Online Algorithms on Antipowers and Antiperiods

Mai Alzamel, Alessio Conte, Daniele Greco, Veronica Guerrini, Costas Iliopoulos, Nadia Pisanti, Nicola Prezza, Giulia Punzi, Giovanna Rosone

Online Algorithms on Antipowers and Antiperiods

Mai Alzamel[2,3], Alessio Conte[1], Daniele Greco[1], Veronica Guerrini[1], Costas Iliopoulos[2], Nadia Pisanti[1], Nicola Prezza[1], Giulia Punzi[1], and Giovanna Rosone[1]

[1] Department of Computer Science, University of Pisa, Italy
[conte,veronica.guerrini,pisanti,nicola.prezza]@di.unipi.it,
grc.daniele.cs@gmail.com, giuliagpunzi@gmail.com, giovanna.rosone@unipi.it
[2] Department of Informatics, King's College London, UK
[mai.alzamel,costas.iliopoulos]@kcl.ac.uk
[3] Department of Computer Science, King Saud University, KSA

Abstract. The definition of *antipower* introduced by Fici et al. (ICALP 2016) captures the notion of being the opposite of a *power*: a sequence of k pairwise distinct blocks of the same length. Recently, Alamro et al. (CPM 2019) defined a string to have an *antiperiod* if it is a prefix of an antipower, and gave complexity bounds for the offline computation of the minimum antiperiod and all the antiperiods of a word. In this paper, we address the same problems in the *online* setting. Our solutions rely on new arrays that compactly and incrementally store antiperiods and antipowers as the word grows, obtaining in the process this information for all the word's prefixes. We show how to compute those arrays online in $O(n \log n)$ space, $O(n \log n)$ time, and $o(n^\epsilon)$ delay per character, for any constant $\epsilon > 0$. Running times are worst-case and hold with high probability. We also discuss more space-efficient solutions returning the correct result with high probability, and small data structures to support random access to those arrays.

Keywords: antiperiod · antipower · power · periodicity · repetition · regularities · online algorithms

1 Introduction

String properties that highlight regularities such as periodicity, powers, repetitions, palindromes, as well as properties that—dually—highlight diversity (e.g. being square-free, non-periodic, etc.), have been extensively investigated in literature. They have been studied both in terms of combinatorial properties [25,26] and of algorithmic methods to detect or certify them in a single string [10,9], or finding maximal common factors that share such properties [8,5,18,19,3,2]. In addition to having a combinatorial interest on their own, such string properties are also very relevant for several applications [27,28,9]. For instance, they are at the core of many problems arising in biological sequence analysis [24,22].

Arguably, one of the most natural notions of regularity is that of an exact repetition (power), i.e. a substring consisting of two or more consecutive identical factors. The number of these factors, or blocks, is called the *order* of the repetition. The study of powers began in the early 1900s with the work of Thue [29], who studied a class of strings that do not contain any substrings that are powers. Recently, a new notion of string diversity in terms of powers has been introduced: an *antipower* of order k (k-antipower) is a string that can be decomposed into k pairwise-distinct blocks of identical length [14,15]. An antipower is the opposite of a power; i.e., a concatenation of blocks that have the same length but are all different.[4] Likewise, the concept of *antiperiod* is symmetrical to that of period: an integer p is a *period* for a word w if and only if w is a prefix of a power with blocks of length p, whereas ℓ is an antiperiod for w if and only if it is a prefix of an antipower with blocks of length ℓ [1].

With respect to *infinite* words, in [14] the authors prove that regardless of the alphabet size, every infinite word must contain powers of any order or antipowers of any order; i.e., the existence of powers or antipowers is an unavoidable regularity (cf. also [15]). Inspired by this seminal work, Defant [11] studied the sequence of lengths of the shortest prefixes of the Thue-Morse word that are k-antipowers, and proved that this grows linearly in k. The latter result is further extended in [7] to a generalization of k-antipowers defined in [15].

For *finite* words, the first algorithmic approach concerning antipowers appeared in [4]. In this work, the authors tackle the problem of finding all the factors of a string w that are k-antipowers. Specifically, they prove that the number of such factors over an alphabet of any size is $\Omega(n^2/k)$, and provide an algorithm that finds them all in $O(n^2/k)$ time and linear space. The latter results are improved in [21], where the authors give an algorithm that counts and reports the number C of substrings of a word w of length n that are k-antipowers, in $O(nk \log k)$ and $O(nk \log k + C)$ time, respectively. Moreover, they are also able to test whether a factor $w[i, j]$ is a k-antipower (i.e, answering an *antipower query* (i, j, k)) in $O(r)$ time, by constructing a data structure of size $O(n^2/r)$ in $O(n^2/r)$ time, for any $r \in \{1, \ldots, n\}$. As far as antiperiods are concerned, Alamro et al. [1] are the first to give algorithmic results. Specifically, they compute all antiperiods of a string of length n in $O(n \log n)$ time, by employing a split-find data structure with initialization time $O(n)$ and linear space, which quickly answers monotone weighted level ancestor queries over the suffix tree. Furthermore, applying recursion to the same solution, they show how to compute just the smallest antiperiod t in $O(n \log^* t)$ time.

In this paper, we extend the problems considered in [1] to the online setting. We show how to efficiently update all the antiperiods (in particular, the minimum one) of a word upon single character extensions. To achieve this, we introduce the notion of *purely antiperiodic* array—i.e. the array containing, for each word's prefix $w[1, i]$, the smallest block length ℓ such that the prefix is an antipower of order i/ℓ—and the more relaxed notion of *antiperiodic* array, which allows the

[4] We remark that a word may be a power/antipower for different orders, even though in some cases [1] the focus is on the smallest such order.

last block to have length less than ℓ. In addition, we provide the more powerful notion of *complete antipower* array, containing for each possible block length ℓ, the greatest i such that the word's prefix $w[1,i]$ is an antipower of order i/ℓ. The complete antipower array stores, implicitly, all antiperiods of all the word's prefixes, as well as the value i/ℓ itself, that is the maximum order for which a prefix of w is an antipower of period i. We show that these arrays can be computed online in $O(n \log n)$ space, $O(n \log n)$ time, and $o(n^\epsilon)$ delay per character, for any constant $\epsilon > 0$. We also show that if we allow a small (inverse-polynomial) probability of failure, then we can compute online the antiperiodic array in $O(n)$ space, $O(n \log t)$ time and $O(n \log t)$ delay (all running times are worst-case and hold with high probability). Finally, we describe small data structures supporting fast random access to those three arrays, without the need to explicitly store all of them.

2 Preliminaries and problem definitions

Let $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ be a finite ordered alphabet of size σ with $c_1 < c_2 < \ldots < c_\sigma$. Given a word (or string) $w = w[1]w[2]\cdots w[n] \in \Sigma^*$ we denote by $|w|$ its length n. We use ϵ to denote the empty word. A *factor* (or substring) of w is written as $w[i,j] = w[i]\cdots w[j]$ with $1 \le i \le j \le n$. A factor of type $w[1,j]$ is called a *prefix* of w, while a factor of type $w[i,n]$ is called a *suffix* of w.

An integer $p \ge 1$ is a *period* of a word $w[1]w[2]\cdots w[n]$ where $w[i] \in \Sigma$ if $w[i] = w[i+p]$ for $i = i, \ldots, n-p$. The smallest period of w is called *the period* of w.

A *power of order* k is a string that is the concatenation of k identical equal-length blocks of letters. More formally, given a finite word w, w^k denotes the word obtained by concatenating k copies of w. For a power of order k and length n, the integer n/k is a period.

Definition 1 ([14]). *An antipower of order k, or simply a k-antipower, is defined as a concatenation of k consecutive pairwise distinct blocks of the same length.*

The length of the pairwise distinct blocks is called *antiperiod*. The notion of antiperiod to words that are not antipowers has been extended in [1]: w has an antiperiod ℓ if it is a prefix of some k-antipower w whose antiperiod is ℓ. Holding this intuition, we formalize a definition of *antiperiodic* words as follows:

Definition 2. *A word $w = u_1 \ldots u_k$ is called ℓ-antiperiodic if (i) $u_i \neq u_j$, for all $i \neq j$; (ii) $|u_i| = \ell$, for all $i < k$; (iii) $|u_k| \le \ell$. The number $\ell = \lfloor \frac{n}{k} \rfloor$ is an antiperiod for w.*

Note that a k-antipower of length n is $\frac{n}{k}$-antiperiodic. Therefore, we also call it *purely $\frac{n}{k}$-antiperiodic.*

Example 1. The word `abbbaa` is a 3-antipower, therefore it is also purely 2-antiperiodic: the distinct-factors partition `ab|bb|aa` testifies this.
The word `ababaab` is not 2-antiperiodic, but it is 3-antiperiodic: `aba|baa|b`.

We now formally define the three arrays that will be the output of our online algorithms. The first two arrays respectively store, for each word's prefix, its minimum antiperiod and its minimum pure antiperiod:

ANTIPERIODIC ARRAY (APD)
INPUT: A word w of length n
OUTPUT: An array of length n where APD$[i]$ is the smallest ℓ such that $w[1,i]$ is ℓ-antiperiodic.

PURELY ANTIPERIODIC ARRAY (PAPD)
INPUT: A word w of length n
OUTPUT: An array of length n where pAPD$[i]$ is the smallest ℓ such that $w[1,i]$ is a $\frac{i}{\ell}$-antipower.

We further consider the following question: given i and ℓ is the prefix $w[1,i]$ (purely) ℓ-antiperiodic? We can answer this question by building our third array, the complete array CAP of w:

COMPLETE ANTIPOWER ARRAY (CAP)
INPUT: A word w of length n
OUTPUT: An array of length n where cAP$[\ell]$ is the maximum index i such that the prefix $w[1,i]$ is purely ℓ-antiperiodic.

Example 2. Let $w =$ `abaabaab`. The antiperiodic array of w is given by APD $= [1,1,2,2,2,2,2,4]$, its purely antiperiodic array pAPD $= [1,1,3,2,5,2,7,4]$, and its complete antipower array is cAP $= [2,6,3,8,5,6,7,8]$.

In the next sections, we show how to build these arrays online, and how to access them using little space (i.e. less space than the three explicit arrays).

2.1 Basic Properties of Antiperiodic Words

First, we provide some properties of (purely) ℓ-antiperiodic words.

Lemma 1. *If ℓ is an antiperiod for some word w of length $> \ell$, then ℓ is also antiperiod for $w' = w[1, |w|-1]$.*

Proof. First, let us assume ℓ does not divide $|w|$. By hypothesis, all blocks $u_s = w[(s-1)\ell+1, s\ell]$ for $s = 1, ..., \lfloor \frac{|w|}{\ell} \rfloor = k-1$ and $u_k = w[\lfloor \frac{|w|}{\ell} \rfloor \ell + 1, |w|]$ are pairwise distinct. Let $u'_k = u_k[1, |u_k|-1]$ (possibly the empty word); then the blocks $u_1, ..., u_{k-1}$ are still distinct, and they are all also distinct from u'_k since they are of different lengths. Therefore, since $w' = w[1,|w|-1] = u_1 \cdots u_{k-1} u'_k$, ℓ is an antiperiod for w'. If ℓ divides $|w|$, the proof still holds with last block being u_{k-1} instead of u_k. □

Proposition 1. *The following properties hold:*

i) If a word w is ℓ-antiperiodic, then for all $i \geq \ell$ the prefix $w[1,i]$ is ℓ-antiperiodic.
ii) If a word w is purely ℓ-antiperiodic, then for all $i \geq \ell$ such that ℓ divides i, the prefix $w[1,i]$ is purely ℓ-antiperiodic.
iii) If a prefix $w[1,j]$ is purely ℓ-antiperiodic, then $w[1,i]$ is ℓ-antiperiodic for all $\ell \leq i < j + \ell$.

Proof. Properties *i*) and *ii*) follow by recursively applying Lemma 1. In order to prove Property *iii*), we assume that $w[1,j]$ is purely ℓ-antiperiodic. By part *i*) we have that the thesis holds for $i \in [\ell, j]$. On the other hand, if $w[1,j]$ is purely ℓ-antiperiodic, then the blocks $u_s = w[(s-1)\ell+1, s\ell]$ for $s = 1, \ldots, \frac{j}{\ell}$ are pairwise distinct. Therefore they are also trivially distinct from $u_h = w[j+1, h]$ for all $h < j + \ell$, since they are of different lengths. This ensures the ℓ-antiperiodicity of $w[1,i]$ for $j < i < j + \ell$.

2.2 Algorithmic and data structures toolkits

We make use of Karp-Rabin fingerprinting [20] in the more modern variant, where the fingerprint is a polynomial modulo a prime number evaluated at a random point. To make the paper self-contained, we recapitulate these notions.

Definition 3 (Karp-Rabin fingerprinting [20]). *The Karp-Rabin hash function (over integer alphabet) is defined as $h_{q,x}(w) = \sum_{i=1}^{|w|} w[i] \cdot x^{|w|-i} \mod q$, where q is a prime and x is a random integer in $[1,q]$.*

The value $h_{q,x}(w)$ will be called *hash value* or *fingerprint* of w interchangeably. In the following, *with high probability* (or inverse-polynomial probability) means with probability at least $1 - n^{-c}$ for an arbitrarily large constant c fixed at construction time, where n is the input's size. We will often say that *running times* of our algorithms *hold with high probability*. This means that the algorithm terminates in the claimed running time (or has the claimed delay) with probability $1 - n^{-c}$ for an arbitrarily large constant c fixed at construction time. On the other hand, with probability n^{-c} the algorithm (or single operations) could take polynomial time to terminate.

A crucial property of Karp-Rabin fingerprinting is that, with high probability, no collision occurs among the factors of a given word. To see this, consider two words $s \neq t$ of length n. The polynomial $h_{q,x}(s) - h_{q,x}(t)$ has at most n roots modulo (prime) q, so the two words collide (i.e. $h_{q,x}(s) - h_{q,x}(t) = 0$) with probability n/q. For big enough q we take care of all possible $O(n^2)$ collisions between the factors of a word of length n:

Lemma 2. *For a sufficiently large prime q such that $\log q \in \Theta(\log n)$, the hash function $h_{q,x}$ is collision-free over all factors of any fixed word $w[1,n]$ with high-probability.*

If n is not known in advance (e.g. in the online setting), in the above lemma we can take instead $\log q \in \Theta(\omega)$, where ω is the machine word size. We thus work

in the word RAM model, where the word size always satisfies $\omega = \Omega(\log n)$ for any input size n and standard arithmetic operations between $\Theta(\omega)$-bits numbers take constant time.

In our application we will need to check collisions between factors of the current word in an online fashion, with small delay per added character. The following standard extension of Karp-Rabin fingerprinting (see also [6]) will allow us to reach this goal.

Definition 4 (Extended Karp-Rabin fingerprinting). *The extended Karp-Rabin hash function is defined as*

$$\kappa_{q,x}(w) = \left\langle h_{q,x}(w[1, 2^{\lfloor \log_2 |w| \rfloor}]), h_{q,x}(w[|w| - 2^{\lfloor \log_2 |w| \rfloor} + 1, |w|]), |w| \right\rangle$$

We say that $\kappa_{q,x}$ is *collision-free* on w if $\kappa_{q,x}$ does not generate collisions among factors of w. Note that $\kappa_{q,x}$ is collision-free on w if and only if $h_{q,x}$ is collision-free among the factors of w whose length is a power of two.

A *dictionary* D is a data structure implementing a set of objects (e.g. integers). Each object x is associated with some satellite information y (e.g. another integer), which is retrieved using the notation $y \leftarrow D[x]$. A dynamic dictionary supports the insertion of pairs (object, satellite information) $\langle x, y \rangle$ as $D[x] \leftarrow y$. In our algorithms we will use the dictionary design of Dietzfelbinger et al. [12]:

Lemma 3 (Dynamic Dictionaries [12]). *There exists a linear-space dynamic dictionary data structure supporting insertion and retrieval operations in constant worst-case time w.h.p.*

The probabilities involved in the performance of our algorithms will depend solely on the random choices they make (e.g. choosing the hash function), not on the input word w (which may be arbitrary). However, we do require that the word w is fixed before the algorithm starts, i.e. before the random choices are made. This is a standard assumption with randomized dynamic data structures; violating this assumption is equivalent to using fully-deterministic structures, for which strong lower-bounds apply (see, e.g., the case of dictionaries considered by Dietzfelbinger et al. [13], who also make the same assumption).

3 Online algorithms

In this section, we first discuss how to compute the antiperiodic array APD (by using two different approaches), and then extend the solution to *p*APD and to *c*AP arrays. The final goal of this section will be proving the following theorems:

Theorem 1. *The antiperiodic array (*APD*) of a word w of length n can be computed with an online solution working in $O(n)$ space, $O(n \log t)$ time, and $O(n \log t)$ delay per character, where t is the smallest antiperiod of w. Running times are worst-case and hold w.h.p. The returned solution is correct w.h.p.*

Theorem 2. *The antiperiodic array (*APD*), purely antiperiodic array (p*APD*) and complete antipower array (c*AP*) of a word w of length n can be computed with an online solution working in $O(n \log n)$ space, $O(n \log n)$ time, and $o(n^\epsilon)$ delay per character for any constant $\epsilon > 0$. Running times are worst-case and hold w.h.p.*

The proof of Theorem 1 is given in Section 3.1 and the proof of Theorem 2 follows immediately from Lemmas 6, 7, and 8.

Whenever we say we use a dictionary or an array, it is assumed we use the data structure of Lemma 3. This way, we do not need to assume that the final size n of the text is known. Note that this dictionary offers stronger guarantees than classic hash tables, where query times are constant only in expectation. Similarly, a simple array can be used to represent a dynamic text with right-append operations; however, also in this case resizing operations (e.g. doubling techniques) require $O(n)$ delay in the worst case. In order to achieve a small delay in our online algorithms, we cannot afford paying such overheads and thus opt for the structure of Lemma 3.

The following dynamic string data structure stands at the core of our results. It extends standard techniques (see e.g. [6]) to the online setting to efficiently compute the extended Karp-Rabin fingerprint of any factor and check collisions among them.

Lemma 4. *There is a $O(n \log n)$-space data structure on a word $w[1, n]$ that computes $\kappa_{q,x}(w[i, j])$ in constant time for any $1 \leq i \leq j \leq n$, where $\kappa_{q,x}$ is collision-free. The structure can be updated in $O(\log n)$ time by appending a new character to the end of w, possibly changing function $\kappa_{q,x}$ if a collision is detected. Space and update time can be reduced to $O(n)$ and $O(1)$, respectively, at the cost of using a hash function that is collision-free with high probability only. In all cases, running times are worst-case and hold w.h.p.*

Proof. Similarly to Bille et al. [6] (where they consider more space-efficient variants), we store the hash value of every prefix of w. Then, the fingerprint value of any factor can be computed by subtracting (modulo q) the hashes of two prefixes (the hash of the shortest prefix need also to be multiplied by a suitable power of x, so we compute also all powers of x modulo q). To extend the structure for the word w by one character c, it is sufficient to compute the fingerprint of wc. This can be done with one multiplication and one addition (mod q) to combine the fingerprints of w and c. This is already sufficient to obtain the linear-space version of the structure that is correct with high probability and that supports queries and updates in constant time.

To make the structure collision-free, we check collisions of $h_{q,x}$ among factors whose length is a power of 2. This suffices to ensure that $\kappa_{q,x}$ is collision-free among factors of any length. For each $e = 1, \ldots, \log n$ we keep a dictionary C_e storing all fingerprints of factors of length 2^e. To each such fingerprint X, we associate a position i such that $h_{q,x}(w[i, i + 2^e - 1]) = X$. Assume that the function is collision-free among factors of w whose length is a power of 2. To extend the property to $w' = wc$, for a character c, first we extend the

fingerprinting structure as seen above. Then, for $e = 1, \ldots, \log n$ (in this order) we do as follows. Assume that the suffix of length 2^{e-1} of w' does not generate collisions, which at the beginning ($e = 1$) can be checked in constant time by accessing dictionary C_1 and the word. We look for $X = h_{q,x}(w'[|w'|-2^e+1, |w'|])$ in C_e. If it does not occur, then the suffix of w' of length 2^e does not generate collisions. We insert $\langle X, |w'| - 2^e + 1\rangle$ in the dictionary and we proceed with $e+1$. Otherwise, let $i < |w'|-2^e+1$ be the position found in the dictionary such that $h_{q,x}(w'[i, i+2^e-1]) = X$. Since by assumption $h_{q,x}$ is collision-free among factors of length 2^{e-1}, we can check if $w[i, i+2^e-1] = w'[|w'| - 2^e + 1, |w'|]$ in constant time by comparing the collision-free fingerprints of their two halves. If we detect a collision, then we re-build the whole structure with a new random x. This happens with probability at most n^{-c} for an arbitrarily large constant c, so updates take $O(\log n)$ time w.h.p. □

The *optimal dynamic strings* of Gawrychowski et al. [17] could also be used to replace the structure of Lemma 4. However, they support factor comparison in $O(\log n)$ time[5], whereas Lemma 4 achieves $O(1)$ time (at the price of being less space-efficient and supporting less powerful queries). This is crucial to obtain the claimed time bounds in our algorithms.

3.1 Online antiperiodic array: linear space construction

Our first solution to build the antiperiodic array relies on its monotonicity.

Proposition 2. *The antiperiodic array* APD *is non-decreasing; that is, for all* $i < |w|$, $\text{APD}[i] \leq \text{APD}[i+1]$.

Proof. We know that $\text{APD}[i+1]$ is an antiperiod for $w[1, i+1]$. By Lemma 1, $\text{APD}[i+1]$ is also an antiperiod for $w[1, i]$. Since $\text{APD}[i]$ is the minimum antiperiod for this word, $\text{APD}[i] \leq \text{APD}[i+1]$ holds. □

We now describe a simple online solution for computing array APD that achieves linear space and $O(n \log t)$ running time, where t is the smallest antiperiod of the word, proving Theorem 1.

Proof (Theorem 1). Assume we have processed $w[1, i]$ and computed $\text{APD}[1, i]$, with $\text{APD}[i] = \ell$. We keep the dynamic linear-space data structure of Lemma 4 on w (i.e. the version that is correct w.h.p. only). We also keep a dictionary R containing the fingerprints of all blocks of length ℓ up to position i. If ℓ does not divide $i+1$, then we write $\text{APD}[i+1] = \ell$ and proceed to $i+2$. Otherwise, $w[i, i+1]$ ends a block of length ℓ. We insert the fingerprint of the last block of length ℓ in the dictionary R. If the fingerprint is not already in the dictionary, we can write $\text{APD}[i+1] = \ell$ and proceed to $i+2$. Otherwise, ℓ is not an antiperiod of $i+1$. We empty the dictionary R and look for a new antiperiod $\ell' = \ell+1, \ell+2, \ldots$.

[5] Using their interface, factor comparisons can be achieved by extracting (splitting) the factors in logarithmic time, then comparing them in constant time (or, alternatively, by navigating their grammar in logarithmic time without performing splits).

Proposition 2 guarantees that only these antiperiods need to be tried, since APD is non-decreasing. To check antiperiod ℓ', we insert the fingerprints of all blocks of length ℓ' in R in $O((i+1)/\ell')$ time. If a duplicate is found, we empty R and proceed to $\ell'+1$. We stop at the smallest integer ℓ' that does not generate duplicates in R (and we keep the current elements in R).

Let t be the smallest antiperiod of the whole word $w[1,n]$. Overall, we spend time $O(\sum_{\ell=1}^{t} n/\ell) = O(n \log t)$. Since we keep only one dictionary at a time, the space is $O(n)$. In the worst case, it could be that we are processing a position $i \in \Theta(n)$ and that we need to check most of the antiperiods ℓ' smaller than t, so also the delay is $\Theta(n \log t)$ in the worst case. Moreover, since we do not check collisions between Karp-Rabin fingerprints, the solution is correct with high probability only. □

3.2 Online antiperiodic array: reducing the delay

In this section we improve the delay of the solution described in the previous section with an algorithm returning always the correct solution (at the cost of increasing space usage and running time).

Our solution will require to compute the divisors of all numbers $i = 1, \ldots, n$. With the next lemma we show how to achieve this goal in an online fashion and constant time per element. Running time is w.h.p. because we assume that n is not known in advance and we use the dictionary of Lemma 3 to implement a dynamic array; otherwise, if n is known one can use a simple array and remove randomization.

Lemma 5. *We can list all $d(i)$ divisors of the integers $i = 1, \ldots, n$ in $O(d(i))$ time per integer and $O(n \log n)$ total space. The running time is worst-case and holds w.h.p.*

Proof. We show how to build a dictionary D such that $D[i]$ is the multi-set of the $O(\log i)$ prime divisors of i: if d is the largest integer such that p^d divides i and p is prime, $D[i]$ contains d copies of p. $D[i]$ can be implemented as a simple array (to re-size it, we can simply double its current size: since at most $O(\log i)$ primes can divide i, $D[i]$ can be re-sized with delay $O(\log i)$, which is acceptable for our purposes). At the beginning, we have $D[1] = \emptyset$. Given $D[i]$, all divisors of i can be enumerated in constant time per element by multiplying the integers of any subset of $D[i]$ (with backtracking, to avoid repeating the same multiplications). We also return the trivial divisors 1 and i (or just one of them if $i = 1$).

Suppose we have computed $D[1,i]$. To proceed to the next position $i+1$, for each $p \in D[i]$ we insert p in $D[j]$, where $j = ((i/p)+1) \cdot p$ is the next multiple of p. If $D[i+1]$ is empty, then we insert $i+1$ in $D[i+1]$. It is easy to see that now $D[i+1]$ contains all primes that divide $i+1$: let p be a prime dividing $i+1$. We have two cases. (i) $i+1 = q \cdot p$, with $q > 1$. Then, by definition of our procedure when we processed $D[(q-1)\cdot p]$ we also inserted p in $D[(q-1+1)\cdot p] = D[i+1]$. (ii) $i+1 = p$. Then, $D[i+1]$ is empty when we visit it and our procedure inserts p in $D[i+1]$. Now, if a prime p divides $i+1$ we can find the largest power p^d dividing

$i+1$ by simply computing the remainder between $i+1$ and p^e, for $e = 1, \ldots, d$. We insert $d-1$ further copies of p in $D[i+1]$. Note that we compute each $D[i]$ in $O(\log i) \subseteq O(d(i))$ time and delay, and that at any time D contains at most $\Theta(n)$ multi-sets of size $O(\log n)$ each (when processing position i, the furthest cell of D that we touch is $D[2 \cdot i]$; this happens precisely when i is prime). □

We can now prove the main result of this section regarding the APD:

Lemma 6. *Theorem 2 holds with respect to the antiperiodic array.*

Proof. Assume we have processed $w[1, i]$ and computed $\text{APD}[1, i]$. We keep the $O(n \log n)$-space data structure of Lemma 4 on w (i.e. the version that always returns the correct result and supports $O(\log n)$-time updates).

We build i dictionaries $H_1, \ldots, H_i$, where H_ℓ stores all the fingerprints $\kappa_{q,x}(w[\ell \cdot (j-1)+1, \ell \cdot j])$ for $j = 1, \ldots, \lfloor i/\ell \rfloor$. At each time step, note that the dictionaries store overall $O(n \log n)$ elements.

We also keep a log-time successor data structure $\mathcal{B}$ (e.g. a red-black tree) storing all the values $\{\ell_1, \ldots, \ell_q\} \subseteq [1, i]$ such that ℓ_j is an antiperiod of $w[1, i]$.

Now, suppose we extend $w[1, i]$ with a character, obtaining $w[1, i+1]$, and assume that the last computed value in the antiperiodic array is $\text{APD}[i] = \ell$. To update our structures, we perform the following operations:

1. First, we insert $i+1$ in $\mathcal{B}$, since $i+1$ is a valid antiperiod of $w[1, i+1]$.
2. Then, we update the dictionary H_j for all j that divide $i+1$, i.e. such that a block of length j ends at the end of $w[1, i+1]$. Each such dictionary H_j is updated by inserting $X = \kappa_{q,x}(w[i-j+2, i+1])$, i.e. the hash of the last block of length j. If X is already in the dictionary, then j cannot be anymore an antiperiod, and we remove it from $\mathcal{B}$. In Lemma 5 we show how to list all $d(i+1)$ divisors of $i+1$ in constant time per item. Any integer x has at most $e^{O(\log x / \log \log x)} = o(x^\epsilon)$ divisors, for any constant $\epsilon > 0$, so $o(n^\epsilon)$ will become our delay. Overall, these operations amortize to $O(n \log n)$ time (since $\sum_{i=1}^{n} d(i) = O(n \log n)$, where $d(i)$ is the number of divisors of i).
3. If $\text{APD}[i] = \ell$ does not divide $i+1$, then we set $\text{APD}[i+1] = \ell$. Otherwise, ℓ divides $i+1$. Then, ℓ is the minimum antiperiod of $w[1, i+1]$ iff $\ell \in \mathcal{B}$. If it is, we set $\text{APD}[i+1] = \ell$. Otherwise, we find the successor ℓ' of ℓ in $\mathcal{B}$ and we set $\text{APD}[i+1] = \ell'$. All these operations take $O(\log n)$ time. □

3.3 Purely antiperiodic array

In this section we extend the solution for computing APD of Section 3.2 to the computation of the *purely antiperiodic* array *p*APD.

Lemma 7. *Theorem 2 holds with respect to the purely antiperiodic array.*

Proof. We build the dictionaries $H_1, \ldots, H_i$ as in the proof of Theorem 6. We mark each dictionary H_ℓ with a flag (one bit) recording whether ℓ is no longer an antiperiod for the current position i. When processing position $i+1$, we scan all divisors of $i+1$ (using Lemma 5) and (i) opportunely update the flags

whenever for some divisor ℓ' of $i+1$, the current prefix $w[1, i+1]$ is no longer a $\frac{i+1}{\ell'}$-antipower (but the prefix $w[1, (i+1)-\ell']$ was), and (ii) find the minimum divisor ℓ for which $w[1, i+1]$ is a $\frac{i+1}{\ell}$-antipower (i.e. the minimum ℓ such that H_ℓ is not marked). This ℓ is the value we write in $p\text{APD}[i+1]$. Our time bounds follow since we spend constant time per divisor of $i+1$. □

3.4 Complete antipower array

In this section we show how to build the complete antipower array $c\text{AP}$. Recall that $c\text{AP}[\ell]$ stores the maximum index j such that the prefix $w[1, j]$ is purely ℓ-antiperiodic. Thus, for any $\ell > \lfloor n/2 \rfloor$, $c\text{AP}[\ell]$ is trivially equal to its index ℓ.

With an easy adaptation of Theorem 7 we obtain:

Lemma 8. *Theorem 2 holds with respect to the complete antipower array.*

Proof. We proceed as in the proof of Theorem 7. We keep i dictionaries H_ℓ, $\ell = 1, \ldots, i$ storing the fingerprints of blocks of length ℓ up to position i. We also mark each dictionary H_ℓ with a flag (one bit) recording (when true) whether ℓ is no longer an antiperiod for the current position i. Assume we have computed $c\text{AP}[1, i]$. When processing position $i+1$, we generate all divisors of $i+1$ with Lemma 5. Whenever for some divisor ℓ' of $i+1$, the current prefix $w[1, i+1]$ is no longer a $\frac{i+1}{\ell'}$-antipower (but the prefix $w[1, (i+1)-\ell']$ was), we set the flag associated with dictionary $H_{\ell'}$. Then, for all divisors ℓ'' of $i+1$ such that the flag of $H_{\ell''}$ is false we write $c\text{AP}[\ell''] \leftarrow i+1$. Our time and space bounds follow since the number of divisors is always upper-bounded by $o(n^\epsilon)$ and the sum of the number of divisors of all $i = 1, \ldots, n$ is $O(n \log n)$. □

4 Relationships between cAP, APD and pAPD

We observe that the array cAP stores more information than arrays APD and pAPD, which only record the *minimum* ℓ for each word's prefix. Moreover, the following lemma shows the relation between the array cAP and the property of being ℓ-antiperiodic.

Lemma 9. *A prefix $w[1, i]$ is ℓ-antiperiodic if and only if $\ell \leq i < c\text{AP}[\ell] + \ell$.*

Proof. If a prefix $w[1, i]$ is ℓ-antiperiodic, it must hold that the length i of the prefix is at least ℓ. Moreover, by definition of cAP, since $c\text{AP}[\ell] = x$ is the rightmost index of w such that $w[1, x]$ is purely ℓ-antiperiodic, it also must hold $i < x + \ell$. Conversely, suppose we have an index i such that $\ell \leq i < x + \ell$, where $x = c\text{AP}[\ell]$. Then the fact that $w[1, i]$ is ℓ-antiperiodic follows by the definition of cAP and part *iii*) of Proposition 1. □

Here we show how cAP can be used to directly obtain or access efficiently the other two structures.

Accessing APD from cAP. We show that the array cAP allows fast random access to APD. Indeed, the entry $\text{APD}[i]$ is the minimum length ℓ such that the

prefix $w[1,i]$ is ℓ-antiperiodic. By Lemma 9, the value $\textsc{apd}[i]$ is the smallest ℓ such that $\ell \leq i < c\textsc{ap}[\ell] + \ell$. To find such value we need to find the leftmost entry in $c\textsc{ap}[1,i]$ such that $c\textsc{ap}[\ell] + \ell$ is greater than i.

Given $c\textsc{ap}$, we build a constant-time range maximum data structure (RMQ) on the array $\mathrm{A} = [c\textsc{ap}[1] + 1, c\textsc{ap}[2] + 2, \ldots, c\textsc{ap}[n] + n]$, which requires only $2n+o(n)$ bits and can be built in linear time [16]. The structure returns, for every range $[i,j]$, the index containing the maximum element in A. Then, accessing $\textsc{apd}[i]$ corresponds to finding the leftmost entry in A that exceeds i. This can be solved in $O(\log n)$ time with binary search using the RMQ on A.

Obtaining APD from cAP. We show that cAP may be used to build APD in $O(n)$ time. Since $\textsc{apd}[i] = \min\{\ell : w[1,i] \text{ is } \ell\text{-antiperiodic}\}$, by Lemma 9 it holds $\textsc{apd}[i] = \min\{\ell : \ell \leq i < c\textsc{ap}[\ell] + \ell\}$. We can thus obtain APD by iterating over $c\textsc{ap}[\ell]$ and each time setting values of APD that have not already been set. More formally, we start from $\ell = 1$ and set an auxiliary index $j = 1$. The index j prevents us from writing twice on the same cell. Given ℓ, we mark $\textsc{apd}[i] = \ell$ for all indices i in the range $[j, c\textsc{ap}[\ell] + \ell - 1]$. Then, we update the index j by setting $j = c\textsc{ap}[\ell] + \ell$ if $j < c\textsc{ap}[\ell] + \ell$, and we repeat the procedure for $\ell + 1$. Note that if $c\textsc{ap}[\ell] + \ell - 1$ is smaller than j for some ℓ, we do nothing apart from increasing ℓ. We stop when APD and cAP are of the same length.

When we set $\textsc{apd}[i] = \ell$, ℓ is the smallest value such that $i < c\textsc{ap}[\ell] + \ell$. The cost is $O(n)$ as each cell of APD is only written once, and each cell of $c\textsc{ap}$ only read once, both arrays having length n.

Accessing pAPD from cAP. We further note that cAP allows random access to pAPD by performing just $o(n^\epsilon)$ accesses to $c\textsc{ap}$ (for any $\epsilon > 0$). Indeed, $p\textsc{apd}[i] = \min\{\ell : w[1,i] \text{ is purely } \ell\text{-antiperiodic}\}$. Note that $i \mod \ell$ must be 0. This means we can find $p\textsc{apd}[i]$ by iterating over each divisor x of i and finding the lowest one such that $c\textsc{ap}[x] \geq i$. As any integer number $i \leq n$ has $o(n^\epsilon)$ divisors, the process takes $o(n^\epsilon)$ accesses to $c\textsc{ap}$.

To obtain $o(n^\epsilon)$ running time, one needs to list the divisors of the index $i \leq n$ in constant time per element. One solution could be dividing i by all numbers $j \leq \sqrt{i}$. This solution successfully finds all divisors in $O(\sqrt{n})$ time. This can be improved to $o(n^\epsilon)$ by explicitly factoring i: integer factorization algorithms like Schnorr-Seysen-Lenstra's [23] find all factors of $i \leq n$ in $o(n^\epsilon)$ expected time.

Obtaining pAPD from cAP. Using the above solution to access each cell of pAPD, running time amortizes to $O(n \log n)$, i.e. the sum of the number of divisors of all numbers $i \leq n$. In this case, the divisors can be found using Lemma 5.

5 Final remarks

In this paper, we showed how to efficiently compute in online fashion the antiperiodic array, the purely antiperiodic array, and the complete antipower array.

Moreover, using the complete antipower array, we can answer in constant time the question posed in Section 2: given i and ℓ is the prefix $w[1,i]$ (purely) ℓ-antiperiodic? Indeed, $w[1,i]$ is purely ℓ-antiperiodic if $i \mod \ell = 0$ and $i \leq c\textsc{ap}[\ell]$, and ℓ-antiperiodic if $\ell \leq i < c\textsc{ap}[\ell] + \ell$.

Note that the definition of ℓ-antiperiodic word $w = u_1 \dots u_k$ poses no constraint on the last block u_k when its length is strictly less than ℓ. We may think of extending/restricting the ℓ-antiperiodic notion by imposing conditions on this incomplete block, such as u_k not being a prefix (or suffix, factor...) of any u_i.

Acknowledgements

The authors would like to thank Roberto Grossi for useful conversations on the topic. GR, NPi are partially, and DG, VG, NPr are supported by the project MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant n. RBSI146R5L.

References

1. Alamro, H., Badkobeh, G., Belazzougui, D., Iliopoulos, C.S., Puglisi, S.J.: Computing the Antiperiod(s) of a String. In: 30th Annual Symposium on Combinatorial Pattern Matching (CPM). LIPIcs, vol. To appear (2019)
2. Alzamel, M., Crochemore, M., Iliopoulos, C.S., Kociumaka, T., Radoszewski, J., Rytter, W., Straszyński, J., Waleń, T., Zuba, W.: Quasi-Linear-Time Algorithm for Longest Common Circular Factor. In: 30th Annual Symposium on Combinatorial Pattern Matching (CPM). LIPIcs, vol. To appear (2019)
3. Ayad, L.A.K., Bernardini, G., Grossi, R., Iliopoulos, C.S., Pisanti, N., Pissis, S.P., Rosone, G.: Longest property-preserved common factor. In: 25th International Symposium on String Processing and Information Retrieval (SPIRE). pp. 42–49. Springer LNCS 11147 (2018)
4. Badkobeh, G., Fici, G., Puglisi, S.J.: Algorithms for anti-powers in strings. Information Processing Letters **137**, 57 – 60 (2018)
5. Bae, S.W., Lee, I.: On finding a longest common palindromic subsequence. Theoretical Computer Science **710**, 29–34 (2018)
6. Bille, P., Gørtz, I.L., Knudsen, M.B.T., Lewenstein, M., Vildhøj, H.W.: Longest common extensions in sublinear space. In: Combinatorial Pattern Matching. pp. 65–76. Springer International Publishing, Cham (2015)
7. Burcroff, A.: (k,λ)-anti-powers and other patterns in words. Electronic Journal of Combinatorics **25**, P4.41 (2018)
8. Chowdhury, S., Hasanl, M., S.Iqbal, Rahman, M.: Computing a longest common palindromic subsequence. Fundam. Inf. **129**(4), 329–340 (2014)
9. Crochemore, M., Ilie, L., Rytter, W.: Repetitions in strings: Algorithms and combinatorics. Theoretical Computer Science **410**(50), 5227 – 5235 (2009), mathematical Foundations of Computer Science (MFCS 2007)
10. Crochemore, M., Rytter, W.: Jewels of stringology. World Scientific (2002)
11. Defant, C.: Anti-power prefixes of the Thue-Morse word. Electronic Journal of Combinatorics **24**, P1.32 (2017)
12. Dietzfelbinger, M., Meyer auf der Heide, F.: A new universal class of hash functions and dynamic hashing in real time. In: International Conference on Automata, Languages and Programming. pp. 6–19. Springer Berlin Heidelberg, Berlin, Heidelberg (1990)

13. Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Meyer auF der Heide, F., Rohnert, H., Tarjan, R.E.: Dynamic perfect hashing: Upper and lower bounds. SIAM Journal on Computing **23**(4), 738–761 (1994)
14. Fici, G., Restivo, A., Silva, M., Zamboni, L.Q.: Anti-Powers in Infinite Words. In: ICALP 2016. Leibniz International Proceedings in Informatics (LIPIcs), vol. 55, pp. 124:1–124:9. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2016)
15. Fici, G., Restivo, A., Silva, M., Zamboni, L.Q.: Anti-powers in infinite words. Journal of Combinatorial Theory, Series A **157**, 109 – 119 (2018)
16. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM Journal on Computing **40**(2), 465–492 (2011)
17. Gawrychowski, P., Karczmarz, A., Kociumaka, T., Łacki, J., Sankowski, P.: Optimal dynamic strings. In: Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 1509–1528. SODA '18, Society for Industrial and Applied Mathematics (2018)
18. Inenaga, S., Hyyrö, H.: A hardness result and new algorithm for the longest common palindromic subsequence problem. Information Processing Letters **129**, 11–15 (2018)
19. Inoue, T., Inenaga, S., Hyyrö, H., Bannai, H., Takeda, M.: Computing longest common square subsequences. In: 29th Symposium on Combinatorial Pattern Matching (CPM). LIPIcs, vol. 105, pp. 15:1–15:13 (2018)
20. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development **31**(2), 249–260 (1987)
21. Kociumaka, T., Radoszewski, J., Rytter, W., Straszyński, J., Waleń, T., Zuba, W.: Efficient representation and counting of antipower factors in words. In: Language and Automata Theory and Applications. pp. 421–433. Springer International Publishing, Cham (2019)
22. Kolpakov, R., Bana, G., Kucherov, G.: mreps: efficient and flexible detection of tandem repeats in DNA. Nucleic Acids Research **31**(13), 3672–3678 (2003). https://doi.org/10.1093/nar/gkg617
23. Lenstra, H.W., Pomerance, C.: A rigorous time bound for factoring integers. Journal of the American Mathematical Society **5**(3), 483–516 (1992)
24. Li, L., Jin, R., Kok, P.L., Wan, H.: Pseudo-periodic partitions of biological sequences. Bioinformatics **20**(3), 295–306 (2004)
25. Lothaire, M.: Combinatorics on Words. Cambridge University Press (1997)
26. Lothaire, M.: Algebraic Combinatorics on Words. Cambridge University Press (2002)
27. Lothaire, M.: Applied Combinatorics on Words. Encyclopedia of Mathematics and its Applications, Cambridge University Press (2005). https://doi.org/10.1017/CBO9781107341005
28. Lothaire, M.: Review of applied combinatorics on words. SIGACT News **39**(3), 28–30 (2008)
29. Thue, A.: Uber unendliche zeichenreihen. Norske Vid Selsk. Skr. I Mat-Nat Kl. (Christiana) **7**, 1–22 (1906)