Hide and seek with repetitions[☆]

Paweł Gawrychowski

Institute of Computer Science, University of Wrocław, Wroclaw, Poland

Florin Manea

Department of Computer Science, Kiel University, Kiel, Germany

Robert Mercaş

Department of Computer Science, Loughborough University, UK

Dirk Nowotka

Department of Computer Science, Kiel University, Kiel, Germany

Abstract

Pseudo-repetitions are a natural generalization of the classical notion of repetitions in sequences: they are the repeated concatenation of a word and its encoding under a certain morphism or antimorphism. Thus, such occurrences can be regarded as hidden repetitive structures of, or within, a word. We solve fundamental algorithmic questions on pseudo-repetitions by application of insightful combinatorial results on words. More precisely, we efficiently decide whether a word is a pseudo-repetition and find all the pseudo-repetitive factors of a word. We also approach the problem of deciding whether there exists an anti-/morphism for which a word is a pseudo-repetition. We show that some variants of this latter problem are efficiently solvable, while some others are NP-complete.

Keywords: Stringology, Pattern matching, Combinatorics on words, Repetition, Pseudo-repetition

1. Introduction and examples

The notions of repetition and primitivity are fundamental concepts on sequences used in a number of fields, among them being algorithmics on strings

[☆]This paper is an extended version of [1] and [2]. Florin Manea was supported by the *DFG* grant 596676. Dirk Nowotka was supported by the *DFG Heisenberg* grant 590179.

Email addresses: gawry@cs.uni.wroc.pl (Paweł Gawrychowski), flm@informatik.uni-kiel.de (Florin Manea), R.G.Mercas@lboro.ac.uk (Robert Mercaş), dn@informatik.uni-kiel.de (Dirk Nowotka)

(stringology), cryptography, and algebraic coding theory. A word is a repetition (or power) if it equals a repeated catenation of one of its prefixes. We consider a more general concept here, namely *pseudo-repetitions in words*, introduced by Czeizler et al. [3]. A word w is a pseudo-repetition if it equals a repeated catenation of one of its proper prefixes t and its image $f(t)$ under some morphism or antimorphism (for short "anti-/morphism") f, thus $w \in t\{t, f(t)\}^+$.

In this paper, we first investigate the following basic algorithmic problems: decide whether a word w is a pseudo-repetition for a given anti-/morphism f, find all k-powers of pseudo-repetitions occurring as factors in a word w, for an f as above. We establish algorithms and complexity bounds for these problems for various types of anti-/morphisms. Further, we investigate how efficiently we can discover whether an input word has a hidden repetitive structure, i.e, find an anti-/morphism f for which that word becomes an f-repetition. This last problem seems natural to us: basically, we look at a text and want to find an encoded repetitive structures in it, without actually knowing the encoding scheme. In the case of this problem, we identify both a series of cases when it can be solved efficiently and some cases where it is intractable. Apart from the application of standard stringology tools, like suffix arrays or longest common prefix data structure, our algorithms are based both on involved combinatorics on words results and on a series of diverse efficient data structures. To this end, our approach is aimed to provide a better understanding of the concept of pseudo-repetition itself as well as to enrich a set of algorithmic tools that can be actually used in its originating fields, computational biology and natural computing.

1.1. Background, Motivation, Previous Work

The concept of pseudo-repetitions (introduced in [3]) draws its original motivations from computational biology and natural computing, namely the facts that the Watson-Crick complement can be formalised as an antimorphic involution and both a single-stranded DNA and its complement (that is, its image under such an involution) encodes, in general, the same information. Repetitions are fundamental structures that occur in DNA sequences. Tandem repeats occur when occurrences of a DNA fragment consisting of one or more nucleotides are repeated one after the other in the DNA sequence; they are important in determining an individual's inherited traits. Inverted repeats are fragments of a DNA sequence that are followed somewhere downstream by their reverse complement. Pseudo-repetitions combine these two notions: they were defined as repeated consecutive occurrences of a fragment of a string and its reversed complement (i.e., image under the antimorphic involution modelling the Watson-Crick complement).

Other situations in which one encounters pseudo-repetitions are art-related: whether we consider visual or performing arts, the author may use repetitions of the exact same fragment or slightly modified variants of it to highlight this fragment as an important part of the created work of art. For instance, pseudo-repetitions may be seen as a mathematical model of the repetitive structures

studied in musical theory. Repetitions (also called restatements) of some fragment, in its initial form or on a different pitch, are used to provide unity to a musical piece. Moreover, the concept of *ternary (song) form* is also used: three consecutive musical fragments such that the first and third ones are identical, while the second one is constructed in order to provide a contrast to the other two (which can be formalised sometimes by seeing it as the image of the other two parts under some simple anti-/morphism). This musical concept appears in even more general forms, involving the approximate repetition of more than three factors.

It is worth noting that both in biology and in musical theory, the function that is applied to obtain the pseudo-repetition is structurally simple: it usually just rewrites one letter (symbolic representation of nucleotides or notes) into another one (thus, it is literal) and acts as a morphism (i.e., just writes the images of the letters one after the others in the order from left to right) or as an antimorphism (i.e., writes these images one after the other in reversed order).

Besides the examples above in which pseudo-repetitions occur, the concept seems to be of intrinsic theoretical interest, as they generalise a combinatorics on words concept that is central both in theory and applications. For instance, if we consider even palindromes as a natural modification of squares (a repetition of a word, once written from left to right, once written from right to left), repetitive structures containing both normal occurrences of some factor and mirrored occurrences of the same factor seem to be one of the most natural, thus, interesting, concepts derived from the classical repetitions. Generally, pseudo-repetitions can be seen as words that have an intrinsic, yet not obvious, repetitive structure.

The main results obtained so far on pseudo-repetitions were in the area of combinatorics on words: they were generalisations of classical results regarding repetitions in words to this more general setting of pseudo-repetitions. For instance, [3, 4] present generalisations of the Fine and Wilf theorem, [5, 6, 7, 8] present a complete characterisation of generalised Lyndon-Schuützenberger equations, while [9, 10, 11] present ways to construct infinite words that do not contain cubes under permutations (i.e., particular forms of pseudo-cubes). Except [1, 2], the conference papers on which this paper is based, not many investigations were done in the area of algorithmics of pseudo-repetitions. The papers [12, 13] investigate the problem of identifying all pseudo-repetitive factors in a word, in the setting when f is an antimorphic involution; [12] as well as the more recent [14] deal with the problem of checking whether a word contains at least one pseudo-repetitive pattern whose form is given as input. The results of this paper and comparisons to the existing literature are presented in Section 1.3.

1.2. Some Basic Concepts

For more detailed definitions we refer to [15, 16].

Let V be a finite alphabet. We denote by V^* the set of all words over V and by V^k the set of all words of length k. The *length* of a word $w \in V^*$ is denoted by $|w|$. The *empty word* is denoted by λ. Moreover, we denote by $\text{alph}(w)$ the alphabet of all letters that occur in w. In the problems discussed in this paper

we are given as input a word w of length n and we assume that the letters of w are in fact integers from $\{1, 2, \ldots, n\}$; thus w is seen as a sequence of integers. This is a common assumption in algorithmics on words (see, e.g., [17]).

A word u is a *factor* of a word v if $v = xuy$, for some x, y; also, u is a *prefix* of v if $x = \lambda$ and a *suffix* of v if $y = \lambda$. We denote by $w[i]$ the symbol at position i in w and by $w[i..j]$ the factor $w[i]w[i+1]\cdots w[j]$ of w starting at position i and ending at position j. For simplicity, we assume that $w[i..j] = \lambda$ if $i > j$. A word u occurs in w at position i if u is a prefix of $w[i..|w|]$. Also, we write $w = u^{-1}v$ when $v = uw$. Finally, if $w \in V^n$, we denote by w^R the mirror image of w, namely the word obtained by writing the letters of w right to left: $w^R = w[n]w[n-1]\cdots w[1]$.

The powers of a word w are defined recursively by $w^0 = \lambda$ and $w^n = ww^{n-1}$ for $n \geq 1$. If w cannot be expressed as a power of another word, then w is *primitive*. If $w = u^n$ with $n \geq 2$ and u primitive, then u is called the primitive root of w. A *period* of a word w over V is a positive integer p such that $w[i] = w[j]$ for all i and j with $i \equiv j \pmod{p}$. By $per(w)$ we denote the smallest period of w.

The following classical result is extensively used in our investigation:

Theorem 1 (Fine and Wilf [18])**.** *Let u and v be in V^*. If two words $\alpha \in u\{u, v\}^+$ and $\beta \in v\{u, v\}^+$ have a common prefix of length greater than or equal to $|u| + |v| - \gcd(|u|, |v|)$, then u and v are powers of a common word of length $\gcd(|u|, |v|)$.*

A function $f : V^* \to V^*$ is a morphism if $f(xy) = f(x)f(y)$ for all $x, y \in V^*$; f is an antimorphism if $f(xy) = f(y)f(x)$ for all $x, y \in V^*$. Note that to define an anti-/morphism it is enough to give the definitions of $f(a)$, for all $a \in V$. We say that f is *uniform* if there exists a positive integer k with $f(a) \in V^k$, for all $a \in V$; if $k = 1$ then f is called *literal*. If $f(a) = \lambda$ for some $a \in V$, then f is called *erasing*, otherwise *non-erasing*. The vector T_f of $|V|$ natural numbers with $T_f[a] = |f(a)|$ is called the length-type of the anti-/morphism f in the following. If $V = \{a_1, a_2, \ldots, a_n\}$, T is a vector of n natural numbers $T[a_1], T[a_2], \ldots, T[a_n]$, and $x = b_1 b_2 \cdots b_k$ with $b_i \in V$ for all i, we denote by $T(x) = \sum_{i \leq k} T[b_i]$, the length of the image of x under any anti-/morphism of length-type T defined on V. An anti-/morphism $f : V^* \to V^*$ is an involution if $f(f(a)) = a$ for all $a \in V$; it is not hard to see that any anti-/morphic involution is also literal.

We say that a word w is an *f-repetition*, or, alternatively, an f-power, if w is an element of $t\{t, f(t)\}^+$, for some prefix t of w; for simplicity, if $w \in t\{t, f(t)\}^+$ then w is called an f-power of root t. If w is not an f-power, then w is *f-primitive*.

We consider the following example.

Example 1. *The word $w = ACGTAC$ is primitive from the classical point of view (i.e., $\mathbf{1}$-primitive, where $\mathbf{1}$ is the identical anti-/morphism). Moreover, w*

is also σ-primitive for the morphic involution σ defined by

$$\begin{aligned} \sigma(A) &= T & \sigma(C) &= G \\ \sigma(T) &= A & \sigma(G) &= C \end{aligned} \tag{1}$$

Finally, consider σ to be an antimorphic involution. Note that in this case σ is, in fact, a formalization of the Watson-Crick complement, from biology. We get that $ACGTAC = AC \cdot \sigma(AC) \cdot AC$, thus, w is a σ-repetition.

Finally, the computational model we use to design and analyse our algorithms is the standard unit-cost RAM (Random Access Machine) with logarithmic word size, which is generally used in the analysis of algorithms. Also, all logarithms appearing here are in base 2.

1.3. Algorithmic problems

In the upcoming algorithmic problems, we assume that the words we process are sequences of integers (called letters, for simplicity). In general, if the input word has length n then we assume its letters are in $\{1, 2, \ldots, n\}$, so each letter fits in a single memory-word. This is a common assumption in algorithmics on words (see, e.g., the discussion in [17]).

In the first problem, which seems to us the most interesting one in the general context of pseudo-repetitions, we approach the fundamental problem of deciding whether a word is an f-repetition, for a fixed anti-/morphism f.

Problem 1. *Let $f : V^* \to V^*$ be an anti-/morphism. Given a word $w \in V^*$, decide whether this word is an f-repetition.*

We solve this problem in the general case of erasing anti-/morphisms in $\mathcal{O}(n \lg n)$ time, where $|w| = n$. However, in the particular case of uniform anti-/morphisms we obtain an optimal solution running in linear time. The latter covers the biologically motivated case of involutions from [3]. This optimal result seems interesting to us, as it shows that pseudo-repetitions can be detected as fast as repetitions, if the way we encode the repeated factor (i.e., the function f) is structurally simple, yet not the identity. We also extend our results to a more general form of Problem 1, testing whether $w \in \{t, f(t)\}^+$ for a proper factor t of w. Except for the most general case (of erasing anti-/morphisms), where we solve this problem in $\mathcal{O}(n^{1+\frac{1}{\lg \lg n}} \lg n)$ time, we preserve the same time complexity as we obtained for Problem 1.

Two other natural algorithmic problems are related to the fundamental combinatorial property of freeness of words, in the context of pseudo-repetitions. More precisely, we are interested in identifying the factors of a word which are pseudo-repetitions.

Problem 2. *Let $f : V^* \to V^*$ be an anti-/morphism and $w \in V^*$ be a given word.*

(1) Enumerate all triplets (i, j, ℓ), where $1 \leq i, j, \ell \leq |w|$, such that there exists t with $w[i..j] \in \{t, f(t)\}^\ell$.

(2) Given a positive integer k, enumerate all pairs (i, j), where $1 \leq i, j \leq |w|$, such that there exists t with $w[i..j] \in \{t, f(t)\}^k$.

Question (2) was originally considered in [13], while the first one represents its natural generalisation. Our approach to question (1) is based on constructing data structures which enable us to retrieve in constant time the answer to queries

$$rep(i, j, \ell)\text{: ``Is there } t \in V^* \text{ such that } w[i..j] \in \{t, f(t)\}^\ell?\text{''}$$

for $1 \leq i \leq j \leq n$ and $1 \leq \ell \leq n$, where n denotes the length of w. For an unrestricted variant of f, we can produce such data structures in $\mathcal{O}(n^{3.5})$ time. When f is non-erasing, the time taken to construct them is $\mathcal{O}(n^3)$, while when f is a literal anti-/morphism we can do it in time $\mathcal{O}(n^2)$. Once we have these structures, we can identify in $\Theta(n^3)$ time, in the general case, all the triples (i, j, ℓ) such that $w[i..j] \in \{t, f(t)\}^\ell$, answering (1) in $\mathcal{O}(n^{3.5})$ time. Similarly, for f non-erasing (respectively, literal) we answer question (1) in $\Theta(n^3)$ (respectively, $\Theta(n^2 \lg n)$) time and show that there are input words on which every algorithm solving this question has a running time asymptotically equal to ours (including the preprocessing time). Unfortunately, the time bound obtained for most general case is not tight.

Exactly the same data structures are used in the simplest case of literal anti-/morphisms to answer the more particular question (2). We obtain an algorithm that outputs in $\mathcal{O}(n^2)$ time, for given w and k, all pairs (i, j) such that $w[i..j] \in \{t, f(t)\}^k$; this time bound is shown to be tight. Taking advantage of the fact that k is given as input (so fixed throughout the algorithm) we can refine our solution for question (1) in order to get a $\Theta(n^2)$-time solution of question (2) for f non-erasing and a $\mathcal{O}(n^2 k)$-time solution for the general case. Both these results represent tight bounds when k is considered to be a constant, instead of being part of the input; as it is, for example, the case when we want to count the number of pseudo-cubes occurring in a word. Our results improve significantly the algorithmic results reported in [13].

In the last problem we approach, we are interested to decide whether there exists an anti-/morphism f for which a given word w is an f-repetition. Basically, we check whether a given word has a hidden repetitive structure. Note that, while in the case of the previous problems the main difficulty consists in finding a prefix x of w such that $w \in x\{x, f(x)\}^*$, this problem seems more involved: not only we need to find two factors x and y such that $w \in x\{x, y\}^*$, i.e., a suitable decompositions of w, but we also have to decide the existence of an anti-/morphism f mapping x to y.

Problem 3. *Given a word $w \in V^+$, decide whether there exist an anti-/morphism $f : V^* \to V^*$ and a prefix t of w such that $w \in t\{t, f(t)\}^+$.*

The unrestricted version of the problem is, however, trivial. We can always give a positive answer for input words of length greater than 2. It is enough to take the (non-erasing) anti-/morphism f that maps the first letter of w, namely $w[1]$, to $w[2..n]$, where $n = |w|$. Clearly, $w = w[1]f(w[1])$, so w is indeed an f-repetition. When the input word has length 1 or 0, the answer is negative.

On the other hand, when we add a series of simple restrictions to the initial statement, the problem becomes more interesting. The restrictions we define are of two types: either we restrict the form of the function f we search for or we restrict the repetitive structure of w by requiring that it consists in at least three repeating factors or that the root of the pseudo-repetition has length at least 2.

In the first case, when the input consists both in the word w and an encoding of the type of function we look for (given as the length-type of the respective anti-/morphism f), we obtain a series of solutions for Problem 3 that run in close-to-linear time. More precisely, in the most general case we can decide whether there exists an anti-/morphism f such that w is an f-repetition in $\mathcal{O}(n(\lg n)^2)$ time. Note that deciding whether a word is an f-repetition when f is known, takes only $\mathcal{O}(n \lg n)$ time, as we will show in the case of Problem 1. When we search for an uniform morphism we solve the problem in optimal linear time. This matches the complexity of deciding, for a given uniform anti-/morphism f, whether a given word is an f-repetition, obtained in [1]. This result covers also the case of literal anti-/morphism, extensively approached in the literature (see, e.g., [3, 9, 8]).

It is worth noting that our knowledge of the length-type of the function whose existence we try to decide reflects also in the fact that the form of the repetition is not arbitrary anymore. For instance, for a certain prefix t we know the length ℓ_t of the image of t through any function that has the given length-type; thus, for instance, if $|t| + \max\{|t|, |f(t)|\}$ is smaller than the length of the input word, it follows that w cannot be pseudo-square.

For the second kind of restrictions, the length-type of f is no longer given. In this case, we want to check, for instance, whether there exist a prefix t and an anti-/morphism f such that w is an f-repetition that consists in the concatenation of at least 3 factors t or $f(t)$. The most general case as well as the case when we add the supplementary restriction that f is non-erasing are NP-complete; the case when f is uniform (but of unknown length-type) is tractable. The problem of checking whether there exists a prefix t, with $|t| \geq 2$, and a non-erasing anti-/morphism f such that $w \in t\{t, f(t)\}^+$ is also NP-complete; this problem becomes tractable for erasing or uniform anti-/morphisms.

1.4. Examples

In this section we will present examples of how our problems convey on a given text. Consider now the following pice of text, called $\mathcal{T}$:

$$ACCACCGGTGGTACCGGTGGTACCGGTGGTACCACC$$
$$ACCACCGGTGGTACCGGTGGTACCACCACCGGTGGT$$

Following the description of the text, our alphabet will consist of the set of symbols $\{A, C, G, T\}$. We look at the three problems proposed above, and their variants.

Problem 1

If we consider first σ given in (1) as being a morphism, we see that our text is σ-primitive. However, in the case when σ is an antimorphic involution, $\mathcal{T}$ becomes in fact σ-periodic, by choosing $t = ACC$. In this case our text can be expressed as:

$$tt\sigma(t)\sigma(t)t\sigma(t)\sigma(t)t\sigma(t)\sigma(t)tttt\sigma(t)\sigma(t)t\sigma(t)\sigma(t)ttt\sigma(t)\sigma(t)$$

Clearly, for $t' = \sigma(t) = GGT$, our word $\mathcal{T}$ can also be expressed as:

$$\sigma(t')\sigma(t')t't'\sigma(t')t't'\sigma(t')t't'\sigma(t')\sigma(t')\sigma(t')\sigma(t')t't'\sigma(t')t't'\sigma(t')\sigma(t')\sigma(t')t't'$$

Problem 2

As seen above for the case of antimorphisms we already have that the whole word is a σ-repetition of root $t = ACC$. So the triple $(1, 72, 24)$ is in the solution set for our problem. Similarly, $(4, 9, 2)$ is also a such a triple, and so on.

However,we might inquire if there exist portions of the text $\mathcal{T}$ that are σ-repetitions if we consider σ to be just a regular morphism. Taking, for instance, f to be the identity morphism we see that for question **(1)** of the problem one of the triples in the solution set is $(31, 42, 4)$ with $t = ACC$; actually, this is the highest value of ℓ we can in fact get in this case.

However, considering a morphism that maps C to A (or, alternatively, A to C) and fixing $k = 12$, in the case of question **(2)** we get the unique solution-pair $(31, 42)$ with $t = C$ (respectively, $t = A$).

Problem 3

Taking the text $\mathcal{T}$ above as input of our problem and checking whether it is an f-repetition under the restriction that f is an literal antimorphisms, the answer to this problem should be yes. One of the literal antimorphisms f for which $\mathcal{T}$ is an f-repetition is σ defined above.

Furthermore, considering the problem under the restriction that f is a uniform morphism with the image of each letter of length 3, we shall get again a positive answer, and a possible choice for f is the function θ, with:

$$\begin{aligned} \theta(A) &= ACC & \theta(C) &= GGT \\ \theta(T) &= AAA & \theta(G) &= AAA \end{aligned} \tag{2}$$

In this case by choosing $t = ACC$ we have that $\mathcal{T}$ is a θ-repetitions:

$$\mathcal{T} = t\theta(t)\theta(t)\theta(t)ttt\theta(t)\theta(t)tt\theta(t)$$

When looking for f as 1-uniform morphisms, we again get a positive answer, with a possible choice for f being the morphism γ defined by:

$$\begin{aligned} \gamma(A) &= C & \gamma(C) &= A \\ \gamma(T) &= C & \gamma(G) &= C \end{aligned}$$

that will transform the text into a 1-γ-periodic text with C as a root.

2. Prerequisites

2.1. Number Theory

We begin this section by presenting several number theoretic properties. Lemma 1 is used in the time complexity analysis of our algorithms, while Lemma 2 and its corollary are utilised in the solutions of Problem 2. Given two natural numbers k and n, we write $k \mid n$ if k divides n. We denote by $d(n)$ the number of divisors of n and by $\sigma(n)$ their sum.

Lemma 1. *Let n be a natural number. The following statements hold:*

(1) $\sum_{1\leq \ell \leq n} d(\ell) \in \Theta(n \lg n)$*;*
$\sum_{1\leq \ell \leq n} d(\ell) \geq n \lg n$*, where* $d(n) \in o(n^{\varepsilon})$ *for all* $\varepsilon > 0$*;*

(2) $\sigma(n) \in \mathcal{O}(n \lg \lg n)$*;*

(3) $\sum_{1\leq \ell \leq n} (n - \ell + 1)d(\ell) \in \Theta(n^2 \lg n)$.

Proof. The first two results are well known (for their proofs for instance see [19]).

For the third statement let $T(n) = \sum_{1\leq \ell \leq n} (n - \ell + 1)d(\ell)$.

We have

$$T(n) = T(n-1) + \sum_{1\leq \ell \leq n} d(\ell) + d(n),$$

for $n \geq 2$. According to Statement 1, we obtain that

$$T(n) \geq T(n-1) + n \lg n + 2.$$

Applying iteratively this reasoning, we obtain that

$$T(n) \geq 2n + \sum_{1\leq \ell \leq n} \ell \lg \ell.$$

An elementary form of the Chebyshev inequality says that if $(a_\ell)_{1\leq \ell \leq n}$ and $(b_\ell)_{1\leq \ell \leq n}$ are two increasing sequences of real numbers, then the following holds

$\sum_{1\le\ell\le n} a_\ell b_\ell \ge \frac{1}{n}(\sum_{1\le\ell\le n} a_\ell)(\sum_{1\le\ell\le n} b_\ell)$. We apply this inequality to obtain a lower bound for $T(n)$, taking $a_\ell = \ell$ and $b_\ell = \lg \ell$. Therefore, we have:

$$T(n) \ge 2n + \sum_{1\le\ell\le n} \ell \lg \ell \ge 2n + \frac{1}{n}(\sum_{1\le\ell\le n} \ell)(\sum_{1\le\ell\le n} \lg \ell) \ge 2n + \frac{n(n+1)}{2n} \cdot \frac{n \lg(n/4)}{4}$$

Thus, $T(n) \in \Omega(n^2 \lg n)$.

Further,

$$T(n) = \sum_{1\le\ell\le n} (n - \ell + 1) d(\ell) \le n \sum_{1\le\ell\le n} d(\ell).$$

According to Statement 1, once more, we obtain that $T(n) \in \mathcal{O}(n^2 \lg n)$.

Therefore, $T(n) \in \Theta(n^2 \lg n)$, and the proof of Statement 3 is concluded. □

Lemma 2. *Let n be a natural number. We can compute in $\mathcal{O}(n^3)$ time a three dimensional array $T[k][m][\ell]$, with $1 \le k, m, \ell \le n$, where $T[k][m][\ell] = 1$ if and only if there exists a divisor s of ℓ and the numbers k_1 and k_2 such that $k_1 + k_2 = k$ and $k_1 s + k_2 sm = \ell$.*

Proof. First we note that given the positive integers m, k, and i, there exist the integers k_1 and k_2 such that $k_1 + mk_2 = i$, $k_1 + k_2 = k$, and $k_1, k_2 \ge 0$ if and only if $i \equiv k \bmod (m-1)$ and $km \ge i \ge k$. Indeed, from $k_1 + mk_2 = i$ we get that $k_1 + k_2 + (m-1)k_2 = i$ so $i \equiv k \bmod (m-1)$; the inequality $km \ge i \ge k$ follows easily from the fact $k_1 + k_2 = k$. The other implication follows by taking $k_2 = \frac{i-k}{m-1}$; it is clear that $i - k \le k(m-1)$ so $k_2 \le k$. Further, we take $k_1 = k - k_2$. This shows the equivalence stated above.

This first remark shows that we can decide in constant time whether for some positive integers m, k, and i there exist the integers k_1 and k_2 such that $k_1 + mk_2 = i$, $k_1 + k_2 = k$, and $k_1, k_2 \ge 0$.

Now, for each $\ell \le n$ we compute the list of its divisors. This can be done in $\mathcal{O}(n \lg n)$ time using the Sieve of Eratosthenes.

Further, we show how the values $T[k][m][\ell]$ can be computed in $\mathcal{O}(n^3)$ time, for $m \le \frac{n}{\lg n}$. For some k, m, and ℓ, we just go through all the divisors s of ℓ and set $T[k][m][\ell]$ to be equal to 1 if there exists s such that for the positive integers m, k, and $\frac{\ell}{s}$ there exist the integers k_1 and k_2 such that $k_1 + mk_2 = \frac{\ell}{s}$, $k_1 + k_2 = k$, and $k_1, k_2 \ge 0$. The time needed for all this process is upper-bounded by $\mathcal{O}(\sum_{1\le k\le n} \sum_{1\le m\le n/\lg n} \sum_{1\le\ell\le n} d(\ell))$ which can be shown to be in $\mathcal{O}(n^3)$ by Lemma 1.

Finally, we show how the values $T[k][m][\ell]$ can be computed in $\mathcal{O}(n^3)$ time, for $m \ge \frac{n}{\lg n}$. Since $\ell \le n$ and $m \ge \frac{n}{\lg n}$ it follows that the divisor s of ℓ is at most $\lg n$. Also, if we fix m and ℓ and a divisor s of ℓ, the number of possible values for k for which there exist numbers k_1 and k_2 such that $k_1 + k_2 = k$ and $k_1 s + k_2 sm = \ell$ is at most $\mathcal{O}(\lg n)$, as well, as $k \le n$ and $k \equiv \frac{\ell}{s} \bmod (m-1)$ (and there are at most $\frac{n}{m-1} \in \mathcal{O}(\frac{n}{n/\lg n}) = \mathcal{O}(\lg n)$ such numbers).

Therefore, we can proceed as follows. For all m and ℓ, we go through all the divisors of ℓ that are less than or equal to $\lg n$. For each such divisor, we set

$T[k][m][\ell] = 1$ for all k such that $k \equiv \frac{\ell}{s} \bmod (m-1)$ and $km \geq \frac{\ell}{s} \geq k$. By the explanations above, this takes $\mathcal{O}(n^2(\lg n)^2)$. This concludes our proof. □

As a consequence of the above we have the following Corollary:

Corollary 1. *Let R be a fixed natural constant, and n and k be given natural numbers. We can compute in $\mathcal{O}(n \lg n)$ time a matrix $T_k[m][\ell]$ with $1 \leq m \leq R$ and $1 \leq \ell \leq n$, where $T_k[m][\ell] = 1$ if and only if there exists a divisor s of ℓ and the numbers k_1 and k_2 such that $k_1 + k_2 = k$ and $k_1 s + k_2 sm = \ell$. The constant hidden by the $\mathcal{O}$-notation depends on R.*

Proof. We use exactly the same construction as in the proof of Lemma 2, noting that we only have to compute the values $T[k][m]][\ell]$, for a fixed k and $m \leq R$. These elements are saved in the matrix T_k, i.e., $T_k[m][\ell] = T[k][m][\ell]$. Clearly, this computation is done in $\mathcal{O}(\sum_{1 \leq m \leq R} \sum_{1 \leq \ell \leq n} d(\ell)) \in \mathcal{O}(n \lg n)$, where the constant hidden by the $\mathcal{O}$-notation depends on R. □

2.2. Combinatorics on Words

The following classical combinatorial results are used in this paper; for proofs and details see the handbook [15, Chapter 9] and the references therein.

Lemma 3. *Let $w \in V^n$ be a word, $PS_w = \{u | u \text{ primitive }, u^2 \text{ prefix of } w\}$, and $PR_w = \{u \mid u \text{ is primitive, } u^2 \text{ is a factor of } w\}$.*

(1) Let $u_1, u_2, u_3 \in PS_w$ be words such that $|u_1| < |u_2| < |u_3|$. Then $2|u_1| < |u_3|$. As a consequence, $|PS_w| \leq 2 \lg n$.

(2) We can compute all the pairs (i, j) such that $w[i..j] = u^2$ for some $u \in PR_w$ in $\mathcal{O}(n \lg n)$ time. Moreover, $|PR_w| \in \mathcal{O}(n)$.

The next lemma follows easily.

Lemma 4. *Let x and y be primitive words such that x^2 is a proper prefix of y^2. Then $|y| > (M-1)|x|$ where $M = \max\{p \mid x^p \text{ is a prefix of } y^2\}$. Consequently, for $w \in V^n$ there are at most $\mathcal{O}(\lg n)$ primitive words z such that z^3 is a prefix of w.*

Proof. If $|y| < (M-1)|x|$ we can apply Theorem 1 to the word x^M whose length is greater than $|x| + |y|$ and is both in $\{x\}^+$ and a prefix of the word yx, which is, at its turn, a prefix of y^2. It follows that both y and x are powers of the same word, a contradiction with the fact that x and y are primitive.

The second part of the Lemma follows immediately from the first result. □

Next result shows an important property of pseudo-repetitions, in the case when we consider non-erasing functions.

Lemma 5. *Let f be a non-erasing anti-/morphism, and x, y, z be words over V such that $f(x) = f(z) = y$. If $\{x, y\}^* x \{x, y\}^* \cap \{z, y\}^* z \{z, y\}^* \neq \emptyset$ then $x = z$.*

Proof. We give the proof only for the case when f is a morphism; a similar argument works for the case when f is antimorphism.

If $\{x,y\}^*x\{x,y\}^* \cap \{z,y\}^*z\{z,y\}^* \neq \emptyset$ then we may assume without losing generality there exists w such that $w = xw'$, $w' \in \{x,y\}^*$, and $w \in \{z,y\}^*z\{z,y\}^*$.

If z is a prefix of w, as $f(x) = f(z)$ and f is non-erasing, we get easily that $x = z$.

Assume now that $w = yzw''$ with $w'' \in \{z,y\}^*$. It is not hard to see that from $|x| \leq |y|$ and $w = xw'$ we obtain that $|x|$ is a period of y, and, thus, $y = x^\ell u$ where $\ell > 0$ and u is a prefix of x. If y and x are powers of the same word v, then, for some $k_1, k_2, k_3 \geq 0$, $x = v^{k_1}$, $y = v^{k_2}$ and $u = v^{k_3}$, so z is also a power of v. Since $f(x) = f(z)$, we conclude again that $x = z$.

Further, assume that x and y are not powers of the same word. Hence, u is a proper prefix of x, i.e., $x = uv$ for $u \neq \lambda \neq v$. Consequently, w' has a prefix of the form x^py, with $p \geq 0$, and it follows that after the first $|y|$ symbols of w both the factor vu and the factor z occur (as vu occurs after the first $|y| - |x|$ symbols of w'). Since $|vu| = |x|$ we get easily that $z = vu$. Indeed, if vu would be a proper prefix of z, we would get that $|f(z)| < |f(vu)| = |f(uv)| = |f(x)|$, a contradiction; a similar argument holds for the case when z is a prefix of vu. So, $|z| = |x|$, $y = f(z) = f(vu) = f(v)f(u)$ and $y = f(x) = f(u)f(v)$. It follows that y is a power of a primitive word t, usually called the primitive root of y, and, moreover, $per(y) = |t|$. As $|x|$ is a period of $y = x^\ell u$ we get that $|t|$ is also a period of x. Since y and x are not powers of the same word, $|y| > |x| > |t|$. Therefore, we have $y = t^k$ and $x = t^it'$, where t' is a proper prefix of t and $k > i > 0$. If $k > i + 1$, we get a contradiction from the fact that $(t^i)^{-1}t^k$ has both t^2 and $t't$ as a prefix, so t' and t are powers of the same word. Moreover, if $k = i + 1 > 2$, then we get that $t^{i+1} = f(x) = (f(t))^if(t')$. As $f(t')$ is a prefix of $f(t)$, this can only happen if t is a power of a word, a contradiction with the primitivity of t, or $f(t) = f(t')$, a contradiction with the fact that t' is a proper prefix of t. In conclusion, we have $y = tt$ and $x = tt'$. This also means that $y = xu$, that is $\ell = 1$, and $t'u = t$. Recall that in the initial decomposition of w, the word w' starts either with xx, xy, or $y = xu$; thus, w always has the prefix xxu, so the prefix $tt'tt'u = tt't^2$ as well. As a consequence of the previous fact, and of the fact that $y = t^2$ is also a prefix of w, we obtain that $|t'|$ is a period of t. Let t'' be the suffix of t, such that $|t''| = |t'|$; because $|t'|$ is a period of t we get that t'' is a cyclic permutation of t'. Since $w = yzw''$ we get that either $t''t$ is a prefix of z or vice versa. However, $|f(tt'')| = |f(tt')| = |f(x)| = |f(z)|$, so, as above, $z = t''t$. Note now that $y = t^2 = f(x) = f(t)f(t')$ and $y = t^2 = f(z) = f(t'')f(t)$. By a well known result, we obtain that $f(t) = (\alpha\beta)^q\alpha$, $f(t') = (\beta\alpha)^r$, and $f(t'') = (\alpha\beta)^r$, where $\alpha\beta$ is primitive and $r > 0$. It follows that $y = t^2 = (\alpha\beta)^{q+r}\alpha$. As t is the smallest period of y and $\alpha\beta$ is a period of y, as well, the previous relation holds either when $q+r = 2$ and $\alpha = \lambda$, or when $q+r = 1$. In the first case we get that $q = r$, so $f(t') = f(t)$, a contradiction with the fact that t' is a proper prefix of t. In the second case we get that $q = 0$, so $|f(t)| < |f(t')|$, a contradiction. This completes the analysis of the case when $w = yzw''$; we showed that this

can happen only if $x = z$, and x and y are powers of the same word.

In the case when $w = yyzw''$ for some $w'' \in \{z, y\}^*$, we can apply Theorem 1 to the prefix of length $2|y|$ of w (which is a prefix of a word from $x\{x, y\}^*$, as well) and obtain that x and y are powers of the same word. Once again, we obtain that $z = x$. □

The next lemmas provide insights to the combinatorial properties of f-repetitions, for f a general morphism, and are utilised in showing the soundness and efficiency of our algorithms. When using them, we take x to be the shorter and y the longer of the words t and $f(t)$.

Lemma 6. *Let $x, y \in V^*$ be words that are not powers of the same factor. If $w \in \{x, y\}^*$, then there exists a unique decomposition of w in factors from $\{x, y\}$.*

Proof. As x and y are not powers of the same word, it immediately follows from Theorem 1, that if w has length greater than $|xy|$, then the result holds. For shorter lengths the result is trivial as for each possible length less than $\max\{|x|, |y|\}$ the decomposition consists in catenations of the shorter word. □

Lemma 7. *Let $x, y \in V^+$ and $w \in \{x, y\}^* \setminus \{x\}^*$ be words such that $|x| \leq |y|$ and x and y are not powers of the same word. Let $M = \max\{p \mid x^p$ is a prefix of $w\}$ and $N = \max\{p \mid x^p$ is a prefix of $y\}$. Then $M \geq N$. Moreover, if $M = N$ then $w \in y\{x, y\}^*$ holds, while if $M > N$ then either it is the case that $w \in x^{M-N}y\{x, y\}^* \setminus x^{M-N-1}yxV^*$, or we have $w \in x^{M-N-1}y\{x, y\}^+ \setminus x^{M-N}yV^*$ and $N > 0$.*

Proof. The fact that $M \geq N$ is immediate. Indeed, as $w \in x^k y\{x, y\}^+$ for some $k \geq 0$, we get that $M \geq N + k$.

Assume, further, that $y = x^N y'$ and $w = x^M w'$, where y' and w' do not have x as a prefix. By Lemma 6 it follows that w has a unique decomposition in factors x and y.

When $M = N$ it is straightforward that the unique decomposition of w in factors from $\{x, y\}$ cannot start with x (otherwise we would get $k > 0$, so $M > N$, a contradiction), hence, $w \in x^{M-N}y\{x, y\}^*$.

We further analyse the case of $M > N$.

We first consider the case when x is not a prefix of y; this means that $N = 0$. As y is longer than x, it follows that there is no position i of w where both x and y occur. Thus, it is rather easy to see that $w = x^M yz$ where $z \in \{x, y\}^*$ and we get that $w \in x^{M-N}y\{x, y\}^*$ and, clearly, $x^{M-N-1}yx$ is not a prefix of w.

The more involved case is when x is a prefix of y; that is, $N > 0$. We split the discussion in two more cases.

The first case is when $|x|$ is not a period of y. Exactly as in the previous case, we get that y' must appear after x^M, so $w \in x^{M-N}y\{x, y\}^*$ and $x^{M-N-1}yx$ is not a prefix of w.

Finally, let us consider the case when $|x|$ is a period of y. Note that y is not a power of x, by hypothesis, so $y = x^N y'$ is a proper prefix of x^{N+1}. Clearly, in

the decomposition of w as the catenation of factors x and y, the word y occurs the first time after one of the prefixes x^k, with $k \leq M - N$. Let us analyse what happens when y occurs after x^k, with $k \leq M - N - 2$. In this case, we obtain that the word $(x^k)^{-1}x^M$ has length at least $(N+2)|x| > |x| + |y|$ and it is both in $\{x\}^+$ and a prefix of a word from $y\{x,y\}^*$. By Theorem 1 we get that both y and x are powers of the same word, a contradiction. Therefore, $k \in \{M - N - 1, M - N\}$.

Let us analyse the case when y appears after x^{M-N-1} in the decomposition of w in factors x and y. As y is strictly shorter than x^{N+1} and y is longer than x we get that y is followed in the above mentioned decomposition by at least one other factor x or y. As $N > 0$, we get that $x^{M-N-1}yx$ is a prefix of w. By Theorem 1 it follows that $x^{M-N}y$ is not a prefix of w, unless x and y are powers of the same word.

Similarly, when $w \in x^{M-N}y\{x,y\}^*$, we get that w cannot have both $x^{M-N}y$ and $x^{M-N-1}yx$ as prefixes. □

As a continuation of the proof above, consider the following two examples.

Example 2. *Take $x = ababa$, $y = (ababa)^3ab$, and $w = (ababa)^7ba(ababa)^2ab = x^3yy$. In this case, $M = 7$, $N = 3$, $w \in x^{M-N-1}y\{x,y\}^+ \setminus x^{M-N}y\{a,b\}^*$, and w has $x^{M-N-1}yx$ as a prefix.*

On the other hand, take $x = ab$, $y = ababa$, and $w = (ab)^5aabab = x^3yx^2$. Clearly, $M = 5$, $N = 2$, and $w \in x^{M-N}y\{x,y\}^+ \setminus x^{M-N-1}yx\{a,b\}^*$.

2.3. Data Structures

We start the presentation of this section first recalling basic facts about the data structures we use. For a word u with $|u| = n$ over $V \subseteq \{1, 2, \ldots, n\}$ we can build in linear time a suffix array structure as well as data structures allowing us to return in constant time the answer to queries "How long is the longest common prefix of $u[i..n]$ and $u[j..n]$?". We denote this by $LCPref(u[i..n], u[j..n])$, or, when the word is not ambiguous, just $LCPref(i, j)$. For more details, see [20, 17], and the references therein. Also, for the word u and an anti-/morphism f, we compute an array len with n elements defined as $len[i] = |f(u[1..i])|$, for $1 \leq i \leq n$. For f non-erasing we also compute an array inv, having $|f(u)|$ elements, such that $inv[i] = j$ if $len[j] = i$ and $inv[i] = -1$ otherwise. These computations are done in $\mathcal{O}(n)$ time.

Note the following result:

Lemma 8. *Let $w \in V^n$ be a word. We compute the values $per[i]$, the period of $w[1..i]$, for all $i \in \{1, 2, \ldots, n\}$ in linear time $\mathcal{O}(n)$. Also, we compute the values $per[i][j]$, the period of $w[i..j]$, for all $i, j \in \{1, 2, \ldots, n\}$ in quadratic time $\mathcal{O}(n^2)$.*

Proof. We only show the first part, and the second follows as a simple consequence.

One may note that this result follows from the preprocessing part of the classical Knuth-Morris-Pratt algorithm. Alternatively, a proof based in *LCPref* queries can be easily given.

Note that $per[1] = 1$ and $per[i] \leq per[j]$ for $i < j$. Consequently, to compute $per[i+1]$ we compute the minimum $j \geq per[i]$ such that $LCPref(w[1..i+1], w[j..i+1]) = i - j + 2$. Using this idea, $per[i+1]$ is computed in $per[i+1] - per[i]$ time. Thus, computing all the values $per[i]$ for $i \in \{1, 2, \ldots, n\}$ takes $\mathcal{O}(\sum_{2 \leq i \leq n}(per[i] - per[i-1]))$ time to which the time needed to construct data structures that allow us answer *LCPref* for w is added. The conclusion of the lemma follows. □

The following results can be easily shown.

Remark 1. *A number p is a period of w if and only if $LCPref(1, p+1) = |w| - p$.* ◁

Lemma 9. *Given two words $x, y \in V^*$, for which we have LCPref-data structures, and an array T of $|V|$ integers, we can decide in $\mathcal{O}(per(x))$ time the existence of an anti-/morphism f of length-type T such that $f(x) = y$.*

Proof. Let us discuss the case of f being a morphism, as the other one is similar. We basically define $f(x[i])$ iteratively, for i from 1 to $per(x)$. Define ℓ_i as the length of the image of $x[1..i]$ and $\ell_0 = 1$; all these numbers can be computed in $\mathcal{O}(per(x))$ time. Now, the image of $f(x[i])$ should be $y[\ell_{i-1}..\ell_i]$. We keep track of the way we defined the image of each letter of $\mathrm{alph}(x)$ (as the pair (ℓ, ℓ'), provided that the respective letter is mapped to $y[i..j]$) while going through the letters of x, and, when we identify the image of another letter $x[i]$, we check whether we already knew the image of that letter, and if yes, whether the old definition is the same as the newly identified one. Each of these checks can be done in constant time using *LCPref* queries, so the total time we need to check whether $x[1..per(x)]$ can be mapped to the corresponding prefix $y[1..m]$ of y, where $m = T(x[1..per(x)])$, is $\mathcal{O}(per(x))$. Further, we check by a *LCPref* query whether m is a period of y. If yes, then x can be mapped to y. □

Remark 2. *Let $w \in V^n$ be a word. First, for $1 \leq i \leq n$, we construct the list of pairs (i, j) such that $w[i..j] = u^2$ with u primitive. By Lemma 3 this takes $\mathcal{O}(n \lg n)$ time. Further, we put together all these lists and sort the resulting list according to the lexicographical order of the words encoded by the pairs ((i, j) encodes $w[i..j]$ and we just choose some order on the alphabet of w). This can be done in $\mathcal{O}(n(\lg n)^2)$ time, using LCPref queries to compare the words encoded by two pairs. Next, we construct in $\mathcal{O}(n \lg n)$ time the set PR_w, which has $\mathcal{O}(n)$ elements by Lemma 3. Each element u of PR_w is encoded by the pair (i, j) such that $w[i..j]$ is the first occurrence of u^2 in w. Moreover, while computing PR_w, we store for each pair (i', j') such that $w[i'..j'] = u^2$ for some $u \in PR_w$ the actual pair (i, j) used to encode u in PR_w.* ◁

3. Solution of Problem 1

3.1. A general solution

We first assume that f is a morphism and let $n = |w|$. We construct in linear time the word $wf(w)$ of length $m = n + |f(w)|$ (which is in $\mathcal{O}(n)$); note that the

length of $wf(w)$ (hence, the constant hidden by the $\mathcal{O}$-notation) depends on the fixed morphism f. Moreover, we build in $\mathcal{O}(n)$ time data structures enabling us to answer *LCPref* queries for $wf(w)$.

First we present in Algorithm 1 a procedure $Check(w[s_0..n], x, y)$ that will return true if and only if the factor $w[s_0..n]$ of the word w can be expressed as the repeated catenation of two other factors x and y of w or $f(w)$ (given by their starting position in $wf(w)$ and their length). This procedure works under the assumption that $|x| \leq |y|$.

Algorithm 1 $Check(w[s_0..n], x, y)$: decides whether $w[s_0..n] \in \{x, y\}^*$

1: **if** $s_0 = n + 1$ **then return true**;
   {Halt and decide that $\lambda = w[s_0..n] \in \{x, y\}^*$.}
2: Let $\ell = |x|$ and $\ell' = |y|$;
3: $M = \max\{p \mid x^p \text{ prefix of } w[s..n]\}$;
4: $N = \max\{p \mid x^p \text{ prefix of } y\}$;
5: **if** $w[s_0..n] = x^M$ **then return true**;
6: **else if** $x^{M-N}y$ occurs at position s **then**
7: $\quad s = s_0 + (M - N)\ell + \ell'$;
8: $\quad Check(w[s..n], x, y)$;
9: **else if** $M > N$ and $x^{M-N-1}yx$ occurs at position s **then**
10: $\quad s = s_0 + (M - N - 1)\ell + \ell'$;
11: $\quad Check(w[s..n], x, y)$;
   {By Lemma 7, $w[s_0..n]$ should have either $x^{M-N}y$ or $x^{M-N-1}yx$ as a prefix. Furthermore, if $x^{M-N-1}yx$ occurs at position s_0, we shall check whether $w[s_0..n] \in x^{M-N-1}y\{x, y\}^+$.}
12: **if** any of the above checks holds **then return true**
13: **else return false**
   {If none of the above checks holds, we have $w[s_0..n] \notin \{x, y\}^+$.}

Let us now analyse the complexity of the procedure $Check(w[s_0..n], x, y)$. As x and y are factor of w or $f(w)$ we can check in constant time using *LCPref* queries for $wf(w)$ whether any of these factors occurs at a certain position in $w[s_0..n]$. The computation in each of the steps $2 - 10$ of the algorithm can be executed in constant time using the data structures we already constructed. Indeed, for some $s_0 \leq n$, we can compute the largest s' such that $w[s_0..s']$ is a power of x in constant time as follows. In the worst case, $s' = s_0 - 1$, or, in other words, $w[s..\ell] = \lambda$, when x does not occur at position s_0. Otherwise, s' is the largest number less than or equal to $LCPref(w[s_0..n], w[s_0 + |x|..n])$ such that $s' - s_0 + 1$ is divisible by $|x|$. This strategy is used in steps 3 and 4 to compute M and N. The verification from step 5 takes clearly constant time: we just check whether $n - s_0 + 1 = M|x|$. Moreover, steps 6 and 9 can also be implemented in constant time using *LCPref* queries; indeed, we know that x^{M-N} occurs at position s_0, and then we just have to check whether y occurs at position $s_0 + (M - N)|x|$ by a *LCPref* query, for step 6, or, respectively, whether yx occurs at position $s_0 + (M - N - 1)|x|$ by two *LCPref* queries, for

step 9. The complexity of the whole procedure is $\mathcal{O}(\frac{n-s_0}{|y|})$ since the recursive calls are executed with the parameter $w[s..n]$ where $s \geq s_0 + |y|$.

Using the previously described data structures and the procedure *Check*, Algorithm 2 tests whether there exists a prefix $t = w[1..i]$ such that w can be expressed as the repeated catenation of t and $f(t)$.

Algorithm 2 $Test(w, f)$: decides whether w is an f-repetition

1: Test whether there exists a word x such that $w = x^k$, with $k \geq 2$. If yes, then we halt and decide that w is an f-repetition. Otherwise, go to step 2. {If the result of the test is positive we decide that w is a (trivial) f-repetition. The algorithm continues for w primitive.}
2: **for** $t = w[1..i]$ with $i < n$, $1 \leq len[i]$, and t and $f(t)$ not powers of a $z \in V^*$ **do**
3: Set $x = t$ and $y = f(t)$ if $i \leq len[i]$ or $x = f(t)$ and $y = t$, otherwise;
4: Set $s = i + 1$;
5: **if** $Check(w[s..n], x, y)$=**true then** Halt and decide that w is an f-repetition
6: **end for**
7: Halt and decide that w is not an f-repetition.

Following the comments inserted in its description, it is not hard to see that Algorithm 2 is sound. In the following, we compute its complexity. The step where we test whether w is a repetition takes $\mathcal{O}(n)$ time, as it can be done by locating the occurrences of w in ww. Further, the iterative process in steps $2-6$ is executed for each prefix $w[1..i]$ of w, and during each iteration the algorithm makes at most $\mathcal{O}(\lfloor \frac{n}{|y|} \rfloor)$ steps, as s can take at most $\lfloor \frac{n}{|y|} \rfloor$ different values (in the *Check* procedure). Since $|y| \geq i$, the overall time complexity of the algorithm is upper-bounded by $\mathcal{O}(\sum_{1 \leq i \leq n} \lfloor \frac{n}{i} \rfloor)$. Thus, the time complexity of Algorithm 2 is $\mathcal{O}(n \lg n)$. As a side note, in the case when f is erasing, $w \in t\{t, f(t)\}^+$ for some t with $f(t) = \lambda$ if and only if $w \in \{t\}^+$; in other words, w should be a repetition. Hence, we run the iterative process starting in step 2 only for prefixes $w[1..i]$ with $len[i] \geq 1$.

The case when f is an antimorphism is similar. We build the same data structures for the word $wf(w)$, and proceed just as in the former case. As the single difference, now we have $w[s + 1..s + len[i]] = f(w[1..i])$ if and only if $LCPref(s + 1, m - len[i] + 1) = len[i]$, where $m = |wf(w)|$.

When f is uniform we can easily obtain a more efficient algorithm. In this case, $|t|$ divides n, so we only need to run the iterative instruction for the prefixes $w[1..i]$ of w with $i \mid n$. Hence, the total running time of the algorithm is, in this case, upper-bounded by $\mathcal{O}(\sum_{i|n} \frac{n}{i}) \in \mathcal{O}(n \lg \lg n)$, by Lemma 1.

3.2. A linear time solution for the case when f is uniform

For the case when f is uniform we obtain an even faster solution for Problem 1 by using some more intricate precomputed data structures in order to

speed-up Algorithm 2. To this end, we analyse again the computation performed by Algorithm 2 on an input word w.

The main phase of the algorithm is the following. For a prefix $t = w[1..i]$ of w with $i \mid n$ we run the *Check* procedure that extends iteratively a prefix $w[1..s-1]$, where $s \geq i+1$, of the word w such that the newly-obtained prefix is in $t\{t, f(t)\}^*$. However, at each iteration the prefix is extended with a word of the form $t^k f(t)$, with $k \geq 0$. As k can be equal to 0, we can only say that the number of iterations of the cycle is upper-bounded by $\frac{n}{|f(t)|} \leq \frac{n}{|t|}$. Here we plug in our speed-up strategy: we try to extend the prefix in each of the iterations of the *Check* procedure with a word that belongs to $\{t, f(t)\}^\alpha$ for some fixed number α that depends on n, but not on t. In this way, we upper bound the number of iterations of the cycle by $\frac{n}{\alpha|t|}$, and the overall complexity of the algorithm by $\mathcal{O}(\frac{n \lg \lg n}{\alpha})$. Finally, in order to obtain an algorithm solving Problem 1 in linear time, we choose $\alpha = \lceil \lg \lg n \rceil$.

Let $R = |f(a)|$, for $a \in \text{alph}(w)$; as f is uniform, the definition of R does not depend on the choice of a from V, and we also have $R = \frac{|f(u)|}{|u|}$, for every $u \in V^+$. Moreover, take $r_t = \max\{\ell \mid t^\ell \text{ prefix of } f(t)\}$. Clearly, $r_t \leq R$ and we can assume without losing generality that $\alpha > R$. Indeed, this holds for $n > 2^{2^R}$, which is the case when we want to optimise Algorithm 2; for smaller n the algorithm runs in constant time $\mathcal{O}(1)$, as $n \lg \lg n \leq R 2^{2^R}$ and R is constant (f being fixed).

It only remains to show how we can implement efficiently the above mentioned extension of the prefix. First, we note that there exists a constant C such that $\frac{(\lg n)^4 (\lg \lg n)^2}{n} \leq C$ for all n. Therefore, running the original form of Algorithm 2 for the prefixes t of w with $|t| > \frac{n}{(\lg n)^2 \lg \lg n}$ and $|t| \mid n$ (that is, at most $(\lg n)^2 \lg \lg n$ prefixes) takes $\mathcal{O}(n)$ time. Hence, from now on, we only consider prefixes t such that $|t| \mid n$, $|t| < \frac{n}{(\lg n)^2 \lg \lg n}$, and, assuming that the input word is not a repetition, t and $f(t)$ are not powers of the same word.

Now consider a prefix t, as above. There are $2^\alpha \in \mathcal{O}(\lg n)$ words in $\{t, f(t)\}^\alpha$. Every such word can be encoded by a bit-string of length α: each occurrence of t is encoded by 0 and an occurrence of $f(t)$ by 1. Denote these bit-strings $v_1, v_2, \ldots, v_{2^\alpha}$, and let $\overline{v}_i$ be the word encoded by v_i, for each $1 \leq i \leq 2^\alpha$. Further, for a bit-string v_ℓ we can determine by binary search two values b_ℓ and e_ℓ such that all the suffixes contained in the suffix array of w between the positions b_ℓ and e_ℓ have the word $\overline{v}_\ell t^{r_t}$ as a prefix. From Theorem 1, applied for two strings $\overline{v}_i t^{r_t}$ and $\overline{v}_j t^{r_t}$ with $i \neq j$, using the facts that t and $f(t)$ are not powers of the same word and r_t is the maximal power of t occurring as a prefix of $f(t)$, we get that the intervals $[b_i, e_i]$ and $[b_j, e_j]$ are disjoint. The time needed to compute these values for each ℓ is $\mathcal{O}(\lg n \lg \lg n)$, as a comparison between the word $\overline{v}_\ell t^{r_t}$ and a suffix of w can be done in $\mathcal{O}(\lg \lg n)$ by looking at the encoding v_ℓ and the string t^{r_t} (a prefix of $f(t)$) and, consequently, comparing only the factors of length $|t|$ and $|f(t)|$ of $\overline{v}_\ell t^{r_t}$ with those of the words from the suffix array. Thus, the time needed to compute b_ℓ and e_ℓ for all ℓ is $\mathcal{O}((\lg n)^2 \lg \lg n)$. Next, we construct a set E_t containing the values e_ℓ ordered increasingly, while

keeping track for each e_ℓ of the corresponding values of ℓ and b_ℓ. Note that E_t contains $\mathcal{O}(\lg n)$ integers from $\{1, 2, \ldots, n\}$.

We need one more result before concluding this preprocessing phase. We want to store a static set $S \subseteq \{1, 2, \ldots, n\}$ so that finding the successor in S of a given $x \in \{1, 2, \ldots, n\}$ takes constant time. Thus, we use a static d-ary tree of depth 2, where $d = \lceil n^{0.5} \rceil$, so that the whole tree has n leaves corresponding to different values of x. We mark all leaves corresponding to the elements of S, and remove all nodes with no marked leaf in the corresponding subtree. At each remaining inner node v we store a table of length d where for each child of v (both remaining and already removed) we store the successor of the rightmost leaf in its corresponding subtree. The total size of the structure is $\mathcal{O}(|S|n^{0.5})$ and we can construct it in the same time if we start with an empty S and add its elements one by one, creating new inner nodes when necessary. Furthermore, using the tables we can find the successor of any x in $\mathcal{O}(1)$ time by traversing the path from the root of the tree towards x as long as the nodes exist and taking the minimum of the successors stored for these nodes. If we store each E_t in this way, then the query time is constant and the total construction time and space is in $\mathcal{O}(d(n)n^{0.5} \lg n) \subseteq \mathcal{O}(n)$, where the final upper bound follows from Lemma 1.

By the previously given explanations, this entire preprocessing takes linear time. We now use it to solve in linear time Problem 1.

Assume now that we want to check whether $w \in t\{t, f(t)\}^*$ for some prefix t of w as above and the word $w[s..n]$ with $s \leq n - (\alpha + r_t)|t| + 1$. There is at most one ℓ such that the index i_s of $w[s..n]$ in the suffix array of w is between b_ℓ and e_ℓ (that is, $\overline{v}_\ell t^{r_t}$ is a prefix of $w[s..n]$). This ℓ can be found, if it exists, in $\mathcal{O}(1)$ using the precomputed data structures (i.e., the sets E_t, organised as described above): return the value ℓ such that e_ℓ is the minimal element of E_t greater than or equal to i_s and $b_\ell \leq i_s$. Then, we repeat the procedure for the word $w[s'..n]$ where $w[s..s'-1] = \overline{v}_\ell$, but only if $n - s' + 1 \geq (\alpha + r_t)|t|$ or $s' = n+1$. If $n - s' + 1 \leq (\alpha + r_t)|t|$ we run the processing of the original algorithm. Clearly, this process takes $\mathcal{O}(\frac{n}{\alpha|t|} + 2\alpha)$ steps for each t, so the complete algorithm runs in $\mathcal{O}(n)$ time.

We only have to show now that this process works correctly, i.e., it decides whether $w \in t\{t, f(t)\}^+$. The soundness is proven by the following remark. If $w[s..n]$ starts with $\overline{v}_j t^{r_t}$ for some $j \leq 2^\alpha$, then it is enough to consider in the next iteration only the word $w[s + |\overline{v}_j|..n]$, and no other word $w[s + |\overline{v}_k|..n]$ where $k \leq 2^\alpha$ such that $\overline{v}_k$ is also a prefix of $w[s..n]$. Indeed, if there exists v_k leading to a solution, we get a contradiction with either the fact that r_t is the maximal power of t occurring as a prefix of $f(t)$, or with the fact that t and $f(t)$ are not powers of the same word.

To conclude, this implementation of Algorithm 2 runs in optimal linear time for f uniform.

3.3. Summary

The following theorem gives an account of the obtained results.

Theorem 2. *Let $f : V^* \to V^*$ be an anti-/morphism and $w \in V^n$ be a given word.*

(1) We can decide whether $w \in t\{t, f(t)\}^+$ in $\mathcal{O}(n \lg n)$ time. ◁

(2) In the case when f is uniform we only need $\mathcal{O}(n)$ time. ◁

3.4. A generalisation

The more general problem of testing whether there exists a word t such that for a fixed anti-/morphism f we have that $w \in \{t, f(t)\}\{t, f(t)\}^+$ is also worth considering. Solving this problem seems to require a different strategy than the one in Algorithm 2. There we take prefixes t of w, which determine uniquely $f(t)$, and check whether $w \in t\{t, f(t)\}^*$. Here, a prefix y does not determine uniquely, in general, a factor x such that $f(x) = y$, so more possibilities have to be considered when checking whether there exists t such that $w \in f(t)\{t, f(t)\}^*$. However, the cases of f non-erasing and uniform anti-/morphisms have solutions based on results in the line of Lemmas 5 and 6, leading to similar complexities as for Problem 1. The case of erasing anti-/morphisms is solved by a more involved algorithm, based on both combinatorics on words and number theoretic insights.

Theorem 3. *Let $f : V^* \to V^*$ be an anti-/morphism and $w \in V^n$ be a given word.*

(1) We can decide whether $w \in \{t, f(t)\}\{t, f(t)\}^+$ in $\mathcal{O}(n(\lg n)^2)$ time.

(2) In the case when f is non-erasing we solve the problem in $\mathcal{O}(n \lg n)$ time.

(3) In the case when f is uniform we solve the problem in $\mathcal{O}(n)$ time.

Proof. As in the previous case, we only present the results for f morphism, as the same reasoning works for the case of antimorphisms.

We first consider the case when f is non-erasing.

Initially, we test in $\mathcal{O}(n \lg n)$ time whether there exists a prefix t of w such that $w \in t\{t, f(t)\}^*$, using Algorithm 2. If yes, we halt. Otherwise, we have to decide whether there exist the words y and x such that $w \in y\{x, y\}^*$ and $f(x) = y$. Clearly, if such words exist we have $|x| \leq |y|$.

Let $y = w[1..i]$ be a prefix of w and take j to be the rightmost position of w such that $w[1..j] = y^\ell$, for some $\ell \geq 1$. Since w is not in $y\{y, f(y)\}^+$, it follows that $j < n$. Once again, note that j can be determined in constant time using *LCPref* queries. Now, if there exists a word x such that $w \in y\{x, y\}^+$ and $f(x) = y$, then $w \in y^k x\{x, y\}^*$. It is important to note that for every $k \leq \ell$ there exists at most one possible x occurring at position $k|y| + 1$ in w such that $f(x) = y$.

Assume, for the sake of contradiction, that $k \leq \ell - 2$ and there exists x occurring at position $k|y| + 1$ that fulfils the above conditions. We have by Theorem 1, applied for $\alpha = y^{\ell-k}$, $\beta = (y^k)^{-1}w$, $u = y$, and $v = x$, that both x and y are powers of another word u, with $|u| \leq |x|$. Thus, we would get

that w is a power of u. The latter is a contradiction with our assumption that $w \notin t\{t, f(t)\}^*$ for any of its prefixes t.

There remain two cases to be analysed: $k = \ell - 1$ and $k = \ell$. In each of the cases, we first determine x, using *LCPref* queries. To this end, recall that for each position j of the word w there exists a unique word x beginning at position $j + 1$ such that $f(x) = y$, although, in general, there may be more than one string whose image through f is y. To actually find x we proceed as follows. First, we verify whether $LCPref(n + len[ki] + 1, 1) \geq |y|$; that is, we verify whether $w[ki + 1..n]$ begins with a word x such that $f(x)$ has the prefix y. If the answer is positive, we check whether $inv[len[ki]+i]$ is defined and when this holds we conclude that $x = w[ki+1..inv[len[ki]+i]]$. If at least one of the above checks does not return a positive answer, then there is no prefix of $w[ki + 1..n]$ that has the image through f equal to $w[1..i] = y$. For each $k \in \{\ell, \ell - 1\}$, once we determined x we check whether $w[ki + 1..n]$ is in $x\{x, f(x)\}^+$ just as in the Algorithm 2. The time complexity of this algorithm is, by arguments similar to the above, $\mathcal{O}(n \lg n)$. Exactly as in the case of the solution for the original problem, this complexity can be decreased to $\mathcal{O}(n)$ when f is uniform.

Finally, we consider the general case, when f can erase letters. The solution, in this case, is much more involved. So, given a word $w \in V^*$ of length n, and a general morphism f, we want to check whether $w \in f(t)\{t, f(t)\}^+$ for some factor t of w. For simplicity, we may assume that w is not in $t\{t, f(t)\}^+$ for any prefix t.

Fact 1. *Lemma 7 and the cycle from steps* $2-6$ *of Algorithm 2 give us a method to decide, for a fixed factor* t *of* w*, whether* $w \in \{t, f(t)\}^*$ *in* $\mathcal{O}\left(\frac{n}{\max\{|t|,|f(t)|\}}\right)$ *time.* ◁

Fact 2. *For* $k \in \{1, 2\}$*, we can decide whether the word* w *is in* $y^k\{x, y\}^*$ *for some factors* x *and* y *of* w *such that* $f(x) = y$ *in* $\mathcal{O}(n \lg n)$ *time.*

Proof. Indeed, we try to guess $y = f(x)$. More precisely, we iterate over all of its possible lengths, and we locate the position where x should start. Having fixed $f(x)$ and the place x should start, we iterate through all places x could end at. There might be more than one such place, as some letters could be mapped by f to the empty word. Hence, for each length of $f(x)$ we iterate through a range of possible endpoints of x, and for each of them apply Fact 1. Observe that all those ranges are in fact disjoint, as whenever we increase $|f(x)|$, we must also move the endpoint of x to the right (otherwise the image of x will actually be smaller then the prefix we chose as $f(x)$). Moreover, the value of $\max\{|f(x)|, |x|\}$ strictly increases after each application of the method in Fact 1. Thus the total complexity is upper-upper-boundedbounded by $\sum_{i=1}^{n} \frac{n}{i} \in \mathcal{O}(n \lg n)$. □

By Fact 2, we only have to check whether $w \in (f(t))^3\{t, f(t)\}^+$ for some factor t of w. Then $f(t)$ is some power of a primitive word u such that u^3 is a prefix of w. Consider all such primitive words u and denote them by $u_1, u_2, \ldots, u_k$, where $|u_i| < |u_{i+1}|$. From Lemma 4 the number k of such different

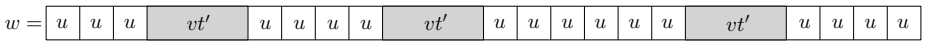

Figure 1: Unique decomposition of w into u and vt'.

possible u's is at most $\mathcal{O}(\lg n)$. We can identify all of them in $\mathcal{O}(n)$ time, by Lemma 8.

We check each of these primitive words u separately.

Consider the largest power u^m being a prefix of w. We want to check all t's such that $f(t)$ is a power of u and t starts after the i-th occurrence of u for some $i \in \{1, 2, \ldots, m\}$. Let v be the factor of length (at most) $|u|$ occurring in w after the prefix u^m. First we iterate through every t being a prefix of u^m, and for each of them apply Fact 1 to decide whether $w \in \{t, f(t)\}^*$. This takes $\mathcal{O}(n \lg n)$ time, and from now on we can assume that t ends after the prefix u^m. Further, assume that t ends after the i-th letter of v. If we fix the endpoint, we can iterate through all possible starting points, and apply Fact 1 for every possibility. For a fixed i the required time is $\sum_{0 \le j \le m} \frac{n}{i+j|u|} \in \mathcal{O}(\frac{n}{|u|} \lg n)$. Summing over all $i \le |v| \le |u|$, we get $\mathcal{O}(n \lg n)$.

From now on we can assume that t ends after v, and $|v| = |u|$.

Fact 3. *If $f(u)$ is not a power of u, for each fixed t' we have at most one single value for i such that $f(t) = f(u^i v t')$ is a power of u. Moreover, we can compute the values of i for all the possible t' in $\mathcal{O}(n)$ time, once u is fixed.*

Proof. Indeed, assume that there are two different values i, namely i_1 and i_2 with $i_1 < i_2$. Then $(f(u))^{i_2 - i_1}$ is a power of u. Hence both u and $f(u)$ are powers of the same word, and, because u is primitive, $f(u)$ is in fact a power of u, which is a contradiction.

Now consider the question of finding this unique value for i. During a preprocessing stage, we iterate through all possible values of i. For each of them we take the image through f of the corresponding suffix of w, and compute the largest power of u which is its prefix. This can be done in constant time using *LCPref* queries on $f(w)$. Next we iterate through all sufficiently large powers, and for each of them mark a range of t'. Because of the above reasoning, any t' is considered at most once. □

If $f(u)$ is not a power of u, we apply the above result and, using Fact 1, we check each generated value of i. Again, the time needed for each choice of t is $\frac{n}{|t|}$, which is upper-bounded by $\mathcal{O}(\frac{n}{|t'|})$. Summing up, the time needed to check all the possible choices for t is upper-bounded by $\mathcal{O}(n \lg n)$. Moreover, we can do this process for all possible u with $u \neq f(u)$, clearly, in $\mathcal{O}(n(\lg n)^2)$.

Now we can consider the case when $f(u) = u^\alpha$ for some $\alpha \ge 0$; the value of α can be found by looking at the word $wf(w)$ for which we already constructed suffix arrays and *LCPref* data structures.

As in the previous case, we fix the place t ends at, and try to consider all $i \le m$ such that $t = u^i v t'$. Because $u \neq v$, w has at most one decomposition into copies of vt' and u, see Figure 1; if $w \in \{t, f(t)\}^*$ then in our decomposition

each factor vt' is the suffix of an occurrence of t in the decomposition of w in factors t and $f(t)$. Moreover, such a decomposition can be found in $\mathcal{O}(\frac{n}{|vt'|})$ time using *LCPref* queries.

Let u^{l_j} be the (maximal) power of u placed just before the j-th occurrence of vt', for $j \in \{1, 2, \ldots, k\}$, and $u^{l_{k+1}}$ be the power of u occurring at the end of w (note that $k \leq \frac{n}{|vt'|}$). Clearly, $l_1 = m$. Now, observe that $f(vt')$ must be a power of u, thus, let $f(vt') = u^{\beta}$; β can be again found in constant time using the structures we already have for the word $wf(w)$. Further, we look for $i \leq \min(l_1, \ldots, l_j)$ such that the following system of equations holds:

$$\alpha i + \beta \mid l_{k+1}, \qquad \alpha i + \beta \mid l_j - i \qquad \forall_{j=1,2,\ldots,k} \tag{3}$$

We can transform (3) to get the following form:

$$\alpha i + \beta \mid l_{k+1}, \quad \alpha i + \beta \mid l_1 - i, \quad \alpha i + \beta \mid l_j - l_1 \qquad \forall_{j=2,3,\ldots,k}. \tag{4}$$

The first step of the transformation clearly requires at most $\mathcal{O}(k)$ time.

The idea to find the possible values of i that fulfil the above system is to note that considering each point where t ends (so each possible value for vt') separately seems a bit time consuming. More precisely, for a fixed u it seems that we can use the information we already computed regarding a certain shorter value of vt' for the new longer values of vt' that have to be considered.

So, for a fixed u we first consider the shortest vt' such that $f(vt')$ is a power of u, i.e., $f(vt') = u^{\beta}$. Then, as above, we decompose w into occurrences of u and vt' and define the numbers l_j, $j \geq 1$, as described already. Note that each factor vt' in this decomposition corresponds to a suffix of the initial value of t that we should try for a fixed u; then, when we try new values for t, each factor vt' in the decomposition corresponds to a factor of each t that appears in a decomposition of w in factors t and $f(t)$.

The case when vt' is a suffix of w, that is, $k = 0$, can be solved in constant time, and no other longer factors t should be considered, so in the following we deal with the case $k \geq 1$. Let us assume that l_p is the smallest of all the numbers l_j with $k+1 \geq j \geq 2$. The first case to be considered is when $p \neq k+1$. Moreover, the case $i = 0$ (so $t = vt'$) can be treated separately in a naive manner, in $\mathcal{O}(\frac{n}{|vt'|})$ time, so let us assume that we look for $i > 0$. Clearly, the interesting case is when $f(t) = f(u)^i f(vt')$ is not the empty word (otherwise, the check is trivial, again).

It is clear that if $l_p \leq 3$ we can easily check in $\mathcal{O}(k) \subseteq \mathcal{O}(\frac{n}{|u|})$ time which of the divisors i of $l_p - l_1$ fulfil (4). Therefore, let us consider in the following the case when $l_p > 3$. Next, we investigate in which way vt' can be extended into u^{l_p} to obtain a new value for t. If t is obtained by appending to vt' a factor whose length is not divisible with $|u|$ we either get that u is not primitive (when the new t ends strictly before the last u factor of u^{l_p}) or that there should be more than one decomposition of w in u and vt' (when the new t ends inside the last u factor of u^{l_p}). So, a new t can only be obtained from vt' by appending a power of u. We will consider each possible such extensions of vt' one by one;

when a new u is added to the current value of t then l_p decreases by a unit, so $l_p - l_1$ also decreases with a unit. This suggests that instead of considering as a possible value of $\alpha i + \beta$ the divisors of $\gcd(l_{k+1}, l_2 - l_1, \ldots, l_k - l_1)$ obtained for each possible value of t and check for each of them whether (4) is fulfilled, we can consider all the divisors of the current $l_p - l_1$, compute the i corresponding to each of them, check (4), then decrease $l_p - l_1$ by one and repeat the procedure. This means that we will have to consider the divisors of all numbers less or equal to the initial $l_p - l_1$ obtained for the initial value of vt', and check for all of them whether (4) holds. By Lemma 1, there are $\mathcal{O}(l_p \lg l_p)$ such divisors, where l_p in this last formula is the initial one, obtained for the shortest value of vt'. Hence, the total time to do the aforementioned check is $\mathcal{O}(k l_p \lg l_p)$, which means that we need at most $\mathcal{O}(\frac{n}{|u|} \lg n)$ time for a fixed u, as $k \leq \frac{n}{l_p |u|}$.

The case when $p = k + 1$ is very similar. Here, the only difference is that we should consider the divisors of $l_p = l_{k+1}$ instead of the divisors of $l_p - l_1$, but the rest of the argument follows in the same manner. The complexity of the processing is again $\mathcal{O}(\frac{n}{|u|} \lg n)$.

Summing this up for all possible choices of the primitively-rooted square prefix u^2 of w, and using Lemma 3, we get that the total processing we perform in this case runs in $\mathcal{O}(n \lg n)$ time.

This concludes our proof of Theorem 3: it can be decided whether there exists t such that $w \in \{t, f(t)\}^+$ in $\mathcal{O}(n(\lg n)^2)$ time. □

4. Solution of Problem 2

Recall that our approach to solve the first question of Problem 2 is based on constructing, for the input word w, data structures that enable us to obtain in constant time the answer to queries

$$rep(i, j, \ell)\text{: ``Is there } t \in V^* \text{ such that } w[i..j] \in \{t, f(t)\}^{\ell}?\text{''}$$

for all $1 \leq i \leq j \leq |w|$ and $1 \leq \ell \leq |w|$. Moreover, a solution for the second question is derived directly from this strategy: we only need to construct similar data structures, that allow us to answer, this time, queries $rep(i, j, \ell)$ for a single ℓ, given as input of the problem together with w.

4.1. The case of erasing morphisms

We start by presenting the solution for the first question of the problem. Given an arbitrary anti-/morphism f and a word w of length n, we can construct the aforementioned data structures in $\mathcal{O}(n^{3.5})$ time. More precisely, we construct an oracle-structure that already contains the answers to every possible such query.

We only give an informal description of our construction. Assume that $|w| = n$. The idea is to compute the $n \times n \times n$ array M such that $M[i][j][k] = 1$ if there exists a word t with $w[i..j] \in \{t, f(t)\}^k$, and $M[i][j][k] = 0$, otherwise. We proceed as follows.

Let i be a position in w. We first consider the prefixes t of $w[i..n]$ such that t and $f(t)$ are not powers of the same word. Observe that, for such a prefix t of $w[i..n]$ with $t \neq \lambda$, $f(t) \neq \lambda$ and $j > i$, there is at most one positive integer k such that $w[i..j] \in t\{t, f(t)\}^{k-1}$. The set of these prefixes is partitioned in $n^{0.5}+1$ sets $S_{i,\delta} = \{t \mid |f(t)| = \delta\}$, for $1 \leq \delta \leq n^{0.5}$, and $S_i = \{t \mid |f(t)| > n^{0.5}\}$; note that some of these sets may actually be empty. Further, for each δ, we compute $f_{i,\delta} = \max\{k \mid x^k \text{ is a suffix of } w[1..i], |x| = \delta\}$.

We first deal with the case when $t \in S_i$, for $1 \leq i \leq n$. We compute for each j the number k such that $w[i..j] \in t\{t, f(t)\}^{k-1}$; this can be done in constant time (for each j) using *LCPref*-queries, as in the previous algorithms. More precisely, for some j we only need to look at the corresponding value for $j - |t|$ and $j - |f(t)|$, increase them with 1 (if they are defined) and store as the value corresponding to j the one obtained from $j - |t|$ if t occurs as a suffix of $w[i..j]$, or the one corresponding to $j - |f(t)|$ if $f(t)$ occurs as a suffix of $w[i..j]$ (due to Lemma 6, at most one case holds); if none of these values was defined, or neither t nor $f(t)$ occurs as a suffix of $w[i..j]$, the value corresponding to j remains undefined. This entire process takes linear time. Then, for j such that $w[i..j] \in t\{t, f(t)\}^{k-1}$ and all $k' \in \{0, 1, \ldots, f_{i,\delta}\}$, where $\delta = |f(t)|$, we set $M[i - k'\delta][j][k + k'] = 1$.

It is not hard to see that for $\delta > n^{0.5}$ we have $f_{i,\delta} < n^{0.5}$, so the process described above takes $\mathcal{O}(n^{0.5})$ time for each j. Now, we repeat the process for all $i \in \{1, 2, \ldots, n\}$ and all prefixes t from S_i and discover all the factors $w[i'..j']$ and numbers k such that $w[i'..j'] \in \{f(t), t\}^k$, with $|f(t)| > n^{0.5}$. The time needed to do the computations described above is $\mathcal{O}(n^{3.5})$.

Further, we consider the case of the words in the sets $S_{i,\delta}$, for some fixed $\delta \leq n^{0.5}$ and all $1 \leq i \leq n$. For each i, for each t in $S_{i,\delta}$, and for each j, we compute and put the pairs (i, k) such that $w[i..j] \in t\{t, f(t)\}^{k-1}$ in a list R_j^δ. This takes roughly $\mathcal{O}(n^3)$ time. Note that the number of elements of the list R_j^δ is also bounded by n^2, as for each i we have a unique decomposition of $w[i..j]$ in k parts, starting with a prefix t.

Now, for each j (and, recall, that δ is fixed), we build an $n \times n$ matrix T_j^δ, initially with all the entries set to 0. Next we partition this matrix in *diagonal arrays* obtained as follows: for ℓ from 1 to n and for p from 1 to n, if the element $T_j^\delta[\ell][p]$ is not stored already in a diagonal array, we construct a new diagonal array that stores the elements

$$T_j^\delta[\ell][p], T_j^\delta[\ell - \delta][p + 1], \ldots, T_j^\delta[\ell - d\delta][p + d],$$

for $0 \leq d < \frac{\ell}{\delta}$. While constructing this matrix we can keep track for each element of the array it belongs to. This procedure takes, clearly, $\mathcal{O}(n^2)$ time. These arrays partition the elements of the matrix T_j^δ so the total number of their elements is n^2.

To continue, for each element (i, k) of the list R_j^δ, we check in which diagonal array (i, k) is and memorise that we should mark (i.e., set to 1) in this array the consecutive elements

$$T_j^\delta[i][k], T_j^\delta[i - \delta][k + 1], \ldots, T_j^\delta[i - f_{i,\delta}\delta][k + f_{i,\delta}].$$

This, again, can be done in $\mathcal{O}(n^2)$ time, as we only need to memorise the first and the last of these elements (called, in the following, margins). When we are done, we have to mark r_d groups of consecutive elements in each diagonal array d, where $\sum_d r_d \in \mathcal{O}(n^2)$. To do the marking we sort the margins of the groups associated with each diagonal array, with the counting sort algorithm, and then traverse each array, keeping track of how many groups contain each of its elements, and mark the elements appearing in at least one group. Sorting the lists of intervals takes $\mathcal{O}(\sum_d r_d) = \mathcal{O}(n^2)$ time, and, thus, the marking takes $\mathcal{O}(n^2)$ time in total. Once the elements of all groups are marked, for all i and k we set $M[i][j][k] = 1$ if and only if $T_j^\delta[i][k] = 1$.

The overall complexity of the computation described above for a fixed δ is $\mathcal{O}(n^3)$. As we iterate through all $\delta \leq n^{0.5}$, we get that this case requires $\mathcal{O}(n^{3.5})$ time, as well. Now, we know all triples (i, j, k) such that $w[i..j] \in \{t, f(t)\}^k$ with t and $f(t)$ not powers of the same word.

Further, we consider the case of triples (i, j, k) such that $w[i..j] \in \{t, f(t)\}^k$, where t and $f(t)$ are powers of the same word. By Lemma 8 we compute in $\mathcal{O}(n^2)$ time the periods of all the factors $w[i..j]$ of w and of the factors $f(w[i..j])$ of $f(w)$. We also compute in cubic time the array T from Lemma 2. Now we can check in constant time using *LCPref* queries whether $per(t) = p$, $p \mid |t|$, and $f(w[i..i + p - 1])$ is a power of $w[i..i + p - 1]$ (i.e., t and $f(t)$ are powers of the same word). If this is the case, we compute $m = \frac{|f(w[i..i+p-1])|}{p}$ and set $M[i][j][k] = 1$ if and only if $T[k][m][j - i + 1] = 1$. Indeed, $M[i][j][k] = 1$ if and only if there exist s, k_1, and k_2 such that $s \mid j - i + 1$, $k_1 + k_2 = k$, and

$$w[i..j] = ((w[i..i + p - 1])^s)^{k_1}(f((w[i..i + p - 1])^s))^{k_2};$$

that is, $sk_1 + smk_2 = j - i + 1$, which is equivalent to $T[k][m][j - i + 1] = 1$.

There is a simple case that remains to be discussed. If $f(w[i..j]) = \lambda$, then $M[i][j][k] = 1$, for all $k \geq 1$. Identifying and storing all such factors takes $\mathcal{O}(n^3)$ time.

By the above case analysis, we conclude that we can compute all the non-zero entries of the matrix M in $\mathcal{O}(n^{3.5})$ time. The answer to $rep(i, j, k)$ is given by the entry $M[i][j][k]$.

Finally, we consider the case when we search f-repetitions with k factors, for a fixed k. This time, we compute a two dimensional matrix M_k such that $M_k[i][j] = M[i][j][k]$, defined previously. Fortunately, M_k can be computed much quicker than the whole matrix M. According to Corollary 1 the case of t and $f(t)$ being factors of the same word can be implemented in quadratic time (the constant R from the statement of the corollary can be taken as the maximum length of $f(a)$, for all letters $a \in \text{alph}(w)$). Further, when t and $f(t)$ are not periods of the same word we just need to compute, for each i, t and j, the number k' such that $w[i..j] \in t\{t, f(t)\}^{k'-1}$ and check (in constant time) whether $f(t)^{k-k'}$ is a suffix of $w[1..i]$; if all of these hold, we get that $M_k[i][j] = 1$. However, note that we do not need to go through all the possible values of j. Indeed, we first generate all the prefixes of $w[i..n]$ that have the form t^ℓ with $\ell \leq k$ and see if one of them is longer than $|t| + |f(t)|$. If yes, we

try to extend the longest such prefix with t or $f(t)$ iteratively until we use k factors t or $f(t)$ in the constructed word. By Lemma 6 we obtain in this process only $\mathcal{O}(k)$ such words, and these are exactly the prefixes of $w[i..n]$ that can be expressed as the catenation of at most k factors t and $f(t)$; in other words, this process provides a set that contains all the values j for which $M_k[i][j] = 1$. According to these, the whole process of computing the non-zero entries of the matrix M' takes $\mathcal{O}(n^2 \cdot k)$ time. Note that the answer to a query $rep(i, j, k)$ is given by $M_k[i][j]$; as we already mentioned, we only ask queries for the value k given as input.

4.2. The case of non-erasing morphisms

For f non-erasing, the oracle matrix M described previously can be computed in $\mathcal{O}(n^3)$ time, where $|w| = n$. Initially, we set $M[i][j][k] = 0$, for $i, j, k \in \{1, 2, \ldots, n\}$.

As in the general case, by Lemma 8 we compute (and store) in quadratic time the periods of all the factors $w[i..j]$ of w and of the factors $f(w[i..j])$ of $f(w)$. We also compute in cubic time the array T from Lemma 2.

First we analyse the simplest case. We can check in constant time using *LCPref* queries whether $per(w[i..j]) = p$, $p \mid (j - i + 1)$, and $f(w[i..i + p - 1])$ is a power of $w[i..i + p - 1]$. If so, we compute $m = \frac{|f(w[i..i+p-1])|}{p}$ and set $M[i][j][k] = 1$ if and only of $T[k][m][j - i + 1] = 1$.

Further we present the more complicated cases.

First, let i be a number from $\{1, 2, \ldots, n\}$. We want to detect the factors $w[i..j]$ that belong to $t\{t, f(t)\}^{k-1}$ for some prefix t of $w[i..n]$ such that t and $f(t)$ are not powers of the same word (this case was already covered) and $k \geq 2$. To do this we try all the possible prefixes t of $w[i..n]$. Once we choose such a $t = w[i..\ell]$ we set $M[i][\ell][1] = 1$. Further, starting from the pair $(\ell, 1)$, we compute, by backtracking, all the pairs (m, e) such that $w[i..m] \in t\{t, f(t)\}^{e-1}$; basically, from the pair (m, e) we obtain the pairs $(m + |t|, e + 1)$ if $w[m + 1..m + |t|] = t$ and the pair $(m + |f(t)|, e + 1)$ if $w[m + 1..m + |f(t)|] = f(t)$. By Lemma 6 we obtain exactly one pair of the form $(m, \cdot)$ (as there is an unique decomposition of $w[i..m]$ into factors t and $f(t)$, as long as t and $f(t)$ are not powers of the same word). Therefore, computing all these pairs takes linear time. Further, if we obtained the pair (m, k) we set $M[i][m][k] = 1$.

The whole process just described can be clearly implemented in $\mathcal{O}(n^3)$ time. At this point we know all the possible triples (i, j, k) such that $w[i..j] \in t\{t, f(t)\}^{k-1}$ for some t. It remains to find also the triples (i, j, k) such that $w[i..j] \in f(t)\{t, f(t)\}^{k-1}$ for some t.

In this case, for each $i \in \{1, 2, \ldots, n\}$ we go through all the prefixes $y = w[i..\ell]$ of $w[i..n]$ and assume that $y = f(t)$. Further, we compute a set of pairs (m, e) such that $w[i..m] = y^e$; this can be done easily in linear time, using *LCPref*-queries. Now, for each of these pairs, say (m, e), we try to find a factor $t = w[m + 1..m']$ such that $f(t) = y$, and t and y are not powers of the same word. Once we found such a factor t (which can be done in constant time using *LCPref* queries and the array *inv*) we store the pair $(m+|t|, e+1)$ and starting

from this pair we compute, as in the previous case, all the pairs (m'', e') such that $w[m+1..m''] \in t\{t,y\}^{e'-e-1}$. The key remark regarding this process is that, by Lemma 5, no two pairs having the first component equal to m'' are obtained for a fixed i. As the number of values that m'' may take is upper-bounded by n, the entire computation of these pairs takes $\mathcal{O}(n)$ time. Once this is completed, we set $M[i][m][k] = 1$ for each (m, k) obtained.

In this way we identified in cubic time all the triples (i, j, k) such that $w[i..j] \in \{t, f(t)\}^k$, for some t, and stored in the array M the answers to all the possible $rep(\cdot, \cdot, \cdot)$ queries.

Now, consider the case when we search for f-repetitions with k factors, for a given k and a given non-erasing f. The computation goes on exactly as in the case of general morphisms with the only difference that when we consider the prefix t of a word $w[i..n]$ we can restrict our search to the prefixes t shorter than $\frac{n}{k}$. Thus, the overall complexity of computing the entries of the matrix M_k decreases to $\mathcal{O}(n \cdot \frac{n}{k} \cdot k) = \mathcal{O}(n^2)$ time. Again, the answer to a query $rep(i, j, k)$ for the given value k is provided by the entry $M_k[i][j]$ of the matrix M_k.

4.3. The case of literal morphisms

In the case when f is literal, we are able to construct faster some data structures enabling us to answer *rep* queries. More precisely, we do not need to construct the entire oracle structure, but only some less complex matrix allowing us to retrieve in constant time the answers to our queries. To this end, we first create for the word $wf(w)$ the same data structures as in the initial solution of Problem 1. Further, we define an $n \times n$ matrix M such that for $1 \leq i, d \leq n$ the element $M[i][d] = (j, i_1, i_2)$ stores the beginning index of the longest word $w[j..i]$ contained in $\{t, f(t)\}^+$ for some word t of length d, as well as the last occurrences $w[i_1..i_1 + |t| - 1]$ of t and $w[i_2..i_2 + |f(t)| - 1]$ of $f(t)$ in $w[j..i]$, such that d divides both $i - i_1 + 1$ and $i - i_2 + 1$. If there exist t and t' with $t \neq t'$ and $w[j..i] \in \{t, f(t)\}^k \cap \{t', f(t')\}^k$, we have $t = f(t')$ and $f(t) = t'$; in this case, $M[i][d]$ equals (j, i_1, i_2) if $i_1 > i_2$ or (j, i_2, i_1), otherwise. The array M can be computed in $\mathcal{O}(n^2)$ time by dynamic programming.

Formally, this is done as follows. Recall that in this case f is literal, but not necessarily bijective.

Let us begin by noting that if $w[j..i] \in \{t, f(t)\}^k$ for some t with $|t| = d$, then $w[j..i]$ contains a factor $w[j + \ell d..j + (\ell + 1)d - 1] = t$, for some $\ell \in \{0, 1, \ldots, \frac{i-j+1}{d}\}$. This is true, as otherwise, $w[j + \ell d..j + (\ell + 1)d - 1] = f(t)$ for every $\ell \in \{0, 1, \ldots, \frac{i-j+1}{d}\}$. However, it follows that $w[j..i] \in \{t'\}^k$ for $t' = w[j..j + d - 1]$ and by chosing $t = t'$ we reach our conclusion.

Now, for $1 \leq i \leq n$ and $1 \leq d \leq n$, we define $M[i][d] = (j, i_1, i_2)$ in three steps:

- First, we define the number j as the minimal such that $w[j..i] \in \{t, f(t)\}^k$ for some $t = w[j + \ell d..j + (\ell + 1)d - 1]$, where $\ell \in \{0, 1, \ldots, \frac{i-j+1}{d}\}$ and $k = \frac{i-j+1}{d}$.

– Second, we define the number i_1 as the maximal such that $j \leq i_1 < i$, $d \mid i-i_1+1$, and $w[j..i] \in \{w[i_i..i_1+d-1], f(w[i_1..i_1+d-1])\}^k$. According to the previously made remark, if $w[j..i] \in \{t, f(t)\}^k$ for some factor t, then i_1 is well defined.

– Third, we define the number i_2 as the maximal such that $j \leq i_2 < i$, $d \mid i - i_2 + 1$, and $w[i_2..i_2 + d - 1] = f(w[i_1..i_1 + d - 1])$; if there exists no factor with $w[h..h + d - 1] = f(w[i_1..i_1 + d - 1])$ and $d \mid h - j$ we set $i_2 = -1$.

Note, however, that it may be the case that there exist t and t' such that $t \neq t'$ and $w[j..i] \in \{t, f(t)\}^k \cap \{t', f(t')\}^k$. However, in this case, we clearly have $t = f(t')$ and $f(t) = t'$. Therefore, $M[i][d]$ is either the triple (j, i_1, i_2) when $i_1 > i_2$ or the triple (j, i_2, i_1), otherwise.

Next, we show that this matrix can be computed by dynamic programming in $\mathcal{O}(n^2)$ time. For all $d \in \{1, 2, \ldots, n\}$ and i with $n \geq i \geq d$ we run the following steps:

Initialisation: for all $\ell = i - d$,

– $M[d+\ell][d] = (\ell, \ell, \ell)$ when $0 \leq \ell \leq n-d$ and $w[\ell+1..d+\ell] = f(w[\ell+1..d+\ell])$.

– $M[d+\ell][d] = (\ell, \ell, -1)$ when $0 \leq \ell \leq n-d$ and $w[\ell+1..d+\ell] \neq f(w[\ell+1..d+\ell])$.

Update the matrix by traversing each of its columns starting with $i = 2d$ to $i = n$

– $M[i][d] = (j, i_1, i - d + 1)$ when $M[i - d][d] = (j, i_1, i_2)$ and $w[i - d + 1..i] = w[i_2..i_2 + d - 1]$. That is, i_2 is updated, since $w[i - d + 1..i] = f(w[i_1..i_1 + d - 1])$.

– $M[i][d] = (j, i - d + 1, i_2)$ when $M[i - d][d] = (j, i_1, i_2)$ and $w[i - d + 1..i] = w[i_1..i_1 + d - 1]$. That is, i_1 is updated, since $w[i-d+1] = w[i_1..i_1+d-1]$.

– $M[i][d] = (i_2+d, i-d+1, i-2d+1)$ when $M[i-d][d] = (j, i_1, i_2)$, $i_1 = i-2d+1$ and $f(w[i-d+1..i]) = w[i_1..i_1+d-1]$, but $w[i-d+1..i] \neq w[i_2..i_2+d-1]$ and $w[i_1..i_1 + d - 1] \neq w[i_2..i_2 + d - 1]$.

– $M[i][d] = (i_1+d, i-d+1, i-2d+1)$ when $M[i-d][d] = (j, i_1, i_2)$, $i_2 = i-2d+1$ and $f(w[i-d+1..i]) = w[i_2..i_2+d-1]$, but $w[i-d+1..i] \neq w[i_1..i_1+d-1]$ and $w[i_1..i_1 + d - 1] \neq w[i_2..i_2 + d - 1]$.

– $M[i][d] = (i_1+d, i-2d+1, i-d+1)$ when $M[i-d][d] = (j, i_1, i_2)$, $i_2 = i-2d+1$ and $w[i-d+1..i] = f(w[i_2..i_2+d-1])$, but $w[i_1..i_1+d-1] \neq w[i_2..i_2+d-1]$.

– $M[i][d]$ remains unchanged, otherwise.

It is not hard to see that $M[i][d]$ is correctly computed by the above strategy. Moreover, the time needed to update the value of $M[i][d]$ is constant, as it only needs to check the value of $M[i-d][d]$ and a series of other conditions for which one can use a constant number of *LCPref* queries on the word $wf(w)$.

The matrix M helps us answer *rep*-queries in constant time. Indeed, the answer to a query $rep(i,j,k)$ is `yes` if and only if $k \mid j-i+1$ and the first component of the triple $M[j][\frac{j-i+1}{k}]$ is lower than or equal to i, and `no`, otherwise.

4.4. Solving Problem 2

We now give the final solutions for the two questions of Problem 2.

Let us begin with the first question. It is straightforward how we can use the computed data structures to identify, given a word w of length n, the triples (i,j,k) such that the factor $w[i..j]$ is in $\{t, f(t)\}^k$ for some t. Indeed, we return the solution-set comprising all the triples (i,j,k) for which the answer to $rep(i,j,k)$ is yes. The time needed to do so is $\Theta(n^3)$ (without the preprocessing), as we go through all possible triples (i,j,k) and check whether $rep(i,j,k)$ returns `yes` or `no`. Furthermore, any algorithm solving this problem needs $\Omega(n^3)$ operations in the worst case. Take, for instance, the non-erasing uniform morphism f defined by $f(a) = aa$ and the word $w = a^n$. It follows that $w[i..j]$ is in $\{a, f(a)\}^k$, for all i and j with $\lfloor (j-i+1)/2 \rfloor \leq k \leq j-i+1$; hence, for these w and f we have $\Theta(n^3)$ triples (i,j,k) in the solution set of our problem.

For f a literal anti-/morphism, we propose a $\Theta(n^2 \lg n)$ algorithm solving the discussed problem. Using the Sieve of Eratosthenes, we compute in $\mathcal{O}(n \lg n)$ time the lists of divisors for all numbers ℓ with $1 \leq \ell \leq n$. Further, for each pair $(i, i+\ell-1)$ with $\ell \geq 1$ and all $d \mid \ell$ we check whether $rep(i, i+\ell-1, d)$ returns yes. If so, the triple $(i, i+\ell-1, d)$ is one of those we were looking for. Clearly, the algorithm is correct. Its complexity is, following the result of Lemma 1,

$$\mathcal{O}(n \lg n) + \Theta(\sum_{1 \leq \ell \leq n} (n-\ell+1)d(\ell)) \in \Theta(n^2 \lg n).$$

Moreover, any algorithm solving this problem does $\Omega(n^2 \lg n)$ operations in the worst case. To see this consider the word $w = a^n$ and the anti-/morphism $f(a) = a$. A correct algorithm returns exactly

$$\sum_{1 \leq \ell \leq n} (n-\ell+1)d(\ell) \in \Theta(n^2 \lg n)$$

triples, which proves our claim.

In the case of the second question of our problem, recall that we are given both a word w and a number k. We proceed as follows. To identify the pairs (i,j) such that the factor $w[i..j]$ is in $\{t, f(t)\}^k$ for some t we just have to go through all the possible values for i and j and check the answer of the query $rep(i,j,k)$. Clearly, this takes $\Theta(n^2)$ time. The preprocessing, in which the data structures needed to answer *rep* queries are built, takes in the more efficient case of non-erasing morphisms $\mathcal{O}(n^2)$ time, as well; in the general case, the preprocessing takes $\mathcal{O}(n^2 k)$ time, which is more than the time needed to actually answer all the queries. Our result on non-erasing morphism improves, in a more general framework, the results reported in [13], where the same problem, considering only involutions, was solved in time $\mathcal{O}(n^2 \lg n)$. Finally, note that when k is

constant (e.g., we look for pseudo-squares or pseudo-cubes, or, generally, pseudo-k^{th}-powers for a fixed k inside the input words), the bounds obtained both for general and non-erasing morphisms are tight, since all the factors of length $k\ell$ of $w = a^n$ are equal to $(a^\ell)^k$, thus being solutions to our problem, no matter what anti-/morphism f is used. Hence, the number of elements in the solution-set of question (2) of Problem 2 for a fixed k and a word of length n is in $\Theta(\frac{n^2}{k}) = \Theta(n^2)$.

4.5. Summary

Before concluding this section, recall that the key idea in our approach was to solve both parts of Problem 2 using *rep* queries. In order to assert the efficiency of this method note that, once data structures allowing us to answer such queries are constructed, our algorithms solve the two parts of Problem 2 efficiently. In particular, no other algorithm solving any of the two questions of Problem 2 can run faster than ours (excluding the preprocessing part), in the worst case. Hence, in general, a faster preprocessing part yields a faster complete solution for the problem. Nevertheless, in the case of non-erasing and, respectively, literal anti-/morphisms (which includes the biologically motivated case of involutions) the preprocessing is as time-consuming as the part where we use the data structures we previously constructed to actually solve the first question of the problem. Thus, the time bounds obtained in these cases are tight. If the number k used in the second question is considered to be a constant, thus, no longer given as input, then the time bounds we obtain are also tight.

Theorem 4. *Let $f : V^* \to V^*$ be an anti-/morphism and $w \in V^*$ be a given word of length n.*

(1) We can identify in time $\mathcal{O}(n^{3.5})$ the triples (i, j, k) with $w[i..j] \in \{t, f(t)\}^k$, for a proper factor t of $w[i..j]$. ◁

(2) We can identify in time $\mathcal{O}(n^2 k)$ the pairs (i, j) such that $w[i..j] \in \{t, f(t)\}^k$ for a proper factor t of $w[i..j]$, when k is also given as input. ◁

(3) For a non-erasing f we solve (1) in $\Theta(n^3)$ time and (2) in $\Theta(n^2)$ time. For a literal f we solve (1) in $\Theta(n^2 \lg n)$ time and (2) in $\Theta(n^2)$ time. ◁

5. Solution of Problem 3 for known length-type

Recall that we are given a word w, and want to check whether there exists an anti-/morphism f such that w is an f-repetition; assume that the length-type T of f is also given as input. Again, we only present the case of morphisms, as the case of antimorphisms is similar; the only difference is, in fact, the way the check in Lemma 9 is implemented.

5.1. Finding suitable decompositions

We begin by noting that if $w \in t\{t, f(t)\}^+$ for a prefix t and a morphism f, then there exists a primitive prefix t' of w such that $w \in t'\{t', f(t')\}^+$. Therefore, we algorithmically construct the set $S = \{(x, y) \mid w \in x\{x, y\}^+ \setminus \{x\}^+, x \text{ primitive}, |y| = T(x)\} \cup \{(x, \perp) \mid x \text{ primitive}, w \in \{x\}^+\}$. In the previous, the vector T of dimension $|V|$ is the length-type of the anti-/morphism f with $T[a] = |f(a)|$ and, for $x = b_1 \cdots b_k$ with $b_i \in V$ for all i, we have $T(x) = \sum_{i \leq k} T[b_i]$, the length of the image of x under any anti-/morphism of length-type T defined on V. Now, if the set S contains the pair $(x, \perp)$, then the input word w is a repetition, and we can already give a positive answer to Problem 3, while if the pair (x, y) belongs to S, then $w \in x\{x, y\}^+$ and there might be a morphism f of length-type T such that $f(x) = y$; this morphism, however, remains to be found.

The following algorithm represents a generalisation of Algorithm 2.

Algorithm 3 *Split*(w, T): outputs the set S

Compute *LCPref*-data structures for w;
Let $n = |w|$ and $m = 0$; Identify the primitive prefixes of w;
for $i = 1$ to n such that $w[1..i]$ is primitive, **do**
  Let $x = w[1..i]$ and $m = m + T[w[i]]$; {We have $m = T(w[1..i])$.}
  Test whether $w = x^k$, with $k \geq 2$. If yes, add $(x, \perp)$ to S; go to step 15;
  Compute j such that $i \mid j - 1$ and $w[j..j + i - 1] \neq x$;
  Let $k = 1$ if $m < i$, $k = \frac{m}{i} - 1$ if $i \geq m$ and $i \mid m$, or $k = \lceil \frac{m}{i} \rceil$, otherwise;
  for $t = k$ downto 0, **do**
    Set $y = w[j - it..j - it + m - 1]$;
    Set $u = x$ and $v = y$ if $i = |x| \leq m = |y|$ or $u = y$ and $v = x$, otherwise;
    Set $s = j - it + 1$;
    if *Check*$(w[s..n], u, v)$=**true then** add (x, y) to S and set $t = 0$;
    {We decide that $w \in x\{x, y\}^+$ and add (x, y) to S.}
  end for
end for
return S {(x, y) is stored in S as $|x|$, $|y|$, and the position y occurs in w. }

Basically, our algorithm works as follows. For each proper and primitive prefix $x = w[1..i]$ of the input word w we check whether $w \in \{x\}^+$. If this is not the case, we must find a factor y, whose length m equals $T(x)$, such that $w \in x\{x, y\}^+$; clearly such a factor y occurs at position j' in w such that $w[1..j'-1] \in \{x\}^+$. To find a factor y as above, we first compute the value j such that $w[1..j-1] \in \{x\}^+$ and x does not occur at position j in w. If $m < i = |x|$, then y may occur at any of the positions $j + 1$ or $j - i + 1$ (otherwise, we get from Theorem 1, that x is not primitive). Further, if $m \geq i$ and $i \mid m$ (which is, in fact, equivalent to $i \mid n$), then y may occur at any of the positions $j - it + 1$ for $0 \leq t \leq \frac{m}{i} - 1$. Finally, if $m > i$ and $i \nmid m$ then y may occur at any of the positions $j - it + 1$ for $0 \leq t \leq \lceil \frac{m}{i} \rceil$ (again, otherwise, we get that x is a proper factor of xx, so it is not primitive, a contradiction). In all cases, for the prefix x

we identify at most $\lceil \frac{m}{i} \rceil + 1$ possibilities to choose the factor y. For each of these possibilities we obtain a pair (x, y) and we check whether $w \in x\{x, y\}^+$. This is done using procedure *Check* defined in Section 3.1. The algorithm adds (x, y) to S whenever $s = n + 1$ (the procedure *Check* returns the value true), and it needs $\mathcal{O}(\frac{n}{\max\{|x|,|y|\}})$ steps to check whether $w \in \{x, y\}^*$. The soundness of this approach follows from the previous explanations, and we can show that its time complexity is $\mathcal{O}(n \lg n)$, provided that we construct and use *LCPref*-data structures for w.

Following the above informal description of the implemented strategy and the comments inserted in its pseudo-code, it is not hard to see that Algorithm 3 is sound.

Finally, we compute the complexity of Algorithm 3. To this end, we are interested in how much time we need to execute steps $3 - 14$, as the first two steps clearly take linear time. Steps $4 - 7$ can be executed in constant time, using *LCPref* queries for step 5. Similarly, steps $9 - 11$ can be executed in constant time. Further, note that the computation of each call of the procedure *Check* takes constant time for some $s \leq n$. Moreover, the *Check* procedure is called $\frac{n}{\max\{|x|,|y|\}} = \frac{n}{\max\{i,m\}}$ times, while the iterative process in steps $8 - 13$ is executed for at most $\lceil \frac{m}{i} \rceil + 1$ times. Consequently, the time complexity of this process is in $\mathcal{O}(\frac{n}{i})$. Finally, the iterative process in steps $3 - 14$ is executed for each primitive prefix $w[1..i]$ of w, and during each iteration the algorithm makes at most $\mathcal{O}(\frac{n}{i})$ steps. Hence, the overall time complexity of the algorithm is upper-bounded by $\mathcal{O}(\sum_{1 \leq i \leq n} \frac{n}{i}) = \mathcal{O}(n \lg n)$, for the case when T is not the subject of any restriction.

5.2. Finding the function

We can now present a solution for Problem 3 in the general case. Assume that the input of our problem consists in a length n word w and a list T of at most n numbers, giving the length of $f(a)$ for all the letters $a \in \text{alph}(w)$, in the order of their appearance in w. A naive solution of the problem runs, clearly, in $\mathcal{O}(n^2 \lg n)$.

Intuitively, our more efficient approach is the following. We try to find in the set S a pair (x, y) such that x can be mapped to y by a morphism of length-type T. However, trying each pair individually takes too much time. Therefore, S is split into several sets whose elements share common combinatorial properties and can be processed simultaneously. Each such set is then analysed separately in an efficient manner. The technical details are described in the following.

Initially, the word w is processed as in Remark 2. Thus, each element u of PR_w is encoded by the pair (i, j) such that $w[i..j]$ is the first occurrence of u^2 in w. Then, for each pair $v \in PR_w$ we construct an empty set $B(v)$; we keep track of the minimal element contained by each such set: when an element is inserted in it, the minimum is updated. Each set $B(v)$ is actually stored as $B(i, j)$ provided that $v = w[i..j] \in PR_w$ (that is, i is the leftmost position where v occurs in w). We also compute inductively (and store) the values $T(w[1..i])$ for $1 \leq i \leq |w|$ in linear time. Finally, for each $j \leq n$ there exists at most one

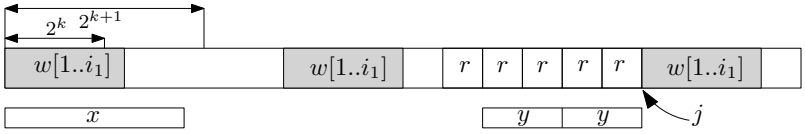

Figure 2: The analysis of the set S_3^k

number i_j such that $w[1..j] = w[1..i_j]y$, with $|y| = T(w[1..i_j])$. As $i = i_j$ if and only if $j = i + T(w[1..i])$, computing these numbers takes linear time.

Next, we compute the set S, as described above. While computing this set we can already split it in two subsets $S_1 = \{(x, y) \mid w \in x^2\{x, y\}^*\}$ and $S_2 = \{(x, y) \mid w \in xy\{x, y\}^*\}$; clearly, $S_1 \cup S_2 = S$, $|S_1| \in \mathcal{O}(n \lg n)$, and $|S_2| \in \mathcal{O}(n)$. The latter, is due to the fact that, for each prefix x of w, the word y such that $(x, y) \in S_2$ is uniquely determined by its length. Moreover, for each $(x, y) \in S_2$ we put i in the set $B(v)$ if $x = w[1..i]$ and v equals the primitive root of y. Using the preprocessing phase described above, this last step takes $\mathcal{O}(n)$ time in total. The rest of our solution consists in a separate analysis of the sets S_1 and S_2, checking whether one contains a pair (x, y) for which there exists a morphism f of length-type T with $f(x) = y$.

We start with S_1. By Lemma 9, we check each pair (x, y) from this set in time $\mathcal{O}(per(x))$. Thus, the time needed to verify all the pairs in S_1 is upper-bounded by

$$\mathcal{O}\left(\sum_{x \in PS_w} |x| \left(|f(x)|/|x| + 2\right)\right) \in \mathcal{O}(n \lg n).$$

Indeed, $|PS_w| \leq 2 \lg n$ by Lemma 3 and for each x we have at most $\lceil |f(x)|/|x| \rceil + 1$ pairs $(x, y) \in S$ to check, and the previously announced upper bound follows.

We continue with the analysis of the set S_2, which is the more involved case. We first split S_2 in two subsets S_3 and S_4. In S_3 we put all the pairs (x, y) with $per(x) > \frac{|x|}{2}$, while $S_4 = S_2 \setminus S_3$. Next we analyse S_3 and S_4 separately.

The analysis of S_3 can be implemented faster than checking its elements one by one. We partition S_3 into the sets $S_3^k = \{(x, y) \in S_3 \mid 2^k \leq |x| < 2^{k+1}\}$, for $0 \leq k \leq \lceil \lg n \rceil$. As for each prefix x of w there is at most one pair (x, y) in S_3^k (and hence in S_3), we can store these sets so that checking whether a pair (x, y) is indeed in S_3^k is done in $\mathcal{O}(1)$ time for every k.

Let us now fix one k, and consider

$$S_3^k = \{(w[1..i_1], y_1), (w[1..i_2], y_2), \ldots, (w[1..i_s], y_s)\},$$

where $i_\ell < i_{\ell+1}$ for $1 \leq \ell \leq s - 1$. Clearly, x starts with $w[1..i_1]$, for all $(x, y) \in S_3^k$. Thus, in a decomposition of w in factors x and y, such that xy occurs as a prefix of w, all the factors x appear at positions where $w[1..i_1]$ occurs in w. Accordingly, we identify the positions where $w[1..i_1]$ occurs in w using a linear time string matching algorithm. There are at most $\frac{n}{2^{k-1}}$ such positions, as $w[1..i_1] \geq 2^k$ and $per(w[1..i_1]) > 2^{k-1}$; let us denote by $j_1, j_2, \ldots, j_p$ these positions.

Since for each $1 \leq \ell \leq s$ the word w has a decomposition in factors $w[1..i_\ell]$ and y_ℓ, there exists $j'_\ell \in \{j_1, j_2, \ldots, j_p, n+1\}$ such that $w[1..j'_\ell - 1] \in w[1..i_\ell]\{y_\ell\}^+$ and $w[j_\ell..n] \in \{w[1..i_\ell], y_\ell\}^*$. Hence, there exist $(x, y) \in S_3^k$ and a morphism f of length-type T such that $f(x) = y$ if and only if there exist $j \in \{j_1, j_2, \ldots, j_p, n+1\}$, $(x, y) \in S_3^k$, and a morphism f of length-type T such that $w[1..j-1] \in x\{y\}^+$ and $f(x) = y$. We check whether there exists j fulfilling these conditions.

For each $j \in \{j_1, j_2, \ldots, j_p, n+1\}$ we run the following processing. We first decide in $\mathcal{O}(2^{k+1})$ time whether there exist two words x and y such that $2^k \leq |x| < 2^{k+1}$, $w[1..j-1] = xy$, and $|y| = T(x)$. If yes, by Lemma 9, we check in $\mathcal{O}(2^{k+1})$ time whether there exists f of length-type T with $f(x) = y$. Moreover, if $(x, y) \in S$ then we found a solution of Problem 3. If no solution is found in this way, we further check whether there exist two words x and y such that $2^k \leq |x| < 2^{k+1}$, $w[1..j-1] \in xy\{y\}^+$, and $|y| = T(x)$. Let us assume that such a pair (x, y) exists. According to Lemma 3, for every j, there exists a set $S_j = \{r_1, r_2, \ldots, r_t\}$ of primitive words, with $t \leq 2 \lg n$, such that r^2 is a suffix of $w[1..j]$ for all $r \in S_j$. As $w[1..j]$ ends with y^2, it follows that $y = r^\ell$ for some $r \in S_j$ and $\ell \geq 1$. Hence, for each $r \in S_j$, we proceed as follows. We go through the prefixes $w[1..i]$ of $w[1..2^{k+1}]$ and try to construct a morphism f of length-type T that maps $w[1..i]$ into a prefix of a power of r. The image of these prefixes can be computed inductively: $f(w[1]) = r^{t_1} r'_1$, where r'_1 is a prefix of r and $t_1|r| + |r'| = T[w[1]]$, and, further, $f(w[i+1]) = r''_i r^{t_{i+1}} r'_{i+1}$, where $r'_i r''_i = r$, r'_{i+1} is a prefix of r, and $t_i|r| + |r'_{i+1}| + |r''_i| = T[w[i+1]]$. Clearly, the image of each letter $w[i]$ can be uniquely associated to the triple $(|r''_{i-1}|, t_i, |r'_i|)$. It is not hard to see that the process of computing these images fails whenever we associate to a letter two different images. On the other hand, each time we find a prefix $w[1..i]$ that can be mapped to r^q, for some $q > 0$, we check whether $|r|$ is a period of $w[i+1..j-1]$ and whether $q|r|$ divides $|j-i-1|$. If all these hold, and, also, $(w[1..i], r^q) \in S_3^k$, then Problem 3 can be answered positively. This concludes the analysis of S_3.

The time needed to analyse S_3 as above is $\mathcal{O}(n(\lg n)^2)$. Constructing and storing the sets S_3^k takes linear time. Then, for each k we run a linear time string matching algorithm, whose output consists in at most $\frac{n}{2^{k-1}}$ positions of w. For each j of these positions, we first check in $\mathcal{O}(2^{k+1})$ time whether $w[1..j-1] = xy$ with $(x, y) \in S_3^k$, and, further, we also check in $\mathcal{O}(2^{k+1} \lg n)$ time whether $w[1..j-1] \in xy\{y\}^+$ with $(x, y) \in S_3^k$. Summing up, the total complexity is

$$\mathcal{O}\left(n + n \lg n + \sum_{k \leq \lg n} \left(\frac{n}{2^{k-1}} \left(2^{k+1} + 2^{k+1} \lg n\right)\right)\right) = \mathcal{O}(n(\lg n)^2).$$

If we did not find any solution in the set S_3, we continue with the analysis of S_4. Again, since for each prefix x of w there is at most one pair $(x, y) \in S_4$, we can store S_4 so that checking whether it contains such a pair takes constant time.

Note that if $(x, y) \in S_4$ then $per(x) = d$ with $w[1..d]$ primitive and $(w[1..d])^2$ a prefix of x. Thus, we can split S_4 into the sets $S_4^d = \{(x, y) \in S_4 \mid per(x) = d\}$, for all d such that $w[1..d]$ is primitive and $(w[1..d])^2$ is a prefix of w. There are at most $2 \lg n$ such sets and they partition S_4. We analyse each of these separately.

We fix a value d such that $w[1..d]$ is primitive and $(w[1..d])^2$ is a prefix of w. It is not hard to see that there exist two numbers e_d and f_d such that $per(w[1..i]) = d$ if and only if $e_d \leq i \leq f_d$. Moreover, if $d' < d$, then $f_{d'} < e_d$. As in the analysis of S_3^k, we run a linear time pattern matching algorithm to locate the positions where $w[1..d]$ occurs in w. Let $j_1, j_2, \ldots, j_s$ be these positions; note that s depends on d, but we omit writing this in order to keep the notation simpler.

Since $w[1..d]$ is primitive, it occurs at most $\frac{n}{d}$ times in w, so $s \leq \frac{n}{d}$. Recall that for each $j \in \{j_1, j_2, \ldots, j_s\}$ there exists at most one value i_j such that $w[1..j-1] = w[1..i_j]y$ with $y = T(w[1..i])$; we already computed and stored these values and we can retrieve each of them in $\mathcal{O}(1)$ time. Now, for each j, if the value i_j is defined and $e_d \leq i_j \leq f_d$, we check in $\mathcal{O}(d)$ time whether there exists a morphism f of length-type T such that $f(w[1..i_j]) = w[i_j + 1..j-1]$. If yes, and $(w[1..i_j], w[i_j + 1..j-1]) \in S_4$, then the instance of Problem 3 defined by w and T has a positive answer.

Now, for each $j \in \{j_1, j_2, \ldots, j_s\}$ we check whether $w[1..j-1] \in xy\{y\}^+$ for some $(x, y) \in S_4^d$ such that there is a morphism f of length-type T with $f(x) = y$. Let $S_j = \{r_1, r_2, \ldots, r_t\}$ be the set of primitive words whose squares are suffixes of $w[1..j-1]$. As y^2 is a suffix of $w[1..j-1]$ we get that $y \in \{r\}^+$ for some $r \in S_j$.

We first discuss the case when $y = r^p$ with $p > 1$. Clearly, y has a prefix v^2, where $|v| = T(w[1..d])$; $|v|$ is also a period of y. By Theorem 1, since $y = r^p$ with r primitive, it follows that $v = r^s$ for $s = \frac{T(w[1..d])}{|r|}$. Hence, we check whether r^{4s} occurs as a suffix of $w[1..j-1]$ (i.e., $w[1..j-1]$ ends with a long enough power of r, allowing us to find a suitable y) and whether there is a morphism f of length-type T with $f(w[1..d]) = r^s$. This takes $\mathcal{O}(d)$ time, by Lemma 9. If there exists such an f, as well as an $i \in B(r)$ with $e_d \leq i \leq f_d$, Problem 3 can be answered positively. Indeed, we have a pair $(w[1..i], z) \in S$ where $per(w[1..i]) = d$ and z a power of r. Thus, $w \in w[1..i]\{w[1..i], z\}^+$, $T(w[1..i]) = |z|$, and, as f exists, $w[1..d]$ can be mapped to the $z[1..m]$ for $m = T(w[1..d])$. It follows that $w[1..i]$ can be mapped to z, and this proves our point. If no i as above exists, then we must consider the case $y = r$ as well as another choices for d, j, and r. However, checking for each d, j, and r as above whether $B(r)$ contains an element i in the range $[e_d, f_d]$ is not efficient. It is better to do all these checks after we finished considering all cases and identified all the ranges we have to verify. Basically, for each primitively-rooted square r^2 occurring in w we will have $\mathcal{O}(\lg n)$ range queries (at most one for each d), whose ranges do not overlap; checking all of them at once takes $\mathcal{O}(|B(r)|)$ time, by considering the elements of $B(r)$ one by one. So the time needed for all r's is $\mathcal{O}(n \lg n)$, as the total number of elements of the sets $B(r)$ is less than $|S|$.

The case when $y = r$ is, however, simpler. We just have to see if the

minimum i of $B(r)$ fulfils $e_d \leq i \leq f_d$, and check in $\mathcal{O}(d)$ time whether $w[1..i]$ can be mapped to r. If `yes`, we answer Problem 3 positively.

This analysis of the set S_4 takes, again, $\mathcal{O}(n(\lg n)^2)$ time. First we go through all the possibilities of choosing d; there are $\mathcal{O}(\lg n)$ of them. For each such d we determine in linear time all the positions where $w[1..d]$ occurs in w; there are $\mathcal{O}(\frac{n}{d})$ of them. For each such position j we further go through all the primitively-rooted squares r^2 ending at that position (all these roots are collected in the set S_j). Then we check whether d maps to a power of r (or, respectively, to a suitable period of r); this takes $\mathcal{O}(d)$ time. Moreover, we have to check whether there is some i in $B(r)$ such that $per(w[1..i]) = d$; this means that there exists $(w[1..i], y) \in S$ such that $per(w[1..i]) = d$ and y is a power of r. But at this moment we only memorise that we have to do this check, and try another r, then another j, and finally another d. The checks we memorise are grouped according to r, and we do not memorise duplicate checks (that is, checks about the same d and the same r). We then do all the checks for one r at once. Since $e_d \leq f_d < e_{d'} \leq f_{d'}$ for $d < d'$, we just have to go through all the elements of $B(r)$ and keep track in what range $[e_d, f_d]$ we are; if we have a check to be done for that range, we answer it positively. The time needed to run all the checks for some r is obviously $\mathcal{O}(|B(r)|)$. Further, the time needed to run all the checks for all r's is $\mathcal{O}(\sum_r |B(r)|) = \mathcal{O}(|S|)$; so it is upper-bounded by $\mathcal{O}(n \lg n)$. We get that the overall complexity of the analysis of S_4 is:

$$\mathcal{O}(\sum_{d \in PS_w} (n + \frac{nd \lg n}{d}) + n \lg n) = \mathcal{O}(n(\lg n)^2)$$

Now we can say whether there exists a pair $(x, y) \in S$ and a morphism f of length-type T such that $f(x) = y$. Hence, we gave a solution for Problem 3, when the length-type of f is known. This solution's time complexity is $\mathcal{O}(n(\lg n)^2)$.

5.3. Finding uniform morphisms

Problem 3 can be solved optimally, in $\mathcal{O}(n)$ time, when f is uniform. The result is based on several of the data structures we already developed and on a thorough analysis of the combinatorial properties of the elements of S. Once more, our solution is presented only for the case of morphisms, as it can be easily adapted to the case of antimorphisms.

Let us assume that we want to solve Problem 3 for a morphism that is p-uniform, with p given as input. We have that all the elements of the length-type vector T are equal to p. Moreover, $|x|$ divides both $|y|$ and $|w| = n$. In this case, we can easily obtain a more efficient algorithm: we only run the iterative instruction **for** loop in line 3 of Algorithm 3 for $i \mid n$. Hence, in this case, the time needed to compute the set S is upper-bounded by $\mathcal{O}(\sum_{i|n} \frac{n}{i}) \in \mathcal{O}(n \lg \lg n)$. Furthermore, it is not hard to see that the set S has $\mathcal{O}(n)$ elements. Indeed, let S' be the set that contains for each prefix x with $|x| \leq \frac{n}{p}$ the pair $(x, \perp)$ and the p pairs (x, y), where all the possibilities of choosing y are discussed in the informal description of Algorithm 3 and formally defined in its lines $7 - 9$,

and for each other prefix x only the pair $(x, \perp)$. This set has $2n$ elements and $S \subseteq S'$. We claim that S can be constructed in linear time.

To this end, we look at a refinement of Algorithm 3. Namely, we run the cycle in step 3 first for all the primitive prefixes $x[1..i]$ such that $x[1..i]^2$ is also a prefix of w (that is, $x[1..i] \in PS_w$). By the arguments already given, the time needed to do this is upper-bounded by $\mathcal{O}(\sum_{x[1..i] \in PS_w} \frac{n}{i})$, which is in $\mathcal{O}(n)$ by Lemma 3. Therefore, we can construct the set $Q = \{(x, y) \in S \mid x \in PS_w\}$ in linear time. Note that for each prefix $x \in PS_w$ we may have at most p pairs (x, y) in Q (therefore, in S as well).

We only have to analyse now the prefixes x not contained in PS_w. Let $Q' = \{(x, y) \in S \mid x \notin PS_w\}$ and note that for each prefix $x \notin PS_w$ we may have at most one pair (x, y) in Q' (so, in S as well), where y is the factor of length $p|x|$ of w, occurring just after the x prefix. Observe that we can construct the set Q' in $\mathcal{O}(n)$ time. To do this we use the strategy employed in Section 3 to decide whether a word is an f-repetition in the case when f is a given uniform morphism. More precisely, Algorithm 3 works as follows. For a prefix $x = w[1..i]$ of w with $i \mid n$ we determine the single word y that may be equal to $f(x)$, as described above. Further, we execute a cycle that extends iteratively a prefix $w[1..s-1]$, where $s \geq i+1$, of the word w such that the newly-obtained prefix is in $x\{x, y\}^*$. However, at each iteration the prefix is extended with a word of the form $x^k y$, with $k \geq 0$. As k can be in fact equal to 0, we can only say that the number of iterations of the cycle is upper-bounded by $\frac{n}{|y|} \leq \frac{n}{|x|}$. Further, this approach can be actually made to work in linear time, as shown in Section 3.2. Indeed, the idea is to extend the prefix every time with a word that belongs to $\{x, y\}^\alpha$ for some fixed number α that depends on n, but not on x or y. In this way, we upper bound the number of iterations of the cycle by $\frac{n}{\alpha|x|}$, and the overall complexity of the algorithm by $\mathcal{O}(\frac{n \lg \lg n}{\alpha})$. Finally, in order to obtain a linear algorithm we choose $\alpha = \lceil \lg \lg n \rceil$. The details of the exact implementation of this idea in linear time are given in the aforementioned section. Therefore, we can construct S in linear time. Moreover, during the construction of S we can also memorise for each pair $(x, y) \in S$ whether the decomposition of w in factors x and y starts with $xx, xyx, xyyx$, or $xyyy$.

Further, we proceed as follows. We partition S into four smaller subsets, according to the form of the decomposition of w in factors x and y. In the first set S_1 of the partition we put the pairs $(x, y) \in S$ such that xx is a prefix of the decomposition of w. In the second set S_2 of the partition we put the pairs $(x, y) \in S$ such that xyx is a prefix of the respective decomposition of w. In the third set S_3 of the partition we put the pairs $(x, y) \in S$ such that $xyyx$ is a prefix of the decomposition of w. Finally, in the fourth set S_4 of the partition we put the pairs $(x, y) \in S$ such that $xyyy$ is a prefix of the decomposition of w. This partitioning can be done in linear time.

The case of S_1 can be treated just as above. By Lemma 3 there are at most $2 \lg n$ primitive words x such that x^2 occurs as a prefix of w and a pair (x, y) is a member of this set and the sum of their lengths is $\mathcal{O}(n)$. As each prefix x of this kind appears in at most one pair in this set, we obtain that we can check in

$\mathcal{O}(|x|)$ time for each x whether there exists a pair (x, y) in S_1 such that x can be mapped to y by a p-uniform morphism. Summing up, we get that the time needed to check all the pairs in S_1 is $\mathcal{O}(n)$.

The case of S_2 is more involved. Obviously, if $(x, y) \in S_2$ then there is no other pair $(x, z) \in S_2$. Let $S_2 = \{(x_0, y_0), (x_1, y_1), \ldots, (x_s, y_s)\}$, such that $|x_i| < |x_{i+1}|$, for $0 \leq i \leq s-1$. Now, for each pair in this set, we check whether x_i can be mapped to y_i; this takes $\mathcal{O}(per(x_i))$ time. We show that the total time needed for these checks is linear. Let us define the constant $q = \frac{3}{2}$. When $\frac{|x_{i+1}|}{|x_i|} < q$, it follows that $2(|x_{i+1}| - |x_i|)$ is a period of x_i. Thus, the time needed to check whether x_i can be mapped to y_i is $\mathcal{O}(2(|x_{i+1}| - |x_i|))$. If $\frac{|x_{i+1}|}{|x_i|} \geq q$, then the time needed to check whether x_i can be mapped to y_i is $\mathcal{O}(|x_i|) \subseteq \mathcal{O}(2(|x_{i+1}| - |x_i|))$, as the prefix of length $|x_i| \leq 2(|x_{i+1}| - |x_i|)$. Consequently, we can check in

$$|x_s| + 2\ell \sum_{0 \leq i \leq s-1} (|x_{i+1}| - |x_i|) \in \mathcal{O}(|x_s|)$$

time whether there exists i and a p-uniform morphism f that maps x_i to y_i. Hence, the time needed to find a pair (x, y) in S_2 such that there exists a p-uniform morphism that maps x to y is $\mathcal{O}(n)$.

Further, the case of S_3 is very similar to the above. For the proof of the complexity we just have to redefine the constant q as $\frac{4}{3}$ instead of $\frac{3}{2}$, and modify the rest of the arguments accordingly. Hence, the time needed to find a pair (x, y) in S_3 such that there exists a p-uniform morphism that maps x to y is $\mathcal{O}(n)$.

Finally, these ideas cannot be applied directly in the case of S_4. Consider the constant $q = \frac{4}{3}$ again, and partition S_4, furthermore, in some subsets $S_4^t = \{(x, y) \in S_4 \mid q^t \leq |x| < q^{t+1}\}$, for $0 \leq t \leq \lceil \lg_q \frac{n}{p} \rceil$.

As in the previous case, we fix a number t and analyse more careful the words from the set S_4^t. Let $S_4^t = \{(x_0, y_0), (x_1, y_1), \ldots, (x_s, y_s)\}$, such that $|x_i| < |x_{i+1}|$, for $0 \leq i \leq s-1$. Note that s and the elements of S_4^t depend on t (however, we omit writing this explicitly, as it complicates the notation).

Clearly, $q \leq \frac{3p+1}{2p+1}$, so $x_j y_j^2$ is a prefix of $x_i y_i^3$ for $j > i$. It follows that y_j occurs as a factor of y_i^2, at position $d = |x_j| - |x_i|$. Consequently, y_j occurs as a factor of y_j^2 at position $|y_j| - pd$. Thus, y_j is a repetition of a word u, such that $|u|$ divides pd. It follows immediately that there exists d_0 such that all y_i's are repetitions of factors pd_0 (where d_0 is a divisor of every number d with $d \in \{|x_j| - |x_i| \mid 0 \leq i \leq j \leq s\}$). Moreover, if $y_i = r_i^{s_i}$ with $|r_i| = pd_0$ and $y_j = r_i^{s_i}$ with $|r_i| = pd_0$, then $r_i = uv$ where $r_i = vu$ and $d_0 \mid |v|$. Consequently, we can partition S_4^t even further into the p sets $S_4^{t,u} = \{(x, y) \in S_4^t \mid y \in \{u\}^+\}$, for $u \in \{y_s[hd_0 + 1..(h+1)d_0] \mid 0 \leq h \leq p-1\}$. Now we note that $S_4^{t,u}$ contains a pair (x, y) for which we can construct a p-uniform morphism f that maps x to y if and only if the pair $(x, y) \in S_4^{t,u}$ with the shortest x fulfils this condition (as always x will have to be mapped to the corresponding power of u). Of course, this can be checked in $\mathcal{O}(|x_s|)$ time. Therefore, the total time needed to check whether S_4^t contains a pair (x, y) as above is $\mathcal{O}(p\ell_t)$, where $\ell_t = |x_s|$.

Let us note now that $q\ell_{t-1} \leq \ell_t$ and $\ell_t \leq \frac{n}{p}$ for all t. Therefore, the time needed to analyse all the sets S_4^t is $\mathcal{O}(\sum_{0 \leq t \leq \lceil \lg_q \frac{n}{p} \rceil} pq^t) = \mathcal{O}(p\frac{n}{p}) \subseteq \mathcal{O}(n)$. On that account, we can check in $\mathcal{O}(n)$ time whether there exists a pair (x, y) in S_4 and a p-uniform morphism that maps x to y.

According to the case analysis we just made, we can check in linear time whether S contains a pair (x, y) for which a p-uniform morphism that maps x to y exists. Therefore, Problem 3 can be solved optimally when we are interested in finding p-uniform morphisms that make the input word a pseudo-repetition.

5.4. Summary

A summary of the results obtained in this section is given by the following theorem.

Theorem 5. *Let $w \in V^+$ be a length n word and T be a vector of $|V|$ numbers.*

(1) We decide whether there exists an anti-/morphism f of length-type T such that $w \in t\{t, f(t)\}^+$ in $\mathcal{O}(n(\lg n)^2)$ time. ◁

(2) We decide whether there exists a uniform anti-/morphism f of length-type T (all images of T are equal) such that $w \in t\{t, f(t)\}^+$ in $\mathcal{O}(n)$ time. ◁

6. Solution of Problem 3 for unknown length-type

As already mentioned, the most general form of Problem 3 is trivial. Besides considering the cases when the length-type of the function we search is given, there are two other natural ways to restrict Problem 3 in order to make it non-trivial for functions f of unknown length-type. One variant is obtained by asking that the root t has at least two letters, and another one by asking that w is an f-repetition consisting of at least three factors.

6.1. Tractable cases

Consider the following problem:

> **Problem 4.** *Given a word $w \in V^*$, decide whether there exist an anti-/morphism $f : V^* \to V^*$ and a prefix t of w with $|t| \geq 2$ such that $w \in t\{t, f(t)\}^+$.*

Clearly, in a solution of the problem there must be at least one letter that appears in w and is not deleted by f. If $|w| \leq 2$ the answer to the problem is negative: w cannot be expressed as an f-repetition. If $w = a^{2n}$, for some letter $a \in V$ and integer $n \geq 2$, we take f to be the function that maps a to a and let $t = a^2$, and we obtain that w is indeed an f-repetition. If $w = a^{2n+1}$ and $2n + 1$ is prime, then the problem has no solution. Indeed, if $t = a^k$, then $2n + 1$ would be divisible by k. Therefore, $k = 1$ or $k = 2n + 1$, which creates a contradiction. Otherwise, when $2n + 1$ is not prime, we can take $t = a^p$, where

p is a divisor of $2n+1$, and f to be the function mapping a to a, and, obviously, w is an f-repetition. If w contains at least two different letters, we take the unique prefix of w that has the form $t = a^k b$ for $a \neq b$ and $k \geq 1$, and f the function with $f(a) = \lambda$ and $f(b) = w'$ for $w = tw'$. Therefore, Problem 4 can be easily decided, in linear time.

In the case when we impose the restriction that f is uniform, we can obviously solve the problem in time $\mathcal{O}(n^2)$ time just by running the algorithms from the previous sections for all the possible lengths of the image of a letter under f.

6.2. Hard cases

There are cases of the restricted variants of Problem 3 that are computationally hard. Recall first the pattern-description problem:

> **Problem 5.** *Given two words $x, y \in V^*$ decide the existence of a morphism g with $g(x) = y$.*

This problem is NP-complete and it remains as hard for g non-erasing (see [21]).

We begin by considering the restriction requiring that the root of the pseudo-repetition has at least 2 letters. The only case left open is when the function we look for is non-erasing; otherwise the problem can be solved efficiently, as it was described in the previous section.

> **Problem 6.** *Given a word $w \in V^+$ decide whether there exist a non-erasing anti-/morphism $f : V^* \to V^*$ and a prefix t of w with $|t| \geq 2$ such that $w \in t\{t, f(t)\}^+$'.*

This problem is clearly in NP, either if the searched function f is a morphism or an anti-morphism. We show for each case that it is NP-complete by giving a polynomial-time reduction from Problem 5, in the case when the morphism g from that problem is non-erasing.

First, let us consider the variant where we search for a morphism f. Assume that we have an input instance of the pattern-description problem, namely two words x and y, over an alphabet V, and want to decide whether there exists a non-erasing morphism g such that $g(x) = y$. Let $w = a^n x b^n y$, where $n = 2\max\{|x|, |y|\}$ and $a, b \notin V$; note that $\max\{|x|, |y|\} = |y|$ as g is non-erasing and $\text{alph}(w) = V \cup \{a, b\}$. We show there exists a non-erasing morphism g such that $g(x) = y$ if and only if there exist a non-erasing morphism $f : \text{alph}(w)^* \to \text{alph}(w)^*$ and a prefix t of w with $|t| \geq 2$ such that $w \in t\{t, f(t)\}^+$. The left-to-right implication is immediate. For the other implication, assuming first that $t = a^k$ we obtain that b must appear in $f(t)$ as, otherwise, w would not be in $t\{t, f(t)\}^*$. Further, since $k \geq 2$ we obtain a contradiction, as w should be a k-repetition, and it is not such a repetition. Therefore, $t = a^n x'$. We obtain that $f(t) = (f(a))^n f(x')$. But the only two factors of the form u^n of w are a^n

and b^n, so $(f(a))^n = b^n$ and $f(a) = b$. Now, it follows that $x' = x$ and $f(x) = y$, so we can take $g = f$. This concludes the morphism case.

Let us consider now the antimorphism variant of the problem. Thus assume that we have an input instance of the pattern-description problem, namely two words x and y, over an alphabet V, and we want to decide whether there exists a non-erasing morphism g such that $g(x) = y$. Let $w = a^n x b^n d e c^n y^R g^n a^n x b^n d$, where $n = 2|y|$ and $a, b, c, d, e, g \notin V$. We show there exists a non-erasing morphism g such that $g(x) = y$ if and only if there exist a non-erasing antimorphism $f : \text{alph}(w)^* \to \text{alph}(w)^*$ and a prefix t of w with $|t| \geq 2$ such that $w \in t\{t, f(t)\}^+$. The left-to-right implication is trivial. We show the other one. If $t = a^k$ with $k \geq 2$ it follows that w ends with $f(a)$, so $f(a)$ ends with d. As d occurs only two times in w, it follows that $f(a)$ occurs exactly two times in w. Due to the form of the word w, this is impossible. Therefore t cannot have the form a^k, with $k \geq 2$. Thus, t has a^n as a prefix; it follows easily that w ends with t (otherwise, w should end with $f(a)^n$ and $f(a) \neq \lambda$, a contradiction). As $t \neq w$ it follows that $t = a^n x b^n d$. But now immediately have that $f(a) = g$, $f(b) = c$, $f(d) = e$ and $f(x) = y$. Moreover, this shows the existence of a morphism g that makes $g(x) = y$; this morphism is defined for the letters $s \in V$ as $g(s) = (f(s))^R$. This concludes the proof of the antimorphism case.

Hence, we exhibited a polynomial time reduction from Problem 5 to Problem 6. Therefore, we conclude that this problem is NP-complete.

We further consider the case when w is an f-repetition of at least three factors. The most general form of this variant of Problem 3 is NP-complete:

Problem 7. *Given a word $w \in V^+$ decide whether there exist an anti-/morphism $f : V^* \to V^*$ and a prefix t of w such that $w \in t\{t, f(t)\}\{t, f(t)\}^+$*

We first show the NP-completeness of this problem when we search for a morphism f. It is easy to see that this problem is in NP. Next, we give a polynomial time reduction from the pattern description problem. Assume we have an input instance of the pattern-description problem, namely two words x and y over an alphabet V and want to decide whether there exists a morphism g such that $g(x) = y$. Take (x, y) to be one of the hard instances of the pattern-description problem, defined in [21]. In this case we may assume that the alphabets of the two strings x and y are disjoint, the alphabet of y consists of only two letters, while x contains no squares and its first symbol appears only once in x. Set $n = 2\max\{|x|, |y|\}$ and for $i \in \{1, 2, \ldots, n\}$ let a_i and b_i be $2n$ new symbols not contained in the alphabets of x or y. Let

$$w = a_1 a_2 \cdots a_n x b_1 b_2 \cdots b_n y a_1 a_2 \cdots a_n x.$$

We show that there exists a morphism g such that $g(x) = y$ if and only if there exists a morphism $f : V^* \to V^*$ and a prefix t of w such that $w \in t\{t, f(t)\}\{t, f(t)\}^+$. The left-to-right implication is rather trivial. Let us now

show the other one. If $t = a_1a_2\cdots a_k$, it is clear that $f(a_1a_2\cdots a_k) \neq \lambda$. Moreover, $f(a_1a_2\cdots a_k)$ is a suffix of w. It is not hard to note that $b_1b_2\cdots b_ny$ is a factor of $f(a_1a_2\cdots a_k)$. Since there is only one occurrence of $b_1b_2\cdots b_ny$, we reach the conclusion that only one factor $f(a_1a_2\cdots a_k)$ appears in w, which is impossible. Therefore, t has $a_1a_2\cdots a_n$ as a prefix. Assume now that t is a proper prefix of $a_1a_2\cdots a_nx$. Hence, $tf(t)$ is a prefix of w and $f(t)$ starts with a letter of x. Again, we easily get that $b_1b_2\cdots b_ny$ is a factor of $f(t)$. So, once more, we have exactly one occurrence of $f(t)$ in w. The only possibility is that $tf(t) = a_1a_2\cdots a_nxb_1b_2\cdots b_ny$; otherwise w would not be in $t\{t, f(t)\}\{t, f(t)\}^+$). But this is also impossible, as $a_1a_2\cdots a_nx$ cannot be written as an element of $\{t, f(t)\}^+$. Assume now that t has $a_1a_2\cdots a_nx$ as a proper prefix. Hence, exactly one factor of w equals t. Therefore, $f(t)$ is a suffix of w and it appears more than once in w (as $w \in t\{t, f(t)\}\{t, f(t)\}^+$). The only possibility is when $f(t)$ is a proper suffix of x (otherwise, one occurrence of $f(t)$ would not contain symbols of x, a contradiction). However, this is a contradiction with the fact that x contains no square. Therefore, the only possibility is that $t = a_1a_2\cdots a_nx$. In this case, by a case analysis similar to the above, we get that $f(t) = b_1b_2\cdots b_ny$. Now let g be the morphism defined on the alphabet of x that maps any letter s from this alphabet, except for the first letter of x, into $g(s)$ from which we delete the occurrences of any letter from $\{b_1, b_2, \ldots, b_n\}$. Let P_1 be the first letter of x (following the notation of [21]). This letter is mapped by f to $g(a_1a_2\cdots a_nP_1)$ from which we delete any letter from $\{b_1, b_2, \ldots, b_n\}$. Clearly, $g(x) = y$. The equivalence that we have just shown exhibits a polynomial time reduction from the pattern-description problem to our problem. Therefore, our problem is NP-complete, as well.

The case when we search for an antimorphism f is treated similarly, choosing

$$w = a_1a_2\cdots a_nxc_1c_2\cdots c_nb_1b_2\cdots b_ny^Rd_1d_2\cdots d_na_1a_2\cdots a_nxc_1c_2\cdots c_n.$$

Finally, we can show by the exact same reductions and very similar proofs that the restriction of the above problem to the case when we look for non-erasing anti-/morphisms f is NP-complete, as well.

Let us now overview the case when w is an f-repetition of at least three factors. In the most general variant, the problem asks to decide, for a given word $w \in V$, whether there exist an anti-/morphism $f : V^* \to V^*$ and a prefix t of w such that $w \in t\{t, f(t)\}\{t, f(t)\}^+$. This variant of the problem is NP-complete, just as its restriction to the case when the function f is non-erasing.

6.3. Summary

We summarise the main results of this section in the following theorem:

Theorem 6. *Let $w \in V^+$ be a word of length n.*

(1) We decide whether there exists a (general) anti-/morphism $f : V^* \to V^*$ and a prefix t of w such that $w \in t\{t, f(t)\}^+$ with $|t| \geq 2$ in linear time. ◁

(2) To decide whether there exists a (general) anti-/morphism $f : V^* \to V^*$ and a prefix t of w such that $w \in t\{t, f(t)\}\{t, f(t)\}^+$ is NP-complete. ◁

(3) To decide whether there exists a non-erasing anti-/morphism $f : V^* \to V^*$ and a prefix t of w such that $w \in t\{t, f(t)\}^+$ with $|t| \geq 2$ is NP-complete. ◁

(4) To decide whether there exists a non-erasing anti-/morphism $f : V^* \to V^*$ and a prefix t of w such that $w \in t\{t, f(t)\}\{t, f(t)\}^+$ is NP-complete. ◁

(5) We decide whether there exists a uniform anti-/morphism $f : V^* \to V^*$ and a prefix t of w such that $w \in t\{t, f(t)\}^+$ with $|t| \geq 2$ in $\mathcal{O}(n^2)$ time. ◁

(6) We decide whether there exists a uniform anti-/morphism $f : V^* \to V^*$ and a prefix t of w such that $w \in t\{t, f(t)\}\{t, f(t)\}^+$ in $\mathcal{O}(n^2)$ time. ◁

7. Conclusion

We start this section with a simple remark regarding the way the function f defining the f-repetitions is given in Problems 1 and 2. We assumed that f is fixed; however, we can consider the function f as part of the input: basically, after w is given, we can assume that we are given, one by one, the words $f(w[1]), f(w[2]), \ldots, f(w[|w|])$. This does now affect the complexity of the results: we need linear time in the size of the input to construct suffix arrays and *LCPref*-data structures for the word $wf(w)$ and then $\mathcal{O}(|w|)$ time to solve the problems as explained already.

As a conclusion of our work, we want to emphasise the results in Theorem 2 and 5 regarding uniform morphisms. The first theorem states that deciding whether a word is an f-repetition for a given uniform anti-/morphism f can be done optimally in linear time, that is, as efficiently as deciding whether the given word is a classical repetition. More surprisingly, deciding whether there exists a uniform anti-/morphism f, for which we know the length of the images of the letters, such that a given word is an f-repetition can still be done in linear time, so, again, as efficiently as deciding whether a word is an f-repetition for a given f or as deciding that a word is a classical repetition. Recall that f-repetitions where firstly introduced with a biological motivation: f was considered to be an antimorphic involution modelling the Watson-Crick complementarity; in that case, f was literal so deciding whether a word is an f-repetition can be done optimally, in linear time.

The cases of f being a non-uniform anti-/morphism are not solved optimally and no lower bounds for their solutions were obtained. It remains an open problem to see whether there are more efficient solutions for these cases, as well.

Problem 2 raised the question of identifying all the factors of a word that are f-repetitions, for a given f. We presented solutions that perform as fast as any other algorithm in the worst case: there are words for which just writing the output required by the problem takes more than the processing we did to solve the problem. However, our algorithms are not optimal, in the sense that in general the processing we execute takes more than outputting its results. It would be interesting to see whether we can obtain algorithms solving this

problem in various cases, which are optimal when their running time is measured with respect to the size of the output.

Problem 2 also hints another possible continuation of this work. In this paper we looked for all the repetitive factors of a word. However, we may also be interested in testing only whether the given word contains an f-repetition that has a specific form (e.g., a given exponent, or is structured according to a predefined pattern, etc.). This problem was already addressed in [12] and [14], but only in the case when f is literal; the more general cases of f uniform, or non-erasing, or unrestricted remain open.

Acknowledgments

The authors thank Cătălin Tiseanu for valuable comments and suggestions on the initial draft of [1]. Also, the authors thank the anonymous referees of the conference papers [1, 2], whose comments were helpful in preparing this journal version.

[1] P. Gawrychowski, F. Manea, R. Mercaş, D. Nowotka, C. Tiseanu, Finding pseudo-repetitions, in: 30th International Symposium on Theoretical Aspects of Computer Science (STACS), Vol. 20 of LIPIcs, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 257–268.

[2] P. Gawrychowski, F. Manea, D. Nowotka, Discovering hidden repetitions in words, in: 9th Conference on Computability in Europe (CiE), Vol. 7921 of LNCS, Springer, 2013, pp. 210–219.

[3] E. Czeizler, L. Kari, S. Seki, On a special class of primitive words, Theoretical Computer Science 411 (2010) 617–630.

[4] F. Manea, R. Mercaş, D. Nowotka, Fine and Wilf's theorem and pseudo-repetitions, in: 37th International Symposium on Mathematical Foundations of Computer Science (MFCS), Vol. 7464 of LNCS, Springer, 2012, pp. 668–680.

[5] E. Czeizler, E. Czeizler, L. Kari, S. Seki, An extension of the Lyndon-Schützenberger result to pseudoperiodic words, Information and Computation 209 (2011) 717730.

[6] L. Kari, B. Masson, S. Seki, Properties of pseudo-primitive words and their applications, International Journal of Foundations of Computer Science 22 (2) (2011) 447–471.

[7] F. Manea, M. Müller, D. Nowotka, On the pseudoperiodic extension of $u^\ell = v^m w^n$, in: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2013, Vol. 24 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013, pp. 475–486.

[8] F. Manea, M. Müller, D. Nowotka, S. Seki, Generalised Lyndon-Schützenberger equations, in: Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Vol. 8634 of Lecture Notes in Computer Science, Springer, 2014, pp. 402–413.

[9] F. Manea, M. Müller, D. Nowotka, The avoidability of cubes under permutations, in: 16th International Conference on Developments in Language Theory (DLT), Vol. 7410 of LNCS, Springer, 2012, pp. 416–427.

[10] J. D. Currie, F. Manea, D. Nowotka, Unary patterns with permutations, in: Developments in Language Theory - 19th International Conference, DLT 2015, Vol. 9168 of Lecture Notes in Computer Science, Springer, 2015, pp. 191–202.

[11] F. Manea, M. Müller, D. Nowotka, Cubic patterns with permutations, J. Comput. Syst. Sci. 81 (7) (2015) 1298–1310. doi:10.1016/j.jcss.2015.04.001. URL http://dx.doi.org/10.1016/j.jcss.2015.04.001

[12] Z. Xu, A minimal periods algorithm with applications, in: 21st Annual Symposium on Combinatorial Pattern Matching (CPM), LNCS, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 51–62.

[13] E. Chiniforooshan, L. Kari, Z. Xu, Pseudopower avoidance, Fundamenta Informaticae 114 (1) (2012) 55–72.

[14] P. Gawrychowski, F. Manea, D. Nowotka, Testing generalised freeness of words, in: 31st International Symposium on Theoretical Aspects of Computer Science (STACS), Vol. 25 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014, pp. 337–349.

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

[16] M. Lothaire, Combinatorics on Words, Cambridge University Press, 1997.

[17] J. Kärkkäinen, P. Sanders, S. Burkhardt, Linear work suffix array construction, Journal of the ACM 53 (2006) 918–936.

[18] N. J. Fine, H. S. Wilf, Uniqueness theorem for periodic functions, Proceedings of the American Mathematical Society 16 (1965) 109–114.

[19] T. M. Apostol, Introduction to analytic number theory, Springer, 1976.

[20] D. Gusfield, Algorithms on strings, trees, and sequences: computer science and computational biology, Cambridge University Press, New York, NY, USA, 1997.

[21] A. Ehrenfeucht, G. Rozenberg, Finding a Homomorphism Between Two Words is NP-Complete, Information Processing Letters 9 (2) (1979) 86–88.