Abelian periodicity of strings has been studied extensively over the last years. In 2006 Constantinescu and Ilie defined the abelian period of a string and several algorithms for the computation of all abelian periods of a string were given. In contrast to the classical period of a word, its abelian version is more flexible, factors of the word are considered the same under any internal permutation of their letters. We show two $O(|y|^2)$ algorithms for the computation of all abelian periods of a string $y$. The first one maps each letter to a suitable number such that each factor of the string can be identified by the unique sum of the numbers corresponding to its letters and hence abelian periods can be identified easily. The other one maps each letter to a prime number such that each factor of the string can be identified by the unique product of the numbers corresponding to its letters and so abelian periods can be identified easily. We also define weak abelian periods on strings and give an $O(|y|\log(|y|))$ algorithm for their computation, together with some other algorithms for more basic problems.

Keywords: strings; algorithms; abelian periods.
Introduction

The notion of periodicity in strings is well studied in many fields like combinatorics on words, pattern matching, data compression and automata theory (see [25, 26]), because it is of paramount importance in several applications, not to talk about its theoretical aspects.

A string \( u \) is a period of \( y \), if \( y \) is a prefix of \( u^k \) for some positive integer \( k \) (i.e. \( y \) is a prefix of \( uy \)). The period of \( y \), denoted by \( \text{Period}(y) \), is the length of the shortest period of \( y \). A lot of research has been concentrated on classical periods, e.g. algorithms for finding all periods of a string, algorithms for the computation of the period array of a string [23], etc.

Abelian periods are more flexible than classical ones and are defined in terms of Parikh vectors as in [15]. The Parikh vector of a string \( y \), denoted by \( \mathcal{P}(y) \), enumerates the cardinality of each letter of \( \Sigma \) in \( y \). That is \( \mathcal{P}[i-1] \) is the cardinality of the \( i \)th letter of \( \Sigma \) in \( y \), where \( 0 \leq i \leq |\Sigma| - 1 \). A string \( y \) is said to have an abelian period \((h, p)\) if \( y = u_0u_1...u_{k-1}u_k \) such that: \( \mathcal{P}(u_0) \subset \mathcal{P}(u_1) = ... = \mathcal{P}(u_{k-1}) \supset \mathcal{P}(u_k) \) and \(|\mathcal{P}(u_0)| = h, |\mathcal{P}(u_1)| = p\).

Abelian periodicity has been extensively studied over the last years [4, 5, 6, 7, 11, 16, 17, 27]. Early efficient algorithms for abelian pattern matching were given in [18, 19] and later some linear time algorithms have been designed in [9, 10, 14]. Recently Fici et al gave five algorithms for the computation of all abelian periods of a string [20]. They have proposed two off line algorithms, a brute force algorithm and one that uses a select array, that run in \( O(|y|^2|\Sigma|) \) and three online algorithms, where the first two run in \( O(|y|^3|\Sigma|) \) and the other one runs in \( O(|y|^3\log(|y|)|\Sigma|) \). Experimentally the off line algorithm that makes use of the select array is said to be the fastest in practice.

In this article, we show two \( O(|y|^2) \) algorithms for the computation of all abelian periods of a string \( y \). The first one maps each letter to a suitable number such that each factor of the string can be identified by the unique sum of the numbers corresponding to its letters. The other one maps each letter to a prime number such that each factor of the string can be identified by the unique product of the numbers corresponding to its letters. We are then able to perform the required checks of parikh vectors, necessary to identify abelian periods, with just one operation. Additionally we define weak abelian periods on strings and give an \( O(|y|\log(|y|)) \) algorithm for their computation. Some other algorithms for basic problems on identification of periods which form the basis of the previous ones are also analyzed.

The rest of the article is structured as follows. In Section 1 we present the basic definitions used throughout the article and we define the problems solved. In Section 2 we prove some properties of abelian periods, Parikh vectors and their relation to the \( S\)-signature and \( P\)-signature of factors of the string and we also quote some properties of prime numbers which are used later for the design and analysis of the provided algorithms. In Section 3 we describe our algorithms for solving the stated problems. Finally, we briefly conclude, and give some future proposals in
1. Definitions and Problems

We define an alphabet $\Sigma$ as a finite, non-empty set of symbols. An ordering can be defined via a bijection $\phi : \Sigma \rightarrow \{1, 2, \ldots, |\Sigma|\}$. Throughout this article we consider a string $y$, $|y| = n$, composed by letters drawn from an alphabet $\Sigma = \{\Sigma_1, \Sigma_2, \ldots, \Sigma_\sigma\}$, where $|\Sigma| = \sigma \leq n$. It is represented as $y[0 \ldots n-1]$. A string $w$ is a factor of $y$ if $y = uuwv$ for two strings $u$ and $v$. It is a prefix of $y$ if $u$ is empty and a suffix of $y$ if $v$ is empty. A string $u$ is a border of $y$ if $u$ is both a prefix and a suffix of $y$. The border of $y$, denoted by $\text{Border}(y)$, is the length of the longest border of $y$. A string $u$ is a period of $y$, if $y$ is a prefix of $u^k$ for some positive integer $k$ (i.e. $y$ is a prefix of $uy$). The period of $y$, denoted by $\text{Period}(y)$, is the length of the shortest period of $y$.

Definitions relative to Parikh vectors are as in [15, 20]. The Parikh vector of a string $y$, denoted by $P(y)$, enumerates the cardinality of each letter of $\Sigma$ in $y$. That is $P[i-1]$ is the cardinality of the $i_{th}$ letter of $\Sigma$ in $y$, where $0 ? i ? \sigma - 1$. We denote by $P_y(i, m)$ the Parikh vector of the factor of $y$ of length $m$ starting at position $i$. The sum of the components of a Parikh vector is denoted by $|P|$. Given two Parikh vectors $P, Q$ we write $P \subset Q$ if $P[i] \leq Q[i]$, for every $0 ? i ? \sigma - 1$ and $|P| \leq |Q|$. A string $y$ is said to have an abelian period $(h, p)$ if $y = u_0u_1\ldots u_{k-1}u_k$ such that:

- $P(u_0) \subset P(u_1) = \ldots = P(u_{k-1}) \supset P(u_k)$
- $|P(u_0)| = h, |P(u_1)| = p$

Factors $u_0$ and $u_k$ are called the head and the tail of the abelian period respectively. A string $y$ is said to have a weak abelian period $p$ if $y = u_0u_1\ldots u_{k-1}u_k$ such that:

- $P(u_0) = P(u_1) = \ldots = P(u_{k-1}) \supset P(u_k)$
- $|P(u_0)| = p$

Example 1. String $y = \text{caabbacabbca}$ has $(2, 5)$ as an abelian period (see Figure 1) and 5 as a weak abelian period (see Figure 2).

Figure 1: $(2, 5)$ is an abelian period of $y$  Figure 2: 5 is a weak abelian period of $y$

A natural order can be defined on abelian periods as follows: let $(h, p)$ and $(h', p')$ be abelian periods of a string $y$, then $(h, p) < (h', p')$ if $p < p'$ or $(p = p'$ and $h < h')$. Given a mapping $p : \Sigma \rightarrow A$, where $A$ is the set of the first $\sigma$ prime numbers, such that $p(\Sigma_i) = i_{th}$ prime number, the $P$-signature of a word $y$ is defined to be equal
Identifying all abelian periods of a string in quadratic time and relevant problems

Michalis Christou, Maxime Crochemore and Costas S. Iliopoulos

to $\prod_{i=0}^{|y|-1} p(y[i])$. We remind the reader that a prime is a positive integer greater than 1 having exactly one positive divisor other than 1.

Given a mapping $s : \Sigma \rightarrow B$, where $B$ is the set of the first $\sigma - 1$ powers of $n + 1$ and 0, such that:

$$s(\Sigma_i) = \begin{cases} 0, & i = \{1\} \\ (n + 1)^{(i-2)}, & \text{otherwise} \end{cases}$$  \hspace{1cm} (1)

the $S$-signature of a word $y$ is defined to be equal to $\sum_{i=0}^{|y|-1} s(y[i])$.

The array $Pr$, where $Pr[i] = \prod_{j=0}^i p(y[j])$, is useful in computing the $P$-signature of substrings of $y$, as:

$$P\text{-signature}(y[q..k]) = \begin{cases} Pr[k] / Pr[q-1], & q \neq 0 \\ Pr[k], & q = 0 \end{cases} \hspace{1cm} (2)$$

The array $S$, where $S[i] = \sum_{j=0}^i s(y[j])$, is useful in computing the $S$-signature of substrings of $y$, as:

$$S\text{-signature}(y[q..k]) = \begin{cases} S[k] - S[q-1], & q \neq 0 \\ S[k], & q = 0 \end{cases} \hspace{1cm} (3)$$

We consider the following problems:

**Problem 1 (Abelian period decision)** Decide if $(h, p)$, where $0 \leq h < \min(p, \left\lfloor \frac{n-1}{2} \right\rfloor + 1)$ and $1 \leq p \leq n$, is an abelian period of some string $y$.

**Problem 2 (String-Abelian period decision)** Decide if a string $x$, where $|x| = m < n$, composed from the same alphabet $\Sigma$ as a string $y$ can be an abelian period of $y$, i.e. there exist an abelian period $(h, p)$ of $y$ such that $y[h..h+p-1]$ is a permutation of $x$.

**Problem 3 (String-Abelian periods)** Output all abelian periods $(h, p)$ of $y$ such that $y[h..h+p-1]$ is a permutation of a string $x$, where $|x| = m < n$ and $x$ is composed from the same alphabet $\Sigma$ as $y$.

**Problem 4 (Computing all weak abelian periods of a string)** Compute all weak abelian periods of some string $y$.

**Problem 5 (Computing all abelian periods of a string)** Compute all abelian periods of some string $y$. 
2. Properties

In this section, we prove some useful properties for abelian periods and we also quote some fundamental properties of primes that prove to be useful for the analysis of our algorithms.

**Theorem 2.** (Fundamental Theorem of arithmetic) [21]
Every positive integer, except 1, can be represented in exactly one way apart from permutations as a product of one or more primes.

**Theorem 3.** (Prime Number Theorem) [21]
\[ \pi(n) \sim \frac{n}{\ln n}, \text{ where } \pi(n) \text{ is the number of primes less than } n. \]

**Corollary 4.** [21] \( p_n \sim n \log n \), where \( p_n \) is the \( n \)th prime number.

**Theorem 5.** [3] There exists an algorithm that gives the prime numbers up to a natural number \( N \) in time \( O\left(\frac{N}{\log \log N}\right) \).

**Theorem 6.** [21] \( \lim_{k \to \infty} \sum_{i=1}^{k} \frac{1}{i} \to \ln(n) + \gamma \), where \( \gamma \) is the Euler-Mascheroni constant.

**Lemma 7.** There exists an algorithm that gives the first \( n \) primes in time \( O(\frac{n \log n}{\log \log(n \log n)}) \).

**Proof.** Immediate consequence of Theorem 3 and Corollary 4.

**Lemma 8.** Two strings \( x, y \) of same length are represented by the same Parikh vector iff they share the same P-signature.

**Proof.** Immediate consequence of Theorem 2.

**Lemma 9.** Two strings \( x, y \) of same length are represented by the same Parikh vector iff they share the same S-signature.

**Proof.** Direct: Suppose \( x \) and \( y \) are strings of the same length and share the same S-signature, i.e.:
\[
S\text{-signature}(x) = \sum_{i=0}^{||x||-1} s(x[i]) = \sum_{i=0}^{k} a_i (n+1)^i
\]
\[
S\text{-signature}(y) = \sum_{i=0}^{||y||-1} s(y[i]) = \sum_{i=0}^{q} b_i (n+1)^i
\]
, where \( a_i \) is the cardinality of \( \Sigma_{i+1} \) in \( x \) and \( b_i \) is the cardinality of \( \Sigma_{i+1} \) in \( y \). W.l.o.g. consider \( k \geq q \).
\[
S\text{-signature}(y) = \sum_{i=0}^{||y||-1} s(y[i]) = \sum_{i=0}^{q} b_i (n+1)^i \leq n(n+1)^q \text{ as } b_i \leq n
\]
and so \( S\text{-signature}(y) < (n+1)^{q+1} \).

Therefore \( q = k \) and by using similar arguments:
\[
\sum_{i=0}^{q-1} b_i (n+1)^i \leq n(n+1)^q - 1 < (n+1)^q \text{ and so } a_k = b_k.
\]
Similarly it follows that \( a_j = b_j \) for every \( j \in \{0,1,\ldots,k\} \).

**Reverse:** Trivial
Lemma 10. Let \( rs[i] = \text{minimum } j \text{ such that } P(y[0..i-1]) \subset P(y[i..j]) \), where \( i \in \{1, 2, \ldots, n\} \). Then \( P(y[0..i-1]) \subset P(y[i..q]) \) for all \( q \in \{rs[i], rs[i]+1, \ldots, n-1 \} \) and \( rs[i] \leq rs[i]+1 \) for all \( i \in \{1, 2, \ldots, n-1 \} \).

Proof. First part:
Let \( rs[i] = \text{minimum } j \text{ such that } P(y[0..i-1]) \subset P(y[i..j]) \) for some \( i \in \{1, 2, \ldots, n\} \). Then for \( q \in \{rs[i], rs[i]+1, \ldots, n-1 \} \) holds that \( P(y[i..q]) = P(y[i..rs[i]]) + P(y[rs[i]+1..q]) \) and hence \( P(y[0..i-1]) \subset P(y[i..q]) \).
Second part:
By definition \( rs[i+1] = \text{minimum } j \text{ such that } P(y[0..i]) \subset P(y[i+1..j]) \)
= minimum \( j \text{ such that } P(y[0..i-1]) \subset P(y[i..j]) + \text{max}(0, \text{minimum } k \text{ such that } (P(y[i+1..i+k]) - P(y[0..i]))[y[i]] > 0 - rs[i]) \geq rs[i] \).

Lemma 11. Let \( re[i] = \text{maximum } j \text{ such that } P(y[n-i..n-1]) \subset P(y[j..n-i-1]) \), where \( i \in \{1, 2, \ldots, n\} \). Then \( P(y[n-i..n-1]) \subset P(y[q..n-i-1]) \) for all \( q \in \{re[i], re[i]+1, \ldots, n\} \) and \( re[i] \geq re[i]+1 \) for all \( i \in \{1, 2, \ldots, n-1 \} \).

Proof. Similar to the proof of Lemma 10.

3. The algorithms

In this section, we describe our algorithms for solving Problems 1–5. Firstly we describe some data structures that are used throughout the algorithms. Then we show how to solve the more basic problems and we extend these ideas to solve Problem 4 and Problem 5, ending with some comments on the analysis of the given algorithms.

3.1. Preprocessing

Before proceeding with the algorithms we will need some preprocessing to compute the following:

- The \( S\)-signature of each prefix of \( y \) is precomputed and stored in array \( S \), such that \( S[i] = S\text{-signature}(y[0..i]) \) for \( 0 \leq i \leq n-1 \). The necessary powers of \( n+1 \) can be computed in \( O(\sigma) \) time and stored in an array \( S \) such that they don’t have to be computed every time they are called. Then we fill the array using the properties \( S[0] = s(y[0]) \) and \( S[i] = S[i-1] + s(y[i]) \) for \( 1 \leq i \leq n-1 \).

- The \( P\)-signature of each prefix of \( y \) is precomputed and stored in array \( Pr \), such that \( Pr[i] = P\text{-signature}(y[0..i]) \) for \( 0 \leq i \leq n-1 \). We assume that the necessary primes can be easily found from a library in the computer. Otherwise we can produce them fast using a prime sieve as in [3] (see also Theorem 5). Then we fill the array using the properties \( Pr[0] = p(y[0]) \) and \( Pr[i] = Pr[i-1]p(y[i]) \) for \( 1 \leq i \leq n-1 \).
The array $rs$, where $rs[i] = \text{minimum } j \text{ such that } P(y[0..i-1]) \subseteq P(y[i..j])$ is computed in linear time using the properties $rs[i+1] \geq rs[i]$ and $re[i+1] \leq re[i](\text{Lemma 10})$. We use a simple sliding window approach keeping $P(y[0..rs[h-1]]) - P(y[0..h-1])$ in $PV$ array. When trying to find $rs[h]$, if $PV[\phi[y[h-1]] - 1] \geq 0$ then $rs[h] = rs[h-1]$, otherwise we search for $y[h-1]$ in $\{y[rs[h]..n-1]\}$ and use that length as an answer, if not found we assign $n$ to $rs[h]$ (\text{Lemma 10}).

The array $re$, where $re[i] = \text{maximum } j \text{ such that } P(y[n-i..n-1]) \subseteq P(y[j..n-i-1])$ is computed in a similar manner to the way we compute $rs$ so we only give the algorithm to find $rs$.

\begin{algorithm}
    \begin{algorithmic}
    \State $rs[0] \leftarrow 0$;
    \For{$i \leftarrow 0$ to $\sigma - 1$} \Do
        \State $PV[i] \leftarrow 0$;
        \For{$h \leftarrow 1$ to $\left\lceil \frac{n-1}{2} \right\rceil$} \Do
            \If{$PV[\phi[y[h-1]] - 1] \leftarrow PV[\phi[y[h-1]] - 1] - 2$;}
            \Else
                \State $q \leftarrow rs[h-1]$;
                \While{$(PV[\phi[y[h-1]] - 1] \neq 0)$ and $q < n - 1$} \Do
                    \State $q \leftarrow q + 1$;
                    \State $PV[\phi[y[q]] - 1] \leftarrow PV[\phi[y[q]] - 1] + 1$;
                    \If{$PV[\phi[y[h-1]] - 1] \neq 0$} \Then
                        \State $rs[h] \leftarrow n$;
                    \Else
                        \State $rs[h] \leftarrow q$;
                    \EndIf
                \EndWhile
            \EndIf
        \EndFor
    \EndFor
    \end{algorithmic}
\end{algorithm}

\section{3.2. Preliminary problems}

In this section, we describe algorithms for solving Problems 1-3. These problems are quite basic and our algorithms for Problem 4 and Problem 5 use similar ideas. The weak abelian period version of the first two problems is solved in the same manner.

Problem 1 is solved in $O(n)$ time by checking the required conditions for $(h, p)$ to be an abelian period, i.e. the necessary Parikh vectors, using either the $S$-signature or the $P$-signature of factors of $y$ (as of \text{Lemmas 8 and 9}). A careful sliding window implementation would also be able to solve the problem in $O(n)$ time.
Problem 2 is solved in \(O(n)\) time by the following steps:

- If \(\mathcal{P}(x) \not\subset \mathcal{P}(y[0 \ldots 2|x|-1])\) then the answer is immediately no.
- We calculate the array \(Pr\) or \(S\) for rapid comparison of Parikh vectors.
- We check each \((h, m)\), where \(0 \leq h < \min(m, \left\lfloor \frac{n-1}{2} \right\rfloor +1)\), if it is an abelian period of \(y\) until we find the first one that is. Clearly we go over at most \(\left\lfloor \frac{n}{m} \right\rfloor\) factors during each period check. We check at most \(m\) different periods and hence the algorithm is linear.

Problem 3 is solved in the same way but in the last step we keep checking for abelian periods after we find the first one. Clearly we go over at most \(\left\lfloor \frac{n}{m} \right\rfloor\) factors during each period check. We check \(m\) different periods and hence the linearity of the algorithm.

### 3.3. Identifying all weak abelian periods

This algorithm uses basic ideas from the above preliminary algorithms to solve Problem 4. Before proceeding with the algorithm the \(S\)-signature or the \(P\)-signature of each prefix of \(y\) is precomputed and stored in the array \(S\) or the array \(Pr\) respectively. We also precompute the array \(re\), where \(re[i] = \max j \text{ such that } \mathcal{P}(y[i \ldots n-i-1]) \subset \mathcal{P}(y[j \ldots n-i-1])\) in linear time using the properties of Lemma 11. We only show the version of the algorithm that uses the \(S\)-signature as it is almost the same as with the version using the \(P\)-signature.

**Algorithm All-Weak-Abelian-Periods-S**

\[
\begin{align*}
\text{ALGORITHM All-Weak-Abelian-Periods-S}(y, n, S, re) \\
1: & \text{ for } p \leftarrow 1 \text{ to } n \text{ do} \\
2: & \text{ if } p \geq n - re(n \mod p) - n \mod p \text{ then} \\
3: & \quad i \leftarrow 1; \\
4: & \quad \text{while } ((i < \left\lfloor \frac{n}{p} \right\rfloor) \text{ and } (S[(i + 1) \cdot p - 1] - S[ip - 1]) = S[p - 1]) \text{ do} \\
5: & \quad i \leftarrow i + 1; \\
6: & \quad \text{if } i = \left\lfloor \frac{n}{p} \right\rfloor \text{ then} \\
7: & \quad \text{Output } p;
\end{align*}
\]

**Theorem 12.** Algorithm All-Weak-Abelian-Periods-S runs in time \(O(n\log n)\).

**Proof.** Computation of the arrays \(S\) and \(re\) is done in linear time as it is easy to see that each letter is checked at most once during that phase of preprocessing. During the execution of the main algorithm we go over only from some factors of \(y\) which are checked at most once. These determine the complexity of our algorithm:

\[
\sum_{i=1}^{\lfloor |y| \rfloor} \frac{n}{\lfloor \frac{n}{i} \rfloor} \leq n \sum_{i=1}^{\lfloor |y| \rfloor} \frac{1}{\frac{1}{i}}
\]
Theorem 6 states that \( \lim_{k \to \infty} \sum_{i=1}^{k} \frac{1}{i} \to \ln(n) + \gamma \), where \( \gamma \) is the Euler-Mascheroni constant, and so we get the above result.

**Theorem 13.** Algorithm **All-Weak-Abelian-Periods-S** has \( \Theta(n) \) best case running time.

**Proof.** Consider an alphabet \( \Sigma \). It is easy to see that the word \( y = \Sigma[1]\Sigma[2] \ldots \Sigma[\sigma] \), where \( \Sigma[i] \) is the \( i \)th letter of \( \Sigma \), has no abelian periods. On executing our algorithms re is full of \(-1\) and therefore we never enter the if part of the main loop of the algorithm, thus only counting from \( p \leftarrow 1 \) to \( n \). No better running time is possible as preprocessing needs \( \Theta(n) \) time.

### 3.4. Identifying all abelian periods

We propose two algorithms for the solution of Problem 5. The first one maps each letter to a suitable number such that each factor of the string can be identified by the unique sum of the numbers corresponding to its letters (S-signature). The other one maps each letter to a prime number such that each factor of the string can be identified by the unique product of the numbers corresponding to its letters (P-signature). We are then able to perform the required checks of Parikh vectors, necessary to identify abelian periods, with just one operation using ideas from algorithms from the preliminary problems.

#### 3.4.1. S-Signature algorithm

This algorithm makes use of the S-signature of factors of \( y \) in order to make rapid comparison of Parikh vectors. It takes as input the string \( y \), its length \( n \) and the arrays \( S, rs \) and \( re \) and outputs all the abelian periods of \( y \) in the required encoding. For each possible \( h \) from 0 to \( \lfloor \frac{n-1}{2} \rfloor \) we check all possible values of \( p \) from \( rs(h) - h + 1 \) to \( n - h \). For \( (h, p) \) to be an abelian period we need:

1. \( \mathcal{P}(y[0 \ldots h - 1]) \subset \mathcal{P}(y[h \ldots h + p - 1]) \), i.e. \( p \geq rs(h) - h + 1 \).
2. \( \mathcal{P}(y[h \ldots h + p - 1]) = \mathcal{P}(y[h+p \ldots h+2p-1]) = \cdots = \mathcal{P}(y[h + ((n-h) \mod p) - 1)p \ldots h + ((n-h) \mod p)p - 1]), \) i.e. \( (S[i+1] * p + h - 1) - S[ip + h - 1]) = S[p + h - 1] - S[h - 1]) \) for all \( i \in \{2,3,\ldots,((n-h) \mod p) - 1\} \).
3. \( \mathcal{P}(y[h + ((n-h) \mod p)p \ldots n - 1]) \subset \mathcal{P}(y[h \ldots h + p - 1]), \) i.e.\( p \geq n - re((n-h) \mod p) - (n-h) \mod p \).

**Theorem 14.** Algorithm **All-Abelian-Periods-S** runs in \( O(n^2) \) time.

**Proof.** Computation of the arrays \( S, rs \) and \( re \) is done in linear time as it is easy to see that each letter is checked at most once during that phase of preprocessing.
Identifying all abelian periods of a string in quadratic time and relevant problems

Michalis Christou, Maxime Crochemore and Costas S. Iliopoulos

**Algorithm**

```
ALGORITHM All-Abelian-Periods-S(y, n, S, rs, re)
1: for h ← 0 to ⌊(n+1)/2⌋ do
2:   for p ← rs(h) to n − h do
3:     if p ≥ n − re((n − h) mod p) − (n − h) mod p then
4:       i ← 1;
5:       while ((i < ⌊n−h/p⌋) and (S[i+1] * p + h − 1) − S[i+1]) =
6:         S[p + h − 1] − S[h − 1]) do
7:         i ← i + 1;
8:     if i = ⌊n−h/p⌋ then
9:       Output (h, p);
```

The necessary powers of \( n + 1 \) are computed in \( \Theta(\sigma) \) time, where \( \sigma \leq n \). During the execution of the main algorithm all the factors of \( y(n(n+1)/2) \) are checked at most once which gives time complexity \( O(n^2) \).

3.4.2. \( P \)-signature algorithm

This algorithm makes use of the \( P \)-signature of factors of \( y \) in order to make rapid comparison of Parikh vectors. It takes as input the string \( y \), its length \( n \) and the arrays \( Pr, rs \) and \( re \) and outputs all the abelian periods of \( y \) in the required encoding. For each possible \( h \) from 0 to \( \lfloor (n+1)/2 \rfloor \) we check all possible values of \( p \) from \( h+1 \) to \( n−h \). For \( (h,p) \) to be an abelian period we need:

1. \( P(y[0..h-1]) \subset P(y[h..h+p-1]) \),
   i.e. \( p \geq rs(h) \).
2. \( P(y[h..h+p-1]) = P(y[h+p..h+2p-1]) = \cdots = P(y[h+((n−h) mod p)p−1]) \),
   i.e. \( Pr[(i+1) \cdot p + h - 1]/Pr[mp + h - 1]) = Pr[p + h - 1]/Pr[h - 1]) \) for all \( i \in \{1, 2, \ldots, ((n−h) mod p) − 1\} \)
3. \( P(y[h+((n-h) mod p)p..n-1]) \subset P(y[h..h+p-1]) \),
   i.e. \( p \geq n - re((n-h) mod p) - (n-h) mod p \).

**Theorem 15.** Algorithm All-Abelian-Periods-P runs in \( O(n^2) \) time.

**Proof.** Computation of the arrays \( Pr, rs \) and \( re \) is done in linear time as it is easy to see that each letter is checked at most once during that phase of preprocessing. During the execution of the main algorithm all the factors of \( y(n(n+1)/2) \) are checked at most once which gives time complexity \( O(n^2) \).

3.4.3. An example

We provide an example, providing the data structures build for the execution of our algorithm on the string \( y = acabbacabbca \).
Identifying All Abelian Periods of a String in Quadratic Time and Relevant Problems

**Algorithm**

```plaintext
ALL-ABELIAN-PERIODS-P(y, n, Pr)
1: for h ← 0 to \(\frac{n-1}{2}\) do
2: for p ← \(rs(h)\) to \(n - h\) do
3: if \(p ≥ rs(h) - h + 1\) then
4: \(i ← 1;\)
5: while ((\(i < \frac{n-h}{p}\)) and (\(Pr[i + 1]*p + h - 1)/Pr[ip + h - 1]\) = \(Pr[p + h - 1]/Pr[h - 1]\)) do
6: \(i ← i + 1;\)
7: if \(i = \frac{n-h}{p}\) then
8: Output \((h, p)\);
```

| i  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|----|---|---|---|---|---|---|---|---|---|---|----|----|
| y[i]| a | c | a | b | a | c | a | b | b | c | a  |
| S[i]| 0 | 13| 13| 14| 15| 28| 28| 29| 30| 43| 43 |
| Pr[i]| 2 | 10| 20| 60| 180| 360| 1800| 3600| 10800| 32400| 324000 |
| rs[i]| 0 | 2 | 6 | 7 | 9 | 12| 12| 12| 12| 12| 12 |
| re[i]| 11| 7 | 6 | 6 | 3 | 2 | -1| -1| -1| -1| -1 |

In order to calculate the P-signature of factors of \(y\) we use the mapping \(p : \{a, b, c\} \rightarrow \{2, 3, 5\}\), such that:
\(p(a) = 2\) \(p(b) = 3\) \(p(c) = 5\)

In order to calculate the S-signature of factors of \(y\) we use the mapping \(s : \{a, b, c\} \rightarrow \{0, 1, 13\}\), such that:
\(s(a) = 0\) \(s(b) = 1\) \(s(c) = 13\)

All abelian periods of \(y\) are:
(0, 5), (0, 7), (0, 8), (0, 9), (0, 10), (0, 11), (0, 12), (1, 6), (1, 7), (1, 8), (1, 9), (1, 10), (1, 11), (2, 5), (2, 6), (2, 7), (2, 8), (2, 9), (2, 10), (3, 5), (3, 6), (3, 7), (3, 8), (3, 9), (4, 5), (4, 6), (4, 7), (4, 8), (5, 6), (5, 7)

3.5. Further comments on the complexity of the above algorithms

In this subsection we give more details on the complexity of the suggested algorithms. We claim that they are optimal under the natural encoding suggested by the definition of the abelian period and that they have a best case linear running time. We also observe that a large alphabet size may lead to the creation of large numbers during the execution of our algorithms. However when dealing with applications \(\sigma\) is very small compared to \(n\) and so our algorithms are efficient.

**Theorem 16.** Algorithm ALL-ABELIAN-PERIODS-P and Algorithm ALL-ABELIAN-PERIODS-S are optimal.
Proof. Consider the word $a^n$. As suggested in [20] it has $O(n^2)$ abelian periods, which is also the worst case running time of our algorithms.

Theorem 17. Algorithm All-Abelian-Periods-P and Algorithm All-Abelian-Periods-S have $\Omega(n)$ best case running time.

Proof. Consider an alphabet $\Sigma$. It is easy to see that the word $y = \Sigma[1]\Sigma[2] \ldots \Sigma[\sigma]$, where $\Sigma[i]$ is the $i_{th}$ letter of $\Sigma$, has no abelian periods. On executing our algorithms $rs$ is full of $n$ and therefore we never enter the second loop of the algorithm, thus only counting from $h \leftarrow 0$ to $\lfloor \frac{2n}{\sigma} \rfloor$. No better running time is possible as preprocessing needs $\Theta(n)$ time.

As mentioned before a large alphabet size may lead to the creation of large numbers during the execution of our algorithms. In particular it is the signatures of the factors that might grow too large. The following theorems show the worst case size that they can have.

Theorem 18. The number of digits of variables used during the execution of Algorithm All-Abelian-Periods-P is $O(n \log(\frac{\sigma}{\log(\sigma)}))$.

Proof. Consider an alphabet $\Sigma$. The biggest variable encountered during the execution of the algorithm is the $P$-signature of the word $y = (\Sigma[\sigma])^n$, where $\Sigma[i]$ is the $i_{th}$ letter of $\Sigma$. That means $P$-signature$(y) = (i_{th}$ prime number$)^n$.

As suggested by Corollary 4, $P$-signature$(y)$ is $O((\frac{\sigma}{\log(\sigma)})^n)$.

Theorem 19. The number of digits of variables used during the execution of Algorithm All-Abelian-Periods-S is $O(\sigma \log(n))$.

Proof. Consider an alphabet $\Sigma$. The biggest variable encountered during the execution of the algorithm is the $S$-signature of the word $y = (\Sigma[\sigma])^n$, where $\Sigma[i]$ is the $i_{th}$ letter of $\Sigma$. That means $S$-signature$(y) = n(n + 1)^{\sigma - 2}$

Fortunately the numbers formed when we execute Algorithm All-Abelian-Periods-P can be further reduced by taking logarithms of the signatures as shown in the definitions below:

- The $P'$-signature of a word $y$ is defined to be equal to $\log(\prod_{i=0}^{n-1} p(y[i]))$.
- The array $Pr'$ is given by $Pr'[i] = \log(\prod_{j=0}^{n} p(y[j]))$

The array $Pr'$ is useful in computing the $P'$-signature of substrings of $y$, as:

$$P'$-signature$(y[q \ldots k]) = \begin{cases} Pr[k] - Pr[q - 1], & q \neq 0 \\ Pr[k], & q = 0 \end{cases}$$  (4)
As before the array $Pr'$ can be easily calculated using the properties $Pr'[0] = \log(p(y[0]))$ and $Pr'[i] = Pr'[i - 1] + \log(p(y[i]))$ for $1 \leq i \leq n - 1$. As of Theorem 18 the number of digits of variables used during the execution of Algorithm \textsc{All-Abelian-Periods-P} is $O(\log(n \log(\frac{\sigma}{\log(\sigma)}))) = O(\log n)$, while the running time of the algorithm is the same.

4. Conclusion—Further Work

Parikh vectors have found applications in bioinformatics, particularly in mass spectrometry data, DNA alignment, SNP discovery, repeated pattern discovery and gene clusters [9]. Recently, Constantinescu and Ilie [15] defined the abelian period of a string and several algorithms for the computation of all abelian periods of a string were given by Fici et al [20].

In this article, we have provided two $O(n^2)$ time algorithms for computing all abelian periods of a given string. We have also introduced the notion of the weak abelian period and we gave an $O(n \log n)$ algorithm for the computation of all weak abelian periods of a given string. Additionally we have analyzed simpler problems for the identification of abelian periods in strings and gave linear time algorithms for their solution. Our algorithms make extensive use of the $P$-signature and $S$-signature of factors of the string, thus being able to quickly compare Parikh vectors.

Further work can be done on designing a faster algorithm for the computation of all weak abelian periods of a string, or as suggested by Fici et al [20] on designing an algorithm for the computation of the Abelian Period array of a given string, i.e. computing the shortest abelian period of each prefix of a string as for its classical analog in [23]. Additionally further work can be done on algorithms around abelian quasiperiodicities, thus extending work done on classical ones, e.g. covers and seeds [1, 2, 8, 12, 13, 21, 22]. Variants of those algorithms are very likely to find applications in other areas such as bioinformatics.

Bibliography

[1] A. Apostolico and D. Breslauer. Of periods, quasiperiods, repetitions and covers. In Structures in Logic and Computer Science, pages 236–248, 1997.
[2] A. Apostolico and A. Ehrenfeucht. Efficient detection of quasiperiodicities in strings. Theor. Comput. Sci., 119(2):247–265, 1993.
[3] A.O.L. Atkin and D.J. Bernstein. Prime sieves using binary quadratic forms. Mathematics of Computation, 73(246):1023–1030, 2004.
[4] S. Avustalinovich, A. Glen, B.V. Halldórsson, and S. Kitaev. On shortest crucial words avoiding abelian powers. Discrete Applied Mathematics, 158(6):605–607, 2010.
[5] S. Avustalinovich, J. Karhumaki, and S. Puzynina. On abelian versions of critical factorization theorem. Proceedings of the 13th Mons. Theoretical Computer Science Days, 2010.
[6] F. Blanchet-Sadri, J. Kim, R. Mercas, W. Severa, and S. Simmons. Abelian square-free partial words. Language and Automata Theory and Applications, pages 94–105, 2010.
[7] F. Blanchet-Sadri, A. Tebbe, and A. Veprauskas. Fine and Wilf's theorem for abelian periods in partial words. *Proceedings of the 13th Mons Theoretical Computer Science Days, Amiens, France*, 2010.

[8] D. Breslauer. An on-line string superprimitivity test. *Inf. Process. Lett.*, 44(6):345–347, 1992.

[9] P. Burcsi, F. Cicalese, G. Fici, and Z. Lipták. On table arrangements, scrabble freaks, and jumbled pattern matching. In *Fun with Algorithms*, pages 89–101, Springer, 2010.

[10] P. Burcsi, F. Cicalese, G. Fici, and Z. Lipták. Algorithms for jumbled pattern matching in strings. (accepted for publication in *International Journal of Foundations of Computer Science*).

[11] J. Cassaigne, G. Richomme, K. Saari, and L.Q. Zamboni. Avoiding abelian powers in binary words with bounded abelian complexity. *International Journal of Foundations of Computer Science*, 22(4):905–920, 2011.

[12] M. Christou, M. Crochemore, C. Iliopoulos, M. Kubica, S. Pissis, J. Radoszewski, W. Rytter, B. Szreder, and T. Waleń. Efficient seeds computation revisited. In *Combinatorial Pattern Matching*, pages 350–363. Springer, 2011.

[13] M. Christou, M. Crochemore, O. Guth, C. S. Iliopoulos, and S. P. Pissis. On the right-seed array of a string. In *Proceedings of the 17th Annual International Computing and Combinatorics Conference (COCOON’11)*, Lecture Notes in Computer Science. Springer, 2011. (accepted).

[14] F. Cicalese, G. Fici, and Z. Lipták. Searching for jumbled patterns in strings. *Proceedings of the Prague Stringology Conference 2009*.

[15] S. Constantinescu and L. Ilie. Fine and Wilf's theorem for abelian periods. *Bulletin of the EATCS*, 89:167–170, 2006.

[16] J.D. Currie and A. Aberkane. A cyclic binary morphism avoiding abelian fourth powers. *Theoretical Computer Science*, 410(1):44–52, 2009.

[17] M. Domaratzki and N. Rampersad. Abelian primitive words. *DLT 2011-Developments in Language Theory*, 2011.

[18] T. Ejaz. Abelian pattern matching in strings. *PhD Thesis, Technischen Universität Dortmund*, 2010.

[19] T. Ejaz, S. Rahmann, and J. Stoye. Online abelian pattern matching. *Research Report, Technischen Universität Dortmund*, 2010.

[20] G. Fici, T. Lecroq, A. Lefeuvre, E. Prieur-Gaston, and MC Equipe. Computing abelian periods in words. *Proceedings of the Prague Stringology Conference 2011*.

[21] G.H. Hardy and E.M. Wright. An introduction to the theory of numbers. 1979.

[22] C. S. Iliopoulos, D. W. G. Moore, and K. Park. Covering a string. *Algorithmica*, 16:289–297, September 1996.

[23] D. E. Knuth, J. H. Morris Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[24] Y. Li and W. F. Smyth. Computing the cover array in linear time. *Algorithmica*, 32(1):95–106, 2002.

[25] M. Lothaire, editor. *Algebraic Combinatorics on Words*. Cambridge University Press, 2001.

[26] M. Lothaire, editor. *Applied Combinatorics on Words*. Cambridge University Press, 2005.

[27] A.V. Samsonov and A.M. Shur. On abelian repetition threshold. *Proc. 13th Mons Days of Theoretical Computer Science. Univ. de Picardie Jules Verne, Amiens*, pages 1–11, 2010.