Medical Science

http://www.aimspress.com/journal/medicalScience/

AIMS Medical Science, 4(3): 332–351
DOI:10.3934/ms.2017.3.332
Received: 29 April 2017
Accepted: 15 August 2017
Published: 30 September 2017

Research article

Subquadratic-Time Algorithms for Abelian Stringology Problems

Tomasz Kociumaka[1], Jakub Radoszewski[1,*] and Bartłomiej Wiśniewski[1]

[1] Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Warsaw, Poland

[*] **Correspondence:** E-mail: jrad@mimw.edu.pl; Tel: +48-22-5544482; Fax: +48-22-5544300

Abstract: We propose the first subquadratic-time algorithms to a number of natural problems in abelian pattern matching (also called jumbled pattern matching) for strings over a constant-sized alphabet. Two strings are considered equivalent in this model if the numbers of occurrences of respective symbols in both of them, specified by their so-called Parikh vectors, are the same. We consider the problems of computing abelian squares, abelian periods, abelian runs, abelian covers, and abelian borders. This work can be viewed as a continuation of a recent very active line of research on subquadratic space and time jumbled indexing for binary and constant-sized alphabets (e.g., Moosa and Rahman, 2012). Our algorithms apply the so-called four Russian technique in a fancy way and work in linear space. The purpose of this work is to show that breaking the $O(n^2)$ barrier is possible for the aforementioned problems. In this paper we extend our previous work, published in a preliminary form at MACIS 2015 conference, with efficient computation of abelian runs.

Keywords: jumbled pattern matching; abelian square; abelian period; abelian run

1. Introduction

Algorithmic abelian stringology has been extensively studied in the recent years. Abelian pattern matching (also called jumbled pattern matching) can be viewed as an approximate variant of regular pattern matching. The problem that has received most attention in this area is jumbled indexing; see [1–7]. In this problem we are to construct a data structure for a given text that can answer queries asking if there is a substring of the text that is commutatively equivalent to a given pattern. In the binary case, initially it was known that there exists a jumbled index of linear size that can answer queries (specified by the number of zeroes and ones) in constant time; see [3]. However, the straightforward construction time of such an index was quadratic. Due to a number of works [1, 6, 7] $O(n^2/\log^2 n)$-time construction algorithm of such an index was obtained. This line of research eventually lead to a breakthrough $O(n^{1.859})$-time algorithm by Chan and Lewenstein [2]. In this work we present the first subquadratic-time algorithms computing several basic notions of abelian stringology in the case of a binary and a constant-sized alphabet. This includes:

- abelian squares, that were first considered from a combinatorial perspective by Erdős [8] and from the algorithmic perspective by Cummings and Smyth [9] (in the latter case, under the name of weak repetitions);
- abelian periods, defined by Constantinescu and Ilie [10];
- abelian runs, introduced by Matsuda et al. [11] and further studied by Fici et al. [12];
- abelian covers, introduced by Matsuda et al. [11];
- natural notions of abelian borders and abelian border array.

1.1. Previous Results

Cummings and Smyth [9] presented an $O(n^2)$-time algorithm for computing all abelian squares in a string of length n.

Fici et al. [13] showed an $O(n^2)$-time algorithm computing abelian periods in a string over a constant-sized alphabet and Crochemore et al. [14] solved this problem in $O(n^2)$ time for any integer alphabet. Other types of abelian periods are known, including regular and full abelian periods (see [15]), for which linear-time or almost linear-time algorithms are known [16].

Abelian runs were first defined by Matsuda et al. [11] who gave an $O(n^2)$-time algorithm for their computation. A different—in a sense more general—notion of abelian runs was introduced by Fici et al. [12]. For clarity, we call the abelian runs defined by Matsuda et al. as anchored abelian runs. Fici et al. [12] presented several algorithms for computing abelian runs and anchored abelian runs; in particular, they show that abelian runs can be computed in $O(n^2)$ time. All these algorithms work for any integer alphabet.

Matsuda et al. [11] used a slightly different definition of abelian covers than ours. More precisely, they considered two abelian covers different if and only if the set of starting positions of the occurrences of the covers are different. They obtained an $O(n^2)$-time algorithm for computing a representation of all such (possibly exponentially many) abelian covers.

Computation of abelian regularities on RLE compressed strings was recently considered in [17, 18].

1.2. Our Results

We assume constant-sized alphabet and the word-RAM model of computation. By n we denote the length of the input string and by SQ the number of its substrings being abelian squares. We present algorithms with the following time complexities that work in linear space excluding the size of the result.

- Computing the longest abelian square, the shortest abelian square, and the number of all abelian squares in $O(n^2/\log^2 n)$ time and listing all abelian squares in $O(n^2/\log^2 n + \mathrm{SQ})$ time. This improves upon the algorithm of Cummings and Smyth [9].
- Computing the shortest (general) abelian period and all its occurrences, and an $O(n^2/\log n)$-size representation of all abelian periods in $O(n^2/\sqrt{\log n})$ time. This improves upon the results of Fici et al. [13] and Crochemore et al. [14] in the case of a constant-sized alphabet.
- Computing the number of anchored abelian runs in $O(n^2/\log n)$ time and listing all anchored abelian runs and abelian runs in $O(n^2/\log n + \mathrm{SQ})$ time. This yields an improvement (especially in the case that $\mathrm{SQ} = o(n^2)$) over the results of Matsuda et al. [11] and Fici et al. [12].
- Computing the shortest abelian cover in $O(n^2/\log n)$ time.
- Computing the abelian border array in $O(n^2/\log^2 n)$ time and a representation of all abelian borders of prefixes of the string in $O(n^2/\log^2 n + \mathrm{BD})$ time where BD is the total number of abelian

borders reported (showing a similarity between computing abelian borders and abelian squares).

In all algorithms we apply in a fancy way the technique called *four-Russian trick*, which was first introduced to abelian pattern matching by Moosa and Rahman [7] in the problem of jumbled indexing.

This is an extended version of a paper that was previously published at the MACIS 2015 conference [19]. The main extension is the computation of abelian runs and anchored abelian runs.

The following section includes definitions of the terms used in abelian stringology with examples as well as the basic notation. In Sections 3-6 we show our results for a binary alphabet. We start with computing abelian squares and abelian borders in Section 3. In Sections 4 and 5 we consider abelian periods and abelian runs, respectively; this is the most involved part of the paper. Finally, in Section 6 we compute abelian covers. In Section 7 we show that all presented algorithms can be applied to the case of any constant-sized alphabet, preserving both time and space complexity. Section 8 contains a brief discussion of possible future work.

2. Preliminaries

Let s be a string of length $n = |s|$ over an alphabet Σ of size $\sigma = |\Sigma|$. By $s[i]$, where $1 \leq i \leq n$, we denote the i-th symbol of s, and by $s[i, j]$ we denote a substring of s consisting of all symbols of s on positions from i to j inclusively. Substrings of the form $s[1, i]$ are called prefixes of s and substrings of the form $s[i, n]$ are called suffixes of s. For technical reasons, if $[i, j] \not\subseteq [1, n]$, we denote $s[i, j] = s[i', j']$ such that $[i', j'] = [i, j] \cap [1, n]$. In particular, if $[i, j] \cap [1, n] = \emptyset$, then we set $[i', j'] = [1, 0] = \emptyset$ and $s[i, j]$ denotes an empty string.

A *Parikh vector* of a string s, $\mathcal{P}(s)$, is a vector of size σ, where the element $\mathcal{P}(s)[l]$ stores the number of occurrences of symbol l in string s, i.e. $\mathcal{P}(s)[l] = x$ if and only if $|\{i : s[i] = l\}| = x$. The *norm* $r = \|R\|$ of a Parikh vector R is the sum of its elements; $r = \Sigma_{l\in\Sigma} R[l]$. For Parikh vectors P and Q, by $P + Q$ and $P - Q$ we denote their component-wise sum and difference.

We say that two strings s and t are *abelian equivalent* (or commutatively equivalent) if s is a permutation (an anagram) of t, or equivalently $\mathcal{P}(s) = \mathcal{P}(t)$, and write $s \approx t$. We say that t is an *abelian factor* of s if and only if $\mathcal{P}(t)[l] \leq \mathcal{P}(s)[l]$ for every symbol l of the alphabet Σ. We say that a position i in a string t is an *abelian occurrence* of a string s if $s \approx t[i, i + |s| - 1]$.

We now proceed with the definitions of the main notions of abelian periodicity that we consider in the paper.

Definition 1. An *abelian square* is a string t of length $2k$ such that $t[1, k] \approx t[k + 1, 2k]$.

The strings 011 101 and 10201 20011 are examples of abelian squares over a binary and a ternary alphabet, respectively.

Definition 2. A pair (i, p) for $1 \leq i \leq p$ is a *(general) abelian period* of s if and only if there exists a positive integer j such that $s[i, i + p - 1] \approx s[i + p, i + 2p - 1] \approx \ldots \approx s[i + (j - 1)p, i + jp - 1]$ and $s[1, i - 1]$ and $s[i + jp, n]$ are abelian factors of $s[i, i + p - 1]$. A *regular abelian period* k of s is a general abelian period of s that starts at the first position of s, i.e. the general abelian period $(1, p)$ of s is a regular abelian period p of s.

For example, $s = 00011101001100$ has, in particular, abelian periods $(3, 5)$: $00 \cdot 01110 \cdot 10011 \cdot 00$ and $(1, 5)$: $00011 \cdot 10100 \cdot 1100$, the latter one being regular.

The substrings $s[i, i + p - 1]$, $s[i + p, i + 2p - 1]$, …, $s[i + (j - 1)p, i + jp - 1]$ are called the *cores* of the abelian period, whereas the substrings $s[1, i - 1]$ and $s[i + jp, n]$ are the *head* and *tail*. The *Parikh period* corresponding to (i, p) is $\mathcal{P}(s[i, i + p - 1])$.

Definition 3. We say that a string s is an *anchored abelian repetition with abelian period* (a, p) if s has an abelian period (a, p) with at least two cores. We further say that s is an *abelian repetition with Parikh period* R if s is an anchored abelian repetition with core having Parikh vector R. An *anchored abelian run* or an *abelian run* is a substring of s that is an anchored abelian repetition or an abelian repetition, respectively, that is inclusion-maximal in s. More precisely, if $s[i, j]$ is an abelian run (anchored abelian run) with Parikh period R (abelian period (a, p), respectively), then each of the substrings $s[i-1, j]$ and $s[i, j+1]$ is either equal to $s[i, j]$ or is not an abelian repetition (anchored abelian repetition) with period R (abelian period $(a \bmod p+1, p)$ and (a, p), respectively).

An abelian run is always an anchored abelian run, but the opposite is not true; see [12]. In the classical notion of runs in a string (see [20, 21]) it suffices to specify the run as $s[i, j]$ since the shortest period can be then retrieved in a unique way. However, for abelian runs this is no longer the case. E.g., the string 00011101001100 is an anchored abelian run with abelian periods $(1, 5)$ and $(3, 5)$ that correspond to Parikh periods $(3, 2)$ and $(2, 3)$.

Definition 4. An *abelian border* t of s is a prefix of s that is abelian equivalent to some suffix of s. A proper abelian border t of s is an abelian border t such that $|t| < n$. An *abelian border array* of s is a table π of length n such that $\pi[i]$ is the length of the longest proper abelian border of $s[1, i]$.

As an example, the string 0101101110 has abelian borders of the following lengths:

- 1: $0 \approx 0$
- 2: $01 \approx 10$
- 5: $01011 \approx 01110$
- 8: $01011011 \approx 01101110$
- 9: $010110111 \approx 101101110$

It is easy to observe that a string of length n has an abelian border of length b if and only if it has an abelian border of length $n - b$ (see [11]).

Definition 5. An abelian border t is an *abelian cover* of s if and only if the abelian occurrences of t in s *cover* the whole string s. That is, if I is a sorted sequence of positions i in s such that $s[i, i+|t|-1] \approx t$, then the first element of I is 1, the last element of I is $n - |t| + 1$ and the differences between every two consecutive elements of I are no greater than $|t|$.

In Sections 3-6 we consider strings over a binary alphabet $\Sigma = \{0, 1\}$. In this case, to check if two strings s and t of the same length are abelian equivalent it suffices to know only the number of 1s in each of them. Similarly, we can use this number of check if a string t is an abelian factor of a string s, provided that we know the lengths of s and t. We denote $ones(s) = |\{i : s[i] = 1\}|$. Let us note that $ones(s[i, j])$ can be computed in constant time using an array precomputed in linear time that stores values of $ones(s[1, i])$ for every $1 \le i \le n$. We assume that $ones(s[i, j]) = 0$ for $i > j$.

In the binary case we assume that $s[i, i+m-1]$ for $m = O(\log n)$ can be stored using a constant number of integers (bit masks). An array that stores the representations of all such substrings for a given m can be computed in linear time.

3. Abelian Squares

In this section our main focus is finding the longest abelian squares which have their centres in every possible index i of string s. That is, for each $1 \le i \le n$ we want to find the largest k such that $s[i-k, i-1] \approx s[i, i+k-1]$.

This problem was solved in $O(n^2)$ time by Cummings and Smyth [9]. Let us briefly restate their solution in the case of a binary alphabet. For every i we take $k = \min(i-1, n-i+1)$ and store in a variable r the difference $ones(s[i-k, i-1]) - ones(s[i, i+k-1])$. If $r = 0$, then $s[i-k, i+k-1]$ is an abelian square. If not, then we decrease k by one, update r, and check again. We continue this way until we find such k for which the condition holds. One can see that this solution works in $O(n^2)$ time and $O(n)$ space. Our algorithm reduces the time complexity of this scheme by employing two optimisations.

3.1. First Optimisation

For the first optimisation we will use a precomputed 3-dimensional auxiliary array A of size $(2m+1) \times 2^m \times 2^m$. The particular value of m will be chosen later; it will satisfy $m = O(\log n)$. If there exists a string w of length $2k \geq 2m$ with an m-letter prefix p, m-letter suffix q and $ones(w[1, k]) - ones(w[k+1, 2k]) = \Delta$, and the longest abelian square centred in the middle of w has length $k-l$ for $0 \leq l \leq m$, then $A[\Delta][p][q] = l$. Otherwise (if such an abelian square does not exist or it is shorter) $A[\Delta][p][q] = -1$.

One can see that when $|\Delta| > m$, then $A[\Delta][p][q] = -1$ for every p and q, so there is no need to store this information explicitly. Each cell of array A can be computed in $O(m)$ time, so the whole array A can be constructed in $O(m^2 4^m)$ time. To compute each cell of the array A we obviously do not need to generate the string w.

When analysing a specific index i of string s and a specific length k we set an m-letter prefix of $s[i-k, i+k-1]$ as p, and an m-letter suffix of $s[i-k, i+k-1]$ as q, i.e. $p = s[i-k, i-k+m-1]$ and $q = s[i+k-m, i+k-1]$. Additionally, we set Δ as the difference in the number of 1s between the first and the second half of $s[i-k, i+k-1]$, i.e. $\Delta = ones(s[i-k, i-1]) - ones(s[i, i+k-1])$; see Fig. 1.

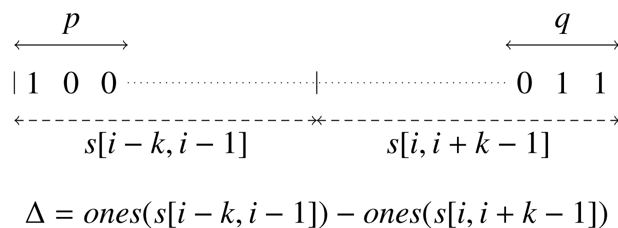

Figure 1. How to choose p, q and Δ for given i, k and $m = 3$. If $\Delta = 1$, $A[\Delta][p][q] = 2$.

If $A[\Delta][p][q] \neq -1$ and k is the largest possible (that is, $k = \min(i-1, n-i+1)$), we have found the longest abelian square with a centre in i. If $A[\Delta][p][q] = -1$, we know that the length of the longest abelian square is not greater than $k-m$. We can now decrement k by m and check the array A again. We finish either when the first abelian square is found or when k drops below m; in the latter case we find the longest abelian square in $O(m)$ time. This way we can compute the length of the longest abelian square with a centre in i in $O(n/m + m)$ time and the lengths of longest abelian squares for every centre index i of s in $O(n^2/m + nm)$ time.

Lemma 1. *The longest abelian squares for every centre index i of a string of length n can be computed in $O(n^2/\log n)$ time using $O(n)$ space.*

Proof. By choosing $m = \left\lfloor \frac{\log n}{3} \right\rfloor$ the running time becomes:

$$O(n^2/m + nm + m^2 4^m) \quad = \quad O(n^2/\log n + n \log n + 4^{\frac{\log n}{3}} \log^2 n)$$

$$= O(n^2/\log n + n^{\frac{2}{3}} \log^2 n)$$
$$= O(n^2/\log n)$$

The space complexity becomes:

$$\begin{aligned} O(n + m4^m) &= O(n + 4^{\frac{\log n}{3}} \log n) \\ &= O(n + n^{\frac{2}{3}} \log n) \\ &= O(n) \end{aligned}$$

□

3.2. Second Optimisation

We will now use a second optimisation, which will allow us to further decrease the time complexity. For this purpose we will precompute an auxiliary array B of size $(2m-1) \times 2^{2m-1} \times 2^{2m-1} \times 2^m \times 2^m$.

If there is a string w of length $2k + m - 1$ ($k \geq m$) that satisfies all the conditions:

- $ones(w[1,k]) - ones(w[k+1,2k]) = \Delta$,
- its $(2m-1)$-letter prefix is p,
- its $(2m-1)$-letter suffix is q,
- $c = w[k+1, k+m-1]$,
- b is a bit mask of length m,

then $B[\Delta][p][q][c][b]$ stores all pairs (j, l) such that $j \leq m$, $l \leq m$, $b[j] = 0$ and $k - l$ is the length of the longest abelian square with a centre in $k + j$.

Every cell of array B can be computed in $O(m^2)$ time with a modified algorithm for finding abelian squares by Cummings and Smyth [9] that we have already recalled at the beginning of this section. We check m centre indices and m lengths of potential abelian squares. The variable r from the algorithm can be initialised in constant time using Δ.

In our algorithm, instead of analysing a specific index i, we will analyse m indices i, $i+1$, ..., $i+m-1$ at a time. When considering i we look at $k = \min(i-1, n-i-m+2)$, set the $(2m-1)$-letter prefix of $s[i-k, i+k+m-2]$ as p and the $(2m-1)$-letter suffix of the same as q. We also take $\Delta = ones(s[i-k, i-1]) - ones(s[i, i+k-1])$ and $c = s[i, i+m-1]$.

Additionally, we maintain a bit mask b telling us for which of these m indices we have already found the longest abelian square (we initialise all bits of b to 0). Now we check the array B with proper values for each dimension (see Fig. 2), save all pairs stored in the corresponding cell of B, replace b with the updated mask and decrease k by m. We continue this way until all bits of b are 1 or until $k < m$, in which case we search for the abelian squares at positions $i, \ldots, i+m-1$ in $O(m^2)$ time. Then we start considering $i + m$.

This solution can be implemented in two nested loops, where both the inner and the outer loop execute $O(n/m)$ runs. This way the whole algorithm works in $O(n^2/m^2 + nm^2)$ time, after constructing array B in $O(m^3 64^m)$ time. Since the array can possibly store more than one value in each cell, it uses $O(m^2 64^m)$ space.

This algorithm computes the lengths of the longest abelian squares with their centres in every index i. It can be easily modified to compute the lengths of the shortest abelian squares. Instead of starting with the largest possible k and decreasing it by m in every run of the inner loop, we may start with

$$\overset{p}{\longleftrightarrow} \qquad \overset{c}{\longleftrightarrow} \qquad \overset{q}{\longleftrightarrow}$$

$$|0\ 1\ 0\ 0\ 1 \cdots |1\ 0\ 1 \cdots 0\ 1\ 1|1\ 1$$

$$\underset{s[i-k,i-1]}{\longleftarrow\!-\!-\!-\!\times\!-\!-\!-\!\longrightarrow}\ s[i,i+k-1]$$

$$\Delta = ones(s[i-k,i-1]) - ones(s[i,i+k-1])$$

Figure 2. How to choose p, q, c and Δ for given i, k and $m = 3$. If $\Delta = 1$ and $b = (0,0,1)$, then $B[\Delta][p][q][c][b] = ((1,1),(2,3))$. As for the pair $(2,3)$, we have $ones(s[i-k+1,i]) - ones(s[i+1,i+k]) = 2$ and $ones(s[i-k+4,i]) = ones(s[i+1,i+k-3])$.

$k = 1$ and increase it by m in every run. The array B also needs to be modified to store information about the shortest abelian squares.

Theorem 1. *The longest (shortest) abelian squares for every centre index i of a string of length n can be computed in $O(n^2/\log^2 n)$ time using $O(n)$ space.*

Proof. By choosing $m = \left\lfloor \frac{\log n}{7} \right\rfloor$ the running time becomes:

$$\begin{aligned} O(n^2/m^2 + nm^2 + m^3 64^m) &= O(n^2/\log^2 n + n\log^2 n + 64^{\frac{\log n}{7}} \log^3 n) \\ &= O(n^2/\log^2 n + n^{\frac{6}{7}} \log^3 n) \\ &= O(n^2/\log^2 n) \end{aligned}$$

The space complexity becomes:

$$\begin{aligned} O(n + m^2 64^m) &= O(n + 64^{\frac{\log n}{7}} \log^2 n) \\ &= O(n + n^{\frac{6}{7}} \log^2 n) \\ &= O(n) \end{aligned}$$

□

3.3. Counting All Abelian Squares

We can modify the values stored in the array B so that we can use them to count the number of all abelian squares in the string s. In each cell of B we may store the number of all abelian squares, when the values Δ, p, q and c are specified. As we are interested only in computing the number of all abelian squares, we may omit maintaining the bit mask b and cut one of the dimensions of the array B. This obviously does not increase the overall time and space complexity.

Using the same approach we may also compute a list of all abelian squares in the string s (i.e. pairs of centre and length for every abelian square). This time the array B stores data about all the respective abelian squares instead of only the longest ones (and also the bit mask b is not needed). The time complexity is $O(n^2/\log^2 n + \mathrm{SQ})$ and the space complexity becomes $O(n + \mathrm{SQ})$.

3.4. Abelian Borders

It is known that a string s has an abelian border of length i if and only if it has an abelian border of length $n - i$ (see Lemma 4 in Matsuda et al. [11]). This way the longest and the shortest abelian borders are determined by each other. We will modify the algorithm for computing abelian squares of s so that it will compute the abelian border array of s.

We still will be analysing m indices $i, i+1, \ldots, i+m-1$ at once, but this time they stand for m ends of consecutive prefixes $s[1, i]$, $s[1, i+1]$, ..., $s[1, i+m-1]$. We will focus on finding the shortest abelian borders for these prefixes. For given i we start with $k = 1$. We set Δ to the difference of numbers of 1s in $s[1, k-1]$ and $s[i+m-k+1, i+m-1]$. We denote $(2m-1)$-letter prefix of $s[k, i+m-k]$, $p = s[k, k+2m-2]$ and $(2m-1)$-letter suffix of the same, $q = s[i-m-k+2, i+m-k]$. We fix $c = s[i, i+m-1]$ and maintain a bit mask b, where the j-th bit of b tells if we have already found the shortest abelian border of $s[1, i+j-1]$.

We will use a precomputed auxiliary array C with the same number of dimensions and the same size of respective dimensions as array B. Given Δ, p, q, c and b the array C stores pairs (j, l) of shortest abelian borders that were found. Every cell of this array can be computed in $O(m^2)$ time.

The algorithm executes two nested loops. The outer loop starts with $i = 1$ and increases i by m after every run. The inner loop starts with $k = 1$ and increases k by m after every run. This leads us to the following corollary.

Corollary 2. *The abelian border array of string s can be computed in $O(n^2/\log^2 n)$ time using $O(n)$ space.*

Similarly as in the case of computing a distribution of abelian squares by their centres, we can compute a representation of all abelian borders of prefixes of s in $O(n^2/\log^2 n + \mathrm{BD})$ time and $O(n)$ additional space.

4. Abelian Periods

The problem of finding all abelian periods was solved in $O(n^2)$ time for any alphabet by Crochemore et al. [14]. Let us briefly recall their solution adapted to our binary alphabet case.

One can see that k is a regular abelian period of $s[i, n]$ ($k \le n - i + 1$) if and only if all the conditions below hold:

- k is a regular abelian period of $s[i+k, n]$ (or $|s[i+k, n]| \le k$),
- if $n \ge i + 2k - 1$, then $s[i+k, i+2k-1] \approx s[i, i+k-1]$,
- if $n < i + 2k - 1$, then $s[i+k, n]$ is an abelian factor of $s[i, i+k-1]$.

Thereby (i, k) for $i \le k$ is a (general) abelian period of s if and only if k is a regular abelian period of $s[i, n]$ and $s[1, i-1]$ is an abelian factor of $s[i, i+k-1]$.

All general abelian periods of s are of the form (i, k), where $1 \le i, k \le n$, thus information about all of them can be represented as an array GP of size $n \times n$, where $GP[i][k] = 1$ if (i, k) is an abelian period of s and $GP[i][k] = 0$ otherwise. We can compute this array using dynamic programming. For this purpose we use an array RP of size $n \times n$, where $RP[i][k] = 1$ if k is a regular abelian period of $s[i, n]$ and $RP[i][k] = 0$ otherwise.

We will first show how to compute RP. We set $RP[i][k] = 0$ if $i + k - 1 > n$. We will consider all remaining pairs (i, k) in a descending order of i. If $i + 2k - 1 > n$, we only need to check if $s[i+k, i+2k-1]$ is an abelian factor of $s[i, i+k-1]$. Otherwise, we check if $RP[i+k][k] = 1$ and

if $s[i, i+k-1]$ and $s[i+k, i+2k-1]$ are abelian equivalent. In each case the check can be done in constant time, so the whole array RP can be computed in $O(n^2)$ time and space.

To compute the array GP we look at each $RP[i][k] = 1$ such that $i \leq k$ and set $GP[i][k] = 1$ if and only if $s[1, i-1]$ is an abelian factor of $s[i, i+k-1]$.

The length of the shortest abelian period of s, the number of all of them and the number of all abelian periods of s can be easily extracted from the GP array.

4.1. Computing RP Faster

For a better picture, let us assume that for all arrays i numbers columns from left to right and k numbers rows from the top to the bottom. Let us take $m < n$ and without loss of generality assume that m is a divisor of n (otherwise we increase n by at most $m-1$). We can code $O(\log n)$ cells of each of the arrays GP, RP into one machine word. In our case we will pack all cells on indices $\{i, i+1, \ldots, i+m-1\} \times \{k, k+1, \ldots, k+m-1\}$ (there are exactly m^2 of them), where $m|(i-1)$ and $m|(k-1)$, into unit square arrays of size $m \times m$. If m is small enough, we can code such a unit array into one machine word, row by row. This way both GP and RP are divided regularly into n^2/m^2 unit square arrays and stored in $O(n^2/m^2)$ space.

From now on we will still reference to a single cell of GP and RP on indices (i, k) or a subarray of any of them, but we are assuming that the physical representation of arrays is as described. To improve time complexity of computing RP and GP we will fill out m cells of RP at a time and m cells of GP at a time.

To compute RP we will use three auxiliary arrays: A of size $(2m+1) \times 2^{m^2} \times 2^{m-1} \times 2^{2m-2} \times (2m-1)$, E of size $2^{m^2} \times 2^{m^2} \times m$, and F of size $2^{m^2} \times 2^m \times m$.

E is an array which, for given two unit arrays X and Y and a shift i, stores a unit array Z such that the first $n-i+1$ columns of Z are the last $n-i+1$ columns of X and the last $i-1$ columns of Z are the first $i-1$ columns of Y.

F is an array which for given unit array X, a column (that is, a 1-d array) C and a shift i, stores a unit array Y which is X with the i-th column replaced by C. We also store an analogous array that can substitute a given row of a unit array.

Finally, $A[\Delta][X][p][q] = C$ if and only if there exists a string w of length at least $2k$ such that all of the conditions below hold:

- $p = w[k+1, k+m-1]$,
- $q = w[2k+1, 2k+2m-2]$,
- $\Delta = ones(w[1, k]) - ones(w[k+1, 2k])$,
- $k = lm + 1$ for some integer l,
- X is a unit array containing information about regular periods on suffixes of w starting on indices from $\{k+1, k+2, \ldots, k+m\}$ and having lengths from $\{k, k+1, \ldots, k+m-1\}$,
- C is a column containing information about regular periods of w with lengths from $\{k, k+1, \ldots, k+m-1\}$.

Note that q may exceed the boundary of w; in this case its length is smaller than $2m-2$. Hence, we also need to store the length of q. For every Δ, X, p and q, there exists exactly one C. If $|\Delta| > m$, then $C = (0, 0, \ldots, 0)$ and we do not store this information explicitly; otherwise we can simply precompute each cell of A in $O(m)$ time.

We will use this array to, instead of calculating one value $RP[i][k]$ at a time, calculate $RP[i][k, k+m-1]$ at once.

The first step is to compute, for every position i, the smallest k_i such that $k_i \geq (n - i + 1)/2$ and k_i is a regular abelian period of $s[i, n]$. To this end we can apply binary search which costs us $O(n \log n)$ time over all positions i. Now we know that $RP[i][k] = 1$ for all $k_i \leq k < n - i + 1$. Knowing the range of values of k, we can set the respective columns of unit arrays of RP to ones (or to ones starting from a given point) in $O(n^2/m)$ time using the array F.

Now for the remaining pairs (i, k) we are interested only in periods with at least two full occurrences. We iterate first by k, starting with $k = n - m + 1$ and decreasing it by m. For a given k we iterate by i from $n - 2k + 1$ to 1, decreasing it by 1.

In each operation we take $w = s[i, n]$, $X = RP[i + k, i + k + m - 1][k, k + m - 1]$ and p, q and Δ respectively; see Fig. 3. X can be extracted from two particular unit arrays from the n^2/m^2 unit arrays that RP consists of, by using array E. After getting a new column C from array A, that corresponds to $RP[i][k, k + m - 1]$, we can update the appropriate unit array in RP, using array F.

p q

| 1 0 | 1 1 0 1

$s[i, i + k - 1]$ $s[i + k, i + 2k - 1]$

$$\Delta = ones(s[i, i + k - 1]) - ones(s[i + k, i + 2k - 1])$$

Figure 3. How to choose p, q and Δ for given i, k and $m = 3$. If $\Delta = 0$ and $X = \begin{pmatrix} \mathbf{1} & 0 & 0 \\ 0 & \mathbf{0} & 0 \\ 0 & 0 & \mathbf{1} \end{pmatrix}$, then $A[\Delta][X][p][q] = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$.

All the operations above are constant-time. Thus the whole RP array can be computed in $O(n^2/m)$ time.

4.2. Computing GP Faster

Let us notice that if $RP[i][k] = 0$, then $GP[i][k] = 0$. Thus the array GP is the array RP with some cells set to 0.

To compute the array GP we will first copy the array RP into it. Then we will update $GP[i][k] = 0$ for all $i > k$. This can be done in $O(n^2/m^2)$ time as before, using the array F.

The remaining cells that have to be set to 0 are determined by all i and k for which $s[1, i - 1]$ is not an abelian factor of $s[i, i + k - 1]$. To detect all such i and k, for each i we compute the minimal k_i, such that $s[1, i - 1]$ is an abelian factor of $s[i, i + k_i - 1]$. This can be done in linear time, since obviously $(i + 1) + k_{i+1} \geq i + k_i$; see [14].

Having i and k_i we set $GP[i][k] = 0$ for all $k < k_i$. Since all such k form an interval, it can be done in $O(n/m)$ time for specific i and k_i using the array F and in $O(n^2/m)$ time for the whole array GP.

The arrays RP and GP use $O(n^2/m^2)$ space. If we are interested only in counting periods of every size and storing all shortest periods, it suffices to store only m consecutive rows of both arrays, i.e. rows $k, \ldots, k + m - 1$ (and all k_i's). For this only $O(n/m)$ unit arrays and $O(n)$ additional space is necessary.

In conclusion, computing arrays A, E and F costs us $O(m^3 2^{m^2} 8^m + m^3 4^{m^2})$ time and $O(m^2 2^{m^2} 8^m + m 4^{m^2})$ space. Computing the arrays RP and GP costs us additionally $O(n \log n + n^2/m)$ time and $O(n)$ space.

Theorem 3. *All shortest abelian periods of a string of length n can be computed in $O(n^2/\sqrt{\log n})$ time using $O(n)$ space.*

Proof. By choosing $m = \left\lfloor \sqrt{\frac{\log n}{3}} \right\rfloor$ the running time becomes:

$$\begin{aligned} O(n^2/m + m^3 2^{m^2} 8^m + m^3 4^{m^2}) &= O(n^2/\sqrt{\log n} + 2^{\frac{\log n}{3}} 8^{\sqrt{(\log n)/3}} \log^{\frac{3}{2}} n + 4^{\frac{\log n}{3}} \log^{\frac{3}{2}} n) \\ &= O(n^2/\sqrt{\log n} + n^{0.34} + n^{\frac{2}{3}} \log^{\frac{3}{2}} n) \\ &= O(n^2/\sqrt{\log n}) \end{aligned}$$

The space complexity becomes:

$$\begin{aligned} O(n + m^2 2^{m^2} 8^m + m 4^{m^2}) &= O(n + 2^{\frac{\log n}{3}} 8^{\sqrt{(\log n)/3}} \log n + 4^{\frac{\log n}{3}} \sqrt{\log n}) \\ &= O(n + n^{0.34} + n^{\frac{2}{3}} \sqrt{\log n}) \\ &= O(n) \end{aligned}$$

□

5. Abelian Runs

In the first subsection we present an efficient algorithm for counting anchored abelian runs. In the next subsection we proceed with efficient enumeration of both types of abelian runs.

5.1. Counting Anchored Abelian Runs

If $s[i, j]$ is an anchored abelian run with abelian period (a, p), then we call the abelian square substring of s that is composed of the first two cores of $s[i, j]$ the *basic square* of $s[i, j]$. Let us start with the following easy observation.

Observation 1. *An abelian square $s[a-p, a+p-1]$ is a basic square of an anchored abelian run in s if and only if $s[a-2p, a-1]$ has length smaller than $2p$ or is not an abelian square. Moreover, if $s[a-p, a+p-1]$ is a basic square of an anchored abelian run, then the run is determined uniquely.*

Proof. Assume that $s[a-p, a+p-1]$ is an abelian square in s. Let j be the maximum index $j \geq a+p-1$ such that $s[a-p, j]$ has a regular abelian period p. Similarly, let i be the minimum index $i \leq a-p$ such that the reversal of $s[i, a+p-1]$ has a regular abelian period p. If $i > a-2p$, then $s[a-2p, a-1]$ is not an abelian square of length $2p$ and indeed $s[a-p, a+p-1]$ is the basic square of the anchored abelian run $s[i, j]$. Otherwise, $s[a-2p, a-1]$ is an abelian square of the specified form and, therefore, $s[a-p, a+p-1]$ is not a basic square of any anchored abelian run. □

Thus, to count anchored abelian runs, it suffices to count abelian squares satisfying the conditions of the observation.

5.1.1. A Different Approach to Computation of Abelian Squares

Let ASQ$[i][k]$ be a Boolean array such that ASQ$[i][k] = 1$ if and only if $s[i-k, i+k-1]$ is an abelian square in s. For simplicity we assume that ASQ$[i][k] = 0$ for i or k being out of range. Let m be an integer parameter such that $m = O(\sqrt{\log n})$. First we will show that, after $O(n)$-time and space preprocessing, we can compute in $O(1)$ time any $m \times m$ subarray of ASQ represented as a constant number of machine words.

It suffices to modify the approach of Section 3.1. Intuitively, consider any abelian square with centre $i \in [i_0, i_0 + m - 1]$ and square half length $k \in [k_0, k_0 + m - 1]$. This abelian square certainly contains all the positions of s from the interval $[i_0 - k_0 + (m-1), i_0 + k_0 - 1]$. Depending on the particular values of i and k, it may also contain positions from $s[i_0 - k_0 - (m-1), i_0 - k_0 + (m-1) - 1]$ and $s[i_0 + k_0, i_0 + k_0 + 2(m-1) - 1]$, both substrings being of length $2m-2$. Hence, to check which i and k in the considered ranges indeed correspond to abelian squares in s, it suffices to know:

- $p = s[i_0 - k_0 - (m-1), i_0 - k_0 + (m-1) - 1]$,
- $q = s[i_0 + k_0, i_0 + k_0 + 2(m-1) - 1]$,
- $\Delta = ones(s[i_0 - k_0 + (m-1), i_0 - 1]) - ones(s[i_0, i_0 + k_0 - 1])$,
- $r = s[i_0, i_0 + m - 1]$.

The substring r needs to be stored in addition to what was stored in Section 3.1, as we do not know the exact centre of the abelian square so we might need to update the number of ones specified by Δ accordingly. In order to encapsulate the degenerate cases, in which some of the sought abelian squares do not exist because they would exceed the bounds of the string s, for each of the strings p, q, r we allow it to have length smaller than $2(m-1)$, $2(m-1)$, and m, respectively. Hence, the strings are stored as their length and the string itself. Again, if Δ is too large or too small, we can safely assume that no abelian square satisfying the requirements exists; in this case a sufficient condition is $|\Delta| > 3m$.

Consequently, we may store an array A of size $(6m+1) \times (2^{2m-2} \times (2m-1))^2 \times (2^m \times (m+1))$ such that $A[\Delta][p][q][r] = X$ if and only if there exists a string w and positive integers i_0 and k_0 such that:

- $p = w[i_0 - k_0 - (m-1), i_0 - k_0 + (m-1) - 1]$,
- $q = w[i_0 + k_0, i_0 + k_0 + 2(m-1) - 1]$,
- $\Delta = ones(w[i_0 - k_0 + (m-1), i_0 - 1]) - ones(w[i_0, i_0 + k_0 - 1])$,
- $r = w[i_0, i_0 + m - 1]$,
- $X = \text{ASQ}[i_0, i_0 + m - 1][k_0, k_0 + m - 1]$ is an $m \times m$ subarray of the ASQ array for w represented in $O(1)$ integers.

By what was written above, for any string w that satisfies the above conditions, the resulting array X is determined uniquely. Each element of the array X can be computed in $O(m)$ time, which gives $O(m^3)$-time computation of a single cell of the array A.

5.1.2. Counting Basic Abelian Squares

Now let us consider a partition of the array ASQ into n^2/m^2 unit arrays of size $m \times m$. We assume for simplicity that $m \mid n$. For each unit array ASQ$[i_0, i_0 + m - 1][k_0, k_0 + m - 1]$ we wish to count the number of abelian squares with centres $i \in [i_0, i_0+m-1]$ and half lengths $k \in [k_0, k_0+m-1]$ that are not preceded by an abelian square whose second half coincides with their first half (as in Observation 1).

We will use an auxiliary array B of size $2^{m^2} \times 2^{2m^2}$ such that $B[X][Y] = c$ if and only if there exist a string w and positive integers i_0 and k_0 such that:

- $X = \text{ASQ}[i_0, i_0 + m - 1][k_0, k_0 + m - 1]$ for the ASQ array for w,
- $Y = \text{ASQ}[i_0 - k_0 - m, i_0 - k_0 + m - 1][k_0, k_0 + m - 1]$ for the ASQ array for w,
- the number of abelian squares $w[i - k, i + k - 1]$ in w such that
  - $i \in [i_0, i_0 + m - 1]$,
  - $k \in [k_0, k_0 + m - 1]$, and
  - $w[i - 2k + 1, i - 1]$ has length smaller than $2k$ or is not an abelian square

  is c.

One can observe that the value of c depends only on X and Y. Indeed, for each $i \in [i_0, i_0 + m - 1]$ and $k \in [k_0, k_0 + m - 1]$ such that $\text{ASQ}[i][k] = 1$, the abelian square $w[i - k, i + k - 1]$ satisfies the condition of Observation 1 if and only if $\text{ASQ}[i - k, k] = 0$. However:

$$i_0 - k_0 - m \;=\; i_0 - (k_0 + m) \;<\; i - k \;\leq\; i_0 + m - 1 - k_0 \;=\; i_0 - k_0 + m - 1$$

so $\text{ASQ}[i - k, k]$ is indeed present in the array Y. This proves that the value of c is the same for every string w that satisfies the aforementioned conditions and that each cell of the array B can be computed in $O(m^2)$ time.

In the end, we compute every $m \times m$ unit subarray X from the partition of ASQ and the corresponding $2m \times m$ subarray Y using the array A in $O(1)$ time. Then we count the anchored abelian runs with basic squares represented by X using the array B, also in $O(1)$ time.

In total, computing the array A costs us $O(m^7 2^{5m})$ time and $O(m^4 2^{5m})$ space and computing the array B costs us $O(m^2 2^{3m^2})$ time and $O(2^{3m^2})$ space. Computing the final result then takes $O(n^2/m^2)$ time, with $O(1)$ additional space.

Theorem 4. *The number of anchored abelian runs in a string of length n can be computed in $O(n^2/\log n)$ time using $O(n)$ space.*

Proof. By choosing $m = \left\lfloor \frac{\sqrt{\log n}}{2} \right\rfloor$ the running time becomes:

$$\begin{aligned} O(n^2/m^2 + m^7 2^{5m} + m^2 2^{3m^2}) &= O(n^2/\log n + 2^{\frac{5}{2}\sqrt{\log n}} \log^{\frac{7}{2}} n + 2^{\frac{3}{4}\log n} \log n) \\ &= O(n^2/\log n + n^{\frac{3}{4}} \log n) \\ &= O(n^2/\log n) \end{aligned}$$

The space complexity becomes:

$$\begin{aligned} O(n + m^4 2^{5m} + 2^{3m^2}) &= O(n + 2^{\frac{5}{2}\sqrt{\log n}} \log^2 n + 2^{\frac{3}{4}\log n}) \\ &= O(n + n^{\frac{3}{4}}) \\ &= O(n) \end{aligned}$$

□

5.2. Computing Anchored Abelian Runs and Abelian Runs

In this section we assume that the set of all abelian squares in s has already been computed; let U denote this set ($|U| = \text{SQ}$). Section 3.3 mentions that U can be computed in $O(n^2/\log^2 n + \text{SQ})$ time and $O(n + \text{SQ})$ space. Our first goal is to compute the set of all anchored abelian runs in s.

We start by identifying abelian squares that satisfy the condition of Observation 1 for being a basic square of some anchored abelian run. Let us first sort all the abelian squares of U by their lengths and by their centres in case of ties. This can be done in $O(n + \mathrm{SQ})$ time using radix sort [22]. Then, for every length k we iterate over the list L of centres of abelian squares with this square half length using two pointers, j_1 and j_2 ($0 \le j_1 < j_2 \le |L|$). The pointers move only along the list. We keep an invariant that $L[j_1]$ is the largest element that does not exceed $L[j_2] - k$; $j_1 = 0$ if no such element exists. First we proceed with j_2 by one. Afterwards we move j_1 along the list as long as the invariant is satisfied. In the end, the abelian square with the centre j_2 is *not* a basic square if and only if $j_1 > 0$ and $L[j_1] = L[j_2] - k$. For a list L this procedure works in $O(|L|)$ time which gives $O(\mathrm{SQ})$ time overall.

Next, for each basic square $s[a-k, a+k-1]$ we would like to compute all the cores of its anchored abelian run. That is, we need to find the largest integer multiple ck of k such that

$$s[a, a+k-1] \approx s[a+k, a+2k-1] \approx \ldots \approx s[a+ck, a+(c+1)k-1].$$

We find this multiple by checking naively the abelian equivalence of the respective substrings. The time complexity of this process over all basic squares amortises by $O(\mathrm{SQ})$, as each substring $s[a + rk, a+(r+1)k-1]$ that turns out to be equivalent to $s[a+(r-1)k, a+rk-1]$ accounts to a distinct abelian square substring in U.

Thus for each anchored abelian run we have computed its period length p and a substring $s[i', j']$ that is the concatenation of all its cores. To retrieve the run $s[i, j]$ itself, we need to find its head and tail, that is, the smallest i such that $s[i, i'-1]$ is an abelian factor of $s[i', i'+p-1]$ and the largest j such that $s[j'+1, j]$ is an abelian factor of $s[j'-p+1, j']$. We focus on computing the requested values of i; the algorithm for computing j is symmetric.

Assume there are Q substrings of the form $s[i', j']$ for a given given i', ordered by the period length p. We will compute the values of i for all the substrings in $O(n/\log n + \mathrm{Q})$ time using a solution to an abstract problem defined in the following subsection. This will yield the requested solution for the whole problem as all the substrings can be sorted by i' and then by p in case of draws in $O(\mathrm{SQ})$ time using radix sort.

5.2.1. Abstract Problem of Centred Abelian Factors

Consider a position i in the string s and the following simple algorithm that finds, for every position $b \ge i$, the smallest position $a \le i$ such that $s[a, i-1]$ is an abelian factor of $s[i, b]$. In the algorithm we increase b one by one, starting from $b = i$. After each incrementation, we decrease a one by one as long as $s[a, i-1]$ is an abelian factor of $s[i, b]$. This algorithm works in $O(n)$ time if the function *ones* is used to check if one substring of s is an abelian factor of the other.

We will speed up this algorithm. Imagine a sequence Z of bits that describes the execution of the algorithm: a 1 denotes incrementation of variable b and a 0 denotes decrementation of variable a. Let $m = O(\log n)$ be a parameter. The sequence Z has length up to n and so it can be divided into at most $n/m + 1$ chunks each of which can be represented by a constant number of machine words. The last—possibly incomplete—chunk is padded with ones. Let Z' denote this representation of the sequence Z.

Example 1. Assume that $s = 01010\,11010$ and $i = 6$. Then the sequence Z is 111001100.

We will show that Z' can be computed in $O(n/m)$ time, left to right. Assume that we have computed a prefix of the sequence Z', after which the variables a and b are at positions a_0 and b_0, respectively. Let $\Delta = ones(s[a_0, i-1]) - ones(s[i, b_0])$ and $\Delta' = (i - a_0) - (b_0 - i + 1) - \Delta$ be the differences

in ones and zeroes, respectively, of substrings $s[a_0, i-1]$ and $s[i, b_0]$. It is easy to observe that, to compute the next element of the sequence Z', it suffices to know the values $\hat{\Delta} = \max(\min(\Delta, m), -m)$ and $\hat{\Delta}' = \max(\min(\Delta', m), -m)$ and the substrings $p = s[a_0 - m, a_0 - 1]$ and $q = s[b_0 + 1, b_0 + m]$. Indeed, the total number of increments of b and decrements of a will be at most m, so the variables will stay within the respective substrings. Moreover, the change of the difference of the number of zeroes and ones in $s[a, i-1]$ and $s[i, b]$ comparing to the corresponding difference between $s[a_0, i-1]$ and $s[i, b_0]$ will be at most m. Note that each of the substrings $s[a_0 - m, a_0 - 1]$ and $s[b_0 + 1, b_0 + m]$ may have length smaller than m.

We use an auxiliary array C of size $(2m+1)^2 \times ((m+1) \times 2^m)^2$. We have that $C[\hat{\Delta}][\hat{\Delta}'][p][q] = Y$ if and only if the next element of the array Z' equals Y. Obviously, each cell of the array C can be computed in $O(m)$ time.

In the word-RAM model we can assume that the numbers of zeroes and ones in Y can be computed in constant time after standard $o(n)$-space and time precomputation. This lets us update a_0 and b_0 in $O(1)$ time, as desired.

Finally, it is not hard to see that during the computation of the sequence Z' we can answer queries for the smallest ℓ such that $s[\ell, i-1]$ is an abelian factor of $s[i, k]$, where k are listed in an increasing order. Assume that b_0 became greater than k after the new element Y of sequence Z' had been computed and that the previous values of the parameters were a'_0 and b'_0. Then we need to find the position r of the $(k+1-b'_0)$-th one in Y (a *select* query). To answer the query for k, we decrease a'_0 by the number of zeroes in Y before the position r (a *rank* query). Both the operations on Y can be performed in $O(1)$ time in the word-RAM model after standard $o(n)$-space and time precomputation. The special case of $k = n$ is handled by returning the last computed value of a_0.

One can further notice that we do not need to store the sequence Z' itself, only its last element.

5.2.2. The Algorithms

Let us summarise the computation of anchored abelian runs. We find all the abelian squares in $O(n^2/\log^2 n + \mathrm{SQ})$ time and $O(n + \mathrm{SQ})$ space. Then we check which of them are basic squares of anchored abelian runs in $O(n + \mathrm{SQ})$ time and find the cores of the respective anchored abelian runs in $O(\mathrm{SQ})$ time. Finally, we find the heads of all the Q anchored abelian runs whose basic square starts at the same position in $O(\mathrm{Q} + n/m)$ time and $O(\mathrm{Q})$ space. Thus computing all heads and all tails takes $O(\mathrm{R} + n^2/m)$ time and $O(\mathrm{R})$ space, where R is the total number of anchored abelian runs. Observation 1 shows that $\mathrm{R} \le \mathrm{SQ}$. The algorithm uses an auxiliary array C that takes $O(m^4 2^{2m})$ space and can be computed in $O(m^5 2^{2m})$ time. We arrive at the following result.

Theorem 5. *All anchored abelian runs in a string of length n can be computed in $O(n^2/\log n + \mathrm{SQ})$ time using $O(n + \mathrm{SQ})$ space.*

Proof. By choosing $m = \left\lfloor \frac{\log n}{3} \right\rfloor$ the running time becomes:

$$
\begin{aligned}
O(n^2/m + n + \mathrm{SQ} + m^5 2^{2m}) &= O(n^2/\log n + \mathrm{SQ} + 2^{\frac{2}{3}\log n} \log^5 n) \\
&= O(n^2/\log n + \mathrm{SQ} + n^{\frac{2}{3}} \log^5 n) \\
&= O(n^2/\log n)
\end{aligned}
$$

The space complexity becomes:

$$
\begin{aligned}
O(n + \mathrm{SQ} + m^4 2^{2m}) &= O(n + \mathrm{SQ} + 2^{\frac{2}{3}\log n} \log^4 n) \\
&= O(n + \mathrm{SQ} + n^{\frac{2}{3}} \log^4 n) \\
&= O(n + \mathrm{SQ})
\end{aligned}
$$

□

Fici et al. [12] show that abelian runs are those anchored abelian runs that are not properly contained in an anchored abelian run with the same Parikh period. They mention that one can filter out anchored abelian runs that do not satisfy this requirement in $O(n + \mathrm{R})$ time provided that all anchored abelian runs are sorted by their Parikh periods. However, for a constant-sized alphabet one can sort anchored abelian runs by their Parikh periods in $O(n + \mathrm{R})$ time by applying radix sort. Since $\mathrm{R} \leq \mathrm{SQ}$, which concludes the final result of this section.

Theorem 6. *All abelian runs in a string of length n can be computed in* $O(n^2/\log n + \mathrm{SQ})$ *time using* $O(n + \mathrm{SQ})$ *space.*

6. Abelian Covers

For given $i \leq n$ we can check if the prefix of length i of string s is an abelian cover of s. This is actually the abelian pattern matching problem, which for a constant-sized alphabet can obviously be solved in linear time.

Let us assume that we have found all occurrences of $s[1, i]$ in s and stored their (sorted) starting positions in a sequence I. Then $s[1, i]$ is an abelian cover of s if and only if the first element of I is 1, the last element of I is $n - i + 1$ and the differences between all pairs of consecutive elements of I are no greater than i.

Checking all $1 \leq i \leq n$, the shortest abelian cover of a string of length n can be found in $O(n^2)$ time.

6.1. Optimising Running Time

Let $m = O(\log n)$ be a parameter. Assume without loss of generality that m is a divisor of n. Instead of finding one occurrence of $s[1, i]$ in s at a time, we will only find the smallest and the largest positions from each of intervals $[1, m]$, $[m + 1, 2m]$, ..., $[n - m + 1, n]$ such that $s[1, i]$ is an abelian match on s starting at these positions.

For this purpose we will use a precomputed auxiliary array A of size $2^{m-1} \times (2m + 1) \times 2^{m-1}$. $A[p][\Delta][q] = (l_1, l_2)$ if and only if there are strings w of length $i + m - 1$ (text) and c of length i (potential cover) such that all of the conditions below hold:

- p is an $(m - 1)$-letter prefix of w,
- q is an $(m - 1)$-letter suffix of w,
- $\Delta = ones(w[1, i]) - ones(c)$,
- $1 \leq l_1 \leq l_2 \leq m$,
- $ones(p[1, l_1 - 1]) - ones(q[1, l_1 - 1]) = \Delta$, i.e. $c \approx w[l_1, l_1 + i - 1]$,
- $ones(p[1, l_2 - 1]) - ones(p[1, l_2 - 1]) = \Delta$, i.e. $c \approx w[l_2, l_2 + i - 1]$,
- l_1 is the smallest possible,
- l_2 is the largest possible.

If no such pair exists, then $A[p][\Delta][q] = (-1, -1)$. Each element of the array A can be precomputed in $O(m)$ time. We consider only $-m \leq \Delta \leq m$, so the whole array can be then precomputed in $O(m^2 4^m)$ time and uses $O(m4^m)$ space.

Our algorithm for finding the shortest abelian cover works as follows. First we check if any prefix of s of length less than $2m$ is a cover of s. This can be done by the straightforward solution in $O(nm)$ time. Now we assume that the shortest cover has length at least $2m$.

We iterate with i from $2m$ to n to check if $s[1, i]$ is an abelian cover of s. For every i we start with $I = ()$. Now we iterate with j from $j = 1$ to $n - i + 1$, increasing j by m each time. We are checking if there is an abelian match on positions $[j, j + m - 1]$, so we are implicitly considering the substring $s[j, j + i + m - 2]$.

For every i and j we denote three values:

- the $(m - 1)$-letter prefix of $s[j, j + i + m - 2]$, $p = s[j, j + m - 2]$,
- the $(m - 1)$-letter suffix of $s[j, j + i + m - 2]$, $q = s[j + i, j + i + m - 2]$ and
- the difference of 1s, $\Delta = ones(s[j, j + i - 1]) - ones(s[1, i])$.

If $|\Delta| > m$, then $s[1, i]$ is not abelian equivalent to any substring of $s[j, j + i + m - 2]$. Otherwise, we can find the first and the last occurrence (in the considered interval) by referencing to the array A and, if they exist, add them to I.

After iterating with j, I contains $O(n/m)$ elements. We can then check if $s[1, i]$ is an abelian cover of s by examining the set I in the same way as we did it in the straightforward solution. This way the whole algorithm works in $O(nm + n^2/m)$ time.

Theorem 7. *The shortest abelian cover of a string of length n can be computed in $O(n^2/\log n)$ time using $O(n)$ space.*

Proof. By choosing $m = \left\lfloor \frac{\log n}{3} \right\rfloor$ the running time becomes:

$$\begin{aligned} O(nm + n^2/m + m^2 4^m) &= O(n \log n + n^2/\log n + 4^{\frac{\log n}{3}} \log^2 n) \\ &= O(n^2/\log n + n^{\frac{2}{3}} \log^2 n) \\ &= O(n^2/\log n) \end{aligned}$$

The space complexity becomes:

$$\begin{aligned} O(n + m4^m) &= O(n + 4^{\frac{\log n}{3}} \log n) \\ &= O(n + n^{\frac{2}{3}} \log n) \\ &= O(n) \end{aligned}$$

□

It is clear that we can use the same algorithm to compute the lengths of all abelian covers of s. If we are interested in detecting all occurrences of all abelian covers of s, then the array A needs to store not only the pair of the smallest l_1 and the largest l_2 that satisfy all mentioned conditions, but all such l. This has no effect on space complexity, but the time complexity becomes $O(n^2/\log n + \text{COV})$, where COV is the number of all occurrences of all abelian covers in s.

The shortest abelian cover is probably of most interest. We can use our algorithm to find it and then we can find all its occurrences in s in $O(n)$ time, using abelian pattern matching.

7. The Case of A Larger Alphabet

We have presented subquadratic-time algorithms for computing abelian squares, abelian periods, abelian runs, abelian covers, and abelian borders for a string over a binary alphabet. In the fundamental problem of abelian stringology—jumbled indexing—switching from binary to a larger constant-sized alphabet increases the hardness of the problem considerably [5, 23]. However, all our algorithms can be applied to the case of any constant-sized alphabet, preserving both time and space complexity.

For this, in each of the presented auxiliary arrays, instead of considering Δ, that is, the difference in the number of 1s between two substrings, we may consider a difference of Parikh vectors of the corresponding substrings. The number of possible differences we should consider now is obviously no greater than $(2m+1)^{\sigma}$, where σ is the size of the alphabet. This increases the size of the auxiliary arrays and the computation time by at most a factor of $O(\log^{\sigma} n)$. These sizes and construction times were always $o(n^{1-\epsilon})$ for some $\epsilon > 0$. Hence, they will remain sublinear for a constant σ.

Instead of using the function *ones*, we may use an analogous constant-time function that returns a Parikh vector for a substring. In our algorithms we assume that a substring of length m can be stored using one machine word. For this to be still true for $\sigma > 2$, we need to divide the proposed values of m in Theorems 1, 5 and 7 by a factor of $\log \sigma$ and in Theorems 3 and 4 by a factor of $\sqrt{\log \sigma}$.

8. Conclusions

We have presented the first subquadratic-time algorithms for computing abelian squares, abelian periods, abelian runs, abelian covers, and abelian borders for a string over a constant-sized alphabet. An open question is if there exist $O(n^2/\log^c n)$-time algorithms for the problems considered in this paper with a greater constant c. It would be also interesting to construct an $O(n^2/\log^c n + \mathrm{R})$-time algorithm for reporting abelian runs of any type assuming that R is the number of the runs reported.

A further question, for all of the problems, is if there exists an $O(n^{2-\epsilon})$-time algorithm or, possibly, if there is a lower bound on the time complexity of an algorithm solving this problem. In comparison, for the seminal problem in this area, the jumbled indexing, on one hand, $O(n^{2-\epsilon})$-time algorithms are known for any constant-sized alphabet [2], but on the other hand, for sufficiently large alphabets $\sigma > 2$ conditional lower bounds are known for the query vs construction time trade-off [23, 24] based on the hardness of the 3SUM problem.

Acknowledgements

Tomasz Kociumaka is supported by Polish budget funds for science in 2013-2017 as a research project under the 'Diamond Grant' programme. Jakub Radoszewski is supported by the Polish Ministry of Science and Higher Education under the 'Iuventus Plus' programme in 2015-2017 grant no. 0392/IP3/2015/73. The author is also a Newton International Fellow at King's College London.

References

1 Burcsi P, Cicalese F, Fici G, et al. (2010) On Table Arrangements, Scrabble Freaks, and Jumbled Pattern Matching. In: Fun with Algorithms, 5th International Conference. Springer: 89-101.

2 Chan TM, Lewenstein M (2015) Clustered integer 3SUM via additive combinatorics. In: Proceedings of the forty-seventh annual ACM Symposium on Theory of Computing. ACM: 31-40.

3 Cicalese F, Fici G, Lipták Z (2009) Searching for Jumbled Patterns in Strings. In: Stringology: 105-117.

4 Hermelin D, Landau GM, Rabinovich Y, et al. (2014) Binary jumbled pattern matching via all-pairs shortest paths. arXiv preprint arXiv:1401.2065, 2014.

5 Kociumaka T, Radoszewski J, Rytter W (2013) Efficient indexes for jumbled pattern matching with constant-sized alphabet. *Algorithmica* 77: 1194-1215.

6 Moosa TM, Rahman MS (2010) Indexing permutations for binary strings. *Inf Process Lett* 110: 795-798.

7 Moosa TM, Rahman MS (2012) Sub-quadratic time and linear space data structures for permutation matching in binary strings. *J Discrete Algorithms* 10: 5-9.

8 Erdos P (1961) Some unsolved problems. Hungarian Academy of Sciences Mat. Kutató Intézet Közl 6: 221–254.

9 Cummings LJ, Smyth WF (1997) Weak repetitions in strings. *JCMCC* 24: 33-48.

10 Constantinescu S, Ilie L (2006) Fine and Wilf's Theorem for Abelian Periods. *Bulletin of the EATCS* 89: 167-170.

11 Matsuda S, Inenaga S, Bannai H, et al. (2014) Computing Abelian Covers and Abelian Runs. In: Stringology, 43-51.

12 Fici G, Kociumaka T, Lecroq T, et al. (2016) Fast computation of abelian runs. *Theor Comput Sci* 656: 256-264.

13 Fici G, Lecroq T, Lefebvre A, et al. (2014) Algorithms for computing Abelian periods of words. *Discrete Appl Math* 163: 287-297.

14 Crochemore M, Iliopoulos CS, Kociumaka T, et al. (2013) A note on efficient computation of all Abelian periods in a string. *Inf Process Lett* 113: 74-77.

15 Fici G, Lecroq T, Lefebvre A, et al. (2016) A note on easy and efficient computation of full abelian periods of a word. *Discrete Appl Math* 212: 88-95.

16 Kociumaka T, Radoszewski J, Rytter W (2017) Fast algorithms for abelian periods in words and greatest common divisor queries. *J Comput Syst Sci* 84: 205–218.

17 Amir A, Apostolico A, Hirst T, et al. (2016) Algorithms for jumbled indexing, jumbled border and jumbled square on run-length encoded strings. *Theor Comput Sci* 656:146–159.

18 Sugimoto S, Noda N, Inenaga S, et al. (2017) Computing Abelian regularities on RLE strings. arXiv preprint arXiv:1701.02836, 2017.

19 Kociumaka T, Radoszewski J, Wiśniewski B (2015) Subquadratic-time algorithms for Abelian stringology problems. In: International Conference on Mathematical Aspects of Computer and Information Sciences. Springer: 320-334.

20 Kolpakov R, Kucherov G (1999) Finding maximal repetitions in a word in linear time. In: 40th Annual Symposium on Foundations of Computer Science. IEEE: 596-604.

21 Kolpakov RM, Kucherov G (2000) On maximal repetitions in words. *J Discrete Algorithms*: 159–186.

22 Cormen TH, Leiserson CE, Rivest RL, et al. (2009) Introduction to Algorithms. 3 ed. MIT Press.

23 Amir A, Chan TM, Lewenstein M, et al. (2014) On hardness of jumbled indexing. In: International Colloquium on Automata, Languages, and Programming. Springer: 114-125.

24 Goldstein I, Kopelowitz T, Lewenstein M, et al. (2017) How hard is it to find (honest) witnesses? European Symposium on Algorithms. Schloss Dagstuhl: 45:1-45:16.

AIMS Press

© 2017, Tomasz Kociumaka et al., licensee AIMS Press. This is an open access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0)