Computing the Antiperiod(s) of a String

Hayam Alamro
Department of Informatics, King's College London, UK
Department of Information Systems, Princess Nourah bint Abulrahman University, Riyadh, KSA
hayam.alamro@kcl.ac.uk

Golnaz Badkobeh
Department of Computing, Goldsmiths, University of London, UK
g.badkobeh@gold.ac.uk

Djamal Belazzougui
Centre de Recherche sur I'nformation Scientifique et Technique, Algeria
dbelazzougui@cerist.dz

Costas S. Iliopoulos
Department of Informatics, King's College London, UK
costas.iliopoulos@kcl.ac.uk

Simon J. Puglisi
Department of Computer Science, University of Helsinki, Finland
puglisi@cs.helsinki.fi

Abstract

A string $S[1,n]$ is a *power* (or repetition or tandem repeat) of order k and period n/k, if it can be decomposed into k consecutive identical blocks of length n/k. Powers and periods are fundamental structures in the study of strings and algorithms to compute them efficiently have been widely studied. Recently, Fici et al. (Proc. ICALP 2016) introduced an *antipower* of order k to be a string composed of k distinct blocks of the same length, n/k, called the antiperiod. An arbitrary string will have antiperiod t if it is prefix of an antipower with antiperiod t. In this paper, we describe efficient algorithm for computing the smallest antiperiod of a string S of length n in $O(n)$ time. We also describe an algorithm to compute all the antiperiods of S that runs in $O(n \log n)$ time.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms

Keywords and phrases antiperiod, antipower, power, period, repetition, run, string

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.32

Acknowledgements This work was supported by the Academy of Finland under grant 319454. We thank the anonymous reviewers for their detailed comments that greatly improved the quality of this article, in particular for the improvement in Section 5.2 and for pointers on previous works on weighted-level ancestors data structures.

1 Introduction

Algorithms and data structures for finding repeating patterns or *regularities* in strings (see, e.g., [6, 16, 24]) are central to several fields of computer science including computational biology, pattern matching, data compression, and randomness testing. The nature and extent of regularity in strings is also of immense combinatorial interest in its own right [22].

One of the most fundamental notions of regularity is that of powers (also known as repetitions or tandem repeats). A power of order k is defined by a concatenation of k identical blocks of symbols, where k is at least 2. The study of powers began in the early 1900s with the work of Thue [25], who studied a class of strings that do not contain any substrings that are powers. Powers in various forms later came to be important structures in computational biology, where they are associated with various regulatory mechanisms and play an important role in genomic fingerprinting (for further reading see, e.g., [19] and references therein). More recently it has been shown that the number of so-called *maximal* powers (or runs) in a string is less than the length of the string itself [2] – positively settling a long-standing conjecture [20, 5].

Antipowers are an orthogonal notion to that of powers, that were introduced recently by Fici et al. in [9, 10]. In contrast to powers, antipowers insist instead on the diversity of consecutive blocks: an antipower of order k is a concatenation of k pairwise distinct strings of equal length.

In the short period of time following Fici et al.'s work, antipowers have received a great deal of attention, especially from a combinatorial perspective. In a follow-up article, Fici et al. [10], further showed that every infinite string contains powers of any order or antipowers of any order. Moreover, Defant in [7] (see also Narayanan [23]) studied the sequence of lengths of the shortest prefixes of the Thue-Morse string that are k-antipowers, and proved that this sequence grows linearly in k. Gaetz [12] has since extended Defant's results to substrings. Burcroff [4] studied the avoidability of k antipowers in infintite strings, generalizing Fici et al.'s results.

The algorithmic study of antipowers was initiated recently by Badkobeh et al. [1], who describe an algorithm that, given a string S of length n and a parameter k, locates all substrings of S that are antipowers of order k. The algorithm takes $\Theta(n^2/k)$ time, which the authors show to be optimal in the sense that there exist strings containing $\Theta(n^2/k)$ distinct antipowers of order k. Very recently Kociumaka et al. [18] have shown an output sensitive algorithm for the same problem with running time $O(nk \log k + C)$, where C is the number of reported antipowers. They also show that they can be counted within time $O(nk \log k)$.

New Results. In this paper, we describe algorithms for computing the smallest antiperiod and all the antiperiods of a string S of length n. Our definition of antiperiod is slightly more general than that considered by Badkobeh et al., and we define it formally in the next section. The starting point for both our algorithms is an efficient solution to the monotone weighted level ancestor problem, which we describe in Section 3. This leads to an $O(n \log n)$ time algorithm for computing all the antiperiods of a given string, described in Section 4. We then show how to exploit combinatorial properties of antipowers to obtain an algorithm for computing the smallest antiperiod, of S in time $O(n)$. We notice that a very similar result to our monotone weighted level ancestor solution was already shown by Kociumaka et al. [17] (see also Barton et al. [3]). Such a result was unknown to us prior to the initial submission of this paper.

2 Preliminaries

Let $S = S[1, n]$ be a string of length $|S| = n$ over an alphabet Σ of size $|\Sigma| = \sigma$. The empty string ε is the string of length 0. For $1 \leq i \leq j \leq n$, $S[i]$ denotes the ith symbol of S, and $S[i, j]$ the contiguous sequence of symbols (called *factor* or *substring*) $S[i]S[i+1] \dots S[j]$. A substring $S[i, j]$ is a suffix of S if $j = n$ and it is a prefix of S if $i = 1$. A string p is a *repeat* of S iff p has at least two occurrences in S. In addition p is said to be *right-maximal* in S iff there exist two positions $i < j$ such that $S[i, i + |p| - 1] = S[j, j + |p| - 1] = p$ and either $j + |p| = n + 1$ or $S[i, i + |p|] \neq S[j, j + |p|]$.

The *suffix tree* T for a string S of length n over the alphabet Σ is a rooted directed compacted trie built on the set of suffixes of S. The suffix tree has n leaves and its internal nodes have at least two children whiles its edges are labelled with substrings of S. The labels of all outgoing edges from a given node start with a different character. All leaves of the suffix tree are labelled with an integer i, where $i \in \{1 \cdot \cdot n\}$ and the concatenation of the labels on the edges from the root to the leaf gives us the suffix of S which starts at position i. The nodes of the (non-compacted) trie which have branching nodes and leaves of the tree are called *explicit nodes*, while the others are called *implicit nodes*. The occurrence of a substring P in S is represented on T by either an explicit node or implicit node and called the *locus* of P. The *suffix tree* T can be constructed in $O(n)$ time and space. In order to have one-to-one correspondence between the suffixes of S and the leaves of T, a character $\$ \notin \Sigma$ is added to the end of the label edge for each suffix i to ensure that no suffix is a prefix of another suffix. To each node α in T is also associated an interval of leaves $[i..j]$, where $[i..j]$ is the set of labels of the leaves that have α as an ancestor (or the interval $[i..i]$ if α is a leaf labelled by i). The intervals associated with the children of α (if α is an internal node) form a partition of the interval associated with α (the intervals are disjoints sub-intervals of $[i..j]$ and their union equals $[i..j]$). For any internal node α in the *suffix tree* T, the concatenation of all edge labels in the path from the root to the node α is denoted by $\bar{\alpha}$ and the string depth of a node α is denoted by $|\bar{\alpha}|$. For brevity, in what follows, we will always talk about depth instead of string depth [1].

A *power of order* k (or k*-power*) is a string that is the concatenation of k identical strings. The *period* of a k-power of length n is n/k. For example, *abaaba* is a 2-power (or a *square*) of period 3.

Our main subject in this paper is a complementary structure to a power, called an antipower, which we now formally define.

▶ **Definition 1** (Fici et al. [9]). *An* antipower *of order* k *(or* k-antipower*) is a string obtained by the concatenation of* k *pairwise-distinct strings of identical length. The* antiperiod *of a* k*-antipower of length* n *is* n/k*.*

For example, *ababbabab* is a 3-antipower of antiperiod 3.

We now extend the notion of antiperiod to strings that are not necessarily antipowers.

▶ **Definition 2.** *A string* s *has an antiperiod* t *if it is a prefix of some* k*-antipower* $w = p_1p_2 \cdots p_k$ *whose antiperiod is* t*.*

Note that by this definition every string has antiperiods: $\lfloor n/2 \rfloor + 1, \ldots, n-1$.

▶ **Example 3.** Suppose we have the following string:

$s = aabbbbaaaabb|s| = 12$

$t = 6$: s is a 2-antipower with antiperiod 6. $s = (aabbbb)(aaaabb)$;
$t = 5$: s is a prefix of a word, w, that is a 3-antipower with antiperiod 5:
$w = (aabbb)(baaaa)(bbaaa)$;
$t = 4$: s is not a 3-antipower because $s = (aabb)(bbaa)(aabb)$ and it therefore cannot be a prefix of any antipower with antiperiod 4;
$t = 3$: s is a 4-antipower with antiperiod 3. $s = (aab)(bbb)(aaa)(abb)$;
$t = 2$: s is not a 6-antipower because $s = (aa)(bb)(bb)(aa)(aa)(bb)$ and so cannot be a prefix of any antipower with antiperiod 2.
Therefore, the smallest antiperiod of the string s is 3.

[1] Notice that string depth of a suffix tree node is different from the usual definition of node depth in arbitrary trees which refers to the number of nodes in the path from the root to the reffered node.

▶ **Example 4.** Suppose we have the following string:

$s = ababbbaaaaabaa |s| = 14$

$t = 7$: s is a 2-antipower with antiperiod 7. $s = (ababbba)(aaaabaa)$;
$t = 6$: s is a prefix of a 3-antipower, w with antiperiod 6. $w = (ababbb)(aaaaab)(aaaaaa)$;
$t = 5$: s is a prefix of a 3-antipower, w with antiperiod 5. $w = (ababb)(baaaa)(abaaa)$;
$t = 4$: s is a prefix of a 4-antipower, w with antiperiod 4. $w = (abab)(bbaa)(aaab)(aaaa)$;
$t = 3$: s is a prefix of a 5-antipower, w with antiperiod 3. $w = (aba)(bbb)(aaa)(aab)(aac)$;
$t = 2$: s is not a 7-antipower. $s = (ab)(ab)(bb)(aa)(aa)(ab)(aa)$.
Therefore, the smallest antiperiod of the string s is 3.

We study two problems in this paper related to the computation of antiperiods.

▶ **Problem 1.** *Given a string S, find the smallest antiperiod of S.*

▶ **Problem 2.** *Given a string S, find all the antiperiods of S.*

In what follows we denote by $\log x$ the function $\log_2 x$ and by $\ln(x)$ the natural logarithm of x. We will also make use of the iterated logarithm function. We denote by $\log^{(1)}(x) = \log(x)$ and for integer $k > 1$ define $\log^{(k)}(x) = \log(\log^{(k-1)}(x))$. We let $k = \log^*(x)$ be the number k such that $\log^{(k)}(x) \leq 1$.

3 Monotone Weighted Level Ancestor Queries

In this section, we are interested in weighted level ancestor queries over leaves in the suffix tree. We will use such queries in subsequent sections to obtain efficient algorithms for computing antiperiods.

An internal node α *spans* depth d iff the depth of α is at least d and *depth* of the parent of α is less than d. A leaf ℓ spans depth d iff the parent of ℓ has depth less than d.

Abusing notation we will identify each leaf of T by its label. A weighted level ancestor query consists of a pair (ℓ, d), where ℓ is a leaf in the suffix tree and d is a depth. The goal is to return the highest ancestor of ℓ that has depth at least d. Although there exist some solutions to this problem, none of them is satisfactory for our purpose. The solution described in [15] supports constant time queries, but does not have an efficient construction algorithm[2]. All the other solutions to this problem have linear preprocessing time, but query time $O(\log\log n)$ [8, 21]. Here we will describe a solution that can be used to efficiently answer *sequences* of weighted-level ancestor queries as long as the depth argument of the queries in the sequence is non-decreasing.

One of the most frequent applications of weighted level ancestor queries is the determination of the locus of substrings of S. More precisely, the locus of a substring $S[i, j]$ can be determined as follows: first determine the leaf ℓ that corresponds to suffix $S[i, n]$. Then, the locus of $S[i, j]$ is obtained by issuing a weighted level ancestor query with pair $(\ell, j - i + 1)$.

▶ **Definition 5.** *A sequence of weighted-level ancestor queries $(\ell_1, d_1), (\ell_2, d_2) \ldots (\ell_t, d_t)$ is called monotone, iff we have that $d_i \leq d_{i+1}$ for all $i \in [1..t-1]$.*

▶ **Definition 6.** *A split-find data structure is a data structure over the interval $[1..n]$ that starts with a set of disjoint non-empty sub-intervals of $[1..n]$ whose union equals $[1..n]$ (the*

[2] Although not specifically analysed in [15], preprocessing time is $O(n \log^4(n))$ [14].

subintervals form a partition of $[1..n]$*) and in which the query* `find`(k) *returns the only subinterval* $[i..j]$ *such that* $i \le k \le j$*, and the update query* `split`$([i..j], k)$ *with* $k \in [i..j-1]$ *removes the interval* $[i..j]$ *and insert intervals* $[i..k]$ *and* $[k+1..j]$*.*

Before describing our proposed solution, we will start by stating some useful lemmas.

▶ **Lemma 7.** *A node alpha associated with an interval* $[i_\alpha, j_\alpha]$ *is an ancestor of a node* β *associated with interval* $[i_\beta, j_\beta]$ *iff* $[i_\beta, j_\beta] \subseteq [i_\alpha, j_\alpha]$

Proof. The lemma is immediate from the definition of the suffix tree and the definition of the intervals associated with the suffix tree nodes. ◀

▶ **Lemma 8.** *The answer to a weighted-level ancestor query* (ℓ, d) *is a node* α *that spans depth* d *and has an associated interval* $[i..j]$ *such that* $\ell \in [i..j]$*.*

Proof. The fact that α needs to span d is immediate from the definition of the query. The fact that $[i..j]$ needs to include ℓ follows from the definition of the query and from Lemma 7. ◀

▶ **Lemma 9.** *The collection of intervals associated with the set of nodes that span depth* d *forms a partition of the interval* $[1, n]$

Proof. We will first prove that the intervals are disjoint and then prove that their union equals $[1, n]$. We will prove that by showing that every leaf is included in exactly one interval that spans depth d. Suppose that a leaf ℓ is included in two intervals $[i_\alpha, j_\alpha]$ and $[i_\beta, j_\beta]$, then one of the nodes α or β is an ancestor of the other (by Lemma 7), which means that it has depth less than d thus does not span d. We will now prove that ℓ is included in at least one interval. Suppose that the parent of ℓ has stringdepth less than d. Then, ℓ spans d and thus the interval $[\ell, \ell]$ is in the collection. Otherwise, clearly one of the ancestors of ℓ spans d and its corresponding interval includes ℓ. ◀

We will now prove the following lemma.

▶ **Lemma 10.** *The collection of intervals corresponding to nodes that span depth* $d+1$ *can be obtained from the collection of intervals that corresponds to nodes that span depth* d *by replacing the intervals that correspond to internal node at depth* d *with intervals of their children.*

Proof. First of all, observe that each node that spans depth $d+1$ either spans depth d or has a parent that spans depth d. Thus generating all nodes that span depth $d+1$ amounts to consider all (intervals of) nodes that span depth d and for each such node either add it (its interval) to the output set or add (intervals of) its children. We will now look at all nodes that span depth d. Observe that a node α that spans depth d will have depth at least d. Consider now two cases:

1. If α has depth more than d, then it will clearly also span depth $d+1$, since it obviously has depth at least $d+1$ and its parent has depth less than d and thus less than $d+1$. It thus suffices to keep the interval that corresponds to α.
2. If α has depth exactly d, then the depth of all its children is at least $d+1$. Those children will obviously span d, since their depth is at least $d+1$ and the depth of their parent α is $d < d+1$. It thus suffices to replace the interval of α with the intervals of its children. ◀

We are now ready to prove the main theorem of this section.

▶ **Theorem 11.** *Suppose that we have a split-find algorithm over n elements in the interval $[1..n]$ that has initialization time $O(n)$ and that supports the* `find` *operation in constant time and any sequence of k* `split` *operations in amortized $O(1 + \frac{n}{k})$ time per operation. Then we can use such an algorithm to support any sequence of k monotone weighted-level ancestor queries over a suffix tree with n leaves in amortized $O(1 + \frac{n}{k})$ time per query.*

Proof. We assume depth $d_1 \geq 1$ (level ancestor queries for depth 0 are trivial – the returned node is the root). Each node α in the suffix tree has a depth d and an associated interval $[i..j]$, where $[i..j]$ is the set of leaves that have α as an ancestor. We initialize the weighted level ancestor query data structure in time $O(n)$ with the intervals associated with the children of the root. We will use an auxiliary table $P[1..n]$ which will store the data associated with each interval.

More precisely, the data associated with interval $[i..j]$ will be stored in the cell $P[i]$, and in our case will consist of a pair (P_1, P_2), where P_1 is a pointer to the suffix tree node to which the interval $[i..j]$ is associated and P_2 is a pointer to the internal representation of the interval $[i..j]$ in the split-find data structure. We also preprocess the suffix tree, and store a table $L[1..n-2]$, where $L[d]$ points to the list of all nodes at depth d (nodes α such that $|\bar{\alpha}| = d$). This preprocessing can clearly be done in $O(n)$ time [3].

We now describe the algorithm. We proceed in $n - 2$ *update steps* numbered from 2 to $n - 1$. These steps are intermingled with the queries. More precisely, if we have $d_i > d_{i-1}$ for some pair of consecutive queries $(\ell_{i-1}, d_{i-1}), (\ell_i, d_i)$, we proceed to all steps $d_{i-1} + 1, \ldots, d_i$, thus preparing the split-find data structure for the next queries. At update step number $u \in [d_{i-1} + 1, d_i]$, we will induce the collection of intervals of nodes that span depth u from the collection of intervals that span depth $u - 1$. Following Lemma 10, we traverse the list of nodes $L[u - 1]$ and for every node α with associated interval $[i..j]$ and split the interval $[i..j]$ in the split-find structure by using the pointer $P[i].P_2$ that points to the internal representation of the interval in the split-find representation. The interval of α will be replaced with the intervals of its children. If α has t children, then $[i..j]$ will be split into k subintervals $[i..i_1], \ldots, [i_{t-1} + 1..j]$. This splitting can be done by successively splitting $[i..j]$ into $[i..i_1]$ and $[i_1 + 1..j]$, then split $[i_1 + 1, j]$ into $[i_1 + 1, i_2]$ and $[i_2 + 1, j]$ and so on, until we have completed the splitting of the interval $[i_{t-2} + 1, j]$ into subintervals $[i_{t-2} + 1, i_{t-1}]$ and $[i_{t-1} + 1, j]$ (each time using and updating cells $P[i], P[i_1 + 1] \ldots$). At the end, the cells $P[i], P[i_1 + 1] \ldots P[i_{t-1} + 1]$ will point to the suffix tree nodes with associated intervals $[i..i_1], \ldots, [i_{t-1} + 1..j]$.

Notice that after step d_i, the subintervals stored in the split-find data structure will precisely form the collection of intervals associated to nodes that span depth d_i. It is clear that this set of nodes is precisely the set of nodes that can be answers to weighted level ancestor queries for depth d_i. Moreover answering a query (ℓ_i, d_i) amounts to finding the subinterval $[i..j]$ such that $i \leq \ell_i \leq j$ and from there retrieve the pointer to the suffix tree node pointed by $P[i]$.

It remains to analyze the amortized time of queries to the weighted-level ancestor query data structure. The total time can be decomposed into three components: the preprocessing time, the update time and the query time. The preprocessing time is dominated by the construction of lists L which as argued above is $O(n)$. The update time is dominated by the updates to the split-find data structure. We argue that the total time for the updates (between queries) is $O(n)$ assuming that the union-find data structure supports a sequence

[3] We traverse all nodes in the tree in depth-first order, computing the depth of every node α from the depth of its parent β and the length of the label of the edge that connects β to α.

of k updates in amortized time $O(1 + n/k)$ which in total gives $O(n + k)$. It is clear that the total number of split operations is upper bounded by the number of children of internal nodes in the suffix tree, since we perform $t - 1$ update operations for an internal node with t children. The total number of children of all internal nodes is clearly upper bounded by the total number of nodes in the suffix tree which is $2n - 1$. We thus conclude that the total time for all updates is $O(n + 2n - 1) = O(n)$.

Finally the amortized query time is constant for each query, since it is dominated by the query time of the split-find data structure which is constant time. Thus the sum of all query times is $O(k)$. We thus conclude that the total time for preprocessing, updates and queries is $O(n)$ and thus the amortized time per operation is $O((n + k)/k) = O(1 + n/k)$. ◀

Since there exists a split-find algorithm over n elements that has initialization time $O(n)$, uses space $O(n)$ working over the interval $[1..n]$, that supports the `find` operation in constant time and any sequence of t `split` in amortized constant $O(1 + \frac{n}{t})$ time per operation [11], we can state the following corollary.

▶ **Corollary 12.** *Given a suffix tree T with n leaves, we can compute the result of any sequence of t monotone weighted-level ancestor queries over T in $O(1 + \frac{n}{t})$ amortized time.*

4 Computing All the Antiperiods of a String

In the smallest antiperiod problem, we are given a string S of length n and have to find the smallest t such that the string S is a prefix of some string $p_0p_1 \dots p_k$, where $|p_i| = t$ for all $i \in [0..k]$ and $p_i \neq p_j$ for all $i \neq j$. This problem can be solved using the suffix tree of S by making use of the following observation:

▶ **Lemma 13.** *Two substrings $S[i, i + d - 1]$ and $S[j, j + d - 1]$ are equal iff they have the same locus in the suffix tree of S.*

The algorithm proceeds in (at most) $\lfloor n/2 \rfloor$ phases, where in phase i, it is tested whether value $t = i$ is an antiperiod of S. Such a test can be carried out via $\lfloor n/i \rfloor$ monotone weighted level-ancestor queries. That is, in phase i, we compute weighted level ancestor queries for (weighted) level i for the leaves labeled with positions 1, $i + 1$, $2i + 1$, $3i + 1$, and so on. If at any point we visit the same node (i.e. ancestor) a second time, then S cannot have antiperiod i because at least two of the substrings of length i starting at the tested positions are equal. On the other hand if all ancestors are unique so are the corresponding substrings of length i (at tested positions), implying that i is an antiperiod.

Let t be the first phase in which all ancestor nodes returned by the monotone level ancestor queries are unique. It is clear that t is the smallest antiperiod of S. We now analyse the running time. The ith phase involves n/i level ancestor queries. Therefore if we stop the algorithm at phase t, the total number of queries will be $N = O(\sum_{i=1}^{t} n/i) = O(n \log t)$. We now bound the total time taken by all queries. First of all, notice that queries have the monotonicity property, which means that we can apply Theorem 11. Now, by Theorem 11, each of the N queries is supported in amortized $O(1 + \frac{n}{N}) = O(1)$ time per query which means that the total time for all N queries is $O(N) = O(n \log t)$. We have thus obtained the following theorem.

▶ **Theorem 14.** *We can find the smallest antiperiod of a string of length n in $O(n \log t)$ time, where t is the length of the antiperiod.*

Clearly, if instead of stopping at the point at which we have computed the smallest antiperiod we continue to run the algorithm up to phase $\lfloor n/2 \rfloor$, we will compute all the antiperiods of the string, obtaining the following theorem.

▶ **Theorem 15.** *We can find the all the antiperiods of a string of length n in $O(n \log n)$ time.*

5 Faster Computation of the Smallest Antiperiod

In this section we show how to obtain a faster algorithm for the smallest antiperiod problem via the following two easily proved properties of antiperiods.

▶ **Lemma 16.** *If t is a an antiperiod, then any multiple of t is also an antiperiod.*

▶ **Lemma 17.** *If t is the smallest antiperiod, then $1, 2 \ldots, t-1$ are not antiperiods.*

Using the two observations above, we can improve the solution described in the previous section. More precisely, we can reduce the time down to $O(n \log \log t)$ as follows. We first test whether the value $i = 1$ is an antiperiod. If not we proceed as follows. Let us first define $f(x) = x\lceil \log^2 x \rceil$. For increasing values of $i = 2, 3, \ldots$, we test antiperiod $f(i)$, and stop whenever we find a value i such that $f(i)$ is an antiperiod. Clearly this first step takes time:

$$O(n) + \sum_{i=2}^{\infty} O\left(\frac{n}{i\lceil \log^2 i \rceil}\right) = O\left(n + n\sum_{i=2}^{\infty} \frac{1}{i\lceil \log^2 i \rceil}\right) = O(n).$$

This is because the series $\sum_{i=2}^{\infty} 1/(i\lceil \log^2 i \rceil)$ converges to a constant (for completeness this is shown in the appendix).

Now, for any $j \in [2..i-1]$, we have the following two facts:

1. $f(j)$ is multiple of j.
2. $f(j)$ is not an antiperiod.

By Lemma 16, these two facts imply that j is not an antiperiod. We thus have proved that the smallest antiperiod is in the range $[i..f(i)]$. Then we test all antiperiods between i and $f(i)$ (in increasing order), taking time:

$$\begin{aligned} O(n) + \sum_{j=i}^{f(i)} O\left(\frac{n}{j}\right) &= O\left(n + \sum_{j=1}^{f(i)} \frac{n}{j} - \sum_{j=1}^{i} \frac{n}{j} + \frac{n}{i}\right) \\ &= O\left(n + n(\log f(i) - \log(i))\right) \\ &= O\left(n + n(\log(i\lceil \log^2 i \rceil) - \log(i))\right) \\ &= O(n \log\lceil \log^2 i \rceil) = O(n \log \log i) \end{aligned}$$

Clearly we have that $t \in [i..f(i)]$ and so $O(n \log \log i) \in O(\log \log t)$. We thus have proved the following theorem

▶ **Theorem 18.** *We can find the smallest antiperiod t of a string of length n in time $O(n \log \log t)$.*

5.1 Recursive solution

By recursing on the solution described above multiple times we get total time $O(n \log^* t)$ to find the smallest antiperiod. That is, at the second step, instead of testing all antiperiods between i and $f(i)$, we instead test every $\ell = \lceil \log\log i \rceil$ value, spending total time $O(n)$. That is, we test the antiperiods $i \cdot \ell, (i+1)\cdot \ell, \ldots, \lceil \frac{f(i)}{\ell} \rceil \cdot \ell$. The complexity is this time:

$$\begin{aligned} O(n) + \sum_{j=i}^{\lceil \frac{f(i)}{\ell} \rceil} O\left(\frac{n}{j\ell}\right) &= O\left(n + \frac{n}{\ell} \cdot \sum_{j=i}^{\lceil \frac{f(i)}{\ell} \rceil} \frac{1}{j}\right) \\ &\in O\left(n + \frac{n}{\ell} \cdot \sum_{j=i}^{f(i)} \frac{1}{j}\right) \\ &= O(n + \frac{n}{\ell} \cdot (\log(f(i)) - \log(i) + \frac{1}{i}) \\ &= O(n + \frac{n}{\ell} \cdot (\log\log i)) = O(n). \end{aligned}$$

In the formulae above, the $O(n)$ terms in the beginning is due to the monotone weighted-level ancestor data structure. At the end of this second step we will have determined an interval $[i_2, i_2 \cdot \ell]$. In third step, we continue doing the tests but testing every $\ell_2 = \lceil \log\log\log i \rceil$ value as a candidate antiperiod, obtaining interval $[i_3, i_3 \cdot \ell_2]$. We continue that way, at each step k using value $\ell_{k-1} = \lceil \log^{(k)}(i) \rceil$ until we get into an interval $[i_k..i_k \cdot \lceil \log^{(k)}(i) \rceil]$, such that $\log^{(k)}(i) \leq 3$. Then, we can test all values inside that interval. We also have $\log^* i = O(\log^* t)$. Now, since we have $O(\log^* t)$ steps and each step takes $O(n)$ time, the total execution time of our algorithm is $O(n \log^* t)$.

▶ **Theorem 19.** *We can find the smallest antiperiod t of a string of length n in $O(n \log^* t)$ time.*

5.2 Linear time solution

We now show how to further reduce the time for finding the smallest antiperiod from $O(n \log^* t)$ to $O(n)$. We will make use of the substring hashing data structure described by Gawrychowski [13]. Given a string $S[1..n]$, the data structure allows to associate to each distinct substring of S a unique integer in the range $[1..O(n^3)]$. The data structure can be precomputed in $O(n)$ time and returns the unique identifier associated to a given substring $S[s..e]$ in constant time. For testing an antiperiod p, we will make $\lfloor n/p \rfloor$ queries to the substring hashing data structure and collect the obtainted integer identifiers. We can check whether two substrings are equal by using radix-sort to sort all the identifiers in time $O(\sqrt{n} + (n/p) \cdot \frac{\log n}{\log n/2}) = O(\sqrt{n} + n/p)$ using buckets of size $2^{\lceil \log(n)/2 \rceil}$. At the end of the sorting we scan the identifiers to check whether any identifier occurs twice, and if not, conclude that p is an antiperiod. We will modify the recursive algorithm as follows.

At each step $k+1$ we will use $\ell_k = \lceil (\log^{(k)}(i))^2 \rceil$ instead of $\ell_k = \lceil \log^{(k)}(i) \rceil$. The running time of step $k+1$ will thus be

$$O\left(n \frac{\log \ell_{k-1}}{\ell_k}\right) = O\left(n \frac{2\log^{(k)}(i)}{(\log^{(k)}(i))^2}\right) = O\left(\frac{n}{\log^{(k)}(i)}\right)$$

instead of $O(n)$ (notice that we avoind the $O(n)$ term due to the weighted-level ancestor data structure). Summing up over all the steps, the running time will be:

$$O\left(n + \frac{n}{\log\log i} + \frac{n}{\log\log\log i} + \ldots + n\right) = O(n)$$

We thus have proved the following theorem.

▶ **Theorem 20.** *We can find the smallest antiperiod of a string of length n in $O(n)$ time.*

References

1 Golnaz Badkobeh, Gabriele Fici, and Simon J. Puglisi. Algorithms for Anti-Powers in Strings. *Information Processing Letters*, 137:57–60, 2018. arXiv:1805.10042.

2 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "Runs" Theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017.

3 Carl Barton, Tomasz Kociumaka, Chang Liu, Solon P Pissis, and Jakub Radoszewski. Indexing weighted sequences: neat and efficient. *arXiv preprint*, 2017. arXiv:1704.07625.

4 Amanda Burcroff. (k, λ)-Anti-Powers and Other Patterns in Words. *Electronic Journal of Combinatorics*, 25(4):#P4.41, 2018.

5 Maxime Crochemore, Lucian Ilie, and Liviu Tinta. The "runs" conjecture. *Theor. Comput. Sci.*, 412(27):2931–2941, 2011.

6 Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology.* World Scientific, 2002.

7 Colin Defant. Anti-Power Prefixes of the Thue-Morse Word. *Electronic Journal of Combinatorics*, 24(1):#P1.32, 2017.

8 Martin Farach and S Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *Annual Symposium on Combinatorial Pattern Matching*, pages 130–140. Springer, 1996.

9 Gabriele Fici, Antonio Restivo, Manuel Silva, and Luca Q. Zamboni. Anti-Powers in Infinite Words. In *43rd International Colloquium on Automata, Languages, and Programming, (ICALP)*, volume 55 of *LIPIcs*, pages 124:1–124:9. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

10 Gabriele Fici, Antonio Restivo, Manuel Silva, and Luca Q. Zamboni. Anti-powers in infinite words. *J. Comb. Theory, Ser. A*, 157:109–119, 2018. doi:10.1016/j.jcta.2018.02.009.

11 Harold N Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of computer and system sciences*, 30(2):209–221, 1985.

12 Marisa Gaetz. Anti-power j-fixes of the Thue-Morse word. arXiv:1808.01528.

13 Paweł Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: fast, simple, and deterministic. In *European Symposium on Algorithms*, pages 421–432. Springer, 2011.

14 Paweł Gawrychowski. Personal Communication, 2018.

15 Paweł Gawrychowski, Moshe Lewenstein, and Patrick K Nicholson. Weighted ancestors in suffix trees. In *European Symposium on Algorithms*, pages 455–466. Springer, 2014.

16 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

17 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. A Linear Time Algorithm for Seeds Computation. *arXiv preprint*, 2011. arXiv:1107.2422.

18 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, and Wiktor Zuba. Efficient representation and counting of antipower factors in words. In *International Conference on Language and Automata Theory and Applications*, pages 421–433. Springer, 2019.

19 Roman M. Kolpakov, Ghizlane Bana, and Gregory Kucherov. mreps: efficient and flexible detection of tandem repeats in DNA. *Nucleic Acids Research*, 31(13):3672–3678, 2003.

20 Roman M. Kolpakov and Gregory Kucherov. Finding Maximal Repetitions in a Word in Linear Time. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 596–604, 1999.

21 Tsvi Kopelowitz and Moshe Lewenstein. Dynamic weighted ancestors. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 565–574. Society for Industrial and Applied Mathematics, 2007.

22 M. Lothaire. *Algebraic Combinatorics on Words.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2002. doi:10.1017/CBO9781107326019.

23 Shyam Narayanan. Functions on antipower prefix lengths of the Thue-Morse word. arXiv:1705.06310.

24 W. F. Smyth. *Computing Patterns in Strings.* Pearson Addison Wesley, United Kingdom, 2003.

25 Axel Thue. Uber unendliche Zeichenreihen. *Norske vid. Selsk. Skr. I. Mat. Nat. Kl. Christiana*, 7:1–22, 1906.

A Complement of Section 4

We show that the series $\sum_{i=2}^{\infty} 1/(i\lceil\log^2 i\rceil)$ converges to a constant. This can be proved as follows. Since the function $f(x) = 1/(x(\ln x)^2)$ is monotonically decreasing over $[2, \infty)$, we can upper bound the sum

$$\sum_{i=2}^{\infty} \frac{1}{i(\ln i)^2}$$

by the integral

$$\int_2^{\infty} \frac{1}{x(\ln x)^2} \cdot \mathrm{d}x$$

By replacing $x = e^u$, we get

$$\int_2^{\infty} \frac{1}{x(\ln x)^2} \cdot \mathrm{d}x = \int_{\ln 2}^{\infty} \frac{1}{e^u u^2}(e^u \cdot \mathrm{d}u) = \int_{\ln 2}^{\infty} \frac{1}{u^2} \cdot \mathrm{d}u.$$

The antiderviative of $1/u^2$ being equal to $-1/u$, we get that

$$\int_{\ln 2}^{\infty} \frac{1}{u^2} \cdot \mathrm{d}u = \frac{1}{\ln 2} - \frac{1}{\infty} = \frac{1}{\ln 2}.$$

We thus have

$$\begin{aligned}
\sum_{i=2}^{\infty} \frac{1}{i\lceil\log^2 i\rceil} &\le \sum_{i=2}^{\infty} \frac{1}{i\log^2 i} \\
&= 1 + \sum_{i=2}^{\infty} \frac{(\ln 2)^2}{i(\ln i)^2} \\
&\le 1 + (\ln 2)^2 \cdot \int_2^{\infty} \frac{1}{x(\ln x)^2} \cdot \mathrm{d}x \\
&= 1 + (\ln 2)^2 \cdot \frac{1}{\ln 2} = 1 + \ln 2.
\end{aligned}$$