abelian periods. We also give an algorithm finding all Abelian periods, which comes in two variants: an $O(n \log\log n + n \log m)$ time deterministic and an $O(n \log\log n)$ time randomized. All algorithms run in $O(n)$ space in the standard word-RAM model. 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 an error occurs with probability at most $\frac{1}{n^c}$.

As a tool we develop a data structure that after $O(n)$ preprocessing time computes $\gcd(i, j)$ for any $i, j \in \{1, \ldots, n\}$ in $O(1)$ time, which might be of its own interest. We are not aware of any solutions to this problem besides the folklore ones: preprocessing all answers ($O(n^2)$ preprocessing, $O(1)$ queries), using Euclid's algorithm (no preprocessing, $O(\log n)$ queries) or prime factorization ($O(n)$ preprocessing [14], queries in time proportional to the number of distinct prime factors, which is $O(\frac{\log n}{\log\log n})$).

The structure of the paper. Our algorithms use several non-trivial number-theoretic results, which are presented in the next two sections. The data structure for gcd-queries is developed in Section 2 and the tools specific to Abelian periods are described in Section 3. Then in Section 4 we introduce the proportionality relation on Parikh vectors, which provides a convenient characterization of Abelian periods in a string. Further properties of this relation are explored in Section 5. In particular we reduce efficient testing of this relation to a problem of equality of members of certain vector sequences, which are although potentially of $\Theta(nm)$ total size, admit an $O(n)$-sized representation. Deterministic and randomized constructions of an efficient data structure for the vector equality problem (based on such representations) are proposed in Section 6. Finally in Section 7 we conclude with our main algorithms for Abelian periods and full Abelian periods.

2 Greatest Common Divisor queries

The key idea behind our data structure 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 $k = k_1 \cdot k_2 \cdot k_3$ and

$$k_i \leq \sqrt{k} \quad \text{or} \quad k_i \in \mathit{Primes} \quad \text{for } i = 1, 2, 3.$$

▶ **Example 1.** $(2, 64, 64)$ 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 a positive integer ℓ, let $\mathit{MinDiv}[\ell]$ denote the least prime divisor of k.

▶ **Fact 2.** *Let $\ell \leq n$ be a positive integer, $p = \mathit{MinDiv}[\ell]$ and $k = \ell/p$. If (k_1, k_2, k_3) is a special decomposition of k then $(k_1, k_2, k_3) \otimes p$ is a special decomposition of ℓ.*

Proof. Assume that $k_1 \leq k_2 \leq k_3$. If $k_1 = 1$ then $k_1 \cdot p = p$ is prime. Otherwise, we have $p \leq k_1$ and thus

$$(k_1 p)^2 = k_1^2 p^2 \leq k_1^3 p \leq k_1 k_2 k_3 p = \ell.$$

Consequently $k_1 p \le \sqrt{\ell}$ and in both cases $(k_1 p, k_2, k_3)$ is a special decomposition of ℓ. ◀

Fact 2 allows to compute special decompositions provided that the values $MinDiv[k]$ can be computed efficiently. This is, however, a by-product of an $O(n)$ time prime number sieve.

▶ **Lemma 3** ([14], Section 5). *The values $MinDiv[k]$ for all $k = 2, \ldots, n$ can be computed in $O(n)$ time.*

▶ **Theorem 4.** *After $O(n)$ preprocessing, given any $k, \ell \in \{1, \ldots, n\}$ the value $\gcd(k, \ell)$ can be computed in constant time.*

Proof. In the preprocessing phase we compute in $O(n)$ time two tables:
(a) a $Gcd\text{-}small[i, j]$ table such that $Gcd\text{-}small[i, j] = \gcd(i, j)$ for all $i, j \in \{1, \ldots, \lfloor\sqrt{n}\rfloor\}$;
(b) a $Decomp[k]$ table such that $Decomp[k]$ is a special decomposition of k for each $k \le n$.

The *Gcd-small* table is filled using elementary steps in Euclid's subtraction algorithm. and the *Decomp* table is computed according to Fact 2.

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

Algorithm *Query*(k, ℓ)
  $(x_1, x_2, x_3) := Decomp[k]$;
  $(y_1, y_2, y_3) := Decomp[\ell]$;
  $g := 1$;
  foreach $i, j \in \{1, 2, 3\}$ **do**
    if $\max(x_i, y_j) \le \sqrt{n}$ **then**
      $d := Gcd\text{-}small[x_i, y_j]$;
    else if $x_i = y_j$ **then** $d := x_i$;
    else $d := 1$;
    $g := g \cdot d$;
    $x_i := x_i/d$; $y_j := y_j/d$;
  return g;

The algorithm Query(k, ℓ) computes $\gcd(k, \ell)$ using special decompositions (x_1, x_2, x_3) and (y_1, y_2, y_3) of k and ℓ respectively. The values x_i and y_j are altered during the execution of the algorithm, but remain prime or bounded by $\sqrt{n}$. In each step we have $d = \gcd(x_i, y_j)$; if $x_i, y_j \le \sqrt{n}$ then *Gcd-small* table is used and otherwise the gcd can be greater than 1 only if $x_i = y_j \in Primes$. We maintain an invariant that $k = x_1 x_2 x_3 \cdot g$ and $\ell = y_1 y_2 y_3 \cdot g$. At the end $\gcd(x_i, y_j) = 1$ holds for all $i, j \in \{1, 2, 3\}$ and consequently $g = \gcd(k, \ell)$. ◀

3 Number-theoretic tools for Abelian periods

Now we introduce two abstract *filter* operations and show how to perform them efficiently.

For integers $n, k > 0$ let $Mult(k, n)$ be the set of multiplicities of k not exceeding n, i.e.

$$Mult(k, n) = \{m \cdot k : m \in \mathbb{Z}_+, m \cdot k \le n\}.$$

Also denote $Div(n) = \{d \in \mathbb{Z}_+ : d \mid n\}$, the set of divisors of n.

▶ **Lemma 5.** *Let n be a positive integer and $A \subseteq \{1, \ldots, n\}$. There exists an $O(n)$ time algorithm that computes the set*

$$FILTER1(A, n) = \{d \in Div(n) : Mult(d, n) \subseteq A\}.$$

Proof. Let $A' = \{1, \ldots, n\} \setminus A$. Observe that for $d \in Div(n)$

$$d \notin FILTER1(A, n) \iff \exists_{j \in A'}\ d \mid j$$

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

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

These observations lead to the following algorithm.

Algorithm *FILTER1*(A, n)
  $D := Div(n)$; $X := Div(n)$;
  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 $O(1)$ 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 [1]. Consequently, the algorithm runs in $O(n)$ time. ◀

▶ **Lemma 6.** *Let $\approx$ be an arbitrary equivalence relation on $\{k_0, k_0 + 1, \ldots, n\}$ which can be tested in constant time. Then, there exists an $O(n \log\log n)$ time algorithm that computes the set:*

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

Proof. In the algorithm we use the following observation, which holds for $k \in \{k_0, \ldots, n\}$:

$$k \in FILTER2(\approx) \iff \forall_{p \in Primes\,:\, k \cdot p \le n}\ (k \approx k \cdot p \ \wedge\ k \cdot p \in FILTER2(\approx)). \tag{1}$$

The $(\Rightarrow)$ part of the equivalence is obvious. For the proof of the $(\Leftarrow)$ part consider any k satisfying the right hand side of (1) and any integer $\ell \ge 2$ such that $k \cdot \ell \le n$. We need to show that $k \approx k \cdot \ell$. Let p be a prime divisor of ℓ. By the right hand side, we have $k \approx k \cdot p$, and since $k \cdot p \in FILTER2(\approx)$, we get $k \cdot p \approx k \cdot p \cdot (\ell/p) = k \cdot \ell$.

The following algorithm uses (1) to compute $FILTER2(\approx)$ for k decreasing from n to k_0, It uses an invariant $Y = FILTER2(\approx) \cap \{k + 1, \ldots, n\}$ while checking condition of (1) for k.

Algorithm *FILTER2*$(\approx)$
  $Y := \{k_0, \ldots, n\}$;
  for $k := n$ **downto** k_0 **do**
    foreach $p \in Primes,\ p \cdot k \le n$ **do**
($\star$) **if** $k \cdot p \not\approx k$ **or** $k \cdot p \notin Y$ **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 $O(n)$ time, see [14]. For a fixed $p \in Primes$ the instruction $(\star)$ is called for at most $\frac{n}{p}$ values of k. The total number of operations performed by the algorithm is thus $O(n \log\log n)$ due to the following well-known fact from number theory, see [1]:

$$\sum_{p \in Primes,\ p \le n} \frac{1}{p} = O(\log\log n).$$

◀

4 Characterization of Abelian periods by proportionality relation

Let w be a word of length n. Let $\mathcal{P}_i = \mathcal{P}(w[1\,..\,i])$. Two positions $i, j \in \{1, \ldots, n\}$ are called *proportional*, which we denote $i \sim j$, if $\mathcal{P}_i[k] = c \cdot \mathcal{P}_j[k]$ for each k, where c is a real number independent of k, see also Figure 1.

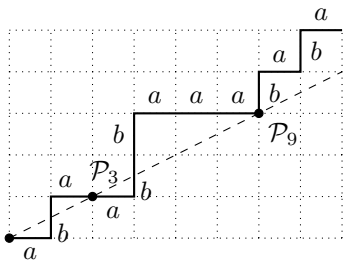

Figure 1 Here $\mathcal{P}_3 = (2, 1)$ (the word *aba*) and $\mathcal{P}_9 = (6, 3)$ (the word *abaabbaaa*), hence $3 \sim 9$. In other words, the points $\mathcal{P}_3$ and $\mathcal{P}_9$ lie on the same line originating from $(0, 0)$.

▶ **Definition 7.** An integer k is called a *candidate* (as a potential Abelian period) if $i \sim k$ for each $i \in \mathit{Mult}(k, n)$, or equivalently $k \sim 2k \sim 3k \sim \ldots$

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

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

▶ **Example 8.** For the Fibonacci word $\mathit{Fib} = \mathit{abaababaabaababaababa}$ of length 21, the first ten elements of the table $\mathit{tail}[i]$ are ∞, the remaining eleven elements are:

i	11	12	13	14	15	16	17	18	19	20	21
$\mathit{Fib}[i]$	a	a	b	a	b	a	a	b	a	b	a
$\mathit{tail}[i]$	∞	10	11	8	8	7	5	5	3	2	2

The following lemma is proved in the Appendix and also (implicitly) in [7].

▶ **Lemma 9.** *Let w be a word of length n. The values $\mathit{tail}[i]$ for $1 \leq i \leq n$ can be computed in $O(n)$ time.*

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

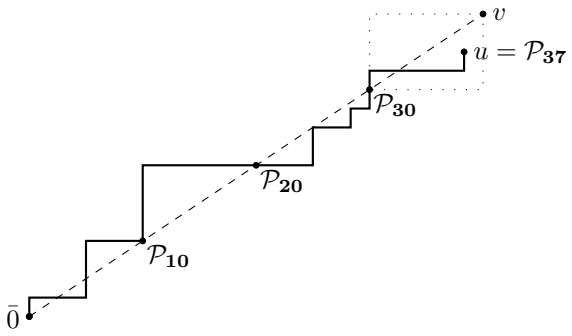

Figure 2 Illustration of Fact 10b. The word $u = \mathit{baaabbbaaabbbbaaaaaaaaabbaabababbaaaaab}$ of length 37 has an Abelian period 10. We have $10 \sim 20 \sim 30$ and $\mathcal{P}_{31,37}$ is dominated by the period ($\mathcal{P}_{10}$), i.e. the graph of the word ends within the rectangle marked in the figure, the point v dominates the point u.

▶ **Fact 10.** *Let w be a word of length n. A positive integer $q \leq n$ is:*
(a) *a full Abelian period of w if and only if $q \mid n$ and q is a candidate;*
(b) *an Abelian period of w if and only if q is a candidate and $\mathcal{P}_{kq+1,n} < \mathcal{P}_{(k-1)q+1,kq}$ for $k = \left\lfloor \frac{n}{q} \right\rfloor$, which is equivalent to $tail[kq+1] < q$ (see Fig. 2).*

5 Efficient implementation of the proportionality relation

Denote by $s = LeastFreq(w)$, the least frequent letter of w. Let q_0 be the position of the first occurrence of s in w. For $i \in \{q_0, \ldots, n\}$ let $\gamma_i = \mathcal{P}_i / \mathcal{P}_i[s]$. Vectors γ_i are introduced in order to deal with vector equality instead of vector proportionality.

▶ **Lemma 11.** *If $i, j \in \{q_0, \ldots, n\}$ 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 $i, j \geq q_0$. 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, so that $i \sim j$. ◀

▶ **Example 12.** Consider the word $w = acbaabacaacb$ for which $m = 3$, $LeastFreq(w) = b$ 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 conclude that $\gamma_4 = \gamma_8 = \gamma_{12}$ and $\gamma_5 = \gamma_{10}$ and consequently $4 \sim 8 \sim 12$ and $5 \sim 10$.

Let us formally define a natural way to store a sequence of vectors with a small total Hamming distance between consecutive elements, like $\mathcal{P}_i$ or, as we prove in Lemma 15, γ_i.

▶ **Definition 13.** 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 $\bar{u}_1, \ldots, \bar{u}_k$ be a sequence of vectors of the same dimension, and let $\xi = (\sigma_1, \ldots, \sigma_r)$ be a sequence of elementary operations. We say that ξ is a *diff-representation* of $\bar{u}_1, \ldots, \bar{u}_k$ if $(\bar{u}_i)_{i=1}^k$ is a subsequence of the sequence $(\bar{v}_j)_{j=0}^r$, where $\bar{v}_j = \sigma_j(\ldots(\sigma_2(\sigma_1(\bar{0})))\ldots)$.

▶ **Example 14.** Let ξ be the sequence: $v[1] := 1$, $v[2] := 2$, $v[1] := 4$, $v[3] := 1$, $v[4] := 3$, $v[3] := 0$, $v[1] := 1$, $v[2] := 0$, $v[4] := 0$, $v[1] := 3$, $v[2] := 2$, $v[1] := 2$, $v[4] := 1$. This sequence is schematically presented as the top rectangle in Fig. 3. 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 15.** **(a)** *$\sum dist_H(\gamma_{i+1}, \gamma_i) \leq 2n - m$, where $dist_H$ is the Hamming distance.*
(b) *An $O(n)$-sized diff-representation of $(\gamma_i)_{i=q_0}^n$ can be computed in $O(n)$ time.*

Proof. To prove (a) observe that $\mathcal{P}_i$ differs from $\mathcal{P}_{i-1}$ only at the coordinate corresponding to $w[i]$. If $w[i] \neq s$, the same holds for γ_i and γ_{i-1}. If $w[i] = s$, vectors γ_i and γ_{i-1} may differ on all coordinates, so m operations might be necessary. However s occurs at most $\frac{n}{m}$ times including the occurrence at the position q_0, which does not participate in the sum.

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

▶ **Lemma 16.** *Let w be a word of length n. The set $[n]_\sim$ can be computed in $O(n)$ time.*

Proof. Observe that if $k \sim n$ then $k \geq q_0$. Indeed, if $k \sim n$, then $\mathcal{P}_k$ is proportional to $\mathcal{P}_n$, so all letters occurring in w also occur in $w[1 \dots k]$. This lets us use the characterization of Lemma 11 and a diff-representation provided by Lemma 15 to reduce the task to the following problem with $\delta_i = \gamma_i - \gamma_n$.

▶ **Claim 17.** *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 $O(m + r)$ time, where m is the size of vectors and r is the size of representation.*

The solution simply maintains the number of non-zero coordinates of δ_i. ◀

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

▶ **Problem 1 (Integer vector equality).** *Assume we are given a diff-representation ξ of a vector sequence $(\bar{u}_i)_{i=1}^k$. Let m be the dimension of vectors and r be the size of representation. Assume vectors have integer components of magnitude $(m + r)^{O(1)}$. Preprocess ξ to answer queries of the form: "Is $\bar{u}_i = \bar{u}_j$?" for $i, j \in \{1, \dots, k\}$.*

In Section 6 we show that after $O(m + r \log m)$ time deterministic or $O(r + m)$ time randomized preprocessing these queries can be answered in constant time. In the latter case, with a small probability we can get false positive answers.

Note that this Lemma can be used for testing proportionality only for $i, j \geq q_0$. In other words, it allows testing $\sim |_{\{q_0,\dots,n\}}$, the restriction of $\sim$ to $\{q_0, \dots, n\}$.

▶ **Lemma 18.** *Let w be a word of length n over an alphabet of size m. There exists a data structure of $O(n)$ 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 $O(n \log m)$ time deterministic or an $O(n)$ time randomized algorithm (Monte Carlo, correct with high probability).*

Proof. By Lemma 11, to answer the proportionality-queries it suffices to efficiently compare the γ vectors, which, by Lemma 15, admit a diff-representation of size $O(n)$. Problem 1 requires integer values, so we split γ into two sequences α and β, of nominators and denominators respectively. We need to store the fractions in a reduced form so that comparing nominators 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 $O(1)$ 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 values of α and β are also positive integers not exceeding n. This allows to use a solution to Problem 1 given in Theorem 28, so that the whole algorithm runs in the desired $O(n)$ or $O(n \log m)$ time and $O(n)$ space. ◀

6 Vector equality in diff-representation

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 of the form: set the j-th component to some value x. We assume that x is an integer of magnitude $(m + r)^{O(1)}$. Let $\bar{v}_0 = \bar{0}$ and for $1 \leq i \leq r$ let $\bar{v}_i$ be a 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 diff-representation.

▶ **Definition 19.** A function $H : \{0, \ldots, r\} \to \{0, \ldots, r\}$ is called an *r-naming* for ξ if $H(i) = H(j)$ holds if and only if $\bar{v}_i = \bar{v}_j$.

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

6.1 Deterministic construction of a naming function

Let $\xi = (\sigma_1, \ldots, \sigma_r)$ be a sequence of operations on a vector of dimension m. Let $A = \{1, \ldots, m\}$ be the set of coordinates. For any $B \subseteq A$, let $rank_B[i]$, where $i \in \{0, \ldots, r\}$, be the number of operations concerning coordinates in B among $\sigma_1, \ldots, \sigma_i$. Moreover, let $select_B[i]$ be the index of the ith operation concerning B among $\sigma_1, \ldots, \sigma_r$.

▶ **Definition 20.** Let ξ be a sequence of operations, A be the set of coordinates and $B \subseteq A$. Let $h : B \to \mathbb{Z}$ be a function. Then define:

$$Squeeze(\xi, B) = \xi_B \quad \text{where } \xi_B[i] = \xi[select_B[i]]$$

$$Expand(\xi, B, h) = \eta_B \quad \text{where} \quad \eta_B[i] = h(rank_B[i]).$$

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 its domain B to the full domain A.

▶ **Example 21.** Let ξ be the sequence from Example 14, here $A = \{1, 2, 3, 4\}$. Let $B = \{1, 2\}$ and assume $H_B = [0, 1, 2, 6, 2, 1, 4, 5, 3]$. Then (see also Fig. 3):

$$Expand(\xi, B, H_B) = (0, 1, 2, 6, 6, 6, 6, 2, 1, 1, 4, 5, 3, 3).$$

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 $\{0, \ldots, r\}$ such that $\eta[i] < \eta[j]$ if and only if $H[i] < H[j]$ for any $i, j \in \{0, \ldots, r\}$.

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

▶ **Fact 22.** *Let ξ be a sequence of elementary operations, $A = B \cup C$ be the set of coordinates and H_B, H_C be r-naming functions for ξ_B, ξ_C respectively. Additionally, let*

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

Then H is an r-naming function for ξ.

The algorithm makes an additional assumption about the sequence ξ.

▶ **Definition 23.** We say that a sequence of operations ξ is *normalized* if for each coordinate j the following condition holds. Let $v[j] := x_1, \ldots, v[j] := x_s$ be all operations from ξ concerning the j-th coordinate. Then $x_i \in \{0, \ldots, s\}$.

If for each operation $v[j] := x$ the value x is of magnitude $(m+r)^{O(1)}$, then normalizing the sequence ξ, i.e., constructing a normalized sequence with the same answers to all equality queries, takes $O(m + r)$ 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 24.** *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 $O(r \log m)$ 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 22, see the pseudocode below and Figure 3.

Algorithm *ComputeH*(ξ)
  if ξ *is empty* **then return** $\bar{0}$;
  if $|A| = 1$ **then** compute H naively;
  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 $Renumber(Align(\eta_A, \eta_B))$;

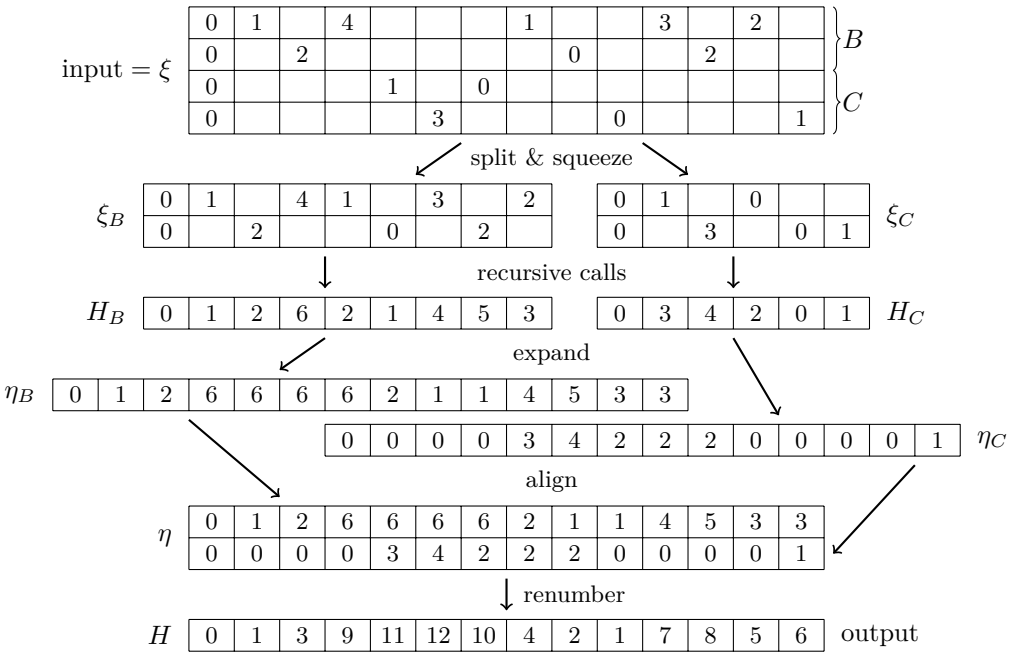

Figure 3 A schematic diagram of performance of algorithm *ComputeH*. The columns correspond to elementary operations and the rows correspond to coordinates of the vectors.

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

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

A solution to this recurrence yields $T(r,m) = O(r \log m)$. ◀

6.2 Randomized construction of a naming function

Our randomized construction is based on fingerprints, see [15]. 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 25.** *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*

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

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 $\leq m$, hence it has at most m roots. Consequently, x_0 is a root of R with probability bounded by m/p. ◀

▶ **Lemma 26.** *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 25 we obtain the conclusion of the lemma:

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

▶ **Lemma 27.** *Let ξ be a sequence of r operations on a vector of dimension m with values of magnitude $n = (m+r)^{O(1)}$. There exists a randomized $O(m+r)$ time algorithm that constructs a function H which is a k-naming for ξ with high probability for $k = (m+r)^{O(1)}$.*

Proof. Assume all values in ξ are bounded by $(m+r)^{c'}$. Let $c \geq c'$. 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 26, 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 $1 \leq i \leq r$ can be computed in $O(m+r)$ 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. Thus we obtain the following result.

▶ **Theorem 28.** *The vector-sequence equality problem can be solved in $O(n)$ space and:*
(a) *in $O(m + r \log m)$ time deterministically or*
(b) *in $O(m + r)$ time using a Monte Carlo algorithm (with one-sided error, correct w.h.p.).*

7 Two main algorithms

In this section we combine our tools to develop efficient algorithms computing all Abelian periods of two types.

▶ **Theorem 29.** *Let w be a word of length n over the alphabet $\{1, \ldots, m\}$. Full Abelian periods of w can be computed in $O(n)$ time.*

Proof. Full Abelian periods are computed using the characterization given by Fact 10a. Recall that $\mathit{FILTER1}([n]_\sim, n) = \{d \mid n : \mathit{Mult}(d, n) \subseteq [n]_\sim\} = \{d \mid n : \mathit{Mult}(d, n) \subseteq [d]_\sim\}$.

Algorithm *Full Abelian periods*
Compute the data structure for answering gcd queries; {Theorem 4}
$A := \{k : k \sim n\}$; {Lemma 16}
$\mathcal{K} := \mathit{FILTER1}(A, n)$; {Lemma 5}
return $\mathcal{K}$;

The algorithms from Lemmas 5 and 16 take $O(n)$ time. Hence, the whole algorithm works in linear time. ◀

▶ **Theorem 30.** *Let w be a word of length n over the alphabet $\{1, \ldots, m\}$. There exist an $O(n \log \log n + n \log m)$ time deterministic and an $O(n \log \log n)$ time randomized algorithms that compute all Abelian periods of w. Both algorithms require $O(n)$ space.*

Proof. Abelian periods are computed using the characterization given by Fact 10b. Recall that Lemma 18 allows testing $\sim|_{\{q_0,\ldots,n\}}$, the restriction of $\sim$ to $\{q_0, \ldots, n\}$, only. Nevertheless all Abelian periods of w are at least q_0 and thus it suffices to initialize Y to the set of candidates greater or equal to q_0, that is the set

$$\{k \in \{q_0, \ldots, n\} : \mathit{Mult}(k, n) \subseteq [k]_\sim\} = \mathit{FILTER2}(\sim|_{\{q_0,\ldots,n\}}).$$

Algorithm *Abelian periods*
Compute the data structure for answering gcd queries; {Theorem 4}
Prepare data structure to answer in $O(1)$ time proportionality-queries; {Lemma 18}
Compute table *tail*; {Lemma 9}
$Y := \mathit{FILTER2}(\sim|_{\{q_0,\ldots,n\}})$; {Lemma 6}
$\mathcal{K} := \emptyset$;
foreach $q \in Y$ **do**
  $j := q \cdot \left\lfloor \frac{n}{q} \right\rfloor + 1$;
  if $\mathit{tail}[j] < q$ **then** $\mathcal{K} := \mathcal{K} \cup \{q\}$;
return $\mathcal{K}$;

The deterministic version of the algorithm from Lemma 18 runs in $O(n \log m)$ time an the randomized version runs in $O(n)$ time. The algorithm from Lemma 6 runs in $O(n \log \log n)$ time and all the remaining algorithms (see Theorem 4, Lemma 9) run in linear time. This implies the required complexity of the Abelian periods' computation. ◀

References

1 Tom M. Apostol. *Introduction to Analytic Number Theory.* Undergraduate Texts in Mathematics. Springer, 1976.

2 Sergey V. Avgustinovich, Amy Glen, Bjarni V. Halldórsson, and Sergey Kitaev. On shortest crucial words avoiding Abelian powers. *Discrete Applied Mathematics*, 158(6):605–607, 2010.

3 Francine Blanchet-Sadri, Jane I. Kim, Robert Mercas, William Severa, and Sean Simmons. Abelian square-free partial words. In Adrian Horia Dediu, Henning Fernau, and Carlos Martín-Vide, editors, *LATA*, volume 6031 of *Lecture Notes in Computer Science*, pages 94–105. Springer, 2010.

4 Francine Blanchet-Sadri and Sean Simmons. Avoiding Abelian powers in partial words. In Mauri and Leporati [17], pages 70–81.

5 Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. Searching for jumbled patterns in strings. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2009*, pages 105–117, Czech Technical University in Prague, Czech Republic, 2009.

6 Sorin Constantinescu and Lucian Ilie. Fine and Wilf's theorem for Abelian periods. *Bulletin of the EATCS*, 89:167–170, 2006.

7 Maxime Crochemore, Costas Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Jakub Pachocki, Jakub Radoszewski, Wojciech Rytter, Wojciech Tyczyński, and Tomasz Waleń. A note on efficient computation of all Abelian periods in a string. *CoRR*, abs/1208.3313, 2012.

8 James D. Currie and Ali Aberkane. A cyclic binary morphism avoiding Abelian fourth powers. *Theor. Comput. Sci.*, 410(1):44–52, 2009.

9 James D. Currie and Terry I. Visentin. Long binary patterns are Abelian 2-avoidable. *Theor. Comput. Sci.*, 409(3):432–437, 2008.

10 Michael Domaratzki and Narad Rampersad. Abelian primitive words. In *Developments in Language Theory*, pages 204–215, 2011.

11 P. Erdös. Some unsolved problems. *Hungarian Academy of Sciences Mat. Kutató Intézet Közl.*, 6:221–254, 1961.

12 Gabriele Fici, Thierry Lecroq, Arnaud Lefebvre, and Élise Prieur-Gaston. Computing Abelian periods in words. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2011*, pages 184–196, Czech Technical University in Prague, Czech Republic, 2011.

13 Gabriele Fici, Thierry Lecroq, Arnaud Lefebvre, Elise Prieur-Gaston, and William Smyth. (quasi-)linear time computation of the abelian periods of a word. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2012*, Czech Technical University in Prague, Czech Republic, 2012.

14 David Gries and Jayadev Misra. A linear sieve algorithm for finding prime numbers. *Commun. ACM*, 21(12):999–1003, December 1978.

15 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

16 Veikko Keränen. Abelian squares are avoidable on 4 letters. In Werner Kuich, editor, *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 41–52. Springer, 1992.

17 Giancarlo Mauri and Alberto Leporati, editors. *Developments in Language Theory - 15th International Conference, DLT 2011, Milan, Italy, July 19-22, 2011. Proceedings*, volume 6795 of *Lecture Notes in Computer Science*. Springer, 2011.

18 Tanaeem M. Moosa and M. Sohel Rahman. Indexing permutations for binary strings. *Inf. Process. Lett.*, 110(18-19):795–798, 2010.

19 P. A. Pleasants. Non-repetitive sequences. *Proc. Cambridge Phil. Soc.*, 68:267–274, 1970.

Appendix

In the lemma below we show how to compute the *tail* table in $O(n)$ time. The same result can also be inferred from [7].

▶ **Lemma 9.** *Let w be a word of length n. The table $tail[i]$ for $1 \le i \le n$ can be computed in $O(n)$ time.*

Proof. Define $tail'[i] = i - tail[i]$. We show how to compute this table in $O(n)$ time using the fact that it is non-increasing:

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

In the algorithm we store the difference $\Delta_i = \mathcal{P}(y_i) - \mathcal{P}(x_i)$ of Parikh vectors of $y_i = w[i\,..\,n]$ and $x_i = w[k\,..\,i-1]$ where $k = tail'[i]$. Note that $\Delta_i[a] \ge 0$ for any $a = 1, 2, \ldots, m$.

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

```
Algorithm Compute-tail(w)
    Δ := (0, 0, ..., 0);
    Δ[w[n]] := 1;                          {boundary condition}
    k := n;
    for i := n downto 1 do
        Δ[w[i]] := Δ[w[i]] − 2;
        while k > 1 and Δ[w[i]] < 0 do
            k := k − 1;
            Δ[w[k]] := Δ[w[k]] + 1;
        if Δ[w[i]] < 0 then k := −∞; tail'[i] := k;
        tail[i] := i − tail'[i];
```

The total number of iterations of the while-loop is $O(n)$, since in each iteration we decrease the variable k and we never increase this variable. Consequently the time complexity of the algorithm is $O(n)$. ◀

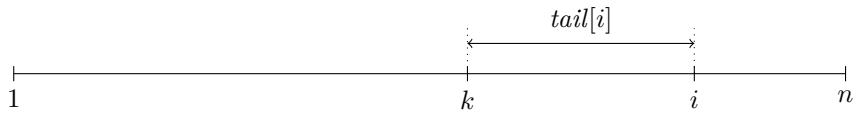

Figure 4 At the end of each iteration of the for-loop, the value $k = tail'[i]$ is as in the figure and $\mathcal{P}_{i,n} \le \mathcal{P}_{k,i-1}$.