Finding Pseudo-repetitions

Paweł Gawrychowski[*1], **Florin Manea**[†2], **Robert Mercaş**[‡3], **Dirk Nowotka**[§2], **and Cătălin Tiseanu**[4]

1 **Max-Planck-Institut 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 natural generalization of the classical notion of repetitions in sequences. 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.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Stringology, Pattern matching, Repetition, Pseudo-repetition

Digital Object Identifier 10.4230/LIPIcs.STACS.2013.257

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 equals 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 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 $w \in t\{t, f(t)\}^+$.

Pseudo-repetitions, introduced in a restricted form by Czeizler et al. [3], lacked so far a well-developed algorithmic part. Given that the motivation for studying these objects originates from bioinformatics, where efficient algorithms are crucial, producing such tools seems not only natural but even necessary. This work is aimed to fill this gap. We investigate the following two basic algorithmic problems: decide whether a word w is a pseudo-repetition for an anti-/morphism f and find all k-powers of pseudo-repetitions occurring as factors in a word w, for an f as above; in these problems w is given as input, while f, although of unrestricted form, is fixed, thus not a part of the input. We establish algorithms and complexity bounds for these problems for various types of anti-/morphisms thereby improving significantly the results from [2]. Apart from the application of standard stringology tools, like suffix arrays, we extend the toolbox by nontrivial applications of results from combinatorics on words.

* Paweł Gawrychowski is supported by the *NCN* grant 2011/01/D/ST6/07164.
† Florin Manea is supported by the *DFG* grants 582014 and 596676.
‡ Robert Mercaş is supported by the *Alexander von Humboldt Foundation*.
§ Dirk Nowotka is supported by the *DFG Heisenberg* grant 590179.

Background and Motivation. The motivation of introducing pseudo-repetition and pseudo-primitivity in [3] 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 involutions, following the original motivation, and the results obtained were mostly of combinatoric nature (e.g., generalizations of the Fine and Wilf theorem - see, e.g., [3, 8]).

A natural extension of these concepts is to consider anti-/morphisms in general, which is done in this paper. Considering that the notion of repetition is central in combinatorics of words and the plethora of applications that this concept has (see [7]), 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 of antimorphic involutions, the general problem of identifying pseudo-repetitions can be seen as a formalization of scenarios where we are interested in identifying sequences having a hidden repetitive structure. Indeed, as each pseudo-repetition is an iterated catenation of a factor and its encoding through some simple function, such words have an intrinsic, yet not obvious, repetitive structure.

Some Basic Concepts. For more detailed definitions we refer to [7].

Let V be a finite alphabet; V^* denotes the set of all words over V and 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. We denote by $\mathrm{alph}(w)$ the alphabet of all letters that occur in w. A word $u \in V^*$ is a *factor* of $v \in V^*$ if $v = xuy$, for some $x, y \in V^*$; 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 $w[i], \dots, w[j]$, where $1 \le i \le j \le n$; we define $w[i..j] = \lambda$ if $i > j$. Also, 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 $n \ge 1$. 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 smallest period of w.

The following classical result is extensively used in our investigation:

▶ **Theorem 1** (Fine and Wilf [4]). *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 $x, y \in V^*$. In order to 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 $ACGTAC$ is primitive from the classical point of view (i.e., $\mathbf{1}$-primitive, where $\mathbf{1}$ is the identical anti-/morphism) as well as f-primitive for the morphic involution f defined by $f(A) = T$, $f(C) = G$, $f(T) = A$, and $f(G) = C$. However, for the antimorphic involution $f(A) = T$ and $f(C) = G$ (which is, in fact, a formalization of the Watson-Crick complement, from biology), we get that $ACGTAC = AC \cdot f(AC) \cdot AC$, thus, it is an f-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.

2 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 $\{1, \ldots, n\}$, so each letter fits in a single memory-word. This is a common assumption in algorithmics on words (see, e.g., the discussion in [6]).

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 $w \in V^*$, 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. 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 simple enough, 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 $w \in V^*$ a given word.
(1) Enumerate all (i, j, ℓ), $1 \le i, j, \ell \le |w|$, such that there exists t with $w[i..j] \in \{t, f(t)\}^\ell$.
(2) Given k, enumerate all (i, j), $1 \le i, j \le |w|$, so there exists t with $w[i..j] \in \{t, f(t)\}^k$.

Question (2) was originally considered in [2], while the first one is 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 $rep(i, j, \ell)$: "Is there $t \in V^*$ such that $w[i..j] \in \{t, f(t)\}^\ell$?", for $1 \le i \le j \le n$ and $1 \le \ell \le n$, where $n = |w|$. For unrestricted f, one 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, again a tight bound, and a $\mathcal{O}(n^2 k)$-time solution for the general case. Our results improve significantly the algorithmic results reported in [2].

2.1 Prerequisites

We begin this section by presenting several number theoretic properties. Lemma 2 is used in the time complexity analysis of our algorithms, while Lemma 3 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 2.** *Let n be a natural number. The following statements hold:*
(1) $\sum_{1\le\ell\le n} d(\ell) \in \Theta(n\lg n)$, $\sum_{1\le\ell\le n} d(\ell) \ge n\lg n$, $d(n) \in o(n^\epsilon)$ for all $\epsilon > 0$ (see [1]); (2) $\sigma(n) \in \mathcal{O}(n\lg\lg n)$ (see [1]); (3) $\sum_{1\le\ell\le n}(n-\ell+1)d(\ell) \in \Theta(n^2\lg n)$. ◀

▶ **Lemma 3.** *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 $k_1 s + k_2 sm = \ell$.* ◀

▶ **Corollary 4.** *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 $1 \le m \le R$ and $1 \le \ell \le n$, 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 $k_1 s + k_2 sm = \ell$. The constant hidden by the $\mathcal{O}$-notation depends on R.* ◀

We briefly present the data structures we use. For a word u with $|u| = n$ over $V \subseteq \{1, \ldots, n\}$ 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 $u[j..n]$?", denoted $LCPref(u[i..n], u[j..n])$. For more details, see [5, 6], and the references therein. Also, for u and an anti-/morphism f, we compute an array len with n elements defined as $len[i] = |f(u[1..i])|$, for $1 \le i \le n$. 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 5.** *Let $w \in V^*$ be a word of length n. We compute the values $per[i]$, the period of $w[1..i]$, for all $i \in \{1, \ldots, n\}$ in linear time $\mathcal{O}(n)$. Also, we compute the values $per[i][j]$, the period of $w[i..j]$, for all $i, j \in \{1, \ldots, n\}$ in quadratic time $\mathcal{O}(n^2)$.* ◀

Next we show an important property of pseudo-repetitions, for non-erasing morphisms.

▶ **Lemma 6.** *Let f be a non-erasing anti-/morphism, and x, y, z be words over V such that $f(x) = f(z) = y$. If $\{x,y\}^*x\{x,y\}^* \cap \{z,y\}^*z\{z,y\}^* \ne \emptyset$ then $x = z$.*

Proof. We sketch the proof only for the case when f is a morphism; a similar argument works for antimorphisms. If $\{x,y\}^*x\{x,y\}^* \cap \{z,y\}^*z\{z,y\}^* \ne \emptyset$ 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 $|x| \le |y|$ 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 $x = v^{k_1}$, $y = v^{k_2}$ 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 $u \ne \lambda \ne v$. Consequently, w' has a prefix of the form $x^p y$, with $p \ge 0$, 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 $z = vu$. 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. By an involved case analysis, it follows that x is a power of the same primitive word as y, a contradiction.

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 7.** *Let x and y be words over V such that x and y are not powers of the same word. If $w \in \{x, y\}^*$ then there exists a unique decomposition of w in factors from $\{x, y\}$.* ◀

▶ **Lemma 8.** *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 $N > 0$.* ◀

2.2 Solution of Problem 1

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

Using these data structures, Algorithm 1 tests whether w is an f-repetition or not.

Algorithm 1 $Test(w, f)$: 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 $t = w[1..i]$, such that $i < n$, $len[i] \geq 1$, $t$ and $f(t)$ are not powers of some $x \in V^*$ do
3:      Set $x = t$ and $y = f(t)$ if $i \leq len[i]$ or $x = f(t)$ and $y = t$, otherwise;
4:      Set $s = i + 1$, $\ell'_i = |y|$, $\ell''_i = |x|$; {We have $\ell'_i = \max\{len[i], i\}$ and $\ell''_i = \min\{i, len[i]\}$}
5:      If $s = n + 1$ halt and decide that $w$ is an $f$-repetition;
6:      Compute $M = \max\{p \mid x^p$ is a prefix of $w[s..n]\}$, $N = \max\{p \mid x^p$ is a prefix of $y\}$;
7:      If $w[s..n] = x^M$, set $s = n + 1$, go to step 5;{If $w[s..n] \in \{x\}^+$ then $w \in t\{t, f(t)\}^*$}
8:      If $x^{M-N}y$ occurs at position $s$, set $s = (M - N)\ell''_i + \ell'_i$, go to step 5;
9:      If $M > N$ and $x^{M-N-1}yx$ occurs at position $s$, set $s = (M-N-1)\ell''_i + \ell'_i$, go to step 5;
        {By Lemma 8, $w[s..n]$ should have either $x^{M-N}y$ or $x^{M-N-1}yx$ as prefix. By Lemma 8,
        if $x^{M-N-1}yx$ occurs at position $s$, we shall check whether $w[s..n] \in x^{M-N-1}y\{x, y\}^+$.}
        {If none of the above holds, we get that $w[s..n] \notin \{t, f(t)\}^+$, so $w \notin t\{t, f(t)\}^+$.}
10: end for
11: Halt and decide that $w$ is not an $f$-repetition.
```

Following the comments inserted in its description, it is not hard to see that Algorithm 1 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, note that the computation 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, $w[s..\ell] = \lambda$, when 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 step 6 to compute M and N. The verification from step 7 takes clearly constant time: we just check whether $n - s + 1 = M|x|$. Moreover, step 8 and 9 can also be implemented in constant time using *LCPref* queries; indeed, we know that x^{M-N} occurs at position s, and then we just have to check whether y occurs at position $s + (M-N)|x|$ by a *LCPref* query, for step 8, or, respectively, whether yx occurs at position $s+(M-N-1)|x|$ by two *LCPref* queries, for step 9. Further, 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 cycle defined by the "go to" instruction from step 8). 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 time complexity of Algorithm 1 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 $w \in \{t\}^+$, that is, w is 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 take $x = wf(w)$, build the same data structures, and proceed just as in the former case. As the single difference, now we have $w[s+1..s+len[i]] = f(w[1..i])$ iff $LCPref(s+1, m - len[i] + 1) = len[i]$, where $m = |x|$.

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 2.

§A linear time solution for the case when f is uniform. We can obtain an even faster solution for Problem 1 for the case when f is uniform by using some more intricate precomputed data structures in order to speed-up Algorithm 1. To this end, we analyse again the computation performed by Algorithm 1 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 a cycle (steps $5-9$) 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 actually 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 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 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 $a \in \mathrm{alph}(w)$; as f is uniform, the definition of R does not depend on the choice of a from V, and we also have $R = \frac{|f(u)|}{|u|}$, $\forall u \in V^*$. Let $r_t = \max\{\ell \mid t^\ell \text{ prefix of } f(t)\}$. 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 1; 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, 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 1 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. Therefore, 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 $v_1, \ldots, v_{2^\alpha}$, and let $\overline{v}_i$ be the word encoded by v_i, for all $1 \leq i \leq 2^\alpha$. 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 $i \neq j$, and 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 $\{1, \ldots, n\}$.

We need one more result before concluding this preprocessing phase. We want to store a static set $S \subseteq \{1, \ldots, n\}$ so that finding the successor in S of a given $x \in \{1, \ldots, n\}$ 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 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 2.

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 run step 5 of the algorithm 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 that it 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 1 runs in optimal linear time for f uniform.

§Summary. We were able to show the following theorem.

▶ **Theorem 9.** *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 we only need $\mathcal{O}(n)$ time.* ◀

The more general problem of testing whether there exists t with $w \in \{t, f(t)\}\{t, f(t)\}^+$ for f a fixed anti-/morphism is also worth considering. Solving this problem seems to require a different strategy than the one in Algorithm 1. 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 6 and 7, 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 10.** *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 is uniform we only need $\mathcal{O}(n)$ time.* ◀

2.3 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 $rep(i, j, \ell)$: "Is there $t \in V^*$ such that $w[i..j] \in \{t, f(t)\}^\ell$?", for all $1 \le i \le j \le |w|$ and $1 \le \ell \le |w|$. 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.

§The case of erasing morphisms. We start by presenting the solution of 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 query.

We only give an informal description of our construction. Assume that $|w| = n$. The idea is to compute the $n \times n \times n$ three dimensional 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. Note that, for such a prefix t of $w[i..n]$, with $t \ne \lambda \ne f(t)$, and $j > i$ there is at most one number 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 $1 \le \delta \le n^{0.5}$, 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 $1 \le i \le n$. 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 7, 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 $i \in \{1, \ldots, n\}$ and all prefixes t from S_i and discover all the factors $w[i'..j']$ and numbers k such that $\{f(t), t\}^k$, 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 of the sets $S_{i,\delta}$, for some fixed $\delta < n^{0.5}$ and all $1 \le i \le n$. 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. Now 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 $0 \le d < \frac{\ell}{\delta}$. 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 $\delta \le n^{0.5}$, we get that this case requires $\mathcal{O}(n^{3.5})$, as well. Now, we know all triples (i, j, k) such that $w[i..j] \in \{t, f(t)\}^k$ and t and $f(t)$ are 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 5 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 3. 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 exists s, k_1, 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 remained to be discussed. If $f(w[i..j]) = \epsilon$, then $M[i][j][k] = 1$, for all $k \ge 1$. Identifying and memorising 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 4 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 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 7 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.

§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 $i, j, k \in \{1, \dots, n\}$.

As in the case of erasing morphisms, by Lemma 5 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 3.

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 $\{1, \dots, n\}$. 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 7 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 $i \in \{1, \dots, n\}$ 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 6, 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 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.

Now, consider the case when we search f-repetitions with k factors, for a given k and f non-erasing. 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 answers to a query $rep(i, j, k)$ for the given value k is given by the entry $M_k[i][j]$ of the matrix M_k.

§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. Intuitively, $M[i][d]$ is obtained in constant time from $M[i - d][d]$ using *LCPref* queries on $wf(w)$.

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.

§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 one 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 $w = a^n$. 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 $\mathcal{O}(n \lg n) + \Theta(\sum_{1 \leq \ell \leq n} (n - \ell + 1) d(\ell))$. Following Lemma 2, 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 $w = a^n$ 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. This proves our claim.

In the case of the second question of our problem, we proceed as follows. Recall that, in this case, we are given both a word w and a number k. 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 $\mathcal{O}(n^2k)$ time, and this is more than the time needed to actually answer all the queries. This 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, note that the bounds obtained for 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 w is in $\Theta(n^2)$.

§Summary. Before concluding this section, recall that the key idea in our approach is 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. However, 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 questions of the problem. Thus, the time bounds obtained in these cases are tight.

▶ **Theorem 11.** *Let $f : V^* \to V^*$ be an anti-/morphism and $w \in V^*$ a given word, $|w| = n$.*
(1) One 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) One can identify in time $\mathcal{O}(n^2k)$ 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.
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.

References

1 T. M. Apostol. *Introduction to analytic number theory.* Springer, 1976.
2 E. Chiniforooshan, L. Kari, and Z. Xu. Pseudopower avoidance. *Fundam. Informat.*, 114(1):55–72, 2012.
3 E. Czeizler, L. Kari, and S. Seki. On a special class of primitive words. *Theoret. Comput. Sci.*, 411:617–630, 2010.
4 N. J. Fine and H. S. Wilf. Uniqueness theorem for periodic functions. *Proc. of the American Mathemat. Soc.*, 16:109–114, 1965.
5 D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, New York, NY, USA, 1997.
6 J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53:918–936, 2006.
7 M. Lothaire. *Combinatorics on Words.* Cambridge University Press, 1997.
8 F. Manea, R. Mercas, and D. Nowotka. Fine and Wilf's theorem and pseudo-repetitions. In *MFCS*, volume 7464 of *LNCS*, pages 668–680. Springer, 2012.