INSTITUT FÜR INFORMATIK

Finding Pseudo-Repetitions

Paweł Gawrychowski, Florin Manea, Robert Mercaş, Dirk Nowotka, Cătălin Tiseanu

Bericht Nr. 1208
June 22, 2012
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Finding Pseudo-Repetitions

Paweł Gawrychowski, Florin Manea, Robert Mercaş, Dirk Nowotka, Cătălin Tiseanu

Bericht Nr. 1208
June 22, 2012
ISSN 2192-6247

e-mail: {flm,dn}@informatik.uni-kiel.de

Finding Pseudo-Repetitions

Paweł Gawrychowski[1], Florin Manea[2⋆], Robert Mercaş[3⋆⋆], Dirk Nowotka[2⋆⋆⋆] and Cătălin Tiseanu[4]

[1] Max-Planck-Institute für Informatik, Saarbrücken, Germany, gawry@cs.uni.wroc.pl
[2] Christian-Albrechts-Universität zu Kiel, Institut für Informatik, D-24098 Kiel, Germany, {flm,dn}@informatik.uni-kiel.de
[3] Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, PSF 4120, D-39016 Magdeburg, Germany, robertmercas@gmail.com
[4] University of Maryland at College Park, Computer Science Department, A.V. Williams Bldg., College Park, MD 20742, USA, ctiseanu@umd.edu

Abstract. Pseudo-repetitions are a generalization of the fundamental notion of repetitions in sequences, considered initially in the framework of DNA computing and bioinformatics. We develop the algorithmic foundations for questions on pseudo-repetitions by the nontrivial application of combinatorial results on words.

1 Introduction

The notions of repetition and primitivity are fundamental concepts on sequences used in a number of fields, among them being stringology and algebraic coding theory. A word is a repetition (or power) if it can be expressed as a repeated catenation of one of its prefixes. We consider a more general concept here, namely *pseudo-repetitions in words*. A word w is a pseudo-repetition if it can be written as a repeated catenation of one of its prefixes t and its image $f(t)$ under some morphism or antimorphism (for short "anti-/morphism") f, thus $w \in t\{t, f(t)\}^+$.

Pseudo-repetitions, introduced in a restricted form by Kari et al.[1], lacked so far a developed algorithmic part, something usually quite important in the field this theory originates from – bioinformatics. This work is aimed to fill this gap. We investigate the following algorithmic problems: decide whether a word w is a pseudo-repetition

⋆ The work of Florin Manea is supported by the *DFG* grant 582014.
⋆⋆ The work of Robert Mercaş is supported by the *Alexander von Humboldt Foundation.*
⋆⋆⋆ The work of Dirk Nowotka is supported by the *DFG Heisenberg* grant 590179.

for some given morphism f, find all k-powers of pseudo-repetitions occurring as factors in a word, for some given f, decide whether there exists a morphism such that w is a pseudo-repetition,. We establish algorithms and complexity bounds for these problems for various types of morphisms thereby improving significantly the results from [2]. Apart from the application of standard tools of string algorithms, like suffix arrays, we extend the toolbox by nontrivial applications of results from combinatorics on words.

Background and Motivation.

The motivation of introducing pseudo-repetition and -primitivity in [1] originated from the field of computational biology, namely the facts that the Watson-Crick complement can be formalized as an antimorphic involution and both a single-stranded DNA and its complement (or its image through such an involution) basically encode the same information. Until now, pseudo-repetitions were considered only in the cases of anti-/morphic involutions, following the original motivation, and the results obtained were mostly of combinatoric nature (e.g., generalizations of the Fine and Wilf theorem).

A natural extension of these concepts is to consider antimorphisms and morphisms in general, which is done in this paper. Considering that the notion of repetition is central in the study of combinatorics of words, and the plethora of applications that this concept has, the study of pseudo-repetitions seems even more attractive, at least from a theoretical point of view. While the biological motivation seems appropriate only for the case when f is an antimorphic involution, one can imagine a series of real-life scenarios where we are interested in identifying factors that can be written as an iterated catenation of a word and its encoding through some simple function f. Indeed, pseudo-repetitions can be seen as strings that have an intrinsic repetitive structure, hidden by rewriting some of the factors that define it through some anti-/morphism.

Some Basic Concepts.

For more detailed definitions we refer to the handbook [3].

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 $\mathrm{alph}(w)$ the alphabet of all letters that occur in w. A word u is a *factor* of a word v, if $v = xuy$, for some x, y. We say that 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 of w starting at position i and ending at position j, consisting of the catenation of the symbols $w[i], \ldots, w[j]$, where $1 \leq i \leq j \leq n$. For simplicity, we assume that $w[i..j] = \lambda$ if $i > j$. Moreover, we write $w = u^{-1}v$ when $v = uw$. 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 nontrivial power of another word, then w is *primitive*. 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 shortest such period for w.

The following well-known result is useful in our investigation:

Theorem 1 (Fine and Wilf [4]). *Let u and v be in V^* and $d = \gcd(|u|, |v|)$. If two words $\alpha \in u\{u, v\}^+$ and $\beta \in v\{u, v\}^+$ have a common prefix of length greater or equal to $|u| + |v| - d$, then u and v are powers of a common word of length d. Moreover, the bound $|u| + |v| - d$ is optimal.*

A function $f : V^* \to V^*$ is a morphism if $f(xy) = f(x)f(y)$ for any words x and y over V. Further, f is an antimorphism if $f(xy) = f(y)f(x)$ for all $x, y \in V^*$. Note that, when we define a morphism or an antimorphism it is enough to give the definitions of $f(a)$, for all $a \in V$. An anti-/morphism $f : V^* \to V^*$ is an involution if $f^2(a) = a$ for all $a \in V$. We say that f is *uniform* if there exists a number 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*. We say that a word w is an *f-repetition*, or, alternatively, an f-power, if w is in $t\{t, f(t)\}^+$, for some prefix t of w. If w is not an f-power, then w is *f-primitive*.

As an example, the word *abcaab* is primitive from the classical point of view (i.e., **1**-primitive, where **1** is the identical morphism) as well as f-primitive, for the morphism f defined by $f(a) = b$, $f(b) = a$ and $f(c) = c$. However, when considering the morphism $f(a) = c$, $f(b) = a$ and $f(c) = b$, we get that *abcaab* is the catenation of ab, $ca = f(ab)$, and ab, thus, being an f-repetition.

Computational Model

We provide some basic information on the computational model we use: the unit-cost RAM (Random Access Machine) with logarithmic word size. For a more detailed explanation, [5] is a good reference (see the explanations in Section 2.2, Analysing Algorithms). In this model (which is generally used in the analysis of algorithms) we assume that each memory cell can store $\mathcal{O}(\lg n)$ bits, or, in other words, that *the machine word size* is $\mathcal{O}(\lg n)$; the constant hidden by the $\mathcal{O}$-notation is at least 1. The instructions are executed one after another, with no concurrent operations. The model contains common instructions: arithmetic (add, subtract, multiply, divide, remainder, shifts and bitwise operations, etc.), data movement (indirect addressing, load the content of a memory cell, store a number in a memory cell, copy the content of a memory cell to another), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time. As it is important in our paper, we point out that testing the equality of two numbers is also assumed to take a constant amount of time; as a consequence, we note that also testing whether a number with $\mathcal{O}(\lg n)$ bits divides another number with $\mathcal{O}(\lg n)$ bits takes constant time, as well. Basically, this model allows us to measure the number of instructions executed in an algorithm, making abstraction of the time spent to execute each of the basic instructions.

As an addition to the above comments, let us also consider the case when the only arithmetic operations that the computational model we use can execute are addition, subtraction, multiplication, shifts and bitwise operations, but not division. In this case, by using the Newton–Raphson division method, one could obtain the result of the integer-division of two $\mathcal{O}(\lg n)$ bits integers in $\mathcal{O}(\lg \lg n)$ time (that is, this many executions of basic steps).

However, detecting all the divisors of a number n, with $\mathcal{O}(\lg n)$ bits, can still be done in $\mathcal{O}(n)$ time, as described in Algorithm 1.

Using this algorithm, one can show that even for this more general model all the results regarding Problem 1 still hold. Moreover, in the case of Problem 2 one can precompute all the pairs (k, ℓ) such that k divides ℓ and $k, \ell \leq n$, using the Eratosthenes Sieve method, in $\mathcal{O}(n \lg n)$ time (and create a very simple look up table for them

Algorithm 1 *Divisors*(n): outputs the divisors of n

```
1: Set m = n;
2: for d = 1 to n do
3:     while d * m > n do
4:         m = m − 1;
5:     end while
6:     if d * m = n then
7:         Output d as a divisor of n;
8:     end if
9: end for
10: Halt and decide that w is not an f-repetition.
```

in $\mathcal{O}(n^2)$ time). So, using such a model has no effect on the results regarding Problem 2, as well.

2 Algorithmic problems

In the upcoming algorithmic problems, when we are given as input a word w of length n we assume that the symbols of w are in fact integers from $\{1, \ldots, n\}$ (i.e., $\text{alph}(w) \subseteq \{1, \ldots, n\}$), and w is seen as a sequence of integers. This is a common assumption in algorithmic on words (see, e.g., the discussion in [6]).

In the first one, which is probably the most interesting in the general context of pseudo-repetitions, we are interested in deciding whether a word is an f-repetition, for some given anti-/morphism f whose size is assumed to be constant.

Problem 1. Let $f : V^* \to V^*$ be an anti-/morphism. Given $w \in V^*$, decide whether for some word t we have $w \in t\{t, f(t)\}^+$.

We solve this problem in the general case in time $\mathcal{O}(n \lg n)$. However, in the particular case of uniform anti-/morphisms we obtain an optimal solution running in linear time. The latter includes the biologically motivated case of involutions from [1]. Further, we extend our results to a more general form of Problem 1, testing whether $w \in \{t, f(t)\}\{t, f(t)\}^+$. Except for the most general case (of anti-/morphisms that can also erase letters), when we solve this problem in $\mathcal{O}(n^{1+\frac{1}{\lg \lg n}} \lg n)$ time , in the rest of the cases we are able to preserve the same time complexity as in the particular case.

Two other natural problems are related to identifying the factors of a word which are pseudo-repetitions. The first one was originally considered in [2]; the second generalizes it. Both these problems are related to testing the fundamental combinatorial property of freeness of words, in the context of pseudo-repetitions.

Problem 2. Let $f : V^* \to V^*$ be an anti-/morphism and $w \in V^*$ a word.
(1) Given the number k enumerate all pairs (i, j) such that $w[i..j] \in \{t, f(t)\}^k$.
(2) Enumerate all triples (i, j, ℓ) such that there exists t with $w[i..j] \in \{t, f(t)\}^\ell$.

Our approach is based on constructing data structures that enable us to obtain in constant time the answer to queries $rep(i, j, \ell)$: "Is there $t \in V^*$ such that $w[i..j] \in \{t, f(t)\}^\ell$?", for $1 \leq i \leq j \leq |w|$ and $1 \leq \ell \leq |w|$. In the general case, one can produce in $\mathcal{O}(n^5)$ time such data structures. When f is non-erasing, the time needed to construct such data structures is $\mathcal{O}(n^3)$, while when f is a literal anti-/morphism we can do it in time $\Theta(n^2)$. In every case, once we have these structures, we can identify in $\Theta(n^2)$ time the pairs (i, j) such that $w[i..j] \in \{t, f(t)\}^k$ for a given k, answering question (1) in Problem 2. Moreover, we exhibit words w and all sorts of (general, non-erasing uniform, or, respectively, literal) anti-/morphisms f for which w has $\Theta(n^2)$ factors from $\{t, f(t)\}^k$. Therefore, for literal anti-/morphisms any algorithm identifying the above pairs has an asymptotically similar running time as ours in the worst case (including the preprocessing part). For general or non-erasing morphisms a similar statement does not hold, the running time of the algorithm being dominated by the preprocessing part. This improves significantly the algorithmic results reported in [2].

As far as question (2) of Problem 2 is concerned, for a general f, provided the word was processed as previously described, we can enumerate all triples (i, j, ℓ) that fulfil the required conditions in cubic time. This matches the worst case complexity of any algorithm solving the problem, but we still have a quite time consuming preprocessing part. When f is non-erasing (respectively, literal), we obtain an algorithm that solves this problem in $\Theta(n^3)$ (respectively, $\Theta(n^2 \lg n)$) time and show that there are input words on which ev-

ery algorithm solving question (2) has a running time asymptotically equal to ours (this time, including the preprocessing time).

Up to now we considered identifying f-repetitions only when f was given. Another important problem seems to us deciding whether there exists f such that a given word has an f-repetition structure. Basically this last problem that we consider checks whether a given word has a hidden repetitive structure.

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

Generally, the problem is trivial. If $w = aw'$ with $a \in V$, we let $t = a$ and f the anti-/morphism with $f(a) = w'$. Therefore, we study two relevant restrictions of the problem (when the prefix t we look for has at least 2 letters and when w has at least three factors from $\{t, f(t)\}$) that make the trivial solution not applicable. In this setting, we obtain both tractable and NP-complete variants of Problem 3, according to the class of functions to which f belongs.

2.1 Prerequisites

Before proving the claims of previous section, we present some number theoretic properties useful in the sequel. Given two natural numbers k and n, we write $k \mid n$ if k divides n. We denote by $d(n)$ the number of n's divisors and by $\sigma(n)$ their sum. Lemma 1 is important in the time complexity analysis of our algorithms.

Lemma 1. *Let n be a natural number. The following statements hold:*
(1). $\sum_{1\leq \ell \leq n} d(\ell) \in \Theta(n \lg n)$*; we also have* $\sum_{1\leq \ell \leq n} d(\ell) \geq n \lg n$.
(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 (cf. [7], for instance).

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 $(a_\ell)_{1\le\ell\le n}$ and $(b_\ell)_{1\le\ell\le n}$ are two increasing sequences of real numbers, then $\sum_{1\le\ell\le n} a_\ell b_\ell \ge \frac{1}{n}(\sum_{1\le\ell\le n} a_\ell)(\sum_{1\le\ell\le n} b_\ell)$. 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 $\mathcal{O}(n^3)$ a three dimensional array $T[k][m][\ell]$, with $1 \le k, m, \ell \le n$, such that $T[k][m][\ell] = 1$ 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 $i \equiv k \mod (m-1)$ and $km \ge i \ge k$. Indeed, from $k_1 + mk_2 = 1$ we get that $k_1 + k_2 + (m-1)k_2 = i$ so $i \equiv k \mod (m-1)$; 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 one can decide in $\mathcal{O}(1)$ 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 $\frac{l}{s}$ 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 $\mathcal{O}(\sum_{1\leq k\leq n}\sum_{1\leq m\leq n/\lg n}\sum_{1\leq \ell\leq n} d(\ell))$ 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 $m \geq \frac{n}{\lg n}$. Since $\ell \leq n$ and $m \geq \frac{n}{\lg n}$ 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 $k \leq n$ and $k \equiv \frac{\ell}{s} \mod (m-1)$ (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 $k \equiv \frac{\ell}{s} \mod (m-1)$ 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. □

Let us briefly present the data structures that we use.

For a string u of length n, over an alphabet $V \subseteq \{1, \ldots, n\}$, we define a suffix-array data structure that contains two arrays Suf, a permutation of $\{1, \ldots, n\}$, and LCP with n elements from $\{0, 1, \ldots, n-1\}$. Basically, Suf is defined such that $u[Suf[i]..n]$ is the i^{th} suffix of x, in the lexicographical order. The array LCP is defined by $LCP[1] = 1$ and $LCP[r]$ is the length of the longest common prefix of $u[Suf[r-1]..n]$ and $u[Suf[r]..n]$. These data structures are constructed in time $\mathcal{O}(n)$. For more details, see [6], and the references therein. Moreover, one can process the array LCP in linear time $\mathcal{O}(n)$ in order to return in constant time the answer to queries "What is the length of the longest common prefix of $u[i..n]$ and $u[j..n]$?", denoted $LCPref(i, j)$. The idea is to first compute a structure S that associates to each i the value $S[i] = \ell$ if and only if $i = Suf[\ell]$; in other words S is the inverse permutation of Suf. Further we compute in linear time a range minimum query data structure for the array LCP (see [8]), and return in constant time the answer to queries "What is the minimum number from $LCP[i], \ldots, LCP[j]$?". Now, $LCPref(i, j)$ is obtained as the minimum from $LCP[i'+1], \ldots, LCP[j']$, where $i' = \min\{S[i], S[j]\}$ and $j' = \max\{S[i], S[j]\}$.

Finally, for a morphism f, compute for each prefix t of x the length of $f(t)$, and save it in an array len, i.e., $len[i] = |f(x[1..i])|$. For f non-erasing we also compute an array with $|f(x)|$ elements inv such that $inv[i] = j$ if $len[j] = i$ and $inv[i] = -1$ otherwise. These are done in $\mathcal{O}(n)$ time.

Further, we give several lemmas regarding combinatoric properties of words. The first one shows a basic property of pseudo-repetitions, while the second is useful to our algorithms.

Lemma 3. *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 $x = z$. Moreover, in this case, if x and y are not powers of the same word then for each $w \in \{x,y\}^*$ there exists a unique decomposition in factors x and y.*

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

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 w has z as a prefix, as well, then from the facts that $f(x) = f(z)$ and f is non-erasing it follows immediately that $x = z$.

Let us now assume 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. If $y = x^\ell$ it follows that y and x are powers of the same word, and we get that z is also a power of that word. Once again we obtain easily that $x = z$. So, let us assume that x and y are not powers of the same word. Therefore, $y = x^\ell u$ where u is a proper prefix of x, that is, $x = uv$ for $u \neq \lambda \neq v$. Note now that w' may start with $x^p y$ with $p \geq 0$. In any case, we get that after the first $|y|$ symbols of w both the factor vu and the factor z occur. There are three cases to be analysed. If vu is a proper prefix of z than we get that $|f(z)| < |f(vu)| = |f(x)|$, a contradiction. Similarly, we get a contradiction if vu is a proper prefix of z. Thus, there is only one possible case, when $z = vu$. But $y = f(z) = f(vu) = f(v)f(u)$ and $y = f(x) = f(u)f(v)$; it follows that y is a power of a word. As x is a period of y we get that x is also a power of the same word, again a contradiction.

Finally, if $w = yyzw''$ 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 $z = x$.

Finally, if there exist two decompositions of a word $w \in \{x, y\}^*$ then, by Theorem 1, we get that x and y are powers of the same word, and the conclusion of the lemma follows. □

Lemma 4. *Let $x, y \in V^+$ and $w \in \{x, y\}^* \setminus \{x\}^*$ be such that $|x| \leq |y|$ and x and y are not powers of the same word. Let $M = \max\{p \mid x^p \text{ is a prefix of } w\}$ and $N = \max\{p \mid x^p \text{ is a prefix of } y\}$. Then exactly one of the following cases holds:*

- *w is in $x^{M-N}y\{x, y\}^*$ and $x^{M-N-1}yx$ is not a prefix of w.*
- *$M - N - 1 \geq 0$, w is in $x^{M-N-1}y\{x, y\}^*$, and $x^{M-N-1}yx$ is a prefix of w.*

Proof. We first consider the case when x is not a prefix of y; this means, clearly, that $N = 0$. Moreover, there is no position i where both x and y occur. Thus, it is rather easy to see that $w = x^M yw'$ where $w' \in \{x, y\}^*$ and the conclusion follows. Note, though, that M may be equal to 0 when w has y as a prefix.

The more involved case is when x is a prefix of y; that is, $N > 0$. It is not hard to see that $M \geq N$. Indeed, w may have y as a prefix so w starts with x^N and we obtain $M \geq N$; otherwise, w has x, but not y, as prefix and, thus, it starts with $xx^{p-1}y$, for some $p > 0$ with $p + N \leq M$, and it follows that $M > N$.

Further, we split the discussion in two more cases. The first case is when x is not a period of y. Since $w \in \{x, y\}^* y\{x, y\}^*$ it follows easily that y occurs in w after one of the prefixes x^k, with $M - N \leq l \leq M$, of the word w. Note that k cannot be strictly smaller than $M - N$, as we would obtain that y has a prefix x^p with $p > N$, a contradiction. However, if $k > M - N$, as y starts with x^N, we get that w has a prefix x^k with $k > M$, a contradiction. Therefore, we have $w \in x^{M-N}y\{x, y\}^*$.

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 is a proper prefix of x^{N+1}. As in the previous case, we obtain that in a 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 $k \leq M-N$, of the word w. 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 $w' = (x^k)^{-1}x^M$ has length at least $(N+2)|x| > |x|+|y|$ 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 see now that $w \notin x^{M-N}y\{x,y\}^* \cap x^{M-N-1}y\{x,y\}^*$. For the sake of contradiction, assume that $w \in x^{M-N}y\{x,y\}^* \cap x^{M-N-1}y\{x,y\}^*$. We get that $w'' = (x^{M-N-1})^{-1}w$ has length at least $|x|+|y|$ and is in $x\{x,y\}^* \cap y\{x,y\}^*$; by Theorem 1 we get that both y and x are powers of the same word, a contradiction. So w is either in $x^{M-N}y\{x,y\}^*$ or in $x^{M-N-1}y\{x,y\}^*$. As y starts with x, it follows that the conclusion of the Lemma holds. □

Further, we give several results regarding basic combinatorial and algorithmic properties of repetitions and periods in words.

Lemma 5. *Let x and y be primitive words such that x^2 is a prefix of y^2. Then $|y| > (M-1)|x|$ where $M = \max\{p \mid x^p$ is a prefix of $y^2\}$.*

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 following result follows easily.

Corollary 1. *Let $w \in V^*$ be a word of length n. There are at most $\mathcal{O}(\lg n)$ primitive words x such that x^3 is a prefix of w.*

Lemma 6. *Let $w \in V^*$ be a word of length n. We can compute the values $per[i]$, the period of $w[1..i]$, for all $i \in \{1, \ldots, n\}$ in linear time $\mathcal{O}(n)$.*

Proof. One may note that the 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 $i \in \{1,\ldots,n\}$ 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 lemma is a simple consequence of Lemma 6.

Lemma 7. *Let $w \in V^*$ be a word of length n. We can compute the values $per[i][j]$, the period of $w[i..j]$, for all $i,j \in \{1,\ldots,n\}$ in quadratic time $\mathcal{O}(n^2)$.*

2.2 Solution of Problem 1

A general solution.

Let us first assume that f is a morphism, and that the input word w has length n. We can construct in $\mathcal{O}(n)$ time the word $x = wf(w)$ of length $m = n + |f(w)|$ (which is in $\mathcal{O}(n)$); notice that the constant hidden by the $\mathcal{O}$-notation and the length of x depend on the morphism f. Moreover, we construct in $\mathcal{O}(n)$ time data structures enabling us to answer $LCPref$ queries for x.

Based on Lemma 4, Algorithm 2 describes a strategy one can use to test whether w is an f-repetition or not.

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 requires locating the occurrences of w in ww. Further, note that the computations in each of the steps $6-9$ of the algorithm can be executed in constant time using the data structures we already constructed. Indeed, for some $s \leq n$, we can compute the largest ℓ such that $w[s..\ell]$ is a power of x in constant time as follows. In the worst case, $\ell = s-1$, or, in other words, $w[s..\ell] = \lambda$ if x does not occur at position s. Otherwise, ℓ is the largest number less than or equal to $LCPref(w[s..n], w[s+|x|..n])$ such that $\ell - s + 1$ is divisible by $|x|$. This strategy is used in steps 6 and 7. Steps 8 and 9 can be also implemented in constant time using $LCPref$ queries. Moreover, the iterative process in steps $3-9$ is executed for each prefix $w[1..i]$ of w, and during each iteration the algorithm makes at most $\mathcal{O}(\lfloor \frac{n}{\ell'_i} \rfloor)$ steps, as s can take at most $\lfloor \frac{n}{\ell'_i} \rfloor$ different values. In the general case, when f can erase letters, we note

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 an f-repetition, as repetitions can be seen as trivial f-repetitions. The algorithm continues for w primitive.}
2: **for** all the prefixes $t = w[1..i]$ of w, with $i < n$ and $len[i] \geq 1$ **do**
3: Set $x = t$ and $y = f(t)$ if $i \leq len[i]$ or $x = f(t)$ and $y = t$, otherwise;
4: Set $s = i + 1$, $\ell'_i = \max\{i, len[i]\}$, $\ell''_i = \min\{i, len[i]\}$;
5: If $s = n + 1$ we halt and decide that w is an f-repetition; {Note that $|x| = \ell'_i$ and $|y| = \ell''_i$.}
6: Compute $M = \max\{p \mid x^p \text{ is a prefix of } w\}$;
7: Compute $N = \max\{p \mid x^p \text{ is a prefix of } y\}$;
8: If $x^{M-N-1}yx$ occurs at position s, set $s = (M - N - 1)\ell''_i + \ell'_i$, go to step 5;
{By Lemma 4, $w[s..n]$ may be in $x^{M-N-1}y\{x, y\}^*$, and it has $x^{M-N-1}yx$ as a prefix, but it is not in $x^{M-N}y\{x, y\}^*$.}
9: If $x^{M-N}y$ occurs at position s, set $s = (M - N)\ell''_i + \ell'_i$, go to step 5;
{By Lemma 4, $w[s..n]$ may be in $x^{M-N}y\{x, y\}^*$ but it does not have $x^{M-N-1}yx$ as a prefix.} {If none of the above case holds, by Lemma 1 we get that $w[i + 1..n] \notin \{t, f(t)\}^+$, so $w \notin t\{t, f(t)\}^+$.}
10: **end for**
11: Halt and decide that w is not an f-repetition.

that $w \in \{t, f(t)\}^+$ for some t such that $f(t) = \lambda$ is equivalent to $w \in \{t\}^+$, i.e., w is a repetition. Therefore, we run the iterative process only for prefixes $w[1..i]$ with $len[i] \geq 1$. Since $\ell'_i \geq i$, 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 complexity of Algorithm 2 is upper bounded by $\mathcal{O}(n \lg n)$.

Fortunately, when f is uniform we obtain a more efficient algorithm. In this case, both $|t|$ and $|f(t)|$ divide 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. This includes the construction of the data structures.

A linear time solution for Problem 1, the case when f is uniform.

We can obtain an even faster solution for Problem 1, in the case when f is uniform, that uses some more intricate precomputed data structures in order to speed-up Algorithm 2. So, let us analyse again the computation performed by that algorithm on an input word w.

The main phase of the algorithm is the following. For a fixed prefix $t = w[1..i]$ of w with $i \mid n$ we run a cycle (in steps $5 - 9$) that extend 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 actually equal to 0, we can only say that the number of iterations of the cycle is upper bounded by $\frac{n}{|t|}$. Here we plug in our speed-up strategy: we try to extend the prefix in each of the iterations of the cycle from steps $5-9$ 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 to $\frac{n}{\alpha|t|}$, and the overall complexity of the algorithm to $\mathcal{O}(\frac{n \lg \lg n}{\alpha})$. Finally, in order to obtain an algorithm solving Problem 1 in linear time, we choose $\alpha = \lceil \lg \lg n \rceil$.

It only remains to show how we can implement the extension of the prefix, mentioned above, efficiently. First of all, let us 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 $|t| \mid n$ (which are at most $(\lg n)^2 \lg \lg n$) takes $\mathcal{O}(n)$ time. Therefore, we will only consider in the following the case when the prefix t fulfils $|t| \mid n$ and $|t| < \frac{n}{(\lg n)^2}$.

Now consider a fixed t, as above. There are at most $2^{\alpha+1} \in \mathcal{O}(\lg n)$ different words in $\cup_{m \leq \alpha} \{t, f(t)\}^m$. Every such word can be clearly encoded by a bit-string of length at most α: each occurrence of a t is encoded by 0 and an occurrence of $f(t)$ by 1. Denote these bit-strings $v_1, \ldots, v_{2^{\alpha+1}}$. We can generate these strings in $\mathcal{O}(\lg n \lg \lg n)$ time and we need the same time to sort them according to the length of the word they encode: the word encoded by v_i is at most as long as the word encoded by v_j, for $i < j$. We also store the length of the word encoded by v_j for each j. Clearly, we can determine by binary search for v_ℓ two values b_ℓ and e_ℓ such that all the suffixes contained in the suffix array of w between positions b_ℓ and e_ℓ have the word encoded by v_ℓ as a prefix. The time needed to compute these values for each ℓ is $\mathcal{O}(\lg n \lg \lg n)$, as a comparison between the word encoded by v_ℓ and a suffix of w can be done in $\mathcal{O}(\lg \lg n)$ by comparing only factors of length $|t|$ and $|f(t)|$ of the two words. Therefore, the time needed to compute these values for all ℓ is $\mathcal{O}((\lg n)^2 \lg \lg n)$. Assume that we construct a set B_t containing the values b_ℓ ordered increasingly, and a set E_t containing the values e_ℓ ordered increasingly. Note that both B_t and E_t contain $\mathcal{O}(\lg n)$ integers from $\{1, 2, \ldots, n\}$.

Before ending the preprocessing phase, we need one more result. The atomic heaps of Fredman and Willard [9] allow us to maintain a collection of sets $S(i) \subseteq \{0, 2, \ldots, n-1\}$ so that inserting, removing and finding successor in each of those sets work in constant time (amortized for insert and remove, worst case for find) as long as $|S(i)| \leq \lg n$ for all i, assuming a $\mathcal{O}(n)$ time and space preprocessing. Note that these time bounds hold in the computational model we use here. Therefore, once the computation of B_t and E_t ends, for all the prefixes of t of w, we can organize these sets using the atomic heaps data structure.

By the explanations given previously, computing all these data structures takes linear time. We now show how the precomputed data structures can be used to solve in linear time Problem 1.

Assume now that we are at step 5 of the algorithm for some prefix t of w with $|t| \mid n$, $|t| \leq \frac{n}{\lg n \lg\lg n}$, and $s \leq n - \alpha|t| + 1$. We may assume that $f(t)$ is not a power of t. We want to check whether $w[s..n]$ starts with a word encoded by one of the bit-strings v_ℓ. We obtain the greatest ℓ such that the index of $w[s..n]$ in the suffix array is between b_ℓ and e_ℓ (that is, the longest word encoded by one of the strings w_ℓ which is a prefix of $w[s..n]$); this can be done in $\mathcal{O}(1)$ using the precomputed data structures. Then, we repeat the procedure for the word $w[s'..n]$ where $w[s..s'-1]$ was the word encoded by v_ℓ, and only if $s' - s \geq \alpha|t|$ or $s' = n + 1$. Clearly, this process takes $\mathcal{O}(\frac{n}{\alpha t})$ steps for each t, so the overall algorithm makes $\mathcal{O}(n)$ steps.

We only have to show that it works correctly. The soundness is proven by two simple remarks. If $w[s..n]$ starts with both x and x', with $|x| > |x'|$ and each of x and x' encoded by some of the strings v_j, it is enough to consider in the next iteration only the word $w[s + |x|..n]$. Indeed, it is plain that if x' would lead to a decomposition of $w[s..n]$ in factors t and $f(t)$ we either get that $f(t)$ is a power of t or $(x')^{-1}x \in \{t, f(t)\}^*$. Now, if $w[s..n]$ starts with x, where x is the longest prefix of $w[s..n]$ encoded by some of the strings v_j, and $s < n - \alpha|t| + 1$ then we have $w[s..n] \in \{t, f(t)\}^*$ if and only if $|x| \geq \alpha|t|$.

To conclude, we described a correct implementation of the general Algorithm 2 that runs in linear time (thus, optimal time) for f an uniform morphism or anti-morphism.

Summary.

The case when f is an antimorphism is very similar. We construct the word $x = wf(w)$ and the same data structures as in the former case. Now we have $w[s+1..s+i] = t$ if and only if $LCPref(s+1, 1) \geq i$, and $w[s+1..s+len[i]] = f(t)$ if and only if $LCPref(s+1, m-len[i]+1) = len[i]$, where $m = |wf(w)|$. The rest of the reasoning remains unchanged.

Therefore, we can state the following theorem.

Theorem 2. *Let $f : V^* \to V^*$ be an anti-/morphism. Given $w \in V^*$, one can decide whether $w \in t\{t, f(t)\}^+$ in $\mathcal{O}(n \lg n)$ time. If f is uniform one can decide whether $w \in t\{t, f(t)\}^+$ in $\mathcal{O}(n)$ time.*

A more general problem.

The more general problem of testing, for an anti-/morphism f, whether a word w is in $\{t, f(t)\}^+$, where t is a factor of w is also worth considering. 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 $|x| \leq |y|$. Assume now that $y = w[1..i]$ is 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 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 exits 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 exist x occurring at position $k|y| + 1$ that fulfils the above conditions, we get 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. Therefore, there are 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 there exists a unique word x beginning on 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 immediately decreased to $\mathcal{O}(n \lg \lg n)$ when f is uniform; moreover, with the same preprocessing described previously we can decrease the complexity even more, in this case, to $\mathcal{O}(n)$.

Finally, we consider the general case, when f can erase letters. The solution, in this case, is much more involved, but it preserves the same time complexity. So, given a word $w \in \Sigma^*$ 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 some prefix t.

Fact 1 *Lemma 4 and the cycle from the steps $3-9$ of Algorithm 2 give us a method to decide, for a fixed word t, whether w is in $\{t, f(t)\}^*$ in $\mathcal{O}\left(\frac{n}{\max\{|t|,|f(t)|\}}\right)$ time.* □

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

Indeed, we try to guess $y = f(x)$. More precisely, we iterate over all of its possible lengths, then 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 map 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 move the endpoint of x to the right. Moreover, the value of $\max\{|f(x)|, |x|\}$ strictly increases after each application of the method in Fact 1. Therefore the total complexity is upper bounded 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 $u_1, u_2, \ldots, u_k$, where $|u_i| < |u_{i+1}|$. From Corollary 1 the number k of such different u possible is at most $\mathcal{O}(\lg n)$. We can identify all of them in $\mathcal{O}(n)$ time, by Lemma 6.

We will 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 such that $f(t) = u^k$ and t starts after the i-th occurrence of u, for some $i \in \{1, \ldots, m\}$. 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 single value of 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.*

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

Now consider the question of finding this unique value of 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 $f(w)$. Then we iterate through all sufficiently large of those powers, and for each of them mark a range of t'. Because of the above reasoning, any t' will be considered at most once there. □

If $f(u)$ is not a power of u, we apply the above fact 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 $\alpha \geq 0$.

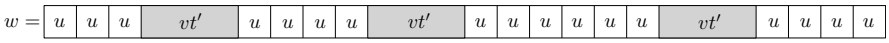

Fig. 1. Unique decomposition of w into u and vt'.

As in the previous case, we fix the place t ends at, and try to consider all $i \leq m$ such that $t = u^i v t'$. Because $u \neq v$, w has unique decomposition into copies of vt' and u, see Figure 1. 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, so let $f(vt') = u^\beta$, and we are looking for $i \leq \min(l_1, \ldots, l_j)$ such that the following system of equations holds:

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

We can transform (1) to get the following form:

$$\begin{aligned} &\alpha i + \beta \mid l_{k+1} \\ &\alpha i + \beta \mid l_1 - i \\ &\alpha i + \beta \mid l_j - l_1 \quad \forall_{j=2,\ldots,k} \end{aligned}$$

and finally:

$$\alpha i + \beta \mid d = \gcd(l_{k+1}, l_2 - l_1, \ldots, l_k - l_1) \tag{3}$$
$$\alpha i + \beta \mid l_1 - i \tag{4}$$

The first step of the transformation clearly requires at most $\mathcal{O}(k)$ time. The second step involves computing the greatest common divisor of k numbers not exceeding n, and can be performed in just $\mathcal{O}(k+\lg n)$ arithmetical operations, by applying the Euclid algorithm iteratively. Assuming we have computed d, we can iterate through all of its divisors, and for each of them check if (4) holds. Note that the number of divisors of a number not exceeding m can be bounded by $Kn^{\frac{1}{\lg\lg n}}$, for a constant K (see Wigert's result on the rate of growth of the divisor function, [7]). As $m \leq \frac{n}{|u|}$, the total complexity for a fixed t' is upper bounded by $\mathcal{O}(\frac{n^{1+\frac{1}{\lg\lg n}}}{|u||vt'|})$. Summing over all t' we get the upper bound $\mathcal{O}(\frac{n^{1+\frac{1}{\lg\lg n}}\lg n}{|u|})$.

For a fixed u we need $\mathcal{O}(\frac{n^{1+\frac{1}{\lg\lg n}}\lg n}{|u|})$ time. Summing up over all u (keeping in mind that $2|u_j| \leq |u_{j+1}|$) we conclude that we can check whether $w \in f(t)\{t, f(t)\}^+$ in $\mathcal{O}(n^{1+\frac{1}{\lg\lg n}}\lg n)$ time.

In conclusion, we showed the following theorem.

Theorem 3. *Let $f : V^* \to V^*$ be an anti-/morphism. Given $w \in V^*$, we decide whether $w \in \{t, f(t)\}\{t, f(t)\}^+$ in $\mathcal{O}(n^{1+\frac{1}{\lg\lg n}}\lg n)$ time. If f is non-erasing we solve the problem in $\mathcal{O}(n\lg n)$ time, while when f we only need $\mathcal{O}(n)$ time.*

2.3 Solution of Problem 2

Solution of Problem 2 for general morphisms:

To begin with, given an arbitrary anti-/morphism f and a word w of length n, we compute in $\mathcal{O}(n^5)$ time data structure that enable us to answer *rep*-queries in constant time. More precisely, we construct an oracle-structure that already contains the answers to every such query.

We only give an informal description of our solution. Assume that we are given an input word of length n. The idea is to compute

the $n \times n \times n$ three dimensional matrix M such that $M[i][j][k] = 1$ if and only if there exists a word t such that $w[i..j] \in \{t, f(t)\}^k$, and $M[i][j][k] = 0$ otherwise. We proceed as follows. Let i be a position in w. For a non-empty prefix t of $w[i..n]$ and $j > i$ we compute all the values k such that $w[i..j] \in t\{t, f(t)\}^{k-1}$. Following the ideas used in the previous algorithms (that is, the methods used to check whether t or $f(t)$ occur at some specific position in w), one can compute these values, for some j, in time $\frac{j}{\ell_t}$, where $\ell_t = \min\{|t|, |f(t)|\}$. Indeed, we only need to look at the corresponding values for $j - |t|$ and $j - |f(t)|$, increase them with 1 and add them to the set of values corresponding to j if t and, respectively, $f(t)$ occur as a suffix of $w[i..j]$; therefore, one may have to check at most $2\frac{j-|t|}{\ell_t}$ values, and, in the end we will have at most $\frac{j}{\ell_t}$ values corresponding to j. This process takes $\mathcal{O}(\frac{(n-i)^2}{\ell_t})$ time. Then, we look for all the positions $j' \leq i$ such that $w[j'..i] \in \{f(t)\}^*$. Now, if $w[j'..i] = (f(t))^k$ and for $j > i$ we have that $w[i..j]$ is in $\{t, f(t)\}^{\ell'}$ for $\ell' \in \{k_1, k_2, \ldots, k_p\}$ then $w[j'..j]$ is in $\{t, f(t)\}^{\ell'}$ for $\ell' \in \{k + k_1, k + k_2, \ldots, k + k_p\}$. The matrix M is updated according to the newly discovered f-repetitions. This process is implemented in $\mathcal{O}(\frac{i(n-i)^2}{\ell_t^2})$ time. This strategy should be repeated for all the prefixes t of $w[i..n]$, that is $\mathcal{O}\left(\sum_t \text{ prefix of } {}_{w[i..n]} \left(\frac{i(n-i)^2}{\ell_t^2} + \frac{(n-i)^2}{\ell_t}\right)\right)$. Finally, we execute the algorithm for all values of i. We get that in the case of an erasing morphism (where we only know that $\ell_t \geq 1$ for all t) the overall complexity of this solution is $\mathcal{O}(n^5)$.

The case of non-erasing morphisms.

For f non-erasing, M can be computed in $\mathcal{O}(n^3)$ time.

Assume that we are given a word w of length n. We compute the array $M[i][j][k]$, for $i, j, k \in \{1, \ldots, n\}$, that contains the answers to all possible *rep*-queries for the word w. Initially, we set all the values $M[i][j][k]$ to be equal to 0.

By Lemma 7 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 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 of $T[k][m][j-i+1] = 1$.

Further we present the more complicated cases.

First, let i be a number from $\{1, \ldots, n\}$. 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 treated) and $k \geq 2$. To do this we try all the possible prefixes t of $w[i..n]$. Once we choose a $t = w[i..\ell]$ we know how to retrieve $f(t)$ (using $LCPref$-queries) and check whether t and $f(t)$ are powers of the same word. If not we continue, and 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 3 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 these words 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 have identified 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 identify also the triples (i, j, k) such that $w[i..j] \in f(t)\{t, f(t)\}^{k-1}$ for some t. The idea is similar to the above. For $i \in \{1, \ldots, n\}$ we go through 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 we store the pair $(m+|t|, e+1)$ and starting from this pair we compute by backtracking, 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 observation regarding this process is that, by Lemma 3, no two pairs having the first component equal to m are obtained. Therefore, as the number of values that m may take is upper bounded by n, we get that the entire computation

of the pairs takes linear time. Once we computed these pairs, we set $M[i][m][k] = 1$ for each pair (m, k) we obtained.

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

The case of literal morphisms

However, in the case when f is a literal anti-/morphism, we are able to construct faster some data structures enabling us to answer *rep* queries. To this end, we first create data structures that allow us to answer in constant time $LCPref$ queries for $wf(w)$, as in the 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 point 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 $w[j..i]$, where d divides both $i - i_1 + 1$ and $i - i_2 + 1$. If there exist t and t' such that $t \neq t'$ 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 Intuitively, $M[i][d]$ is obtained in constant time from $M[i - d][d]$ using $LCPref$ queries on $wf(w)$.

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

Let us begin by noting that $w[j..i] \in \{t, f(t)\}^k$ for some t with $|t| = d$ implies that either $w[j..i]$ contains a factor $w[j + \ell d..j + (\ell + 1)d - 1] = t$, for some $\ell \in \{0, 1, \ldots, \frac{i-j+1}{d}\}$, or $w[j + \ell d..j + (\ell + 1)d - 1] = f(t)$ for all $\ell \in \{0, 1, \ldots, \frac{i-j+1}{d}\}$. In the latter case, it follows that $w[j..i] \in \{t'\}^k$ for $t' = w[j..j + d - 1]$. So, in any case, we can assume that if $w[j..i] \in \{t, f(t)\}^k$ then there exists t' such that $w[j..i] \in \{t', f(t')\}^k$ and there exists a factor $w[j + \ell d..j + (\ell + 1)d - 1] = t'$, for some $\ell \in \{0, 1, \ldots, \frac{i-j+1}{d}\}$.

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 number 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 number 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 number such that $j \leq i_2 < i$, $d \mid i-i_2+1$, and $w[i_2..i_2+d-1] = f(w[i_1..i_1+d-1])$, $d \mid i_2-j$; 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 just set $i_2 = -1$.

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

In other words, $M[i][d]$ stores the beginning point of the longest word $w[j..i]$ contained in $\{t, f(t)\}^+$, for some t of length d, as well as the last occurrences of t and $f(t)$ in this word.

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

– $M[d + \ell][d] = (\ell, \ell, \ell)$, 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: We update the values stored in this matrix, by traversing each of its column for $i = 2d$ 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, $w[i-d+1..i] = f(w[i_1..i_1+d-1])$, so i_2 needs to be updated.
– $M[i][d] = (j, i - d + 1, i_2)$ 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, $w[i-d+1] = w[i_1..i_1 + d-1]$, so i_1 needs to be updated.
– $M[i][d] = (i_2+d, i-d+1, i-2d+1)$ 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 $w[i_1..i_1 + d - 1] \neq w[i_2..i_2 + d - 1]$.
- Otherwise, $M[i][d]$ remains unchanged.

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 $x = wf(w)$.

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. In this case, we did not precompute the answers to all the possible queries.

Solving Problem 2.

Generally, one can use the computed data structures to efficiently identify, given a word w of length n and a number k, the factors $w[i..j]$ from $\{t, f(t)\}^k$ for some t, thus, solve item (i) of Problem 2. Indeed, we put in the solution-set of the problem all the pairs (i, j) for which $rep(i, j, k)$ returns yes. The time needed to do so is $\Theta(n^2)$, as we go through all possible pairs (i, j) and check whether $rep(i, j, k)$ returns yes or no. Note that this time bound does not depend on k. Furthermore, any algorithm solving this problem needs $\Omega(n^2)$ operations in the worst case, as, for instance, all factors $w[i..i+k\ell-1]$ of the word a^n are in $\{a^\ell\}^k$ (no matter what anti-/morphism f is used), thus, the number of elements in the solution-set is in $\Theta(n^2)$. Our result improves in a more general framework the results reported in [2], where the same problem was solved in time $\mathcal{O}(n^2 \lg n)$.

Finally, we can use *rep* queries to efficiently enumerate, given a word w of length n, all the triples (i, j, k) such that $w[i..j] \in \{t, f(t)\}^k$ for some t, thus, solve item (ii) of Problem 2. In the general case, the solution is straightforward: we try all possible values for i, j, and k and return the set of all the triples (i, j, k) for which $rep(i, j, k)$ returns yes. The complexity is $\Theta(n^3)$. Note that for the word a^n and f the erasing morphism that maps a to λ we actually have $\Theta(n^3)$ triples that fulfil the above condition. The same bound holds for f non-erasing. Take $f(a) = aa$ and $w = a^n$ and it follows that $w[i..j]$

is in $\{a, f(a)\}^k$, for all i and j with $\lfloor (j-i+1)/2 \rfloor \leq k \leq j-i+1$; this means that for this choice of w we have $\Theta(n^3)$ triples (i, j, k) that fulfil the conditions of item (ii).

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 verify 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 $\mathcal{O}(n \lg n) + \Theta(\sum_{1 \leq \ell \leq n} (n-\ell+1) d(\ell))$, where $d(\ell)$ is the number of divisors of ℓ. Following Lemma 1, the overall complexity of this algorithm is $\Theta(n^2 \lg n)$. Moreover, any algorithm solving this problem does $\Omega(n^2 \lg n)$ operations in the worst case: for the word a^n and the anti-/morphism $f(a) = a$, a correct algorithm has exactly $\sum_{1 \leq \ell \leq n} (n-\ell+1) d(\ell) \in \Theta(n^2 \lg n)$ triples in the solution set. This proves our claim.

Summary.

Before concluding this section, recall that the key idea in our approach is to use *rep* queries. In order to assert the efficiency of this method note that, when general anti-/morphisms are considered, the most time consuming task is to construct data structures that permit us to answer in constant time to *rep* queries. Once this preprocessing part is completed, our algorithms (essentially based on the possibility of answering *rep* queries) solve the two parts of Problem 2 efficiently. In particular, no other algorithm solving Problem 2 can run better than ours (excluding the preprocessing part), in the worst case. Thus, a faster preprocessing yields a faster complete solution for the problem. The same discussion can be made in the case of our solution for item (1) of the problem in the case of non-erasing morphisms: decreasing the preprocessing from cubic time to quadratic time would yield an optimal solution. However, in the case of item (2) for non-erasing (including uniform) anti-/morphisms, where the preprocessing is done in cubic time, and in the case of both items for literal anti-/morphisms (the case including the biologically motivated case of involutions), where the preprocessing part is done in quadratic time, the solutions are optimal.

Theorem 4. *Let $w \in V^n$ and $f : V^* \to V^*$ an anti-/morphism.*
(1) One can identify in time $\mathcal{O}(n^5)$ the pairs (i, j) such that $w[i..j] \in \{t, f(t)\}^k$ for a prefix t of $w[i..j]$, when k is given as input.
(2) One can identify in time $\mathcal{O}(n^5)$ the triples (i, j, k) with $w[i..j] \in \{t, f(t)\}^k$, for a prefix t of $w[i..j]$.
If f is non-erasing, we solve both questions of Problem 2 in $\mathcal{O}(n^3)$ time. For f literal we solve (1) *in $\Theta(n^2)$ time and* (2) *in $\Theta(n^2 \lg n)$ time.*

2.4 Solution of Problem 3

The tractable cases.

As already mentioned, the most general form of this problem is trivial. In fact, even when we restrict our search to non-erasing or uniform anti-/morphisms, the same solution works.

On the other hand, one can show (see the technical details below), using the algorithmic techniques of the previous sections, that when we look for a literal anti-/morphism f the problem is no longer trivial, and can be solved in time $\mathcal{O}(n \lg \lg n)$. Furthermore, there are two natural ways to restrict the statement of Problem 3 in order to make it nontrivial for even more general functions f. In particular, we may ask that the prefix t has at least two letters, or that $w \in t\{t, f(t)\}\{t, f(t)\}^+$.

Several cases remain tractable, and are also presented below, after this short overview. We can decide in linear time the existence of a general anti-/morphism f and of a prefix t of w with $|t| \geq 2$ such that $w \in t\{t, f(t)\}^+$. Also, building on the algorithms solving Problem 1, we can solve both restricted variants of the problem for f uniform in $\mathcal{O}(n \lg \lg n)$ time, where the constant hidden by the $\mathcal{O}$-notation depends on f.

1: The general problem.
Let us assume in the following that we are interested in finding a literal anti-/morphism f that fulfils the conditions of Problem 3. The problem is solved by checking whether there exist a prefix x and a factor y of w of equal length such that $w \in x\{x, y\}^+$. This strategy is implemented in fashion similar to the solutions of Problem 1. Basically we construct a suffix-array structure for w, and then

check for each prefix x of w, whose length divides $|w|$, whether w is in $x\{x,y\}^+$, where y is the first factor of w, of length equal to $|x|$, which is not equal to x. Once we obtained such x and y, we define a morphism (respectively, antimorphism) f such that $f(x) = y$ by associating to the i^{th} symbol of x the i^{th} symbol of y, or, respectively, the $(|y| - i)^{th}$ symbol of y. A positive answer is given whenever suitable prefixes and anti-/morphisms are found. The total running time of the algorithm is $\mathcal{O}(\sum_{\ell|n}(\ell + \frac{n}{\ell})) = \mathcal{O}(n \lg \lg n)$, where for a prefix of length ℓ we make $\frac{n}{\ell}$ steps to decide if $w \in x\{x,y\}^+$ and another ℓ steps to construct the morphism. Clearly, an identical strategy can be used to decide whether there exist a factor t of w and a literal anti-/morphism f such that $w \in \{t, f(t)\}\{t, f(t)\}^+$. The case when f is uniform, subject to the restriction $k \neq |w|-1$ where k is the length of $f(a)$ for $a \in V$, is solved analogously with the only difference that the word y is now defined as having length $k|x|$.

2: Restricted variants.
We consider first the case of general anti-/morphisms. The problem we solve is: *"Given $w \in V^*$ decide whether there exist an anti-/morphism $f : V^* \to V^*$, and a prefix t of w with $|t| \geq 2$ such that $w \in t\{t, f(t)\}^+$"*.
One can solve this problem efficiently. 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. If $w = a^{2n}$, for some $a \in V$ and $n \geq 2$, we take f to be the function that maps a to a, and $t = a^2$. If $w = a^{2n+1}$ and $2n + 1$ is prime, then the problem has no solution: if $t = a^k$ then $2n + 1$ would be divisible by k, so $k = 1$ or $k = 2n + 1$, 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. 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 $w = tw'$. So the problem above can be easily decided, in linear time.

In the case of literal (and uniform) anti-/morphisms, we begin by noting that the solution in the paragraph **1.** of this section can be easily adapted to find a prefix t and a literal (respectively, uniform)

anti-/morphism such that $w \in t\{t, f(t)\}^+$ with $t \geq 2$ or with $w \in t\{t, f(t)\}\{t, f(t)\}^+$. We just have to search for prefixes x of w that have length greater or equal to 2, in the first case, and to be sure that the number of factors x and y in a solution of the problem found by our algorithm is greater or equal to 3, in the second case.

The computationally hard cases

There exist though other NP-complete cases. When we consider the restriction $|t| \geq 2$ for non-erasing anti-/morphisms the problem is NP-complete: *"Given $w \in V^+$ 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 $w \in t\{t, f(t)\}^+$"*.

Clearly, this problem is in NP. It can be shown to be NP-complete by giving a polynomial-time reduction from the pattern-description problem: *"Given two words x and y decide whether there exists a morphism g such that $g(x) = y$"*. This problem was defined and shown to be NP-complete in [10]; it remains hard for the case when g is considered a non-erasing morphism.

We can prove this result as follows.

We first show the NP-completeness of the case when f is a morphism. Assume now 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 $a, b \notin V$. We show there exists a non-erasing morphism g such that $g(x) = y$ if and only if there exist a non-erasing morphism $f : (V \cup \{a, b\})^* \to (V \cup \{a, b\})^*$ 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 immediately 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, we immediately obtain that $x' = x$ and $f(x) = y$, so we can take $g = f$. This concludes the proof of this implication, and the equivalence that we have just shown exhibits a polynomial time re-

duction from the pattern-description problem to our problem. Thus, our problem is NP-complete.

The case of f being an antimorphism is just a little bit more involved. Clearly, this variant of the problem is in NP, as well. Now we show that it is NP-complete by giving a polynomial-time reduction from the pattern-description problem, used in the case of non-erasing morphisms. 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 $n = 2\max\{|x|, |y|\}$ 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 $f : (V \cup \{a, b, c, d, e, g\})^* \to (V \cup \{a, b, c, d, e, g\})^*$ 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 as a prefix a^n; 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 it follows immediately that $f(a) = g$, $f(b) = c$, $f(d) = e$ and $f(x) = y$. But 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 this implication. The equivalence that we have just shown exhibits a polynomial time reduction from the pattern-description problem to our problem. Therefore, our problem is also NP-complete.

The previous hardness result helped us complete the picture of how complex it is to solve Problem 3 subject to the restriction $|t| \geq 2$.

In the case of the second restriction the most general problem is NP-complete:
"Given $w \in V^+$ 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. We give a polynomial time reduction from the (general) 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 [10]. 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 $i \in \{1, \ldots, n\}$ let a_i and b_i be $2n$ symbols not contained in the alphabets of x or y. Let $w = a_1a_2 \ldots a_nxb_1b_2 \ldots b_nya_1a_2 \ldots a_nx$. 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 $t = a_1 \ldots a_k$, it is clear that $f(a_1 \ldots a_k) \neq \lambda$. Moreover, $f(a_1 \ldots a_k)$ is a suffix of w. It is not hard to note that $b_1b_2 \ldots b_ny$ is a factor of $f(a_1 \ldots a_k)$. Since there is only one occurrence of $b_1 \ldots b_ny$, we reach the conclusion that only one factor $f(a_1a_2 \ldots a_k)$ appears in w, which is impossible. Therefore, t has $a_1 \ldots a_n$ as a prefix. Assume now that t is a proper prefix of $a_1 \ldots a_nx$. Hence, $tf(t)$ is a prefix of w and $f(t)$ starts with a letter of x. Again, we easily get that $b_1 \ldots b_ny$ is a factor of $f(t)$. So, once more, we have exactly one occurrence of $f(t)$ in w. The only possibility is that $tf(t) = a_1 \ldots a_nxb_1 \ldots b_ny$ (otherwise, w would not be in $t\{t, f(t)\}\{t, f(t)\}^+$). But this is also impossible, as $a_1 \ldots a_nx$ cannot be written as an element of $\{t, f(t)\}^+$. Assume now that t has $a_1 \ldots a_nx$ 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 squares. Therefore, the only possibility is that $t = a_1 \ldots a_nx$. In this case, by a case analysis similar to the above, we get that $f(t) = b_1 \ldots b_ny$. 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 $\{b_1, \ldots, b_n\}$. Let P_1 be the first letter of x (following the notations of [10]). This letter is mapped by f to $g(a_1 \ldots a_nP_1)$ from which we delete any letter from $\{b_1, \ldots, b_n\}$. 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 in a similar fashion, choosing:

$$w = a_1 \dots a_n x c_1 \dots c_n b_1 \dots b_n y^R d_1 \dots d_n a_1 \dots a_n x c_1 \dots c_n.$$

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.

Summary.

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

Theorem 5. *Let $w \in V^+$ be a given word.*
1. One can decide in $\mathcal{O}(n)$ time the existence of an anti-/morphism $f : V^* \to V^*$ and a prefix t of w with $|t| \geq 2$ such that $w \in t\{t, f(t)\}^+$. The problem is NP-complete for f non-erasing, and solvable in $\mathcal{O}(n \lg \lg n)$ time for f uniform.
2. The problem of deciding whether there exist an anti-/morphism $f : V^* \to V^*$ and a prefix t of w such that $w \in t\{t, f(t)\}\{t, f(t)\}^+$, is NP-complete in the general case and in the case when f is non-erasing, and is solvable in $\mathcal{O}(n \lg \lg n)$ time when f is uniform.

References

1. Czeizler, E., Kari, L., Seki, S.: On a special class of primitive words. Theoretical Computer Science **411** (2010) 617–630
2. Chiniforooshan, E., Kari, L., Xu, Z.: Pseudopower avoidance. Fundamenta Informaticae **114**(1) (2012) 55–72
3. Lothaire, M.: Combinatorics on Words. Cambridge University Press (1997)
4. Fine, N.J., Wilf, H.S.: Uniqueness theorem for periodic functions. Proceedings of the American Mathematical Society **16** (1965) 109–114
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms (3. ed.). MIT Press (2009)
6. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM **53** (2006) 918–936
7. Apostol, T.M.: Introduction to analytic number theory. Springer (1976)
8. Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, New York, NY, USA (1997)
9. Fredman, M.L., Willard, D.E.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. J. Comput. Syst. Sci. **48**(3) (1994) 533–551
10. Ehrenfeucht, A., Rozenberg, G.: Finding a Homomorphism Between Two Words is NP-Complete. Inf. Process. Lett. **9**(2) (1979) 86–88