Information and Computation 270 (2020) 104463

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

String periods in the order-preserving model ☆

Garance Gourdel [a], Tomasz Kociumaka [b,c,1], Jakub Radoszewski [c,*], Wojciech Rytter [c], Arseny Shur [d,2], Tomasz Waleń [c]

[a] Computer Science Department, ENS Paris-Saclay, Cachan, 5 rue Blaise Pascal, 92220 Bagneux, France
[b] Department of Computer Science, Bar Ilan University, 5290002 Ramat Gan, Israel
[c] Institute of Informatics, University of Warsaw, Banacha 2, 02-097 Warsaw, Poland
[d] Department of Algebra and Fundamental Informatics, Ural Federal University, pr. Lenina 51, 620000 Ekaterinburg, Russia

ARTICLE INFO

Article history:
Received 15 May 2018
Received in revised form 10 January 2019
Accepted 10 January 2019
Available online 4 September 2019

Keywords:
Order-preserving pattern matching
Period
Efficient algorithm

ABSTRACT

In the order-preserving model, two strings match if they share the same relative order between the characters at the corresponding positions. This model is quite recent, but it has already attracted significant attention because of its applications in data analysis. We introduce several types of periods in this setting (op-periods). Then we give algorithms to compute these periods in time $\mathcal{O}(n)$, $\mathcal{O}(n \log\log n)$, $\mathcal{O}(n \log^2 \log n / \log\log\log n)$, $\mathcal{O}(n \log n)$ depending on the type of periodicity. In the most general variant, the number of different op-periods can be as big as $\Omega(n^2)$, and a compact representation is needed. Our algorithms require novel combinatorial insight into the properties of op-periods. In particular, we characterize the Fine–Wilf property for coprime op-periods.

1. Introduction

Study of strings in the *order-preserving* model (*op-model*, in short) is a part of the so-called non-standard stringology. It is focused on pattern matching and repetition discovery problems in the shapes of number sequences. Here the *shape* of a sequence is specified by the relative order of its elements. The applications of the op-model include finding trends in time series which appear naturally when considering e.g. the stock market or melody matching of two musical scores; see [2]. In such problems periodicity plays a crucial role.

One of motivations is given by the following scenario. Consider a sequence D of numbers that models a time series which is known to repeat the same shape every fixed period of time. For example, this could be certain stock market data or statistics data from a social network that is strongly dependent on the day of the week, i.e., repeats the same shape every consecutive week. Our goal is, given a fragment S of the sequence D, to discover such repeating shapes, called here *op-periods*, in S; see Fig. 1. We also consider some special cases of this setting. If the beginning of the sequence S is synchronized with the beginning of the repeating shape in D, we refer to the repeating shape as to an *initial* op-period. If the synchronization takes place also at the end of the sequence, we call the shape a *full* op-period. Finally, we also consider

☆ A preliminary version of this work appeared at STACS 2018 [1].

* Corresponding author.
E-mail addresses: garance.gourdel@ens-paris-saclay.fr (G. Gourdel), kociumaka@mimuw.edu.pl (T. Kociumaka), jrad@mimuw.edu.pl (J. Radoszewski), rytter@mimuw.edu.pl (W. Rytter), arseny.shur@urfu.ru (A. Shur), walen@mimuw.edu.pl (T. Waleń).

[1] Supported by ISF grants no. 824/17 and 1278/16 and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).
[2] Supported by the Ministry of Science and Higher Education of the Russian Federation, project 1.3253.2017.

https://doi.org/10.1016/j.ic.2019.104463
0890-5401/

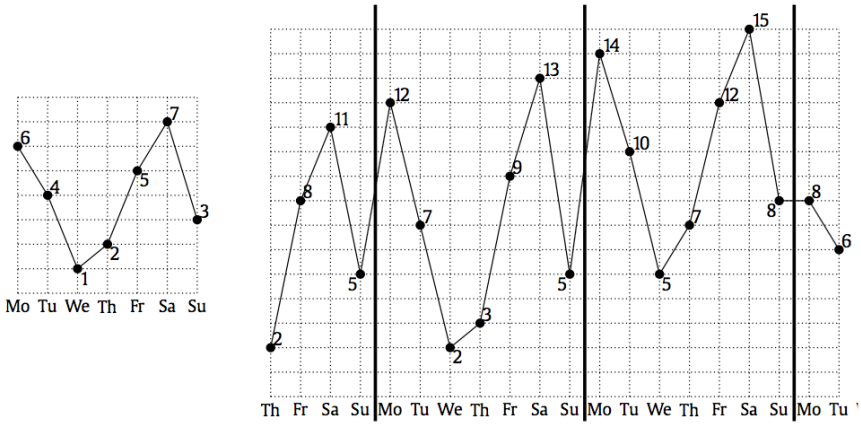

Fig. 1. The pattern to the right shows an example sequence $S = 2\ 8\ 11\ 5\ 12\ 7\ 2\ 3\ 9\ 13\ 5\ 14\ 10\ 5\ 7\ 12\ 15\ 8\ 8\ 6$ of some data reported daily being a fragment of a sequence D with op-period 7. The shape of the op-period is shown to the left (6 4 1 2 5 7 3).

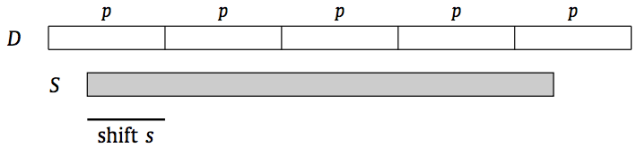

Fig. 2. The structure of a string S having an op-period p with shift s.

sliding op-periods that describe the case when every factor of the sequence D repeats the same shape every fixed period of time.

Order-preserving model. Let $[\![a\,..\,b]\!]$ denote the set $\{a, \ldots, b\}$. We say that two strings $X = X[1]\ldots X[n]$ and $Y = Y[1]\ldots Y[n]$ over an integer alphabet are *order-equivalent* (*equivalent* in short), written $X \approx Y$, if and only if

$$\forall_{i,j\in[\![1\,..\,n]\!]}\ X[i] < X[j] \iff Y[i] < Y[j].$$

Example 1.1. $5\ 2\ 7\ 5\ 1\ 3\ 10\ 3\ 5 \approx 6\ 4\ 7\ 6\ 3\ 5\ 9\ 5\ 6$.

Order-equivalence is a special case of so called *substring consistent equivalence relation* (SCER, in short) that was defined in [3].

For a string S of length n, we can create a new string X of length n such that $X[i]$ is equal to the number of distinct symbols in S that are not greater than $S[i]$. The string X is called the *shape* of S and is denoted by $shape(S)$. It is easy to observe that two strings S, T are order-equivalent if and only if they have the same shape.

Example 1.2. $shape(5\ 2\ 7\ 5\ 1\ 3\ 10\ 3\ 5) = shape(6\ 4\ 7\ 6\ 3\ 5\ 9\ 5\ 6) = 4\ 2\ 5\ 4\ 1\ 3\ 6\ 3\ 4$.

Periods in the op-model. We say that p is a *full op-period* of a string S if there is a decomposition $S = V_1 \cdot V_2 \cdots V_k$ into factors of length p such that $V_1 \approx V_2 \approx \cdots \approx V_k$. The *shape* of such op-period is the common shape $shape(V_1) = \cdots = shape(V_k)$.

More generally, we say that p is a (general) *op-period* of S if p is a full op-period of a superstring of S. If the superstring is $D = X \cdot S \cdot Y$, the value $(p - |X|) \bmod p$ is called a *shift* of the op-period p; see Fig. 2.

One op-period p can have multiple shifts. Let $\mathsf{Shifts}_p \subseteq [\![0\,..\,p-1]\!]$ denote the set of all shifts of p. Note that p is an op-period of S if and only if $\mathsf{Shifts}_p \neq \emptyset$.

We consider further specific notions of periodicity in the op-model; see Fig. 3. An op-period p is called:

- *initial* if $0 \in \mathsf{Shifts}_p$,
- *sliding* if $\mathsf{Shifts}_p = [\![0\,..\,p-1]\!]$.

Initial and sliding op-periods are particular cases of block-based and sliding-window-based periods for SCER, both of which were introduced in [3].

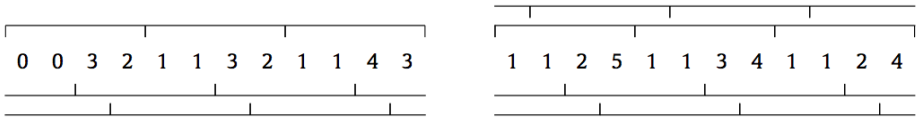

Fig. 3. The string to the left has op-period 4 with three shifts: $\mathsf{Shifts}_4 = [\![0\,..\,0]\!] \cup [\![2\,..\,3]\!]$. Due to the shift 0, the string has an initial—therefore, a full—op-period 4. The string to the right has op-period 4 with all four shifts: $\mathsf{Shifts}_4 = [\![0\,..\,3]\!]$. In particular, 4 is a sliding op-period of the string. Notice that both strings (of length $n = 12$) have (general, sliding) periods 4, but none of them has the op-border (in the sense of [4]) of length $n - 4$.

Table 1
Our algorithmic results.

Type of op-periods	Time	Space
full op-periods	$\mathcal{O}(n)$	$\mathcal{O}(n)$
the smallest initial op-period $p > 1$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
initial op-periods	$\mathcal{O}(n \log\log n)$	$\mathcal{O}(n)$
sliding op-periods	$\mathcal{O}(n \log\log n)$ expected or $\mathcal{O}(n \log^2 \log n / \log\log\log n)$ worst-case	$\mathcal{O}(n)$
general op-periods with all shifts compactly represented	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$

Models of periodicity. In the standard model, a string S of length n has a period p if and only if $S[i] = S[i + p]$ for all $i = 1, \ldots, n - p$. The famous periodicity lemma of Fine and Wilf [5] states that a "long enough" string with periods p and q has also the period $\gcd(p, q)$. The exact bound of being "long enough" is $p + q - \gcd(p, q)$. This result was generalized to arbitrary number of periods [6–8].

Periods were also considered in a number of non-standard models. *Partial words*, which are strings with don't care symbols, possess quite interesting Fine–Wilf type properties, including probabilistic ones; see [9–14]. In Section 2, we make use of periodicity graphs introduced for study of periodicity of partial words in [12,13]. In the *abelian (jumbled)* model, a version of the periodicity lemma was shown in [15] and extended in [16]. Also, algorithms for computing three types of periods analogous to full, initial, and general op-periods were designed [17–21]. In the computation of full and initial op-periods we use some number-theoretic tools initially developed in [20]. Remarkably, the fastest known algorithm for computing general periods in the abelian model has essentially quadratic time complexity [17,21], whereas for the general op-periods we design a much more efficient solution. A version of the periodicity lemma for the *parameterized* model was proposed in [22].

Op-periods were first considered in [3] where initial and sliding op-periods were introduced and direct generalizations of the Fine–Wilf property to these kinds of op-periods were developed. A few distinctions between the op-periods and periods in other models should be mentioned. First, "to have period 1" becomes a trivial property in the op-model. Second, all standard periods of a string have the "sliding" property; the first string in Fig. 3 demonstrates that this is not true for op-periods. The last distinction concerns borders. A standard period p in a string S of length n corresponds to a *border* of S of length $n - p$, which is both a prefix and a suffix of S. In the order-preserving setting, an analogue of a border is an *op-border*, that is, a prefix that is equivalent to the suffix of the same length. Op-borders have properties similar to standard borders and can be computed in $\mathcal{O}(n)$ time [4]. However, it is no longer the case that a (general, initial, full, or sliding) op-period must correspond to an op-border; see [3].

Previous algorithmic study of the op-model. Order-equivalence was introduced in [2,4]. (However, note the related combinatorial studies, originated in [23], on containment/avoidance of shapes of consecutive elements in permutations.) Both [2,4] studied pattern matching in the op-model (*op-pattern matching*) that consists in identifying all consecutive factors of a text that are order-equivalent to a given pattern. We assume that the alphabet is integer and, as usual, that it is polynomially bounded with respect to the length of the string, which means that a string can be sorted in linear time (cf. [24]). Under this assumption, for a text of length n and a pattern of length m, [2] solve the op-pattern matching problem in $\mathcal{O}(n + m \log m)$ time and [4] solve it in $\mathcal{O}(n + m)$ time. Other op-pattern matching algorithms were presented in [25,26].

An *index* for op-pattern matching based on the suffix tree was developed in [2]. For a text of length n it uses $\mathcal{O}(n)$ space and answers op-pattern matching queries for a pattern of length m in optimal, $\mathcal{O}(m)$ time (or $\mathcal{O}(m + Occ)$ time if we are to report all Occ occurrences). The index can be constructed in $\mathcal{O}(n \log\log n)$ expected time or $\mathcal{O}(n \log^2 \log n / \log\log\log n)$ worst-case time. We use the index itself and some of its applications from [2].

Other developments in this area include a multiple-pattern matching algorithm for the op-model [2], an approximate version of op-pattern matching [27], compressed index constructions [28,29], a small-space index for op-pattern matching that supports only short queries [30], and a number of practical approaches [31–35].

Our results. Table 1 lists the complexities of the algorithms that are developed in this paper. In the case of computing general op-periods, the output is the family of sets Shifts_p represented as unions of disjoint intervals. The total number of intervals, over all p, is $\mathcal{O}(n \log n)$.

In the combinatorial part, we characterize the Fine–Wilf periodicity property (aka interaction property) in the op-model in the case of coprime periods. This result is at the core of the linear-time algorithm for the smallest initial op-period.

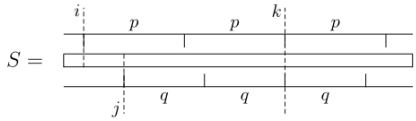

Fig. 4. Op-periods (p, i) and (q, j) synchronized at position k.

Structure of the paper. Combinatorial part of our study is given in Section 2. Then in Section 3 we recall known algorithms and data structures for the op-model and develop further algorithmic tools. The remaining sections are devoted to computation of the respective types of op-periods: all (general) op-periods in Section 4, full and initial op-periods in Section 5, the smallest non-trivial initial op-period in Section 6, and sliding op-periods in Section 7.

2. Fine–Wilf property for op-periods

The following result was shown in [3, Theorem 2]. Note that if p and q are coprime, then the conclusion is void, as every string has the op-period 1.

Theorem 2.1 *([3]). Let $p > q > 1$ and $d = \gcd(p, q)$. If a string S of length $n \geq p + q - d$ has initial op-periods p and q, it has initial op-period d. Moreover, if S has length $n \geq p + q - 1$ and sliding op-periods p and q, it has sliding op-period d.*

However, the case of coprime periods is crucial for the Fine–Wilf property and all its variants and generalizations. The aim of this section is to cover this case for op-periods.

2.1. Notation and observations

For a string S of length n, by $S[i]$ (for $1 \leq i \leq n$) we denote the ith letter of S and by $S[i\,..\,j]$ we denote a *factor* of S equal to $S[i] \dots S[j]$. If $i > j$, $S[i\,..\,j]$ denotes the empty string ε. Whenever a string S has op-period p with shift i we write "S has op-period (p, i)" for short.

We call a string *monotone* if it is either strictly increasing, or strictly decreasing, or constant. A *monotone op-period* of S is an op-period with a monotone shape. Clearly, any divisor of a monotone op-period is a monotone op-period as well.

Below we assume that $n > p > q > 1$. Let a string $S = S[1\,..\,n]$ have op-periods (p, i) and (q, j). If there exists a number $k \in [\![1\,..\,n-1]\!]$ such that $k \bmod p = i$ and $k \bmod q = j$, we say that these op-periods are *synchronized* and k is a *synchronization point* (see Fig. 4).

The proof of Theorem 2.1 given in [3] can be easily adapted to obtain the following.

Theorem 2.2. *Let $p > q > 1$ and $d = \gcd(p, q)$. If op-periods p and q of a string S of length $n \geq p + q - 1$ are synchronized, then S has op-period d, synchronized with them.*

2.2. Periodicity Theorem For Coprime Periods

For a string S, by $trace(S)$ we denote a string Y of length $|S| - 1$ over the alphabet $\{+, 0, -\}$ such that:

$$Y[i] = \begin{cases} + & \text{if } S[i] < S[i+1] \\ 0 & \text{if } S[i] = S[i+1] \\ - & \text{if } S[i] > S[i+1]. \end{cases}$$

Observation 2.3.

(1) *A string is strictly monotone if and only if its trace is a unary string.*
(2) *If S has an op-period (p, i), then $trace(S)$ "almost" has a period p, namely, $trace(S)[j] = trace(S)[k]$ for any $j, k \in [\![1\,..\,n-1]\!]$ such that $j = k \pmod p$ and $j \neq i \pmod p$. (This is because both $trace(S)[j]$ and $trace(S)[k]$ equal the sign of the difference between the same positions of the shape of the op-period of S.)*

Example 2.4. Consider the string 7 5 8 1 4 6 2 4 5. It has an op-period $(3, 1)$ with shape 2 3 1. The trace of this string is:

− + − + + − + +

The positions giving the remainder 1 modulo 3 are shown in gray; the sequence of the remaining positions is periodic.

To study traces of strings with two op-periods, we use *periodicity graphs* (see Fig. 5) very similar to those introduced in [12,13] for the study of partial words with two periods. The periodicity graph $G(n, p, i, q, j)$ represents all strings S of length

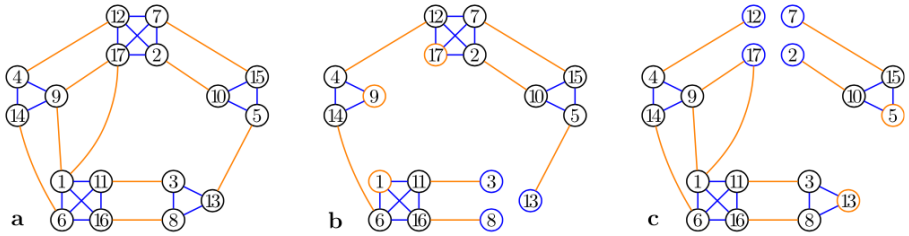

Fig. 5. Examples of periodicity graph: (a) draft graph $H(17, 8, 5)$; (b) periodicity graph $G(17, 8, 1, 5, 3)$; (c) periodicity graph $G(17, 8, 5, 5, 2)$. Orange (blue) are p-edges (resp., q-edges) and the vertices equal to i modulo p (resp., to j modulo q). (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

$n+1$ having the op-periods (p, i) and (q, j). Its vertex set $[\![1\,..\,n]\!]$ is the set of positions of the trace $trace(S)$. Two positions are connected by an edge if and only if they contain equal symbols according to Observation 2.3(2). For convenience, we distinguish between p- and q-edges, connecting positions in the same residue class modulo p (resp., modulo q). The construction of $G(n, p, i, q, j)$ is split in two steps: first we build a *draft graph* $H(n, p, q)$ (see Fig. 5,a), containing all p- and q-edges for each residue class, and then delete all edges of the clique corresponding to the ith class modulo p and all edges of the clique corresponding to the jth class modulo q (see Fig. 5,b,c). If some vertices k, l belong to the same connected component of $G = G(n, p, i, q, j)$, then $trace(S)[k] = trace(S)[l]$ for every string S represented by G. In particular, if G is connected, then $trace(S)$ is unary and S is strictly monotone by Observation 2.3(1).

Example 2.5. The graph $G(17, 8, 1, 5, 3)$ in Fig. 5,b is connected, so all strings having this graph are strictly monotone. On the other hand, some strings with the graph $G(17, 8, 5, 5, 2)$ in Fig. 5,c have no monotonicity properties. Thus, the string 6 18 2 15 17 3 16 1 5 14 4 7 8 10 13 9 11 12 of length 18 indeed has the op-period 8 with shift 5 (and shape 2 8 1 4 7 3 5 6) and the op-period 5 with shift 2 (and shape 1 3 5 2 4).

It turns out that the existence of two coprime op-periods forces every string longer than some bound to have a nontrivial monotone op-period.

Theorem 2.6. *Let S be a string of length n that has op-periods (p, i) and (q, j) such that $\gcd(p, q) = 1$ and $n > p > q > 1$. Then:*

(1) *(q, j) is a monotone op-period of S if one of the following is true:*
  (a) *$n > p+q$ and the op-periods are synchronized;*
  (b) *$n > p+q$ and the op-period p is initial;*
  (c) *$n > p+2q$;*
(2) *moreover, S has a monotone op-period $\min(pq, n)$ if one of the following is true:*
  (a) *$n > 2p$ and the op-periods are synchronized;*
  (b) *$n > \max\{2p-q, p+q\}$, the op-periods are not synchronized, and p is initial;*
  (c) *$n > \max\{2p-q, p+2q\}$ and the op-periods are not synchronized.*

Proof. Take a string S of length n having op-periods (p, i) and (q, j). Let $n' = n - 1$. Consider the draft graph $H(n', p, q)$ (see Fig. 5,a for an example). It consists of q q-cliques (numbered from 0 to $q - 1$ by residue classes modulo q) connected by some p-edges. If $n' = p + q$, there are exactly q p-edges, which connect q-cliques in a cycle due to coprimality of p and q. Thus we have a cyclic order on q-cliques: for the clique k, the next one is $(k+p) \bmod q$. The number of p-edges connecting neighboring cliques increases with the number of vertices: if $n' \geq 2p$, every vertex has an adjacent p-edge, and if $n' \geq p + 2q$, every q-clique is connected to the next q-clique by at least two p-edges.

To obtain the periodicity graph $G(n', p, i, q, j)$, one should delete all edges of the ith p-clique and the jth q-clique from $H(n', p, q)$. First consider the effect of deleting p-edges. If the ith p-clique has at least three vertices, then after the deletion each q-clique will still be connected to the next one. Indeed, if we delete edges between i, $i+p$, and $i+2p$, then there are still the edges $(i+q, i+p+q)$ and $(i+p-q, i+2p-q)$, connecting the corresponding q-cliques. If the p-clique has a single edge, its deletion will break the connection between two neighboring q-cliques if they were connected by a single edge. This is not the case if $n' \geq p+2q$, but may happen for any smaller n'; see Fig. 5,c, where $n' = p+2q-1$.

Now look at the effect of only deleting q-edges from $H(n', p, q)$. If all vertices in the jth q-clique have p-edges (this holds for any j if $n' \geq 2p$), the graph after deletion remains connected; if not, it consists of a big connected component and one or more isolated vertices from the jth q-clique.

Finally we consider the cumulative effect of deleting p- and q-edges. Note that any synchronization point becomes an isolated vertex. In total, there are two ways of making the draft graph disconnected (Fig. 6); let us analyze both.

Way 1 is to break the connection between neighboring q-cliques distinct from the deleted q-clique (Fig. 6,a). If it works, S can have no nontrivial monotone op-periods as in Example 2.5. Way 1 does not work in two situations. First, if the deleted p-edge was adjacent to the deleted q-clique; this means that the op-periods p and q are synchronized. Second, if a p-edge

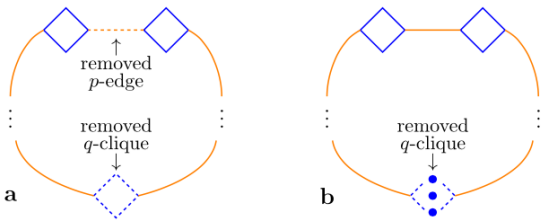

Fig. 6. Disconnecting a draft graph: (a) Way 1: deleting the only edge between neighboring q-cliques distinct from the deleted q-clique; (b) Way 2: getting isolated vertices in the deleted q-clique.

is deleted between q-cliques having more than one connecting p-edge. This happens, in particular, if $n' \geq p + 2q$ (any pair of neighboring q-cliques is connected by at least two edges) or if p is initial (this means $i = 0$, and the q-cliques ($p \mod q$) and ($2p \mod q$) are connected by the edge $(p-q, 2p-q)$ which remains undeleted).

If Way 1 fails, Way 2 is to get isolated vertices in the deleted q-clique (Fig. 6,b). Note that in this case all non-isolated vertices of periodicity graph are connected. Hence all positions of the trace $trace(S)$, except for the isolated ones, contain the same symbol. So all factors of S involving no isolated positions are monotone in the same direction. Since all isolated vertices belong to the jth q-clique, S has monotone op-period (q, j). Way 2 always works if the op-periods of S are synchronized. As mentioned above, if $n' \geq 2p$, then each vertex has a p-edge; hence in this case only synchronization points are isolated. If $n' < 2p$, any vertex without a p-edge becomes isolated if its q-clique is deleted. Note that some clique of the draft graph contains two vertices without p-edges if an only if $n' < 2p - q$; for example, the vertices $p-q$ and p from the q-clique ($p \mod q$) have no p-edges.

We have $n' \geq p + q$ for all statements of the theorem, hence the draft graph contains a cycle of q-cliques. From the previous paragraph we see that S has the monotone period (q, j) if the periodicity graph $G(n', p, i, q, j)$ was not obtained from the draft graph $H(n', p, q)$ by Way 1. But from the analysis of Way 1 we get that it fails if any of the conditions (1a)–(1c) holds. Hence statement (1) is proved.

If all isolated vertices of $G(n', p, i, q, j)$ are synchronization points (in particular, if $n' \geq 2p$), then S has monotone period pq, because two successive synchronization points of S are at distance pq due to coprimality of p and q. This justifies condition (2a). Finally, in the case of non-synchronized op-periods the condition $n' \geq 2p - q$ ensures that $G(n', p, i, q, j)$ has at most one isolated vertex; then all integers are monotone periods of S. Since conditions (2b), (2c) are combinations of the condition $n' \geq 2p - q$ with (1b), (1c), respectively, we obtain statement (2). The theorem is proved. □

Remark 2.7. From the proof of Theorem 2.6 it is clear that all obtained lower bounds for n are sharp. The following table gives the list of examples showing that the statements of Theorem 2.6 no longer hold if we decrease these lower bounds by 1.

p	q	n	Condition	Example string	Property
(8,4)	(5,4)	$p+q$	synchronized periods	1 3 2 4 6 5 8 7 9 11 10 13 12	no monotone periods
(8,0)	(5,4)	$p+q$	p is initial	1 3 2 9 5 4 7 6 8 11 10 13 12	no monotone periods
(8,5)	(5,2)	$p+2q$	non-synchronized periods	6 18 2 15 17 3 16 1 5 14 4 7 8 10 13 9 11 12	no monotone periods
(8,3)	(5,3)	$2p$	synchronized periods	14 15 16 9 10 11 12 13 1 2 3 4 5 6 7 8	not monotone
(10,0)	(3,1)	$2p-q$	p is initial non-synchronized periods	1 2 3 4 5 16 17 13 14 15 6 7 8 9 10 11 12	not monotone

3. Algorithmic toolbox

Let us start by recalling the *encoding for op-pattern matching* (*op-encoding*) from [2,4]. For a string S of length n and $i \in [\![1 \,..\, n]\!]$ we define:

$$\alpha_i(S) \text{ as the largest } j < i \text{ such that } S[j] = \max\{S[k] \,:\, k < i,\ S[k] \leq S[i]\}.$$

If there is no such j, then $\alpha_i(S) = 0$. Similarly, we define:

$$\beta_i(S) \text{ as the largest } j < i \text{ such that } S[j] = \min\{S[k] \,:\, k < i,\ S[k] \geq S[i]\},$$

and $\beta_i(S) = 0$ if no such j exists. Then $(\alpha_1(S), \beta_1(S)), \ldots, (\alpha_n(S), \beta_n(S))$ is the op-encoding of S.

Example 3.1. Consider the string $S = 7\ 5\ 8\ 1\ 4\ 6\ 2\ 4\ 5$. It has the following op-encoding:

$$(0, 0), (0, 1), (1, 0), (0, 2), (4, 2), (2, 1), (4, 5), (5, 5), (2, 2)$$

In particular $(\alpha_5(S), \beta_5(S)) = (4, 2)$ because:

$$j = 4 \text{ is the largest } j < i \text{ such that } S[j] = \max\{S[k] \,:\, k < 5,\ S[k] \le S[5] = 4\} = 1$$

and

$$j = 2 \text{ is the largest } j < i \text{ such that } S[j] = \min\{S[k] \,:\, k < 5,\ S[k] \ge S[5] = 4\} = 5.$$

Op-encodings can be computed efficiently as mentioned in the following lemma.

Lemma 3.2 *([4]). The op-encoding of a string of length n over an integer alphabet can be computed in $\mathcal{O}(n)$ time.*

The op-encoding can be used to efficiently extend a match.

Lemma 3.3. *Let X and Y be two strings of length n and assume that the op-encoding of X is known. If $X[1\,..\,n-1] \approx Y[1\,..\,n-1]$, one can check if $X \approx Y$ in $\mathcal{O}(1)$ time.*

Proof. Let $i = \alpha_n(X)$ and $j = \beta_n(X)$. Lemma 3 from [2] asserts that, if $i \neq j$, then

$$X \approx Y \iff Y[i] < Y[n] < Y[j],$$

and otherwise,

$$X \approx Y \iff Y[i] = Y[n] = Y[j].$$

(Conditions involving $Y[i]$ or $Y[j]$ when $i = 0$ or $j = 0$ should be omitted.) □

3.1. op-PREF *table*

For a string S of length n, we introduce a table op-PREF$[1\,..\,n]$ such that op-PREF$[i]$ is the length of the longest prefix of $S[i\,..\,n]$ that is equivalent to a prefix of S. It is a direct analogue of the PREF array used in standard string matching (see [36]) and can be computed similarly in $\mathcal{O}(n)$ time using one of the standard encodings for the op-model that were used in [26,2,4]; see lemma below.

Lemma 3.4. *For a string of length n, the* op-PREF *table can be computed in $\mathcal{O}(n)$ time.*

Proof. Let S be a string of length n. The standard linear-time algorithm for computing the PREF table for S (see, e.g., [36]) uses the following two properties of the table:

1. Assume that PREF$[i] = k$ and $i < j < i + k$. Let $j' = j - i + 1$. If $j' + \text{PREF}[j'] \le k$, then PREF$[j] = $ PREF$[j']$.
2. If we know that PREF$[i] \ge k$, then PREF$[i]$ can be computed in $\mathcal{O}(\text{PREF}[i] - k)$ time by extending the common prefix character by character.

In the case of the op-PREF table, the first of these properties extends without alterations due to the transitivity of the $\approx$ relation. As for the second property, the matching prefix $S[1\,..\,k] \approx S[i\,..\,i+k-1]$ can be extended character by character using Lemma 3.3 provided that the op-encoding for S is known. The op-encoding can be computed in advance using Lemma 3.2. □

We also use a symmetric array op-PREF^R such that op-PREF$^R[i]$ is the length of the longest suffix of $S[1\,..\,i]$ that is equivalent to a suffix of S. It can be computed in the same way as op-PREF.

Example 3.5. Consider the string $S = 7\ 5\ 8\ 1\ 4\ 6\ 2\ 4\ 5$. The following table contains op-PREF and op-PREF^R arrays:

i	1	2	3	4	5	6	7	8	9
$S[i]$	7	5	8	1	4	6	2	4	5
op-PREF$[i]$		1	2	1	1	2	1	1	1
op-PREF$^R[i]$	1	1	2	1	2	4	1	2	

Let us mention an application of the op-PREF table that is used further in the algorithms. We denote by op-LPP$_p(S)$ ("longest op-periodic prefix") the length of the longest prefix of a string S having p as an initial op-period.

Lemma 3.6. *For a string S of length n, op-LPP$_p(S)$ for a given p can be computed in $\mathcal{O}(\text{op-LPP}_p(S)/p+1)$ time after $\mathcal{O}(n)$-time preprocessing.*

Proof. We start by computing the op-PREF table for S in $\mathcal{O}(n)$ time. We assume that op-PREF$[n+1]=0$. To compute op-LPP$_p(S)$, we iterate over positions $i=p+1, 2p+1, \ldots$ and for each of them check if op-PREF$[i]\geq p$. If i_0 is the first position for which this condition is not satisfied (possibly because $i_0 > n-p+1$), we have op-LPP$_p(S) = i_0 + \text{op-PREF}[i_0]-1$. Clearly, this procedure works in the desired time complexity. □

Remark 3.7. Note that it can be the case that op-LPP$_p(S) \neq p + \text{op-PREF}[p+1]$. See, e.g., the strings in Fig. 3 and $p=4$.

Example 3.8. Consider the string $S = 0\ 0\ 3\ 2\ 1\ 1\ 3\ 2\ 1\ 1\ 4$. The following table contains op-PREF and op-LPP$_p$:

p	1	2	3	4	5	6	7	8	9	10	11
$S[p]$	0	0	3	2	1	1	3	2	1	1	4
op-PREF$[p]$		1	1	1	4	1	1	1	3	1	1
op-LPP$_p(S)$	11	3	4	11	6	7	8	11	10	11	11

In particular, op-LPP$_4(S)=11$ since op-PREF$[5]=4$ and op-PREF$[9]=3$.

3.2. Longest common extension queries for op-model

For a string S, we define a *longest common extension* query op-LCP(i,j) in the order-preserving model as the maximum $k\geq 0$ such that $S[i\,..\,i+k-1] \approx S[j\,..\,j+k-1]$.

Symmetrically, op-LCP$^{\mathrm{R}}(i,j)$ is the maximum $k\geq 0$ such that $S[i-k+1\,..\,i] \approx S[j-k+1\,..\,j]$.

Similarly as in the standard model [37], LCP-queries in the op-model can be answered using lowest common ancestor (LCA) queries in the op-suffix tree, as shown in the following lemma. It also applies to op-LCP$^{\mathrm{R}}$-queries since the construction can be used for the reversed string.

Lemma 3.9. *After preprocessing in $\mathcal{O}(n\log\log n)$ expected time or in $\mathcal{O}(n\log^2\log n/\log\log\log n)$ worst-case time one can answer* op-LCP*-queries in a string of length n in $\mathcal{O}(1)$ time.*

Proof. The *order-preserving suffix tree* (*op-suffix tree*) that is constructed in [2] is a compacted trie of op-encodings of all the suffixes of the text. In $\mathcal{O}(n\log\log n)$ expected time or $\mathcal{O}(n\log^2\log n/\log\log\log n)$ worst-case time one can construct a so-called incomplete version of the op-suffix tree in which each explicit node may have at most one edge whose first character label is not known. Fortunately, for op-LCP-queries the labels of the edges are not needed; the only required information is the depth of each explicit node and the location of each suffix. Therefore, for this purpose the incomplete op-suffix tree can be treated as a regular suffix tree and preprocessed using standard lowest common ancestor data structure that requires additional $\mathcal{O}(n)$ preprocessing and can answer queries in $\mathcal{O}(1)$ time [38]. □

3.3. op-squares

The factor $S[i\,..\,i+2p-1]$ is called an order-preserving square (*op-square*) if $S[i\,..\,i+p-1] \approx S[i+p\,..\,i+2p-1]$. For a string S of length n, we define the set

$$\text{op-Squares}_p = \{i \in [\![1\,..\,n-2p+1]\!] : S[i\,..\,i+2p-1] \text{ is an op-square}\}.$$

Op-squares were first defined in [2] where an algorithm computing all the sets op-Squares$_p$ for a string of length n in $\mathcal{O}(n\log n + \sum_p |\text{op-Squares}_p|)$ time was shown.

We say that an op-square $S[i\,..\,i+2p-1]$ is *right shiftable* if $S[i+1\,..\,i+2p]$ is an op-square and *right non-shiftable* otherwise. Similarly, we say that the op-square is *left shiftable* if $S[i-1\,..\,i+2p-2]$ is an op-square and *left non-shiftable* otherwise.

Using the approach of [2], one can show the following lemma.

Lemma 3.10. *All the (left and right) non-shiftable op-squares in a string of length n can be computed in $\mathcal{O}(n\log n)$ time.*

Proof. We show the algorithm for right non-shiftable op-squares; the computations for left non-shiftable op-squares are symmetric.

Let S be a string of length n. An op-square $S[i\,..\,i+2p-1]$ is called *right non-extendible* if $i+2p-1=n$ or $S[i\,..\,i+p] \not\approx S[i+p\,..\,i+2p]$. We use the following claim.

Claim 3.11 *(See Lemma 18 in [2]). All the right non-extendible op-squares in a string of length n can be computed in $\mathcal{O}(n \log n)$ time.*

Note that a right non-shiftable op-square is also right non-extendible, but the converse is not necessarily true. Thus it suffices to filter out the op-squares that are right shiftable. For this, for a right non-extendible op-square $S[i\,..\,i+2p-1]$ we need to check if op-LCP$(i+1, i+p+1) < p$. This condition can be verified in $\mathcal{O}(1)$ time after $o(n \log n)$-time preprocessing using Lemma 3.9. □

Example 3.12. Consider the string 7 5 8 1 4 6 2 4 5. It contains following op-squares of length greater than two:

$$\begin{array}{ll} \text{op-Squares}_2 = \{1, 2, 5\} & 7\,5\,8\,1,\ 5\,8\,1\,4,\ 4\,6\,2\,4 \\ \text{op-Squares}_3 = \{2, 3\} & 5\,8\,1\,4\,6\,2,\ 8\,1\,4\,6\,2\,4 \end{array}$$

In particular:

- op-square 5 8 1 4 6 2 is right-shiftable but it is right non-extendible,
- op-square 8 1 4 6 2 4 is right non-shiftable.

3.4. Interval representations

An *interval representation* of a set X of integers is

$$X = [\![i_1\,..\,j_1]\!] \cup [\![i_2\,..\,j_2]\!] \cup \cdots \cup [\![i_k\,..\,j_k]\!],$$

where $j_1 + 1 < i_2, \ldots, j_{k-1} + 1 < i_k$; k is called the *size* of the representation.

For an integer set X, by X mod p we denote the set $\{x \bmod p : x \in X\}$. The following technical lemma provides efficient operations on interval representations of sets.

Lemma 3.13.

(a) *Assume that X and Y are two sets with interval representations of sizes x and y, respectively. Then the interval representation of the set $X \cap Y$ can be computed in $\mathcal{O}(x + y)$ time.*
(b) *Assume that $X_1, \ldots, X_k \subseteq [\![0\,..\,n]\!]$ are non-empty sets with interval representations of sizes $x_1, \ldots, x_k$ and $p_1, \ldots, p_k$ be positive integers. The interval representations of the sets $X_1 \bmod p_1, \ldots, X_k \bmod p_k$ can then be computed in $\mathcal{O}(x_1 + \cdots + x_k + n)$ time.*

Proof. To compute $X \cap Y$ in point (a), it suffices to merge the lists of endpoints of intervals in the interval representations of X and Y. Let L be the merged list. With each element of L we store a weight $+1$ if it represents the beginning of an interval and a weight -1 if it represents the endpoint of an interval. We compute the prefix sums of these weights for L. Then, by considering all elements with a prefix sum equal to 2 and their following elements in L, we can restore the interval representation of $X \cap Y$.

Let us proceed to point (b). Note that, for an interval $[\![i\,..\,j]\!]$, the set $[\![i\,..\,j]\!] \bmod p$ either equals $[\![0\,..\,p-1]\!]$ if $j - i \geq p$, or otherwise is a sum of at most two intervals. For each interval $[\![i\,..\,j]\!]$ in the representation of X_a, for $a = 1, \ldots, k$, we compute the interval representation of $[\![i\,..\,j]\!] \bmod p_a$. Now it suffices to compute the sum of these intervals for each X_a. This can be done exactly as in point (a) provided that the endpoints of the intervals comprising representations of $[\![i\,..\,j]\!] \bmod p_a$ are sorted. We perform the sorting simultaneously for all X_a using bucket sort [24]. The total number of endpoints is $\mathcal{O}(x_1 + \cdots + x_k)$ and the number of possible values of endpoints is at most n. This yields the desired time complexity of point (b). □

4. Computing all op-periods

Our goal is to compute a compact representation of all the op-periods of a string that contains, for each op-period p, the interval representation of the set Shifts_p. The procedure relies on the following characterization of shifts; see also Fig. 7.

Fact 4.1. *Let $1 \leq p \leq n$, $s \in [\![0\,..\,p-1]\!]$, and $t = n - (n - s) \bmod p$. We have $s \in \mathsf{Shifts}_p$ if and only if the following three conditions hold:*

(1) *p is a full op-period of $S[s+1\,..\,t]$,*
(2) *$S[1\,..\,\min(n-p, s)] \approx S[p+1\,..\,\min(n, p+s)]$, and*
(3) *$S[1 + \max(0, t-p)\,..\,n-p] \approx S[1 + \max(p, t)\,..\,n]$.*

In particular, if $s < t$, the last two conditions are:

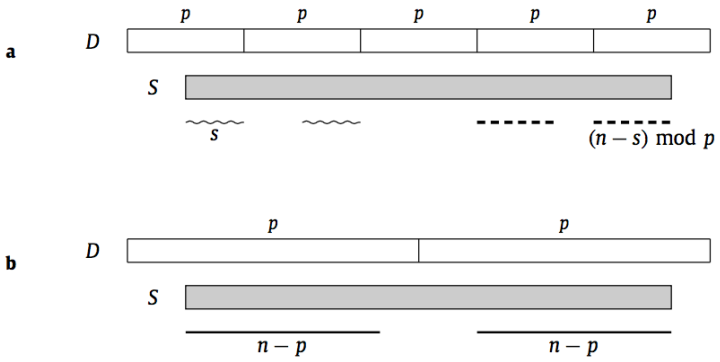

Fig. 7. Let $s \in [\![0\,..\,p-1]\!]$ and $t = n - (n-s) \bmod p$. Part a represents the case that $s < t$ whereas part b the case that $s = t$. Lines of the same type denote order-equivalent strings.

(2) $S[1\,..\,s] \approx S[p+1\,..\,p+s]$,
(3) $S[t+1-p\,..\,n-p] \approx S[t+1\,..\,n]$,

and otherwise they both become:

(2,3) $S[1\,..\,n-p] \approx S[p+1\,..\,n]$.

Proof. If $s \in \mathsf{Shifts}_p$, then p is a full op-period of a superstring $D = X \cdot S \cdot Y$ with $s = (p - |X|) \bmod p$. Without loss of generality, we may assume that $1 \le |X|, |Y| \le p$; otherwise, we can append or remove the leading or trailing p symbols from D. Consequently, $D = V_1 \cdot V_2 \cdots V_k$, where V_i are order-equivalent factors of length p and $S[s+1\,..\,t] = V_2 \cdots V_{k-1}$. Hence, p is indeed a full op-period of $S[s+1\,..\,t]$. Furthermore, $V_1 \approx V_2$ implies

$$S[1\,..\,s] \approx S[p+1\,..\,p+s] \quad \text{if } p+s \le n, \qquad S[1\,..\,n-p] \approx S[p+1\,..\,n] \quad \text{otherwise.}$$

Similarly, $V_{k-1} \approx V_k$ implies

$$S[t-p+1\,..\,n-p] \approx S[t+1\,..\,n] \quad \text{if } t \ge p, \qquad S[1\,..\,n-p] \approx S[p+1\,..\,n] \quad \text{otherwise.}$$

Thus, s satisfies (1)–(3).

For a proof in the other direction, we consider two cases. First, suppose that $s < t$ so that $S[s+1\,..\,t]$ is non-empty. By (1), p is a full op-period of this string, which yields a decomposition $S[s+1\,..\,t]$ into factors of length p with common shape W. Condition (2) further yields $S[1\,..\,s] \approx S[s+1\,..\,s+p]$, i.e., $S[1\,..\,s]$ is order-equivalent to a suffix of W. Hence, we can extend $S[1\,..\,s]$ to the left so that $W = \mathit{shape}(X \cdot S[1\,..\,s])$. Symmetrically, (2) implies $S[t-p+1\,..\,n-p] \approx S[t+1\,..\,n]$, which means that we can extend $S[t+1\,..\,n]$ to the right so that $W = \mathit{shape}(S[t+1\,..\,n] \cdot Y)$. Combining both extensions, we obtain a superstring $X \cdot S \cdot Y$ with full op-period p. Note that $s = p - |X|$, so this proves $s \in \mathsf{Shifts}_p$.

Next, suppose that $s = t$. In this case, (1) is void, while (2) and (3) both simplify to $S[1\,..\,n-p] \approx S[p+1\,..\,n]$. Consequently, we can extend S by p characters to the left and to the right so that $P \cdot S[1\,..\,n-p] \approx S \approx S[p+1\,..\,n] \cdot Q$. Any length-$2p$ factor of $P \cdot S \cdot Q$ is an op-square, and therefore a superstring of S with full op-period p. Thus, $\mathsf{Shifts}_p = [\![0\,..\,p-1]\!] \ni s$. □

We introduce the sets $\mathsf{QShifts}_p$ of *quasi-shifts*, Left_p of *left shifts*, and Right_p of *right shifts*, so that they contain $s \in [\![0\,..\,p-1]\!]$ which satisfy the conditions (1), (2), and (3), respectively. Fact 4.1 clearly yields

$$\mathsf{Shifts}_p = \mathsf{QShifts}_p \cap \mathsf{Left}_p \cap \mathsf{Right}_p.$$

Below, we show how to compute interval representations of the sets $\mathsf{QShifts}_p$, Left_p, and Right_p. As proved in the following lemma, the latter two sets have very simple interval representations, each consisting of at most two intervals.

Lemma 4.2. *For a string of length n, interval representations of the sets Left_p and Right_p for all $p \in [\![1\,..\,n]\!]$ can be computed in $\mathcal{O}(n)$ time.*

Proof. If $S[1\,..\,n-p] \approx S[p+1\,..\,n]$, then $\mathsf{Left}_p = \mathsf{Right}_p = [\![0\,..\,p-1]\!]$. The former condition can be expressed as $\text{op-PREF}[p+1] = n-p$, so it can be verified efficiently. Otherwise, we have

$$\mathsf{Left}_p = \{s \in [\![0\,..\,p-1]\!] : \text{op-PREF}[p+1] \ge s\},$$
$$\mathsf{Right}_p = \{s \in [\![0\,..\,p-1]\!] : \text{op-PREF}^R[n-p] \ge (n-s) \bmod p\}.$$

Consequently, the interval representation of Left_p consists of one interval and the interval representation of Right_p consists of up to two intervals. Moreover, they can both be computed in constant time based on the op-PREF and op-PREF^R arrays. These arrays can be computed in $\mathcal{O}(n)$ time; see Lemma 3.4. □

The interval representations of quasi-shifts are more complicated; fortunately, we can relate $\mathsf{QShifts}_p$ to the set NonSQ_p of all starting positions of length-$2p$ factors which are *not* op-squares:

$$\mathsf{NonSQ}_p = [\![1\,..\,n-2p+1]\!] \setminus \mathsf{op\text{-}Squares}_p.$$

Fact 4.3. *For each $1 \le p \le n$ and $s \in [\![0\,..\,p-1]\!]$, we have $s \notin \mathsf{QShifts}_p$ if and only if $s = (x-1) \bmod p$ for some $x \in \mathsf{NonSQ}_p$.*

Proof. Recall that $s \in \mathsf{QShifts}_p$ if and only if p is a full op-period of $S[s+1\,..\,t]$, where $t = n-(n-s) \bmod p$. We have $s = t \bmod p$, so we can always decompose $S[s+1\,..\,t]$ into factors $W_1 \cdots W_\ell$ of length p. Thus, $s \in \mathsf{QShifts}_p$ if and only if $\mathit{shape}(W_1) = \cdots = \mathit{shape}(W_\ell)$. However, $\mathit{shape}(W_i) \neq \mathit{shape}(W_{i+1})$ is equivalent to $s + (i-1)p + 1 \in \mathsf{NonSQ}_p$. □

Using properties of non-shiftable op-squares, we show that the sets NonSQ_p have efficiently computable interval representations.

Lemma 4.4. *For a string of length n, interval representations of the sets NonSQ_p for all $1 \le p \le n$ can be computed in $\mathcal{O}(n \log n)$ time.*

Proof. The set NonSQ_p is a complement of the set $\mathsf{op\text{-}Squares}_p$; hence, it is enough to describe fast computation of all sets $\mathsf{op\text{-}Squares}_p$. Let us define the following two auxiliary sets:

$$\mathcal{L}_p = \{i \in [\![1\,..\,n-2p+1]\!] : S[i\,..\,i+2p-1] \text{ is a left non-shiftable op-square}\},$$
$$\mathcal{R}_p = \{i \in [\![1\,..\,n-2p+1]\!] : S[i\,..\,i+2p-1] \text{ is a right non-shiftable op-square}\}.$$

By Lemma 3.10, all the sets $\mathcal{L}_p$ and $\mathcal{R}_p$ can be computed in $\mathcal{O}(n \log n)$ time. In particular, $\sum_p |\mathcal{L}_p| = \mathcal{O}(n \log n)$.

If we present the set $\mathsf{op\text{-}Squares}_p$, for any p, as a sum of maximal intervals, then the left and right endpoints of these intervals will be $\mathcal{L}_p$ and $\mathcal{R}_p$, respectively. Hence, $|\mathcal{L}_p| = |\mathcal{R}_p|$. Thus if

$$\mathcal{L}_p = \{\ell_1, \ldots, \ell_k\} \text{ and } \mathcal{R}_p = \{r_1, \ldots, r_k\},$$

then the interval representation of the set $\mathsf{op\text{-}Squares}_p$ is $[\![\ell_1\,..\,r_1]\!] \cup \cdots \cup [\![\ell_k\,..\,r_k]\!]$. Clearly, it can be computed in $\mathcal{O}(|\mathcal{L}_p|)$ time. □

Combining the results above, we derive our algorithm for general op-periods.

Theorem 4.5. *A representation of size $\mathcal{O}(n \log n)$ of all the op-periods of a string of length n can be computed in $\mathcal{O}(n \log n)$ time.*

Proof. Algorithm 1 implements Facts 4.1 and 4.3. Operations "mod" on sets are performed simultaneously using Lemma 3.13(b). The interval representations of sets $\mathsf{QShifts}_p$ take $\mathcal{O}(n \log n)$ space in total, while the representations of sets Left_p and Right_p are both of $\mathcal{O}(n)$ total size. This guarantees the $\mathcal{O}(n \log n)$ overall running time. □

Algorithm 1: Compute the compact representation of all op-periods.

```
1 for p := 1 to n do
2     compute NonSQ_p;                                   ▷Lemma 4.4
3     compute Left_p, Right_p;                           ▷Lemma 4.2
4 for p := 1 to n simultaneously do
5     BadShifts_p := {(x − 1) mod p : x ∈ NonSQ_p};      ▷Lemma 3.13(b)
6 for p := 1 to n do
7     QShifts_p := [[0 .. p − 1]] \ BadShifts_p;         ▷Fact 4.3
8     Shifts_p := QShifts_p ∩ Left_p ∩ Right_p;          ▷Lemma 3.13(a)
9 return (Shifts_p)_{p=1}^n;
```

5. Faster algorithms for full and initial op-periods

For a string S of length n, we define op-PREF$'[i]$ for $i = 0, \ldots, n$ as:

$$\text{op-PREF}'[i] = \begin{cases} n & \text{if op-PREF}[i+1] = n - i \\ \text{op-PREF}[i+1] & \text{otherwise.} \end{cases}$$

Here we assume that op-PREF$[n+1] = 0$. In the computation of full and initial op-periods we heavily rely on this table according to the following obvious observation.

Observation 5.1. *p is an initial op-period of a string S of length n if and only if* op-PREF$'[ip] \geq p$ *for all* $i = 1, \ldots, \lfloor n/p \rfloor$.

5.1. Computing initial op-periods

Let us introduce an auxiliary array P$[1\,..\,n]$ such that:

$$\mathsf{P}[p] = \min\{\text{op-PREF}'[ip] : i = 1, \ldots, \lfloor n/p \rfloor\}.$$

Straight from Observation 5.1 we have:

Observation 5.2. *p is an initial period of S if and only if* $\mathsf{P}[p] \geq p$.

Example 5.3. Consider the string $S = 8\ 7\ 2\ 6\ 5\ 4\ 1\ 2\ 9\ 7\ 1\ 6\ 4\ 3\ 2$ of length $n = 15$. The following table contains the arrays op-PREF$'$ and P:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$S[i+1]$	8	7	2	6	5	4	1	2	9	7	1	6	4	3	2	
op-PREF$[i+1]$		2	1	3	**4**	2	1	1	**6**	2	1	3	**3**	2	1	0
op-PREF$'[i]$		2	1	3	**4**	2	1	1	**6**	2	1	3	**15**	15	15	15
P$[i]$		1	1	1	**4**	1	1	1	6	2	1	3	15	15	15	15

We have op-PREF$'[4]$, op-PREF$'[8]$, op-PREF$'[12] \geq 4$. The numbers in boldface show the calculation of P[4]. We conclude that the initial op-periods of S are 1, 4, 12, 13, 14, and 15.

The table P could be computed straight from definition in $\mathcal{O}(n \log n)$ time. We improve this complexity to $\mathcal{O}(n \log\log n)$ by employing Eratosthenes's sieve. The sieve computes, in particular, for each $j = 1, \ldots, n$ a list of all distinct prime divisors of j. We use these divisors to compute the table via dynamic programming in a right-to-left scan, as shown in Algorithm 2.

Algorithm 2: Compute all initial op-periods of S.

```
1 P := op-PREF';
2 for j := n down to 2 do
3     foreach prime divisor q of j do
4         P[j/q] := min(P[j/q], P[j]);
5 for p := 1 to n do
6     if P[p] ≥ p then p is an initial op-period;
```

Theorem 5.4. *All initial op-periods of a string of length n can be computed in $\mathcal{O}(n \log\log n)$ time and $\mathcal{O}(n)$ space.*

Proof. By Lemma 3.4, the op-PREF table for the string—hence, the op-PREF$'$ table—can be computed in $\mathcal{O}(n)$ time. Then we use Algorithm 2. Each prime number $q \leq n$ has at most $\frac{n}{q}$ multiples below n. Therefore, the complexity of Eratosthenes's sieve the number of updates on the table P in the algorithm is $\sum_{q \in Primes, q \leq n} \frac{n}{q} = \mathcal{O}(n \log\log n)$; see [39].

To implement Algorithm 2 in $\mathcal{O}(n)$ space, we need to avoid storing all prime divisors of every number. We will store a table $L[1\,..\,n]$ of lists such that $L[j]$ will contain all prime divisors of j when j is processed in Line 2. Moreover, it will be guaranteed that the total size of lists in L is always $\mathcal{O}(n)$.

Initially every prime q is stored in the list $L[q \cdot \lfloor n/q \rfloor]$, that is, at its largest multiple in the range. When the list $L[j]$ is processed, every prime number q in the list is copied to the list $L[(j/q - 1) \cdot q]$. Afterwards, the list $L[j]$ is disposed of. Clearly, every prime number will reach the lists of all its multiples in the range in descending order and it will be present in only one list at a time. Hence, the total size of the lists does not exceed $|Primes \cap [\![1\,..\,n]\!]| = \mathcal{O}(n)$. □

5.2. Computing full op-periods

Let us recall an auxiliary data structure of [20] for efficient gcd-computations. We will only need a special case of this data structure to answer queries for $\gcd(x, n)$.

Fact 5.5 *([20, Theorem 4]). After $\mathcal{O}(n)$-time preprocessing, given any $x, y \in [\![1 \,..\, n]\!]$, the value $\gcd(x, y)$ can be computed in constant time.*

Let $Div(i)$ denote the set of all positive divisors of i. In the case of full op-periods we only need to compute $\mathsf{P}[p]$ for $p \in Div(n)$. As in Algorithm 2, we start with $\mathsf{P} = \text{op-PREF}'$. Then we perform a preprocessing phase that shifts the information stored in the array from indices $i \notin Div(n)$ to indices $\gcd(i, n) \in Div(n)$. It is based on the fact that for $d \in Div(n)$, $d \mid i$ if and only if $d \mid \gcd(i, n)$. Finally, we perform right-to-left processing as in Algorithm 2. However, this time we can afford to iterate over all divisors of elements from $Div(n)$. Thus we arrive at the pseudo-code of Algorithm 3.

Algorithm 3: Compute all full op-periods of S.

```
1 P := op-PREF';
2 for i := 1 to n do
3     k := gcd(i, n);
4     P[k] := min(P[k], P[i]);
5 foreach i ∈ Div(n) in decreasing order do
6     foreach d ∈ Div(i) do
7         P[d] := min(P[d], P[i]);
8 foreach p ∈ Div(n) do
9     if P[p] ≥ p then p is a full op-period;
```

Theorem 5.6. *All full op-periods of a string of length n can be computed in $\mathcal{O}(n)$ time.*

Proof. We apply Algorithm 3. The complexity of the **for** loop is $\mathcal{O}(n)$ by Fact 5.5. The first **foreach** loop works in $\mathcal{O}(n)$ time as the sizes of the sets $Div(n)$, $Div(i)$ are $\mathcal{O}(\sqrt{n})$ and the elements of these sets can be enumerated in $\mathcal{O}(\sqrt{n})$ time as well. The total complexity is $\mathcal{O}(n)$. □

6. Computing smallest non-trivial initial op-period

The following theorem provides a useful auxiliary routine in the computation of the smallest initial op-period that is greater than 1.

Theorem 6.1. *All monotone op-periods of a string of length n can be computed in $\mathcal{O}(n)$ time.*

Proof. We show how to compute all strictly increasing op-periods; the computation of strictly decreasing and constant op-periods is the same. Let S be a string of length n and let $X = trace(S)$ (see Section 2.2). Let $A = \{a_1, \ldots, a_k\}$ be the set of all positions $a_1 < \cdots < a_k$ in X such that $X[i] \neq +$. If $A = \varnothing$, every positive integer is a monotone sliding period of S; so let $A \neq \varnothing$. The set A provides a simple characterization of strictly increasing op-periods of S.

Observation 6.2. *(p, s) is a strictly increasing op-period of a non-monotone string S if and only if $a_i = s \pmod p$ for all $a_i \in A$.*

If $A = \{a_1\}$, then by Observation 6.2 every positive integer p is a monotone op-period of S with the shift $s = a_1 \bmod p$. From now on, we assume $|A| > 1$.

For a set of positive integers $B = \{b_1, \ldots, b_k\}$, let us denote $\gcd(B) = \gcd(b_1, \ldots, b_k)$. The claim below follows from Fact 5.5. However, we give a simpler proof.

Claim 6.3. *If $B \subseteq [\![1 \,..\, n]\!]$, then $\gcd(B)$ can be computed in $\mathcal{O}(n)$ time.*

Proof. Let $B = \{b_1, \ldots, b_k\}$ and denote $d_i = \gcd(b_1, \ldots, b_i)$. We want to compute d_k.

Note that $d_i \mid d_{i-1}$ for all $i = 2, \ldots, k$. Hence, the sequence (d_i) contains at most $\log n + 1$ distinct values.

Set $d_1 = b_1$. To compute d_i for $i \geq 2$, we check if $d_{i-1} \mid b_i$. If so, $d_i = d_{i-1}$. Otherwise $d_i = \gcd(d_{i-1}, b_i) < d_{i-1}$. Hence, we can compute d_i using Euclid's algorithm in $\mathcal{O}(\log n)$ time. The latter situation takes place at most $\log n + 1$ times; the conclusion follows. □

Consider the set $B = \{a_2 - a_1, a_3 - a_2, \ldots, a_k - a_{k-1}\}$. By Observation 6.2, (p, s) is a strictly increasing op-period of S if and only if $p \mid \gcd(B)$ and $s = a_1 \bmod p$. The value $\gcd(B)$ can be computed in $\mathcal{O}(n)$ time by Claim 6.3. After that, we find all its divisors and report the monotone op-periods in $\mathcal{O}(\sqrt{n})$ time. □

Example 6.4. Consider a string $S = 5\ 7\ 1\ 3\ 4\ 6\ 7\ 9\ 2\ 3\ 5\ 7\ 8$ of length $n = 13$ and $X = trace(S)$:

i	1	2	3	4	5	6	7	8	9	10	11	12	13
$S[i]$	5	7	1	3	4	6	7	9	2	3	5	5	8
$X[i]$	+	−	+	+	+	+	+	−	+	+	0	+	

The string S has a monotone op-period 3 with shift 2. The set

$$A = \{i \in [\![1 \,..\, n-1]\!] : X[i] \neq +\} = \{2, 8, 11\}$$

and all elements $a \in A$ satisfy $a = 2 \pmod 3$.

We proceed with the following simple property.

Lemma 6.5. *The shape of the smallest non-trivial initial op-period of a string has no shorter non-trivial full op-period.*

Proof. A full op-period of the initial op-period of a string S is an initial op-period of S. □

The following property of initial op-periods, immediate from Theorem 2.6(1), is the basis of our linear-time algorithm.

Lemma 6.6. *If a string of length n has initial op-periods $p > q > 1$ such that $p + q < n$ and $\gcd(p, q) = 1$, then q is strictly monotone.*

Algorithm 4: Compute the smallest non-trivial initial op-period of S.

```
1 if S has a non-trivial monotone op-period then
2     return smallest such op-period;
3 p := the length of the longest monotone prefix of S plus 1;
4 while p ≤ n do
5     k := op-LPP_p(S);
6     if k = n then return p;
7     p := max(p + 1, k − p);
8 return n;
```

Theorem 6.7. *The smallest initial op-period $p > 1$ of a string S of length n can be computed in $\mathcal{O}(n)$ time.*

Proof. We follow the lines of Algorithm 4. We can compute all monotone initial op-periods of S using Theorem 6.1. If non-trivial such op-periods exist, we take the smallest one; let it be $q > 1$. Then none of $2, \ldots, q-1$ is an initial op-period of S. Hence, we can safely return q.

Let us now focus on the correctness of the **while** loop. The invariant is that there is no initial op-period of S that is smaller than p. The initial value of p is safe because S has no non-trivial initial monotone op-periods. If the value of $k = \text{op-LPP}_p(S)$ equals n, then p is an initial op-period of S and we can safely return it. Otherwise, we can advance p by 1. There is also no smallest initial op-period p' such that $p < p' < k - p$. Indeed, if the string $S[1 \,..\, k]$ had initial op-periods p and p', Lemma 6.6 would imply that p is strictly monotone if $\gcd(p, p') = 1$ (which is impossible due to the initial selection of p) and Theorem 2.1 would imply an initial op-period of $S[1 \,..\, p']$ that is smaller than p' and divides p' if $\gcd(p, p') > 1$ (which is impossible due to Lemma 6.5). This justifies the way p is increased.

Now let us consider the time complexity of the algorithm. The algorithm for monotone op-periods of Theorem 6.1 works in $\mathcal{O}(n)$ time. By Lemma 3.6, k can be computed in $\mathcal{O}(k/p + 1)$ time. If $k \leq 3p$, this is $\mathcal{O}(1)$. Otherwise, p at least doubles; let p' be the new value of p. Then $\mathcal{O}(k/p + 1) = \mathcal{O}((p + p' - 1)/p + 1) = \mathcal{O}(p' + 1)$. The case that p doubles can take place at most $\mathcal{O}(\log n)$ times and the total sum of p' over such cases is $\mathcal{O}(n)$. □

7. Computing sliding op-periods

For a string S of length n, we define a family of strings $SH_1, \ldots, SH_n$ such that $SH_k[i] = shape(S[i \,..\, i + k - 1])$ for $1 \leq i \leq n - k + 1$. Sliding op-periods admit an elegant characterization based on the strings SH_k; see Fig. 8.

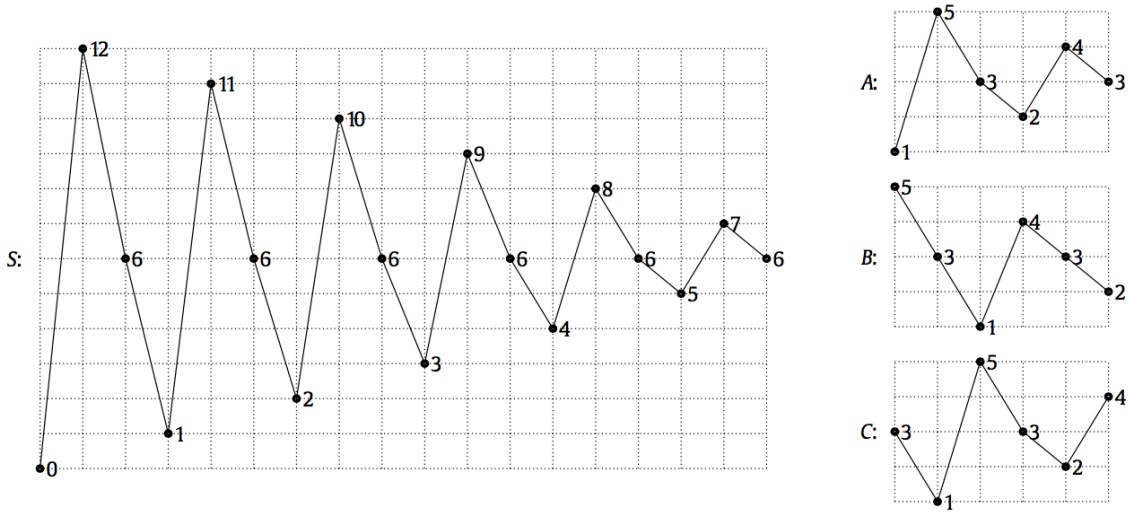

Fig. 8. A string $S = 0\ 12\ 6\ 1\ 11\ 6\ 2\ 10\ 6\ 3\ 9\ 6\ 4\ 8\ 6\ 5\ 7\ 6$ is graphically illustrated above (the ith point has coordinates $(i, S[i])$). We have $SH_6 = ABCABCABCA$, where $A = 1\ 5\ 3\ 2\ 4\ 3$, $B = 5\ 3\ 1\ 4\ 3\ 2$, and $C = 3\ 1\ 5\ 3\ 2\ 4$. Since 6 is a period of SH_6, it is a sliding op-period of S. Moreover, 3 is a period of SH_6, hence of SH_3, so it is a sliding op-period of S.

Lemma 7.1. *An integer p, $1 \le p \le n$, is a sliding op-period of S if and only if*

(1) $p \le \frac{1}{2}n$ *and p is a period of SH_p, or*
(2) $p > \frac{1}{2}n$ *and $S[1\,..\,n-p] \approx S[p+1\,..\,n]$.*

Proof. First, we prove that if $p \le \frac{1}{2}n$ is a sliding op-period of S, then p is a period of SH_p. For this, we need to show that $SH_p[i] = SH_p[i+p]$ for an arbitrary position i with $1 \le i \le n-2p+1$. Note that $(i-1) \bmod p \in \mathsf{QShifts}_p = \mathsf{Shifts}_p$ by Fact 4.1(1). Hence, Fact 4.3 implies $i \in \text{op-}\mathsf{Squares}_p$ and $SH_p[i] = SH_p[i+p]$.

For a proof in the other direction, suppose that $p \le \frac{1}{2}n$ is a period of SH_p. Equivalently, $\text{op-}\mathsf{Squares}_p = [\![1\,..\,n-2p+1]\!]$ and $\mathsf{NonSQ}_p = \emptyset$, so $\mathsf{QShifts}_p = [\![0\,..\,p-1]\!]$. Additionally, observe that $S[1\,..\,p] \approx S[p+1\,..\,2p]$, so $S[1\,..\,s] \approx S[p+1\,..\,p+s]$ for every $s \in [\![0\,..\,p-1]\!]$. Consequently, $\mathsf{Left}_p = [\![0\,..\,p-1]\!]$. Symmetrically, $S[n-2p+1\,..\,n-p] \approx S[n-p+1\,..\,n]$ implies $\mathsf{Right}_p = [\![0\,..\,p-1]\!]$. Thus, Fact 4.1 yields $\mathsf{Shifts}_p = [\![0\,..\,p-1]\!]$ and that p is a sliding op-period of S.

Next, we consider the case of $p > \frac{1}{2}n$. If p is a sliding op-period of S, then Fact 4.1(2) yields $p-1 \in \mathsf{Left}_p = [\![0\,..\,p-1]\!]$. Due to $p-1 \ge n-p$, this implies $S[1\,..\,n-p] \approx S[p+1\,..\,n]$. For the converse implication, we observe that $S[1\,..\,n-p] \approx S[p+1\,..\,n]$ immediately yields $\mathsf{Left}_p = \mathsf{Right}_p = [\![0\,..\,p-1]\!]$. Moreover, $\mathsf{QShifts}_p = [\![0\,..\,p-1]\!]$ holds unconditionally for $p > \frac{1}{2}n$. Hence, p is a sliding op-period of S by Fact 4.1. □

Note that the characters of the strings SH_k are shapes, but we can interpret them as abstract characters. This lets us switch from *order-preserving stringology* to *standard stringology*—from op-strings to classic strings. The total length of strings SH_k is quadratic in n, so we will not compute those strings explicitly. Instead, we use the following abstraction.

Definition 7.2. We say that strings $W_1, W_2, \ldots, W_n$ form a *special family* if they satisfy the following conditions:

(a) $|W_t| = n-t+1$ for $1 \le t \le n$;
(b) if $t \ge 1$ and $1 \le i, j \le |W_t|$, then $W_t[i] = W_t[j]$ implies $W_{t-1}[i\,..\,i+1] = W_{t-1}[j\,..\,j+1]$;
(c) for each $1 \le i, j \le n$, in constant time one can evaluate

$$\mathsf{max\text{-}eq}(i, j) := \max\{t : i, j \le |W_t| \text{ and } W_t[i] = W_t[j]\}$$

and

$$\mathsf{max\text{-}eq}^R(i, j) := \max\{t : i, j \le |W_t| \text{ and } W_t^R[i] = W_t^R[j]\}.$$

We assume that $\mathsf{max\text{-}eq}(i, j)$ and $\mathsf{max\text{-}eq}^R(i, j)$ are the only way to access the strings W_t.

A period of a string is non-trivial if it is smaller than the length of the string. In Section 7.1, we prove the following result on special families.

Proposition 7.3. *Given a special family $W_1, \ldots, W_n$, in $\mathcal{O}(n)$ time one can compute the set*

$$\{p : p \text{ is a non-trivial period of } W_p\}.$$

7.1. Periods in special families (Proof of Proposition 7.3)

In this section, we develop an $\mathcal{O}(n)$-time algorithm that for a special family $W_1, \ldots, W_n$ computes all indices p such that p is a period of W_p. Since $|W_t| = n - t + 1$, these values satisfy $p \leq \frac{n}{2}$.

Let us recall that the *border table* $\mathbf{B}$ of a string S of length n is a table of size n such that $\mathbf{B}[i]$ is the length of the longest proper border of $S[1\,..\,i]$. A string of length n has a border of length b if and only if it has a period $n - b$. In the algorithms below we use the Knuth–Morris–Pratt algorithm [37] that computes the border table of a string in $\mathcal{O}(n)$ time. We also apply it in the form of the following lemma.

Lemma 7.4. *The longest border of length at most k of a string S of length n can be computed in $\mathcal{O}(k)$ time.*

Proof. It suffices to apply the Knuth–Morris–Pratt algorithm for the string $S[1\,..\,k]\,\$\,S[|S| - k + 1\,..\,|S|]$, where $\$$ is an arbitrary character that does not occur in S. □

As a warm-up, we first show how to implement our algorithm if no string W_p has a *highly periodic* border. Our procedure for the general case, described in Section 7.1.3, relies on highly periodic prefixes and suffixes of the strings W_t. Hence, in the previous Section 7.1.2 we implement an auxiliary procedure identifying such prefixes and suffixes.

7.1.1. The case of no highly periodic borders

As a warm-up, below we implement a simple algorithm for Proposition 7.3, which is efficient provided that no string W_t has a highly periodic border.[3] Our algorithm relies on the following simple property:

Observation 7.5. *If p is a period of W_t, then p is also a period of $W_{t'}$ for each $t' < t$.*

Algorithm 5: Compute $\{p : p \text{ is a non-trivial period of } W_p\}$.

```
1 p := 1;
2 while p ≤ (1/2)n do
3     if W_p[1..n − 2p + 1] = W_p[p + 1..|W_p|] then report p;
4     p := min{q > p : q is a period of W_p};          ▷ p := next(p)
```

In the analysis below, for an integer p such that $1 \leq p \leq \frac{1}{2}n$, we denote $\min\{q > p : q \text{ is a period of } W_p\}$ by $\text{next}(p)$.

Fact 7.6. *Algorithm 5 is correct.*

Proof. Note that p is a period of W_p if and only if W_p has a border of length $|W_p| - p = n - 2p + 1$. Hence, p is reported in Line 3 if and only if it is a period of W_p. Thus, we only need to prove that we do not skip any valid output while increasing p in Line 4. This immediately follows from Observation 7.5. □

Next, we bound the running time of a single iteration of the **while** loop.

Fact 7.7. *Lines 3 and 4 can be implemented in $\mathcal{O}(n - 2p + 1)$ time.*

Proof. Let $k = n - 2p + 1$. The claim is trivial for Line 3, while in Line 4, we need to find the longest border of W_p that is shorter than k. This can be done in $\mathcal{O}(k)$ time with Lemma 7.4. □

In the analysis of the overall running time, we finally use our assumption.

Lemma 7.8. *Algorithm 5 takes $\mathcal{O}(n)$ time if strings W_t have no highly periodic borders.*

[3] Recall that a string X is *highly periodic* if its shortest period $\text{per}(X)$ satisfies $\text{per}(X) \leq \frac{1}{3}|X|$.

Proof. Let us introduce a potential $\phi(p) = p + \text{next}(p)$. By Observation 7.5, next is non-decreasing. Hence, $1 < \phi(1) < \phi(2) < \cdots < \phi(\lfloor \frac{1}{2}n \rfloor) = \mathcal{O}(n)$. To bound the overall running time with $\mathcal{O}(n)$, due to Fact 7.7, it suffices to show that $\phi(\text{next}(p)) - \phi(p) > \frac{1}{3}(n - 2p + 1)$ for $p \le \frac{1}{2}n$. For convenience we restrict the proof to the case that $\text{next}(\text{next}(p)) \le \frac{1}{2}n$; the final two iterations of the loop take $\mathcal{O}(n)$ time by Fact 7.7.

By Observation 7.5, both $\text{next}(\text{next}(p))$ and $\text{next}(p)$ are periods of W_p. This yields two borders of W_p, both shorter than $n - 2p + 1$. The longer one is not highly periodic, so

$$|W_p| - \text{next}(\text{next}(p)) < \tfrac{2}{3}(|W_p| - \text{next}(p)) \le \tfrac{2}{3}(|W_p| - p)$$

i.e., $n - p + 1 - \text{next}(\text{next}(p)) < \frac{2}{3}(n - 2p + 1)$. This immediately implies the claimed inequality:

$$\phi(\text{next}(p)) - \phi(p) = \text{next}(\text{next}(p)) - p > \tfrac{1}{3}(n - 2p + 1). \quad \square$$

7.1.2. Highly periodic prefixes in special families

We define a *high period* $\mathsf{hper}(S)$ of a string S as $\mathsf{per}(S)$ if S is highly periodic and otherwise leave $\mathsf{hper}(S)$ undefined. More formally,

$$\mathsf{hper}(S) = \begin{cases} \mathsf{per}(S) & \text{if } \mathsf{per}(S) \le \frac{1}{3}|S|, \\ \perp & \text{otherwise.} \end{cases}$$

For integers ℓ, t with $1 \le \ell \le |W_t|$, we denote

$$PER_\ell[t] = \mathsf{hper}(W_t[1\,..\,\ell]).$$

In other words, PER_ℓ represents high periods of length-ℓ prefixes of strings W_t contained in the special family. Our goal in this section is to compute the vectors PER_ℓ for $1 \le \ell \le n$. We shall prove that the subsequent vectors PER_ℓ and $PER_{\ell-1}$ on average differ on just $\mathcal{O}(1)$ positions; hence, we can represent all these vectors in linear space despite the fact that their total size is quadratic.

Before we compare subsequent vectors PER_ℓ, let us state basic properties of a single such vector. For this, we denote

$$\text{top}(\ell) = \min\{t : |W_t| < \ell \text{ or } PER_\ell[t] = \perp\}.$$

Fact 7.9. *For each ℓ, the vector PER_ℓ satisfies the following conditions:*

(a) *if $t \ge \text{top}(\ell)$, then $PER_\ell[t] = \perp$;*
(b) *if $1 < t < \text{top}(\ell)$, then $PER_\ell[t]$ is a multiple of $PER_\ell[t-1]$.*

Proof. By Definition 7.2, if p is a period of $W_t[1\,..\,\ell]$, then p is also a period of $W_{t'}[1\,..\,\ell]$ for each $t' \le t$. This immediately yields (a). For $1 < t < \text{top}(\ell)$, we conclude that $PER_\ell[t] \le \frac{1}{3}\ell$ is a period of $W_{t-1}[1\,..\,\ell]$, so $\mathsf{per}(W_{t-1}[1\,..\,\ell]) \mid PER_\ell[t]$ due to the Fine–Wilf periodicity lemma. Hence, (b) also follows. $\square$

Next, we compare subsequent vectors $PER_{\ell-1}$ and PER_ℓ; see Fig. 9.

Fact 7.10. *For every ℓ with $1 < \ell \le n$, we have $\text{top}(\ell) \ge \text{top}(\ell-1) - 1$. Moreover, for each $t < \text{top}(\ell)$:*

$$PER_\ell[t] = \begin{cases} PER_{\ell-1}[t] & \text{if } t < \text{top}(\ell-1) \\ \frac{1}{3}\ell & \text{otherwise.} \end{cases}$$

Consequently, in total there are less than $2n$ differences between subsequent vectors $PER_1, \ldots, PER_n$.

Proof. Let us denote $\tau = \text{top}(\ell-1)$. First, suppose that $\tau > 1$ and let $p = PER_{\ell-1}[\tau - 1]$. Note that $W_{\tau-1}[\ell - 1 - p] = W_{\tau-1}[\ell - 1]$, so Definition 7.2 yields $W_t[\ell - p] = W_t[\ell]$ for $1 \le t \le \tau - 2$. By Fact 7.9, the periods $PER_{\ell-1}[t]$ are divisors of p. Hence, $W_t[\ell] = W_t[\ell - p] = W_t[\ell - PER_{\ell-1}[t]]$, i.e., $PER_\ell[t] = PER_{\ell-1}[t]$ for $t \le \tau - 2$.

We may also have $PER_\ell[\tau - 1] = p = PER_{\ell-1}[\tau - 1]$ (provided that $W_{\tau-1}[\ell] = W_{\tau-1}[\ell - p]$). Otherwise, $\mathsf{per}(W_{\tau-1}[1\,..\,\ell])$ is not a multiple of p. By the Fine–Wilf periodicity Lemma for $W_{\tau-1}[1\,..\,\ell - 1]$, this yields $\mathsf{per}(W_{\tau-1}[1\,..\,\ell]) > \ell - p$, i.e., $PER_\ell[\tau - 1] = \perp$.

The argument above implies $\text{top}(\ell) \ge \text{top}(\ell-1) - 1$ and $PER_\ell[t] = PER_{\ell-1}[t]$ for $t < \min(\text{top}(\ell), \text{top}(\ell-1))$. It remains to show that $PER_\ell[t] = \frac{1}{3}\ell$ for $\text{top}(\ell-1) \le t < \text{top}(\ell)$. A period of $W_t[1\,..\,\ell]$ is also a period of $W_t[1\,..\,\ell - 1]$, but this string is not highly periodic due to $PER_{\ell-1}[t] = \perp$. Hence, we must have $\frac{1}{3}\ell \ge PER_\ell[t] > \frac{1}{3}(\ell - 1)$, i.e., $PER_\ell[t] = \frac{1}{3}\ell$, as claimed.

Finally, observe that the Hamming distance between $PER_{\ell-1}$ and PER_ℓ is $|\text{top}(\ell) - \text{top}(\ell-1)|$; see again Fig. 9. Let us track the subsequent values of $x = \text{top}(\ell)$. Initially $x = \text{top}(1) = 1$ and in the end $x = \text{top}(n) \le 2$. This variable can only decrease

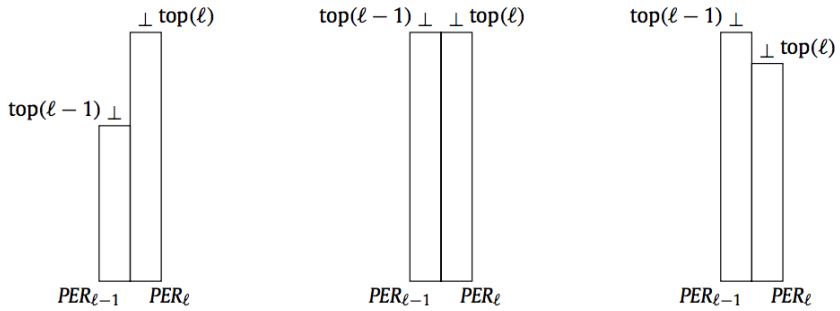

Fig. 9. Three possible cases of transition $PER_{\ell-1} \mapsto PER_\ell$. The column PER_ℓ can differ from $PER_{\ell-1}$ only in positions between $\mathrm{top}(\ell-1)$ and $\mathrm{top}(\ell)-1$ and at position $\mathrm{top}(\ell)$ in the third case. In the first case all newly computed values (above and including $\mathrm{top}(\ell-1)$) are equal to $\frac{1}{3}\ell$. The values above top elements are equal $\perp$ (undefined). $\mathrm{top}(\ell)$ can decrease only by one, but can increase arbitrarily.

by one, which happens at most $n-1$ times. Hence, x can increase by at most n. Consequently, $\sum_{\ell=2}^{n} |\mathrm{top}(\ell) - \mathrm{top}(\ell-1)| \le 2n-1$. □

Fact 7.10 lets us design an algorithm computing the subsequent vectors PER_ℓ. Algorithm 6 gives a high-level description of our procedure. Later, we develop its efficient implementation; see Algorithm 7.

Algorithm 6: Compute PER_ℓ for $\ell = 1, \ldots, n$ (without details).

```
1  P[1..n] := [⊥, ..., ⊥]; τ := 1;
2  for ℓ := 2 to n do                          ▷Invariant: P = PER_{ℓ−1} and τ = top(ℓ − 1)
3      if τ > 1 and P[τ − 1] is not a period of W_{τ−1}[1..ℓ] then
4          P[τ − 1] := ⊥;
5          τ := τ − 1;
6      else
7          while (1/3)ℓ is a period of W_τ[1..ℓ] do
8              P[τ] := (1/3)ℓ;
9              τ := τ + 1;
```

Observation 7.11. *Algorithm 6 correctly computes vectors $PER_1, \ldots, PER_n$. It takes $\mathcal{O}(n)$ time plus the time required to evaluate the tests in Lines 3 and 7.*

Our goal is therefore to provide efficient implementation of Lines 3 and 7. The former task is easy: since $p := \mathsf{P}[\tau-1]$ is a period of $W_{\tau-1}[1..\ell-1]$, it suffices to compare $W_{\tau-1}[\ell]$ with $W_{\tau-1}[\ell-p]$. Thus, the condition in Line 3 can be replaced with the following one, which we can evaluate in constant time:

$$\tau > 1 \textbf{ and } W_{\tau-1}[\ell] \neq W_{\tau-1}[\ell - \mathsf{P}[\tau-1]].$$

As for Line 7, we only aim at amortized constant time. The following result shows that this condition evaluates to **false** unless τ has not changed during many previous iterations of the **for** loop.

Fact 7.12. *If $PER_\ell[t] = \frac{1}{3}\ell$ for some $\ell \in \{1, \ldots, |W_t|\}$, then*

$$\mathrm{top}(\ell-1) = \cdots = \mathrm{top}\left(\left\lfloor \tfrac{1}{2}\ell \right\rfloor\right).$$

Proof. By Fact 7.10, we may assume without loss of generality that $t = \mathrm{top}(\ell-1)$. Let $\ell' < \ell$ be the smallest position such that $\mathrm{top}(\ell') = \cdots = \mathrm{top}(\ell-1) = t$. To prove that $\ell' \le \frac{1}{2}\ell$, we consider two cases.

If $\mathrm{top}(\ell'-1) < t$, then Fact 7.10 applied for $\ell', \ldots, \ell$ yields $PER_{\ell'}[t-1] = \cdots = PER_\ell[t-1] = \frac{1}{3}\ell'$. By Fact 7.9, $PER_\ell[t]$ $(= \frac{1}{3}\ell)$ is a multiple of $PER_\ell[t-1]$ $(= \frac{1}{3}\ell')$. Hence, ℓ is a proper multiple of ℓ', so $\ell' \le \frac{1}{2}\ell$, as claimed.

In the remaining case, we have $\mathrm{top}(\ell'-1) > t$. Both $p := PER_{\ell'-1}[t]$ and $\frac{1}{3}\ell$ are periods of $W_t[1..\ell'-1]$; p by definition and $\frac{1}{3}\ell$ by the assumption of the fact. If $\ell' > \frac{1}{2}\ell$, we have $p + \frac{1}{3}\ell - 1 \le \frac{1}{3}\ell' + \frac{1}{3}\ell - 1 < \ell' - 1$, so $p \mid \frac{1}{3}\ell$ by the Fine–Wilf periodicity lemma. However, $\frac{1}{3}\ell$ is a period of $W_t[1..\ell]$, but p is not. This contradiction concludes the proof. □

In other words, if τ changes during the iteration with $\ell = k$ of Algorithm 6, then the **else** branch can be ignored during iterations with $\ell = k+1, \ldots, 2k-1$. This saves us $\mathcal{O}(k)$ time, which we spend during iteration with $\ell = 2k$ for some

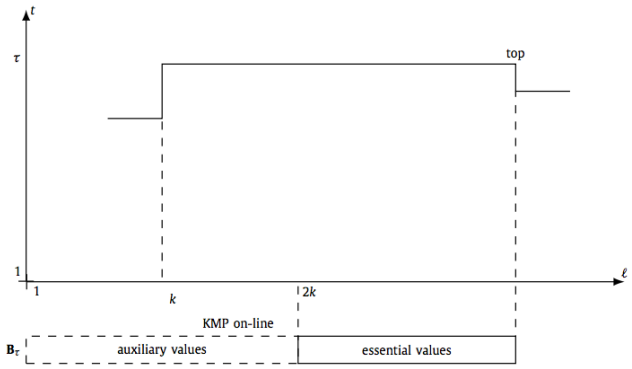

Fig. 10. The use of the Knuth–Morris–Pratt algorithm. We activate this procedure when $\ell = 2k$, where $\ell = k$ was the last iteration when τ changed. At $\ell = 2k$, we precompute auxiliary values $\mathbf{B}_\tau[i]$ for $i \le \ell$ (Line 7 in Algorithm 7). For each $\ell > 2k$, we then compute an essential value $\mathbf{B}_\tau[\ell]$ (Line 8) and compare it with $\frac{2}{3}\ell$ to check if $\frac{1}{3}\ell = \mathrm{per}(W_t[1\,..\,\ell])$. As soon as τ changes again, we abandon the procedure.

preprocessing of τ so that Line 7 can be efficiently evaluated until τ changes again. More specifically, we compute the $2k$ initial entries of the border table $\mathbf{B}_\tau$ of W_τ. In the following iterations, we keep computing subsequent entries $\mathbf{B}_\tau[\ell]$ of this border table using the Knuth–Morris–Pratt algorithm; see Fig. 10. Since $\mathrm{per}(W_\tau[1\,..\,\ell]) = \ell - \mathbf{B}_\tau[\ell]$, this can be used to evaluate Line 7.

If the first evaluation of Line 7 is **true**, we still need to test this condition at subsequent iterations of the **while** loop. Here, ℓ remains constant, so we exploit the full power of max-eq queries to compute the largest index t' such that $\frac{1}{3}\ell$ is a period of $W_{t'}[1\,..\,\ell]$. This takes $\mathcal{O}(\ell)$ time, which we can afford: due to Fact 7.12, we will not enter the **else** branch during the next ℓ iterations of the **for** loop. The resulting procedure is given as Algorithm 7.

Algorithm 7: Compute PER_ℓ for $\ell = 1, \ldots, n$ (with details).

```
1  P[1..n] := [⊥, ..., ⊥]; τ := 1; k := 1;
2  for ℓ := 2 to n do
       ▷Invariant: P = PER_{ℓ−1}, τ = top(ℓ − 1) = ··· = top(k) ≠ top(k − 1)
3      if τ > 1 and W_{τ−1}[ℓ] ≠ W_{τ−1}[ℓ − P[τ − 1]] then
4          P[τ − 1] := ⊥;
5          τ := τ − 1; k := ℓ;
6      if ℓ ≥ 2k then
7          if ℓ = 2k then compute B_τ[1..ℓ];          ▷ℓ iterations of KMP
8          else compute B_τ[ℓ];                       ▷1 iteration of KMP
9          if B_τ[ℓ] = (2/3)ℓ then
10             τ' := min{max-eq(i, i + (1/3)ℓ) : 1 ≤ i ≤ (2/3)ℓ};   ▷τ' = top(ℓ) − 1
11             while τ ≤ τ' do
12                 P[τ] := (1/3)ℓ;
13                 τ := τ + 1;
14             k := ℓ;
```

Lemma 7.13. *Given a special family $W_1, \ldots, W_n$ of strings, Algorithm 7 in $\mathcal{O}(n)$ time computes a compact representation of vectors PER_ℓ for $\ell = 1, \ldots, n$.*

Proof. First, let us focus on correctness of Algorithm 7, interpreting it as an implementation of the Algorithm 6 which we have already shown to be correct. As discussed above, the condition in Line 3 is equivalent to the original test. The new **if** statement in Line 6 is correct due to Fact 7.12. The condition in Line 9 is equivalent to the original test in Line 7 of Algorithm 6 due to the relation between longest borders and shortest periods. Finally, the value τ' computed in Line 10 is clearly the largest t such that $\frac{1}{3}\ell$ is a period of $W_t[1\,..\,\ell]$. Hence, the condition in Line 7 of Algorithm 6 is equivalent to $\tau \le \tau'$. Hence, Algorithm 7 is indeed correct.

As for the running time analysis, we only need to bound the cost of Lines 7–10; the other operations take $\mathcal{O}(n)$ time due to Observation 7.11. Line 7 takes $\mathcal{O}(\ell)$ time, but if it is executed at iteration ℓ of the **for** loop, then it was not executed in the past $k - 1 = \frac{1}{2}\ell - 1$ iterations. Hence, the overall time spent on Line 7 is $\mathcal{O}(n)$. The amortized cost of Line 8 is constant by the properties of the Knuth–Morris–Pratt algorithm. Finally, executions of Line 10 take $\mathcal{O}(\ell)$ time and enjoy the same pattern as of Line 7. Consequently, the overall running time is indeed $\mathcal{O}(n)$. □

7.1.3. General case

Our algorithm for arbitrary special families resembles Algorithm 5. However, it uses the vectors PER_ℓ and their symmetric counterparts PER^R_ℓ (for the reverse strings $W^R_1, \ldots, W^R_n$) to expedite the computations.

Algorithm 8: Compute $\{p : p \text{ is a non-trivial period of } W_p\}$.

```
1  p := 1;
2  while p < (1/2)n do
3      m := n − 2p + 1;
4      if PER_p[m] = PER^R_p[m] ≠ ⊥ then
5          q := PER_p[m];
6          if W_p[1..q] = W_p[p+1..p+q] then report p;
7          p := min{p' > p : p' is a period of W_p[1..p+2q]}
8      else if PER_p[⌈(3/4)m⌉] ≠ ⊥ and PER^R_p[⌈(3/4)m⌉] ≠ ⊥ then
9          p := p + 1;
10     else
11         if W_p[1..n−2p+1] = W_p[p+1..n−p+1] then report p;
12         p := min{p' > p : p' is a period of W_p};
```

Recall that in Section 7.1.1, we defined next(p) so that a single iteration of the **while** loop in Algorithm 5 updates p to next(p). By next$'(p)$, we denote an analogous transformation with respect to Algorithm 8. In other words, next$'(p)$ is the value assigned to the variable p in Line 7, 9, or 12, depending on which branch is executed.

Lemma 7.14. *Algorithm 8 correctly computes the set* $\{p : p \text{ is a non-trivial period of } W_p\}$.

Proof. As in the proof of Fact 7.6, we prove two claims for an arbitrary iteration of the **while** loop, associated with the initial value of p:

- the algorithm reports p if and only if p is a period of W_p,
- no p' with $p < p' <$ next$'(p)$ is a period of $W_{p'}$.

Recall that p is a period of W_p if and only if $W_p[1..n-2p+1] = W_p[p+1..n-p+1]$. Moreover, in the second claim it suffices to prove next$'(p) \le$ next(p).

We analyze each branch separately. In the first branch, both the prefix $W_p[1..m]$ and the suffix $W_p[p+1..p+m]$ are highly periodic with common period q. Hence, $W_p[1..m] = W_p[p+1..p+m]$ if and only if $W_p[1..q] = W_p[p+1..p+q]$, so Line 6 is correct. Moreover, next$'(p) \le$ next(p) because a period of W_p must also be a period of the prefix $W_p[1..p+2q]$.

In the second branch, we just need to prove that p is not a period of W_p. For a proof by contradiction, suppose that $W_p[1..m] = W_p[p+1..p+m]$ are both equal. Then their prefix and suffix of length $\lceil \frac{3}{4}m \rceil$ are highly periodic and they overlap on a fragment of length

$$2\lceil \tfrac{3}{4}m \rceil - m \ge 2\lceil \tfrac{3}{4}m \rceil - \tfrac{4}{3}\lceil \tfrac{3}{4}m \rceil = \tfrac{2}{3}\lceil \tfrac{3}{4}m \rceil.$$

By the Fine–Wilf periodicity lemma, the periods of these two factors must be synchronized, which means that $W_p[1..m] = W_p[p+1..p+m]$ is highly periodic. This is a contradiction because in that case we would be in the first branch.

In the third branch, we explicitly check p and set next$'(p) =$ next(p). □

Finally, we bound the running time of Algorithm 8 using the same potential $\phi(p) = p +$ next(p) as in the proof of Lemma 7.8.[4]

Lemma 7.15. *Algorithm 8 can be implemented in* $\mathcal{O}(n)$ *time.*

Proof. In the preprocessing, we run Algorithm 7 for the input special family $W_1, \ldots, W_n$ and for its reverse. This takes $\mathcal{O}(n)$ time due to Lemma 7.13.

Next, we shall prove that any iteration of the **while** loop can be implemented in $\mathcal{O}(\phi(\text{next}'(p)) - \phi(p))$ time, where $\phi(p) = p +$ next(p).

In the first branch, the running time is $\mathcal{O}(q)$: this is trivial for Line 6, while in Line 7 we seek for the longest border of $W_p[1..p+2q]$ shorter than $2q$, which can be done in $\mathcal{O}(q)$ time with Lemma 7.4. Thus, it suffices to prove that $\phi(\text{next}'(p)) - \phi(p) \ge q$. If next$'(p) - p \ge q$, this is trivial by monotonicity of the next function: next(next$'(p)) \ge$ next(p). Hence, suppose that $p' :=$ next$'(p) < p + q$, which implies $W_p[1..q] = W_p[p'+1..p'+q]$. As $W_p[1..m]$ and $W_p[p'+$

[4] Note that we do not adapt the potential by changing next(p) to next$'(p)$.

$1\,..\,p+m]$ both have period q, this yields a border of W_p of length $|W_p|-p'$. The shortest period of this border is q (since $|W_p|-p' > m-q \geq 2q$), so $\text{next}(p') \geq p'+q$. Moreover, p' is a period of W_p, so $p' = \text{next}(p)$ (we always have $\text{next}'(p) \leq \text{next}(p)$; see Lemma 7.14). Consequently, $\phi(p') - \phi(p) = \text{next}(p') - p \geq p' + q - p \geq q$, as claimed.

In the second branch, the running time is constant while the increase in potential is at least one: $\phi(p+1) - \phi(p) = 1 + \text{next}(p+1) - \text{next}(p) \geq 1$.

In the third branch, the running time is $\mathcal{O}(m)$ (see Fact 7.7) and $\text{next}'(p) = \text{next}(p)$. Thus, we shall prove that $\phi(\text{next}(p)) - \phi(p) \geq \frac{1}{4}m$. For a proof by contradiction, suppose that $\text{next}(\text{next}(p)) - p < \frac{1}{4}m$. By Observation 7.5, $\text{next}(\text{next}(p))$ and $\text{next}(p)$ are both periods of W_p. Both these periods correspond to borders of W_p of length at least $\lceil \frac{3}{4}m \rceil$. Moreover, $\text{next}(\text{next}(p)) - \text{next}(p) \leq \frac{1}{4}m$ is a period of these borders, and, in particular, of the prefix and the suffix of W_p of length $\lceil \frac{3}{4}m \rceil$. However, we are not in the second branch, a contradiction. This completes the proof of Lemma 7.15 and Proposition 7.3. □

7.2. Algorithm for sliding op-periods

An efficient algorithm follows from Proposition 7.3.

Theorem 7.16. *All sliding op-periods of a string of length n can be computed in $\mathcal{O}(n \log\log n)$ expected time or $\mathcal{O}(n \log^2 \log n / \log\log\log n)$ worst-case time, using $\mathcal{O}(n)$ space in either case.*

Proof. Let us argue that $SH_1, \ldots, SH_n$ form a special family. The length condition is satisfied. Note that the order equivalence is hereditary, so $S[i\,..\,i+p-1] \approx S[j\,..\,j+p-1]$ indeed yields

$$S[i\,..\,i+p-2] \approx S[j\,..\,j+p-2] \quad \text{and} \quad S[i+1\,..\,i+p-1] \approx S[j+1\,..\,j+p-1].$$

Finally, let us observe that $\mathsf{max\text{-}eq}(i,j) = \text{op-LCP}(i,j)$ and $\mathsf{max\text{-}eq}^R(i,j) = \text{op-LCP}^R(n+1-i, n+1-j)$ which can be computed in constant time after the preprocessing of Lemma 3.9.

The sliding op-periods p of S are computed using the characterization provided by Lemma 7.1. For $p > \frac{1}{2}n$, it suffices to check if $\text{op-LCP}(1, p+1) \geq n-p$. On the other hand, for $p \leq \frac{1}{2}$, we apply Proposition 7.3. The overall running time is $\mathcal{O}(n)$ plus the preprocessing of Lemma 3.9. □

8. Conclusions and open problems

Several combinatorial and algorithmic results related to periods in the order-preserving model (op-periods) were proposed in this work. In particular, linear-time and near-linear-time algorithms computing different types of op-periods were presented. An open question is to verify if all initial and sliding op-periods can be computed in linear time; our algorithms for these types of op-periods run in $\mathcal{O}(n \log\log n)$ time. We also show an algorithm that computes a representation of all general op-periods in $\mathcal{O}(n \log n)$ time (the number of all general op-periods in a string can be $\Omega(n^2)$). It would be interesting to know if there is an algorithm that computes the shortest (non-trivial) general op-period faster; in comparison, for the shortest (non-trivial) initial op-period we proposed an algorithm running in $\mathcal{O}(n)$ time.

We discussed potential applications of op-periods for discovering repeating *shapes* in time series. In reality, the data in time series can often contain some irregularities introduced by influence of random, unforeseen factors. Therefore, it would be interesting to know if one can efficiently compute *approximate* op-periods of strings allowing for a small number of such errors. A model for approximate order-preserving pattern matching was considered by Gawrychowski and Uznański [27].

Declaration of competing interest

None declared under financial, general, and institutional competing interests.

Acknowledgments

A part of this work was done during the workshop *StringMasters in Warsaw 2017* that was sponsored by the Warsaw Center of Mathematics and Computer Science. The authors thank the participants of the workshop, especially Hideo Bannai and Shunsuke Inenaga, for helpful discussions.

References

[1] G. Gourdel, T. Kociumaka, J. Radoszewski, W. Rytter, A.M. Shur, T. Waleń, String periods in the order-preserving model, in: R. Niedermeier, B. Vallée (Eds.), 35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, in: LIPIcs, vol. 96, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2018, pp. 38:1–38:16, https://doi.org/10.4230/LIPIcs.STACS.2018.38.

[2] M. Crochemore, C.S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S.P. Pissis, J. Radoszewski, W. Rytter, T. Waleń, Order-preserving indexing, Theor. Comput. Sci. 638 (2016) 122–135, https://doi.org/10.1016/j.tcs.2015.06.050.

[3] Y. Matsuoka, T. Aoki, S. Inenaga, H. Bannai, M. Takeda, Generalized pattern matching and periodicity under substring consistent equivalence relations, Theor. Comput. Sci. 656 (2016) 225–233, https://doi.org/10.1016/j.tcs.2016.02.017.

[4] M. Kubica, T. Kulczyński, J. Radoszewski, W. Rytter, T. Waleń, A linear time algorithm for consecutive permutation pattern matching, Inf. Process. Lett. 113 (12) (2013) 430–433, https://doi.org/10.1016/j.ipl.2013.03.015.

[5] N.J. Fine, H.S. Wilf, Uniqueness theorems for periodic functions, Proc. Am. Math. Soc. 16 (1) (1965) 109–114, https://doi.org/10.2307/2034009.

[6] M.G. Castelli, F. Mignosi, A. Restivo, Fine and Wilf's theorem for three periods and a generalization of Sturmian words, Theor. Comput. Sci. 218 (1) (1999) 83–94, https://doi.org/10.1016/S0304-3975(98)00251-5.

[7] J. Justin, On a paper by Castelli, Mignosi, Restivo, RAIRO, Theor. Inform. Appl. 34 (5) (2000) 373–377, https://doi.org/10.1051/ita:2000122.

[8] R. Tijdeman, L. Zamboni, Fine and Wilf words for any periods, Indag. Math. 14 (1) (2003) 135–147, https://doi.org/10.1016/s0019-3577(03)90076-0.

[9] J. Berstel, L. Boasson, Partial words and a theorem of Fine and Wilf, Theor. Comput. Sci. 218 (1) (1999) 135–141, https://doi.org/10.1016/S0304-3975(98)00255-2.

[10] F. Blanchet-Sadri, D. Bal, G. Sisodia, Graph connectivity, partial words, and a theorem of Fine and Wilf, Inf. Comput. 206 (5) (2008) 676–693, https://doi.org/10.1016/j.ic.2007.11.007.

[11] F. Blanchet-Sadri, R.A. Hegstrom, Partial words and a theorem of Fine and Wilf revisited, Theor. Comput. Sci. 270 (1–2) (2002) 401–419, https://doi.org/10.1016/S0304-3975(00)00407-2.

[12] A.M. Shur, Y.V. Gamzova, Partial words and the interaction property of periods, Izv. Math. 68 (2004) 405–428, https://doi.org/10.1070/im2004v068n02abeh000480.

[13] A.M. Shur, Y.V. Konovalova, On the periods of partial words, in: J. Sgall, A. Pultr, P. Kolman (Eds.), 26th International Symposium on Mathematical Foundations of Computer Science, MFCS 2001, in: LNCS, vol. 2136, Springer, 2001, pp. 657–665, https://doi.org/10.1007/3-540-44683-4_57.

[14] L.A. Idiatulina, A.M. Shur, Periodic partial words and random bipartite graphs, Fundam. Inform. 132 (1) (2014) 15–31, https://doi.org/10.3233/FI-2014-1030.

[15] S. Constantinescu, L. Ilie, Fine and Wilf's theorem for Abelian periods, Bull. Eur. Assoc. Theor. Comput. Sci. 89 (2006) 167–170, http://eatcs.org/images/bulletin/beatcs89.pdf.

[16] F. Blanchet-Sadri, S. Simmons, A. Tebbe, A. Veprauskas, Abelian periods, partial words, and an extension of a theorem of Fine and Wilf, RAIRO, Theor. Inform. Appl. 47 (3) (2013) 215–234, https://doi.org/10.1051/ita/2013034.

[17] M. Crochemore, C.S. Iliopoulos, T. Kociumaka, M. Kubica, J. Pachocki, J. Radoszewski, W. Rytter, W. Tyczyński, T. Waleń, A note on efficient computation of all abelian periods in a string, Inf. Process. Lett. 113 (3) (2013) 74–77, https://doi.org/10.1016/j.ipl.2012.11.001.

[18] G. Fici, T. Lecroq, A. Lefebvre, É. Prieur-Gaston, Algorithms for computing abelian periods of words, Discrete Appl. Math. 163 (2014) 287–297, https://doi.org/10.1016/j.dam.2013.08.021.

[19] G. Fici, T. Lecroq, A. Lefebvre, É. Prieur-Gaston, W.F. Smyth, A note on easy and efficient computation of full abelian periods of a word, Discrete Appl. Math. 212 (2016) 88–95, https://doi.org/10.1016/j.dam.2015.09.024.

[20] T. Kociumaka, J. Radoszewski, W. Rytter, Fast algorithms for abelian periods in words and greatest common divisor queries, J. Comput. Syst. Sci. 84 (2017) 205–218, https://doi.org/10.1016/j.jcss.2016.09.003.

[21] T. Kociumaka, J. Radoszewski, B. Wiśniewski, Subquadratic-time algorithms for abelian stringology problems, AIMS Med. Sci. 4 (3) (2017) 332–351, https://doi.org/10.3934/ms.2017.3.332.

[22] A. Apostolico, R. Giancarlo, Periodicity and repetitions in parameterized strings, Discrete Appl. Math. 156 (9) (2008) 1389–1398, https://doi.org/10.1016/j.dam.2006.11.017.

[23] S. Elizalde, M. Noy, Consecutive patterns in permutations, Adv. Appl. Math. 30 (1) (2003) 110–125, https://doi.org/10.1016/s0196-8858(02)00527-4.

[24] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, MIT Press, 2009, https://mitpress.mit.edu/books/introduction-algorithms-third-edition.

[25] D. Belazzougui, A. Pierrot, M. Raffinot, S. Vialette, Single and multiple consecutive permutation motif search, in: L. Cai, S. Cheng, T.W. Lam (Eds.), 24th International Symposium on Algorithms and Computation, ISAAC 2013, in: LNCS, vol. 8283, Springer, 2013, pp. 66–77, https://doi.org/10.1007/978-3-642-45030-3_7.

[26] S. Cho, J.C. Na, K. Park, J.S. Sim, A fast algorithm for order-preserving pattern matching, Inf. Process. Lett. 115 (2) (2015) 397–402, https://doi.org/10.1016/j.ipl.2014.10.018.

[27] P. Gawrychowski, P. Uznański, Order-preserving pattern matching with k mismatches, Theor. Comput. Sci. 638 (2016) 136–144, https://doi.org/10.1016/j.tcs.2015.08.022.

[28] T. Chhabra, M.O. Külekci, J. Tarhio, Alternative algorithms for order-preserving matching, in: J. Holub, J. Zdárek (Eds.), Prague Stringology Conference 2015, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2015, pp. 36–46, http://www.stringology.org/event/2015/p05.html.

[29] G. Decaroli, T. Gagie, G. Manzini, A compact index for order-preserving pattern matching, Softw. Pract. Exp. 49 (6) (2019) 1041–1051, https://doi.org/10.1002/spe.2694.

[30] T. Gagie, G. Manzini, R. Venturini, An encoding for order-preserving matching, in: K. Pruhs, C. Sohler (Eds.), 25th Annual European Symposium on Algorithms, ESA 2017, in: LIPIcs, vol. 87, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017, pp. 38:1–38:15, https://doi.org/10.4230/LIPIcs.ESA.2017.38.

[31] D. Cantone, S. Faro, M.O. Külekci, An efficient skip-search approach to the order-preserving pattern matching problem, in: J. Holub, J. Zdárek (Eds.), Prague Stringology Conference 2015, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2015, pp. 22–35, http://www.stringology.org/event/2015/p04.html.

[32] T. Chhabra, S. Faro, M.O. Külekci, J. Tarhio, Engineering order-preserving pattern matching with SIMD parallelism, Softw. Pract. Exp. 47 (5) (2017) 731–739, https://doi.org/10.1002/spe.2433.

[33] T. Chhabra, E. Giaquinta, J. Tarhio, Filtration algorithms for approximate order-preserving matching, in: C.S. Iliopoulos, S.J. Puglisi, E. Yilmaz (Eds.), 22nd International Symposium on String Processing and Information Retrieval, SPIRE 2015, in: LNCS, vol. 9309, Springer, 2015, pp. 177–187, https://doi.org/10.1007/978-3-319-23826-5_18.

[34] T. Chhabra, J. Tarhio, A filtration method for order-preserving matching, Inf. Process. Lett. 116 (2) (2016) 71–74, https://doi.org/10.1016/j.ipl.2015.10.005.

[35] S. Faro, M.O. Külekci, Efficient algorithms for the order preserving pattern matching problem, in: R. Dondi, G. Fertin, G. Mauri (Eds.), 11th International Conference on Algorithmic Aspects in Information and Management, AAIM 2016, in: LNCS, vol. 9778, Springer, 2016, pp. 185–196, https://doi.org/10.1007/978-3-319-41168-2_16.

[36] M. Crochemore, W. Rytter, Jewels of Stringology, World Scientific, 2002.

[37] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on Strings, Cambridge University Press, 2007.

[38] M.A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, P. Sumazin, Lowest common ancestors in trees and directed acyclic graphs, J. Algorithms 57 (2) (2005) 75–94, https://doi.org/10.1016/j.jalgor.2005.08.001.

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