Repetition Detection in a Dynamic String

Amihood Amir
Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
amir@esc.biu.ac.il

Itai Boneh
Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
barbunyaboy2@gmail.com

Panagiotis Charalampopoulos
Department of Informatics, King's College London, London, UK
Efi Arazi School of Computer Science, The Interdisciplinary Center Herzliya, Herzliya, Israel
panagiotis.charalampopoulos@kcl.ac.uk

Eitan Kondratovsky
Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
eit414@gmail.com

Abstract

A string UU for a non-empty string U is called a square. Squares have been well-studied both from a combinatorial and an algorithmic perspective. In this paper, we are the first to consider the problem of maintaining a representation of the squares in a dynamic string S of length at most n. We present an algorithm that updates this representation in $n^{o(1)}$ time. This representation allows us to report a longest square-substring of S in $\mathcal{O}(1)$ time and all square-substrings of S in $\mathcal{O}(\mathsf{output})$ time. We achieve this by introducing a novel tool – maintaining prefix-suffix matches of two dynamic strings.

We extend the above result to address the problem of maintaining a representation of all runs (maximal repetitions) of the string. Runs are known to capture the periodic structure of a string, and, as an application, we show that our representation of runs allows us to efficiently answer periodicity queries for substrings of a dynamic string. These queries have proven useful in static pattern matching problems and our techniques have the potential of offering solutions to these problems in a *dynamic* text setting.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string algorithms, dynamic algorithms, squares, repetitions, runs

Digital Object Identifier 10.4230/LIPIcs.ESA.2019.5

Funding *Amihood Amir*: Supported by Israel Science Foundation (ISF) grant 1475/18 and United States – Israel Binational Science Foundation (BSF) grant 2018141.
Panagiotis Charalampopoulos: Partially supported by Israel Science Foundation (ISF) grant 794/13.

Acknowledgements We warmly thank Tomasz Kociumaka for useful discussions.

1 Introduction

A string UU, where U is not empty, is called a *square* or a *tandem repeat*. Squares are a fundamental construct in word combinatorics, and algorithms for finding all squares have been sought as early as the 1980's [15, 10, 37]. The problem turned out to be central in computational biology causing much algorithmic work to have taken place since then [12, 27]. The approximate version is also of great interest [36, 19, 41, 40].

A run is a periodic fragment of the text that cannot be extended to either direction without increasing its period. Kolpakov and Kucherov, in their seminal paper [35], showed that there are $\mathcal{O}(n)$ runs in a text of length n, and presented an algorithm to compute them

in $\mathcal{O}(n)$ time. After a long line of research, the breakthrough result of Bannai et al. [11] showed that a string of length n can have at most n runs. Runs have been used as an algorithmic tool, for example, for extracting the k-powers in a string (note that a square is a 2-power) and for efficient computation of the periods of substrings of a string [17, 33, 34].

Due to the importance, both theoretical and practical, of squares and runs, it is surprising that the problem of computing or maintaining them in a dynamic text has not been studied. Of course, one can re-run a square/run detection algorithm after every change in the text, but this is clearly a very inefficient way of handling the problem.

In the 1990's the active field of dynamic graph algorithms was started, with the motive of answering questions on graphs that dynamically change over time. For an overview see [18]. Recently, there has been a growing interest in dynamic pattern matching. This natural interest grew from the fact that the biggest digital library in the world - the web - is constantly changing, as well as from the fact that other big digital libraries - genomes and astrophysical data, are also subject to change through mutation and time, respectively.

Historically, there has been much interest in dynamic string matching algorithms. Amir and Farach [7] introduced dynamic dictionary matching, which was later improved by Amir et al. [8]. Idury and Scheffer [29] introduced an automaton-based dynamic dictionary algorithm. Gu et al. [26] and Sahinalp and Vishkin [39] developed a dynamic indexing algorithm, where a dynamic text is indexed. Further progress in dynamic indexing and dictionary matching was achieved by Ferragina et al. [20, 21] and Mehlhorn et al. [38]. Pattern matching algorithms where the text is dynamic and the text is static were also considered [2, 9].

In the last few years there was a resurgence of interest in dynamic string matching. In 2017 a theory began to develop with its nascent set of tools. Bille et al. [13] investigated dynamic relative compression and dynamic partial sums. Amir et al. [5] considered the longest common factor (LCF) problem in the case of one revertible edit (see also [1]). Special cases of the dynamic LCF problem were discussed by Amir and Boneh [3]. An algorithm for the fully dynamic LCF problem was presented by Amir et al. [6]. (A similar line of work has taken place for the problem of maintaining a longest palindrome in a dynamic string [23, 24, 6, 4].) Gawrychowski et al. [25] settled the complexity of maintaining a dynamic collection of strings under operations: concatenate, split, makestring, lexicographic comparison, and finding the longest common prefix of two strings.

We continue this line of work by considering squares and runs in a dynamic string. We present our algorithms for the case where the allowed update operations are substitutions. We first show our techniques in the setting of the following problem.

Dynamic Longest Square
Input: A string S.
Query: For given index i (and character α), set $S[i] = \alpha$ and compute $\mathsf{LS}(S)$.

Our contributions. We make a step forward in the exciting area of dynamic pattern matching. We give efficient dynamic solutions for a number of important problems:

1. Fully dynamic pattern matching in a text and pattern where the text length is twice the pattern length. In fact, to our knowledge, this is the first known algorithm that does not require $\Omega(\mathsf{occ})$ time to report all pattern occurrences, i.e. it may report them in time smaller than their number, by reporting occurrences via an arithmetic progression.
2. Dynamic maintenance of the *longest* square in a text in $n^{o(1)}$ time per string update, after an $\tilde{\mathcal{O}}(n)$-time preprocessing and using $\tilde{\mathcal{O}}(n)$ space. [1]

[1] The $\tilde{\mathcal{O}}(\cdot)$ notation suppresses $\log^{\mathcal{O}(1)} n$ factors.

3. Dynamic maintenance of all *runs* in a text within the same complexities. It is noteworthy that although a single substitution can destroy/create $\Omega(n)$ squares/runs, we can maintain a compact representation of them in subpolynomial time.
4. We conclude by showing that our representation of runs can be employed to efficiently maintain k-powers in a dynamic text and answer queries about periodicity of substrings, adapting the static solutions of Crochemore et al. [17]. Detecting internal periodicities has proven useful in several static string matching applications, thus our dynamic algorithm can potentially help the dynamic version of such problems. An overview of the literature for internal queries in static texts can be found in [32].

We introduce a *n*ew technique, which we expect will be a powerful tool in other dynamic string matching problems, that of dynamically maintaining *prefix-suffix matches*. This enabled us to efficiently maintain all runs in a dynamic string, which, in turn, enabled the applications presented in this paper.

2 Preliminaries

We begin with basic definitions and notation generally following [16]. Let $S = S[1]S[2]\cdots S[n]$ be a *string* of length $|S| = n$ over an integer alphabet Σ. For two positions i and j on S, we denote by $S[i\,..\,j] = S[i]\cdots S[j]$ the fragment of S that starts at position i and ends at position j (it is the empty string ε if $j < i$). A string Y, of length m with $0 < m \leq n$, is a substring of S if there exists a position i in S such that $Y = S[i\,..\,i+m-1]$. In this case we say that there exists an *occurrence* of Y in S, or, more simply, that Y *occurs in* S at *(starting) position* i. A substring is called *proper* if it is shorter than the whole string. A fragment $S[1\,..\,j]$, $j < n$, is called a prefix of S, and, analogously, a fragment $S[i\,..\,n]$, $i > 1$, is called a suffix. A fragment of S that is neither a prefix nor a suffix of S is called an infix. A string B that occurs both as a proper prefix and a proper suffix of S is called a *border* of S. A positive integer p is called a *period* of S if $S[i] = S[i+p]$ for all $i = 1, \ldots, n-p$. String S has a period p if and only if it has a border of length $n-p$. We refer to the smallest period $\mathsf{per}(S)$ of S as *the period* of the string and, analogously, to the longest border as *the border* of the string. A string S is *periodic* if $\mathsf{per}(S) \leq |S|/2$.

By ST and S^k we denote the concatenation of strings S and T and k copies of the string S, respectively. A string of the form S^2 for some $S \in \Sigma^+$ is called a *square* and a string of the form S^k is called a *k-power*.

A *run* (also known as *maximal repetition*) is a periodic fragment $R = S[a\,..\,b]$ which cannot be extended to the left nor to the right without increasing the period $p = \mathsf{per}(R)$, that is, $S[a-1] \neq S[a+p-1]$ and $S[b-p+1] \neq S[b+1]$. The number of runs in a string of length n is at most n [11] and all runs can be computed in $\mathcal{O}(n)$ time [35].

By $\mathsf{lcpstring}(S,T)$ we denote the longest common prefix of S and T, by $\mathsf{lcp}(S,T)$ we denote $|\mathsf{lcpstring}(S,T)|$, and by $\mathsf{lcp}(r,t)$ we denote $\mathsf{lcp}(S[r\,..\,n], S[t\,..\,n])$. The longest common suffix lcs is defined analogously. We refer to queries returning $\mathsf{lcp}(r,s)$ or $\mathsf{lcs}(r,s)$ as longest common extension queries (*LCE queries*).

It is known that by maintaining Karp-Rabin fingerprints [31] for the substrings of length 2^j starting at a positions $i = 1 \pmod{2^j}$ for all $1 \leq j \leq \log n$ one can obtain the following lemma. (More involved solutions with better complexities in the $\mathcal{O}$-notation can be obtained by applying for instance the results of [25], cf. [6].)

▶ **Lemma 1.** *A dynamic string can be maintained with $\tilde{\mathcal{O}}(1)$-time per edit operation so that LCE queries can be answered in $\tilde{\mathcal{O}}(1)$ time, using $\tilde{\mathcal{O}}(n)$ space.*

3 An $\tilde{\mathcal{O}}(n^{2/3})$-time algorithm

In this section we present an algorithm that reports a longest square $\mathsf{LS}(S)$ in time $\tilde{\mathcal{O}}(n^{2/3})$ after each substitution operation. We actually present two algorithms, one for each of the cases of $\mathsf{LS}(S)$ being short or long. Let m, which is to be chosen later, be the distinguishing threshold between those cases. We can assume that m is a power of 2.

Main idea. In order to handle the case of $\mathsf{LS}(S)$ being short, we split our text into $\mathcal{O}(n/m)$ overlapping fragments of length $2m$. Each substitution operation affects only two of these fragments, and we thus just recompute the longest square in them in $\mathcal{O}(m)$ time. As for the case that the $\mathsf{LS}(S)$ UU is long, we note that the first occurrence of U must contain a fragment of length $m/4$ for some $i = 1 \pmod{m/4}$. The idea is to use such fragments as anchors and maintain all of their occurrences in the string using dynamic renaming. Then for every such fragment and every occurrence of it we would like to check whether there is any square UU that contains them "aligned" in the two occurrences of U. We show how to process fragments that have many occurrences efficiently by exploiting periodicity.

3.1 $|\mathsf{LS}(S)| \leq m$

Preprocessing. We split the string S into overlapping fragments, each of length $2m$, starting at positions $i = 1 + j \cdot m$ for $j = 0, 1, \ldots \lceil n/m \rceil$. (Note that the last two fragments could be shorter than $2m$.) We use a linear-time algorithm ([17, 28]) to compute all squares in each of these fragments, requiring time $\mathcal{O}(m \cdot n/m) = O(n)$ in total. For each fragment, we will maintain a representative longest square-substring, which is chosen arbitrarily in case of ties. We store the representatives of all fragments in a max heap, with their lengths being the keys. The max heap can be built in time $\mathcal{O}(n)$.

Query. Every position of the string is contained in at most two fragments. After each substitution operation we use a linear-time algorithm to recompute all squares in the affected fragments, requiring time $\mathcal{O}(m)$. We then update the heap in time $\mathcal{O}(\log n)$ by deleting the previous representatives (to which we have stored pointers) and inserting the new ones. We then simply retrieve the longest element in the max heap in $\mathcal{O}(1)$ time. The overall query-time complexity is $\mathcal{O}(m + \log n)$.

Correctness. The correctness of the described algorithm follows directly from the observation that each substring of S of length at most m is fully contained in at least one of the $O(n/m)$ $2m$-length fragments.

3.2 $|\mathsf{LS}(S)| \geq m = 4k$

Let us start with an observation.

▶ **Observation 2.** *In a square-substring UU of S, with $|U| \geq 2k$, the first occurrence of U contains $S[i \,..\, i + k - 1]$ for some $i = 1 \pmod{k}$.*

This observation guarantees that long square-substrings of S can be identified using the $\mathcal{O}(n/k)$ fragments starting at positions $i = 1 \pmod{k}$ as anchors. To this end, we maintain names for all k-length fragments of the string, such that two fragments of S have the same name if and only if they are equal. We first briefly describe the renaming technique, originating from [30], and then show how to use it in the dynamic setting.

The renaming technique. We recursively (consistently) rename pairs of letters of a string S. Let us consider the original string $S = S_0$ as the string at level 0 and the resulting string S_λ after λ iterations of renaming as the string at level λ. At level λ, the letter of S_λ at position i corresponds to $S[i\,..\,i+2^\lambda-1]$; in other words $S_\lambda[i] = S_\lambda[j]$ if and only if $S[i\,..\,i+2^\lambda-1] = S[j\,..\,j+2^\lambda-1]$.

Given string S_λ we rename as follows. We consider the multiset V of all pairs $p_\lambda(i) = (S_\lambda[i], S_\lambda[i+2^\lambda-1])$ and radix sort them in $\mathcal{O}(n)$ time. We then assign a distinct integer identifier $f(v)$ from $\{1,\ldots,n\}$ to each distinct element of V. Finally, we set $S_{\lambda+1}[i] = f(p_\lambda(i))$. This process terminates after $\log n$ iterations and thus requires time $\mathcal{O}(n\log n)$ in total.

Dynamic renaming. We only maintain $\log k$ levels of renaming, i.e. strings $S_0,\ldots,S_{\log k}$. We maintain the different pairs of letters at each level λ in a balanced binary search tree (bBST) B_λ; each node of B_λ stores the name given to this pair and a counter of its occurrences in the string. We also maintain a bBST C_λ storing the letters from $\{1,\ldots,n\}$ that are not currently used to rename pairs at this level. Each substitution affects at most k k-length fragments. We update their names in a bottom up manner in time $\mathcal{O}(k\log n)$ as follows. For each affected pair of letters at a level λ that changed, for example, from (a,b) to (a,c), we search for (a,b) in B_λ and decrement its counter. In addition, if the counter reaches 0, the name given to this letter is now free and we update C_λ accordingly. We then search for (a,c) in B_λ and, (a) if we find it, we increment the counter and use the stored name, (b) else we insert (a,c) to B_λ and assign to this pair of letters the smallest element of C_λ.

In addition, for each name a of a k-length fragment (i.e. letter at level $\lambda = \log k$) we store the positions of its occurrences in S_λ in a predecessor data (bBST) structure P_a. We can perform insertions and deletions as well as perform predecessor/successor queries in P_a in $\mathcal{O}(\log n)$ time each. Below, after a brief discussion on periodicity, we will present a modification on P_a in order to compactly store the occurrences of a.

Computing squares. We would like to pair each of the k-length fragments starting at a position $i = 1 \pmod k$, with name a, with all of its other occurrences in S, which can be retrieved from the predecessor structure P_a. Let $j \neq i$ be the position of such an occurrence, and denote such a pair as (i,j). We can assume without loss of generality that $i < j$; the other case is symmetric. For each pair, we want to check whether a square $S[a\,..\,b] = UU$, such that $a \le i < j \le b$ and $j - i = |U|$, exists. We call each such square an (i,j)-*square*. Observation 2 guarantees that every square of length at least $m = 4k$ will be identified in this manner. The following lemma shows how to perform the described check efficiently.

▶ **Lemma 3.** *Given two positions $i < j$, we can check whether an (i,j)-square exists and report all (i,j)-squares compactly in time $\tilde{\mathcal{O}}(1)$.*

Proof. The following observation essentially reduces computing all (i,j)-squares to answering two LCE queries. Inspect Figure 1 for an illustration.

▶ **Observation 4.** *An (i,j)-square UU, where i is the t-th letter of the first occurrence of U exists if and only if $\mathsf{lcs}(i,j) \ge t$ and $\mathsf{lcp}(i,j) \ge |U| - t + 1$.*

Now $1 \le t \le |U|$ and $t = i - a + 1$, where a is the starting position of such a square. Hence $a = i + 1 - t$ for $1 \le t \le |U|$ such that $\mathsf{lcs}(i,j) \ge t$ and $\mathsf{lcp}(i,j) \ge |U| - t + 1$ are the starting positions of all (i,j)-squares. Equivalently, the (i,j)-squares are the fragments $S[a\,..\,a+2|U|-1]$, for $i + 1 - \min\{\mathsf{lcs}(i,j), |U|\} \le a \le \min\{i + \mathsf{lcp}(i,j) - |U|, i\}$. We employ Lemma 1 to efficiently answer LCE queries. ◀

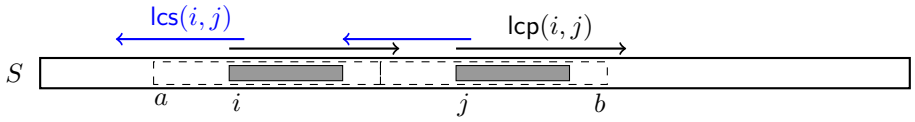

Figure 1 The setting in the proof of Lemma 3. The two occurrences of U in an (i, j)-square UU are denoted by dashed rectangles. The two equal k-length fragments starting at positions i and j are denoted by gray rectangles.

Aperiodic k-length substrings. If the k-length fragment $S[i\,..\,i+k-1]$ to be processed is aperiodic, it occurs $\mathcal{O}(n/k)$ times in S. We can thus afford to employ Lemma 3 for each pair (i, j), where $j \neq i$ is a position where $S[i\,..\,i+k-1]$ occurs. The time required to process $S[i\,..\,i+k-1]$ is thus $\tilde{\mathcal{O}}(n/k)$.

Periodic k-length substrings. If the k-length substring is periodic then we cannot process each pair individually as there could be $\Omega(n)$ of them. To overcome this, we exploit periodicity to process the pairs in batches. The lemma below follows directly from the periodicity lemma, which states that if a string has a period p and a period q, such that $p + q \leq n + \gcd(p, q)$, then $\gcd(p, q)$ is also a period of this string [22].

▶ **Fact 5.** *The distance between the starting positions of two consecutive occurrences of a periodic string Y with period p in a string S is either p or greater than $|Y|/2$.*

We now present an algorithm to process a k-length substring Y with period p that occurs more than $3n/k$ times in S. We can treat periodic substrings that occur fewer than $3n/k$ times with the algorithm for the aperiodic ones. Note that this is in fact necessary, as we cannot afford to compute the period of each relevant substring. Instead, we identify periodic substrings that occur frequently as follows. Remember that we have stored the positions where a k-length fragment Y with name a occurs in a predecessor data structure P_a. Then, in light of Fact 5, if Y occurs more than $3n/k$ times, two of its occurrences will have to be at distance $\mathsf{per}(Y)$. We will identify this by checking the distance of each newly inserted element in the predecessor data structure with its predecessor and successor. If it happens to be below $|Y|/2$, we store this distance, which is $\mathsf{per}(Y)$, as satellite information in P_a.

We call a set of positions $A = \{j + t \cdot p | t = 0, \ldots, r\}$ a p-cluster of Y in S if $p = \mathsf{per}(Y)$, $S[a\,..\,a+k-1] = Y$ for all $a \in A$ and $S[j-p\,..\,j-p+k-1] \neq Y \neq S[j+(t+1)p\,..\,j+(t+1)p+k-1]$. It follows directly from Fact 5 that there are $\mathcal{O}(n/k)$ p-clusters of Y in S. We maintain these p-clusters by storing p-cluster A as an arithmetic progression $(\min A, p, |A|)$ with key $\min A$ in P_a. We merge p-clusters if needed by using predecessor/successor queries in P_a upon insertions, and similarly split p-clusters if needed upon deletions.

▶ **Observation 6.** *Let Y be a periodic string. An occurrence of Y in S is a fragment of exactly one run R with $\mathsf{per}(R) = \mathsf{per}(Y)$. We say that R extends Y. The p-cluster containing this occurrence of Y corresponds to the occurrences of Y in R.*

▶ **Lemma 7.** *Given a periodic fragment Y and $p = \mathsf{per}(Y)$, the run R that extends Y can be computed using a constant number of LCE queries. $R = S[i - a + 1\,..\,i + p + b - 1]$, where $a = \mathsf{lcs}(i, i + p)$ and $b = \mathsf{lcp}(i, i + p)$.*

We next show how to process the pairs yielded by each of the p-clusters in $\tilde{\mathcal{O}}(1)$ time.

▶ **Theorem 8.** *Given a position i in S, where Y occurs, and a p-cluster A of Y in S, we can compute a longest (i, j)-square over all $j \in A$ in time $\tilde{\mathcal{O}}(1)$. In particular, if $i \notin A$, we return a superset of all (i, j)-squares for $j \in A$ that are of length at least $4k$ in a compact form.*

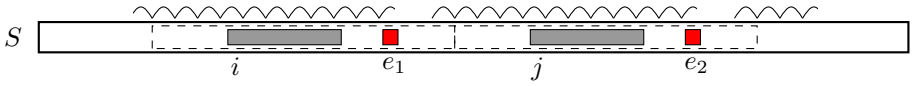

Figure 2 An illustration of the setting in Case 1 in the proof of Theorem 8. As before, the two occurrences of U in an (i,j)-square UU are denoted by dashed rectangles and the two equal k-length fragments starting at positions i and j are denoted by gray rectangles.

Proof. If it so happens that $i \in A$, then the longest (i,j)-square can be easily retrieved as it must lie entirely within the run $R = S[a\,..\,b]$ corresponding to A. Let $r = b - a \pmod{2p}$. It can be readily verified that either $S[a+r\,..\,b]$ or $S[a\,..\,b-r]$ is a longest (i,j)-square over all $j \in A$. (See also [17].)

In the other case, that is $i \notin A$, we first compute the unique run $R_1 = S[s_1\,..\,e_1]$ that extends the occurrence of Y at position i, and similarly the run $R_2 = S[s_2\,..\,e_2]$ corresponding to the occurrences of Y in A. This can be done in time $\tilde{\mathcal{O}}(1)$ by performing a constant number of LCE queries, cf. Lemmas 7 and 1.

Our assumption that $i < j$ implies that $s_1 < s_2$. Let UU be an (i,j)-square with $j \in A$. We have the following cases for the occurrence of U in which $S[e_1+1]$ lies.

1. The first occurrence, in which case the endpoints $S[e_1]$ and $S[e_2]$ of the two runs must be aligned (i.e. be at distance $|U|$), since $\mathsf{lcp}(i,j) > e_1 + 2 - i$. In other words, $S[e_1]$ and $S[e_2]$ must both occur as the t-th letter of an occurrence of U in the square for some t – inspect Figure 2 for an illustration. In this case we compute the longest (e_1,e_2)-square (or all (e_1,e_2)-squares) in $\tilde{\mathcal{O}}(1)$ time using Lemma 3.
2. The second occurrence, in which case, the situation is more interesting. We have the following two subcases.
   a. If $e_1 + 1 < s_2$, by an argument symmetric to that for the first case, the starting points $S[s_1]$ and $S[s_2]$ of the two runs must be aligned – one can think of Figure 2 reversed. As in Case 1, we can compute the longest (all) (s_1,s_2)-square(s) in $\tilde{\mathcal{O}}(1)$ time using Lemma 3.
   b. Else, we have that the first and second occurrences of U are fragments of runs R_1 and R_2, respectively.

We now look into the structure yielded by the condition in Case 2b and show how to compute and represent all (possibly many) squares that satisfy it, and are essentially defined by runs R_1 and R_2, efficiently.

▶ **Definition 9.** *For two runs R_1 and R_2, with period $\mathsf{per}(R_1) = \mathsf{per}(R_2) = p$ that overlap, we define $\mathsf{sq}(R_1,R_2)$ to be the set of squares UU of length at least $4p$ such that the first and second occurrences of U lie entirely within R_1 and R_2, respectively.*

In what follows, we show how to compute $\mathsf{sq}(R_1,R_2)$, which is a superset of the (i,j)-squares of length at least k for $j \in A$ since $4p \le 4k/2 \le 2k$. We obtain a constant number of arithmetic progressions that represent all such squares. Let us start with an example that captures the structure of $\mathsf{sq}(R_1,R_2)$.

▶ **Example 10.** Consider string $(\mathtt{baa})^4\mathtt{a}(\mathtt{baa})^3$. There are two runs with period $p = 3$, namely $R_1 = S[1\,..\,12]$ and $R_2 = S[12\,..\,22]$. See Figure 3 for an illustration and for the squares that satisfy the condition of Case 2b. One can see that we can get $\Omega(n)$ such squares for a string of length $\mathcal{O}(n)$, by extending this paradigm and considering string $(\mathtt{baa})^n\mathtt{a}(\mathtt{baa})^n$. This example shows that a single substitution can create/destroy $\Omega(n)$ squares; think of first setting $S[n+1] := \mathtt{c}$ and then $S[n+1] := \mathtt{a}$.

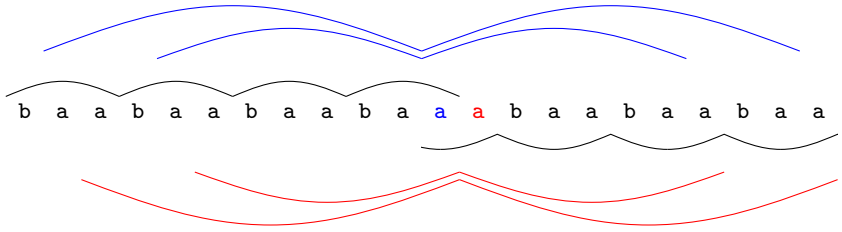

Figure 3 The two runs with period 3 are represented by black. The squares UU of length at least $4p$, such that the two occurrences of U are fully contained in the two runs are shown in red and blue, partitioned with respect to the first letter of the second occurrence of U.

▷ Claim 11. Let us suppose that we are given two runs $R_1 = S[s_1 \,..\, e_1]$ and $R_2 = S_2[s_2 \,..\, e_2]$, with $\mathsf{per}(R_1) = \mathsf{per}(R_2) = p$, such that $R_1[f \,..\, f + p - 1] = R_2[1 \,..\, p - 1]$ for some given $f \leq s_1 + p - 1$ and such that $s_1 \leq s_2 \leq e_1 \leq e_2$. We can compute a representation of $\mathsf{sq}(R_1, R_2)$ in $\mathcal{O}(1)$ time.

Proof. The following fact implies that Example 10 resembles the structure of the problem.

▶ **Fact 12** ([32]). *Two runs with period p cannot overlap by more than $p - 1$ positions.*

Due to the condition that the first and second occurrences of U must be fragments of runs R_1 and R_2, respectively, we have that the second occurrence of U can only start at one of the positions in $C = \{s_2, \ldots, e_1 + 1\}$, where $|C| \leq p$ by Fact 12. Let us consider some $c \in C$ and characterize all squares $S[a \,..\, b] = UU$ with $c = a + |U|$ and $|U| \geq 2p$.

$S[c - p \,..\, c - 1]$ is a rotation of $S[c \,..\, c + p - 1]$, i.e. there exists some $\delta < p$ such that $S[c - p \,..\, c - 1] = S[c + \delta \,..\, c + p - 1]S[c \,..\, c + \delta - 1]$. In particular, $\delta = s_2 - f \pmod p$.

$|U|$ must equal $t \cdot p + \delta$ in order for the two occurrences of U to start at the same offset mod p from f and s_2; this is necessary, since otherwise we would have two different rotations of $R_2[1 \,..\, p-1]$ matching, which is impossible as it would imply that $\mathsf{per}(R_2) < p$. In addition, all $|U|$'s of the form $t \cdot p + \delta$ for $t \geq 2$ and for which the two occurrences of U lie entirely within runs R_1 and R_2, respectively, define valid squares. We can thus compute all these squares in $\mathcal{O}(1)$ time and represent them as an arithmetic progression with respect to $|U|$.

▶ **Example 13** (Continued.). For position 12 of $(\mathtt{baa})^4\mathtt{a}(\mathtt{baa})^3$, the blue a in Figure 3, we have $\delta = 1$ and hence the squares UU that we obtain with this as starting position of the second occurrence of U are for $|U| = 1 + 3t$, for $t = 2, 3$.

Iterating over $c \in C$ in increasing order, we only have to (a) shift all squares by 1 position each time, and (b) identify the – at most two – shifts that yield an increment/decrement in the length of the arithmetic progression due to one more/less square being allowed after the shift. We can infer the values of c for which we must increment/decrement in $\mathcal{O}(1)$ time from the endpoints of the two runs and δ. These values, p, and the arithmetic progression for $c = s_2$ are our representation of $\mathsf{sq}(R_1, R_2)$. ◁

We can straightforwardly extract the longest (i, j)-square for $j \in A$ if it is of length at least k from this representation, and this concludes the proof of the theorem. ◀

To summarize, we spend $\tilde{\mathcal{O}}(k)$ time for the dynamic renaming and then process each of the $\mathcal{O}(n/k)$ fragments starting at positions $i = 1 \pmod k$ in time $\tilde{\mathcal{O}}(n/k)$, using Lemma 3 and Theorem 8. The overall time complexity of this algorithm is thus $\tilde{\mathcal{O}}(n^2/k^2 + k)$.

Wrap-up. By setting $m = 4k = n^{2/3}$ and combining the algorithms for $\mathsf{LS}(S) \leq m$ and $\mathsf{LS}(S) \geq m$ we obtain the following result.

▶ **Theorem 14.** Dynamic Longest Square *queries can be answered in time* $\tilde{\mathcal{O}}(n^{2/3})$*, using* $\tilde{\mathcal{O}}(n)$ *space, after an* $\tilde{\mathcal{O}}(n)$*-time preprocessing.*

4 An $n^{o(1)}$-time algorithm

Main Idea. If we manage to get rid of the $\mathcal{O}(m)$ time dedicated to renaming in the algorithm for computing long squares, we can then recursively obtain faster algorithms. This can be achieved by using our fastest $o(m)$ query-time dynamic algorithm for each updated $2m$-length fragment for the case that $\mathsf{LS}(S) \leq m$ instead of recomputing them in $\mathcal{O}(m)$ time using the static algorithm. We would then obtain a faster algorithm, and could plug this in turn for the case that $\mathsf{LS}(S) \leq m$; and so on.

Towards the goal of getting rid of renaming, we first observe that it is wasteful to keep track of the occurrences of all k-length substrings of S. It would be sufficient to keep track of the occurrences of each k-length substring that occurs at a position $i = 1 \pmod k$. This could be solved by maintaining $\mathcal{O}(n/m)$ instances of dynamic pattern matching with pattern $S[i \,..\, i + k - 1]$, for each $i = 1 \pmod k$, and text S. (Note that both the pattern and the text would have to be dynamic.) The main complication stems from the need to maintain p-clusters efficiently. To the best of our knowledge, the known pattern matching algorithms in the dynamic setting require $\Omega(\mathsf{occ})$ time to report the occ occurrences of the pattern in the text after each update, which is unsatisfactory in our case.

2-1 Dynamic Pattern Matching. A further observation, is that we can reduce the problem to an even easier one by applying the standard trick as follows. For every substring of length k occurring at a position $i = 1 \pmod k$, we maintain a dynamic pattern matching instance with every substring of length $2k$ starting at a position $i = 1 \pmod k$. Note that every possible occurrence of the k-length fragments of interest is contained in one (and at most two) of these $2k$-length fragments. At first glance, it may seem like this partition will be less efficient to maintain because now instead of $\mathcal{O}(n/k)$ instances of dynamic pattern matching we have $\mathcal{O}((n/k)^2)$ instances of 2-1 Dynamic Pattern Matching – to be formally defined soon. However, this is actually lossless, since every change in S only affects $O(n/k)$ such instances. Let us formally define the problem in scope.

> 2-1 Dynamic Pattern Matching
> Given two strings P and T with $|T| = 2|P| = 2n$, return all occurrences of P in T after each substitution operation on either of P, T.

We want to exploit the constant ratio between the lengths of the pattern and the text to obtain an efficient algorithm for 2-1 Dynamic Pattern Matching. We further reduce this problem to another, simpler one. A partition of the text T to its n-length prefix and suffix, analogously partitions any occurrence of P at some position i. Specifically, this occurrence is partitioned to the prefix $P[1 \,..\, m]$ of P, corresponding to the suffix $T[i \,..\, n]$ of $T[1 \,..\, 2n]$ and the suffix $P[m+1 \,..\, n]$ of P, corresponding to the prefix $T[n+1 \,..\, i+n-1]$ of $T[n+1 \,..\, 2n]$. Thus, if we know all the prefixes of P that are suffixes of $T[1 \,..\, n]$, we can extend each of them in order to compute all the occurrences of P in T. (This will be a bit more involved as they will be given as arithmetic progressions, see Lemma 20.) We call a prefix of P that is a suffix of T a *prefix-suffix match of* P *and* T.

4.1 Dynamic Prefix-Suffix

We now focus on the following problem.

> Dynamic Prefix-Suffix
> Given two strings P and S of the same length n, report all the prefixes of P that are suffixes of S, after each substitution operation on either of P, S.

We partition each of P and S to $\lfloor n/m \rfloor$ m-length fragments and possibly an extra shorter fragment. Specifically, we partition P to $P_1, P_2, \ldots, P_{\lceil n/m \rceil}$ with $P = P_1 P_2 \cdots P_{\lceil n/m \rceil}$ and S to $S_{\lceil n/m \rceil}, S_{\lceil n/m \rceil - 1}, \ldots, S_1$ with $S = S_{\lceil n/m \rceil} S_{\lceil n/m \rceil - 1} \cdots S_1$. Fragments $P_{\lceil n/m \rceil}$ and $S_{\lceil n/m \rceil}$ are allowed to be of length less than m.

▶ **Observation 15.** *Let X be a prefix of P that is also a suffix of S. Let $x = \lceil |X|/m \rceil$ and $r = |X| \pmod m$. Every pair of fragments (P_i, S_j) that satisfies $i + j - 1 = x$, will satisfy that the prefix of length r of P_i will be equal to the suffix of the same length of S_j. (Inspect Figure 4 for an illustration.)*

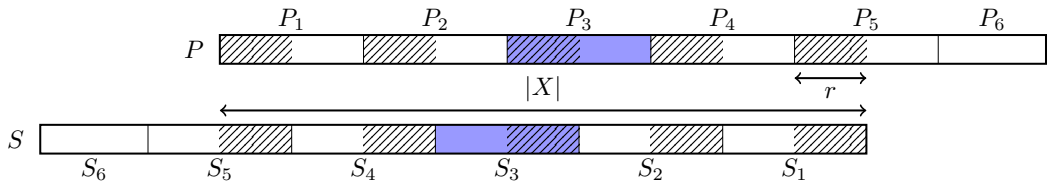

Figure 4 An illustration of the setting in Observation 15 with $x = 5$.

Algorithm. Relying on Observation 15, we design a recursive algorithm. For every $1 \leq x \leq \lceil n/m \rceil$ we will maintain an instance of Dynamic Prefix-Suffix between some pair (P_i, S_j) that satisfies $i + j - 1 = x$. Namely, for a given x, we will consider the pair $(P_{\lceil y \rceil}, S_{\lfloor y \rfloor})$, where $y = (x+1)/2$. It can be readily verified that $\lceil y \rceil + \lfloor y \rfloor - 1 = x$. Note that each P_i, S_j is in at most two of the considered pairs. Hence, each update in P or S results in no more than 2 such pairs being affected.

The prefix-suffix matches of each pair (S_i, P_j) are witnesses for possible prefix-suffix matches between P and S. All $\mathcal{O}(|S_i|) = \mathcal{O}(m)$ witnesses of a given pair can be confirmed with a logarithmic number of LCE queries, exploiting periodicity – the details are omitted due to space constraints.

We efficiently maintain the prefix-suffix matches for all relevant pairs using predecessor structures, analogously to how we maintained all starting positions of the occurrences of a substring corresponding to some name in Section 3, relying on the following lemma.

▶ **Lemma 16** (cf. [34, 6])**.** *The prefixes of a string P that are suffixes of a string S, with $|P|, |S| = \mathcal{O}(n)$, of lengths between 2^j and $2^{j+1} - 1$ form an arithmetic progression. If it has at least three elements, all these prefix-suffix matches have the same period, equal to the difference of the progression.*

Given a change, we recursively update the witnesses for the two affected pairs. At each level of the recursion, we confirm *all* witnesses. This is necessary since a witness that was not affected by the last substitution and was not an instance of a prefix-suffix match between P and S may have just become a prefix-suffix match between P and S due to the last substitution. The opposite case is possible as well.

After obtaining all the prefix-suffix matches we iterate over them to merge consistent periodic clusters as follows. For every $j \in \{1, \ldots, \lceil \log n \rceil\}$, we group the prefix-suffix matches of lengths $s \in [2^{j-1}, \ldots, 2^j - 1]$ and represent them as an arithmetic progression, relying on Lemma 16. The merging is necessary for the output of the algorithm to be in a compact form at every level of the recursion. It should also then be clear that the arithmetic progressions the algorithm returns are non-overlapping, as there is a unique such progression for the elements of length between 2^{j-1} and $2^j - 1$ for each j.

Complexity. The time complexity is $T(n) = 2T(m) + \tilde{\mathcal{O}}(\lceil n/m \rceil) = 2T(\lfloor n/k \rfloor) + \tilde{\mathcal{O}}(k)$ for $k = \lceil n/m \rceil$. $2T(m)$ for updating the two affected pairs and $\tilde{\mathcal{O}}(n/m)$ for confirming and merging all witnesses. We omit the proof of the following fact.

▶ **Fact 17.** *If $T(n) = 2T(\lfloor n/k \rfloor) + c_1 k \log^{c_2} k$ for all $n \geq N_0$, where $k = \left\lceil 2^{\sqrt{\log n}} \right\rceil$, c_1, c_2 are constants, and $T(C) = \mathcal{O}(1)$ for all $C = \mathcal{O}(1)$, then $T(n) = n^{o(1)}$.*

We arrive at the following theorem for Dynamic Prefix-Suffix.

▶ **Theorem 18.** *A representation of all prefix-suffix matches as $\mathcal{O}(\log n)$ arithmetic progressions of their ending positions in P can be maintained with $n^{o(1)}$ time per substitution.*

By maintaining a Dynamic Prefix-Suffix instance for $S = P$ we obtain the following corollary, as prefix-suffix matches correspond to borders of S.

▶ **Corollary 19.** *The period of a string $|S|$ can be maintained with $|S|^{o(1)}$ time per substitution.*

4.2 Wrap-up and complexity

The proof of the following lemma, which uses Theorem 18 as a black box, is omitted due to space constraints.

▶ **Lemma 20.** 2-1 Dynamic Pattern Matching *can be solved with $n^{o(1)}$ time per substitution, reporting the starting positions of all occurrences of P in T as an arithmetic progression.*

For the computation of long squares, after each substitution we proceed as follows.

1. We update each of the $\mathcal{O}(n/k)$ affected 2-1 Dynamic Pattern Matching instances in $f(k) = k^{o(1)}$ time, employing Lemma 20.
2. We apply Lemma 3 and Theorem 8 a total of $\mathcal{O}(n^2/k^2)$ times.

The $\tilde{\mathcal{O}}(n^2/k^2)$ term dominates the time complexity if $n/k \geq f(k)$. We note that f is an increasing function and hence it suffices to have $k \leq n/f(n)$.

Let us express the complexity of our best algorithm for Dynamic Longest Square as $n^\alpha f(n) \log^\beta n$, for $\alpha < 1$ with $n^\alpha \geq (f(n))^2$ and for β being the maximum of the powers of $\log n$ hidden by the $\tilde{\mathcal{O}}(\cdot)$ notation in Lemma 3 and Theorems 8 and 14. (Thus Theorem 14 shows an $\mathcal{O}(n^{2/3} \log^\beta n)$-time algorithm.) Then, for $k = (n^2/f(n))^{1/(\alpha+2)}$, we have that

$$\mathcal{O}(n^2/k^2 \log^\beta n + k^\alpha f(k) \log^\beta n) = \tilde{\mathcal{O}}(n^{2\alpha/(\alpha+2)}((f(n))^{2/\alpha+2} + (f(n))^{\alpha+1/\alpha+2}) \log^\beta n) = \tilde{\mathcal{O}}(n^{2\alpha/(\alpha+2)} f(n) \log^\beta n).$$

Note that this k satisfies the condition $k \leq n/f(n)$, since

$$k = (n^2/f(n))^{1/(\alpha+2)} \leq n/f(n) \iff f(n) \leq n^{\alpha/(\alpha+1)},$$

and the latter holds due to our assumptions on the value of α.

One can show by induction that $g^{(2t)}(1) = 1/t$ for $g(x) = 2x/(2+x)$; note that $g(1) = 2/3$. We can thus construct an algorithm requiring time arbitrarily close to $\tilde{\mathcal{O}}((f(n))^3) = n^{o(1)}$ time per update, recursively, since we can obtain an $\tilde{\mathcal{O}}(n^{g^{t+1}(1)} f(n))$-time algorithm by using the $\tilde{\mathcal{O}}(n^{g^t(1)} f(n))$-time algorithm for short squares. We thus arrive at the following result.

▶ **Theorem 21.** Dynamic Longest Square *queries can be answered in time* $n^{o(1)}$*, using* $\tilde{\mathcal{O}}(n)$ *space, after an* $\tilde{\mathcal{O}}(n)$*-time preprocessing.*

5 Maintaining all runs and applications

In this section, we first discuss how to modify the algorithm to maintain all runs instead of computing the longest square. Afterwards, by adapting the solutions of [17] for the static setting, we show several types of queries that can be answered with our representation of runs. In particular, we show how to maintain the number of all k-powers in $n^{o(1)}$ time and report the longest k-power in S for some fixed k within the same time complexity. All – possibly $\Theta(n^2)$ – k-powers can be reported in a compact way in $\tilde{\mathcal{O}}(\mathsf{runs})$ time, where runs denotes the number of runs in S. Finally, we show how to answer the following queries in $\tilde{\mathcal{O}}(1)$ time: given a fragment determine if it is periodic, and, if so, compute its period.

We start by describing how to maintain all runs in the $\tilde{\mathcal{O}}(n^{2/3})$-time solution.

For short runs, we use the $\mathcal{O}(m)$-time algorithm of [35]. For each $2m$-length fragment, we only maintain runs that do not touch its endpoints, as we do not want to maintain a run that may extend to other fragments. (We only waive this restriction for runs that are suffixes/prefixes of S and are of length smaller than m.) This is sufficient as every run R such that $|R| < m$ will be fully contained in one (and at most two) of the $2m$-length fragments. Upon a substitution we just recompute the runs for the two affected fragments.

As for runs of length at least $m = 4k$, we recompute all of them. Let us first amend Observation 2 as follows.

▶ **Lemma 22.** *A run* $R = S[a\,..\,b]$*, of length at least* $4k$*, contains a fragment* $S[i\,..\,i+k-1]$*, for some* $i = 1 \pmod k$*, that also occurs at position* $i + \mathsf{per}(R)$*.*

Proof. The first $2k$-length fragment of the run must appear again somewhere in the run (otherwise it is not even a square). This fragment, being of length $2k$, must contain a fragment $S[i\,..\,i+k-1]$ with $i = 0 \pmod k$ and $i \le a+k-1$. $S[i\,..\,i+k-1]$ will certainly occur at position $i + \mathsf{per}(R)$, since $i + \mathsf{per}(R) + k - 1 \le a + k - 1 + |R|/2 + k - 1 \le a + |R|/4 - 1 + |R|/2 + |R|/4 - 1 \le b$. ◀

We then proceed as in Section 3. We first define the *(i,j)-run* to be the unique run R containing $S[i\,..\,j]$, in which the difference between i and j is consistent with the period of the run; formally, $j = i \pmod p$, where $p = \mathsf{per}(R)$. Now observe that every run R that is longer than $4k$ is an (i,j)-run for some $i = 0 \pmod k$ and some j for which $S[i\,..\,i+k] = S[j\,..\,j+k]$ due to Lemma 22; in particular, the smallest such j is $j = i + \mathsf{per}(R)$.

Observation 4 can be modified analogously as follows.

▶ **Observation 23.** *An* (i,j)*-run* R*, exists if and only if* $\mathsf{lcs}(i,j) + \mathsf{lcp}(i,j) \ge |j-i|+1$*. If* R *exists it is* $S[i - \mathsf{lcs}(i,j) + 1\,..\,j + \mathsf{lcp}(i,j) - 1]$*.*

The above observation allows us to efficiently process names with less than $3n/k$ occurrences in S. As for names corresponding to k-length substrings with more occurrences, the proof of Theorem 8 shows that we can process the k-length substring $S[i\,..\,i+k-1]$ with a

p-cluster A of its occurrences as follows. If $i \in A$ we are done as we simply report the run with period p corresponding to A. Otherwise, using the notation of the proof of Theorem 8, we compute the (e_1, e_2)-run and the (s_1, s_2)-run if we are in Case 1 or 2a.

Note that a run $S[a\,..\,b]$ may be identified multiple times. We remove duplicates and store $S[a\,..\,b]$ as (a, b, p), where p is the minimum $j - i$ for which this run was obtained as the (i, j)-run; by Lemma 22 we have that $p = \mathsf{per}(S[a\,..\,b])$.

Case 2b is again more tricky. We adapt our solution for squares (see Theorem 8) in order to compute and maintain such runs compactly using arithmetic progressions.

▶ **Observation 24.** *All elements of $\mathsf{sq}(R_1, R_2)$ of equal length are extended by the same run.*

We denote the elements of $\mathsf{sq}(R_1, R_2)$ of length $2|U|$ by $G_{|U|}$.

▶ **Lemma 25.** *If the minimum starting position among the elements of $G_{|U|}$ is $u > s_1$ and the maximum ending position is $v < e_2$, then the run extending the elements of $G_{|U|}$ is $R = S[u\,..\,v]$ and $\mathsf{per}(R) = |U|$.*

Proof. We have $u + |U| = s_2$ since $u > s_1$. If the run extending the squares of $G_{|U|}$ started at some position smaller than u, this would imply $S[u-1\,..\,s_2-2] = S[s_2-1\,..\,s_2+|U|-2]$, which in turn would imply that the right hand side of the equation is a string with period p. This would contradict R_2 being a run. The argument for the other side is symmetric.

As for arguing that $\mathsf{per}(R) = |U|$, let us assume for the sake of contradiction that it has a period $q < |U|$. Then, as $|U|$ is also a period of R, the periodicity lemma implies that $q' = \gcd(|U|, q) \leq |U|/2$ is also a period of R.

We can apply the periodicity lemma again, since p is also a period of U and $p + q' \leq |U|$. We then have that $p' = \gcd(p, q') < |p|/2$ is a period of $S[s_2\,..\,s_2 + |U| - 1]$ and a divisor of p. This is a contradiction as $S[s_2\,..\,s_2 + p - 1]$ is a primitive string, i.e. is not of the form T^k for a string T and $k > 1$, since otherwise $p' < p$ would also be a period of R_2. ◀

We maintain all such runs $\mathsf{runs}(R_1, R_2)$ compactly as a constant number of arithmetic progressions with respect to $|U|$; one for each of the at most three distinct group sizes.

Only two groups of squares may contain a square that starts in the first position of R_1 or ends in the last position of R_2. These groups of squares are the only ones such that the run extending them may not be fully contained in $S[s_1\,..\,e_2]$. This could be the case for example if we had a run R_3 with the same period and appropriate overlap with R_2. We compute the run extending a square of each of the at most two relevant groups using LCE queries and maintain these runs explicitly.

All (explicitly or compactly represented) runs are stored in a way that allows for efficient deletion, using a key with respect to their origin, i.e. the substring at some level of the recursion for which they were computed. After each substitution, the algorithm computes a all the relevant runs for the substring that contains the updated position at each level of the recursion from scratch. Thus, for every substring for which we recompute runs, we first delete all runs that have this substring as key.

▶ **Theorem 26.** *We can maintain all runs $R = S[a\,..\,b]$ of a string of length n, as triplets $(a, b, \mathsf{per}(R))$ and arithmetic progressions with $n^{o(1)}$ time per operation, using $\tilde{\mathcal{O}}(n)$ space, after $\tilde{\mathcal{O}}(n)$-time preprocessing.*

5.1 Application I: k-powers

Let us recall that a k-power is a string of the form U^k for some non-empty string U. The authors of [17] show that given a run R as (a, b, p), one can compute in $\mathcal{O}(1)$ time:

1. a longest substring U^k of R with $\mathsf{per}(R) \mid |U|$;
2. the number of all fragments $S[a\,..\,b] = V^k$, with $\mathsf{per}(R) \mid |V|$, that lie entirely within R.

For any fixed $k \geq 3$, we can maintain the longest k-power by storing a heap keeping the longest that each explicitly stored run contributes and maintain the count on the number of k-powers (not distinct) in S within the time complexities of Theorem 26. Note that by Lemma 25 and the fact that $v - u \leq 2|U| + p$ and $|U| \geq 2p$, where u, v, p and $|U|$ are as in the statement of that lemma, we have that the runs stored as arithmetic progressions do not contribute any k-powers for $k \geq 3$. Finally, we can extract all – possibly $\Theta(n^2)$ – non-distinct k-powers in $\tilde{\mathcal{O}}(\mathsf{runs})$ time in a compact form from the runs [17].

▶ Remark 27. As for maintaining the $\mathcal{O}(n)$ distinct k-powers efficiently, we should first be able to group runs by their Lyndon roots (the Lyndon root of a run R is the lexicographically smallest rotation of a $\mathsf{per}(R)$-length substring of R). It is not clear how to amend our solution to maintain the runs in this way.

5.2 Application II: 2-Period Queries

2-Period Queries Given a fragment $S[i\,..\,j]$ of S, decide whether $S[i\,..\,j]$ is periodic and, if so, compute its period.

2-Period Queries in a static string. 2-Period Queries were introduced in [17], while general internal period queries were intoduced in [33]. The authors of [34] showed how to optimally answer 2-Period Queries in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$-time preprocessing. In these works, it is shown, that in order to answer the query for $S[i\,..\,j]$ it suffices to find the run R that extends $S[i\,..\,j]$, or conclude that there is no such run. In other words, it suffices to find the run R with the smallest period among the runs fully containing $S[i\,..\,j]$. Then, if $\mathsf{per}(R) < (j - i)/2$, the fragment is periodic with period $\mathsf{per}(R)$ and otherwise it is not.

To the best of our knowledge there is no prior work on answering internal period queries in a dynamic string. In what follows we sketch the proof of the following result – the details are omitted due to space constraints.

▶ **Theorem 28.** 2-Period Queries *can be answered in $\tilde{\mathcal{O}}(1)$ time in a string S of length n, with each substitution operation processed in time $n^{o(1)}$, after an $\tilde{\mathcal{O}}(n)$-time preprocessing. The required space is $\tilde{\mathcal{O}}(n)$.*

In order to compute the run with the smallest period that contains $S[i\,..\,j]$, the authors of [17] show that it is enough to be able to answer orthogonal range minimum queries in 2-d, over the following collection of points: for each run (a, b, p) we have point (a, b) with weight p. The desired run then corresponds to the point with minimum weight in the rectangle $[1, i] \times [j, n]$. A restricted version of the main result of [14], is that one can maintain a collection of $\mathcal{O}(n)$ points in $[n]^d$, for any constant d, with $\tilde{\mathcal{O}}(1)$ time per update, such that orthogonal range emptiness queries can be answered in $\tilde{\mathcal{O}}(1)$ time. We note that 2-d orthogonal range minimum queries reduce to 3-d orthogonal range emptiness queries via binary search. We maintain this data structure over the runs that are maintained explicitly, see Theorem 26. The above discussion covers the case that the run extending $S[i\,..\,j]$ has

been stored explicitly. In particular, following our discussion in Section 5.1, we are already able to answer 3-Period Queries, i.e. whether a substring $S[i\,..\,j]$ has period at most $|j-i|/3$, and if so, return this period.

As for 2-Period Queries, we now provide the intuition for handling the case that the run of minimum period that contains $S[i\,..\,j]$ is stored implicitly. We want to check all runs extending some square $UU \in \mathsf{sq}(R_1, R_2)$ that is a prefix of $S[i\,..\,j]$, for some runs R_1, R_2. Note that our definition of $\mathsf{sq}(R_1, R_2)$ implies that $\mathsf{per}(U) = \mathsf{per}(R_1) \leq |U|/2$; i.e. R_1 extends U. The following lemma implies that we can only have a logarithmic number of such squares.

▶ **Lemma 29** ([32, Corollary 5.1.3]). *Let U_1, U_2, U_3 be periodic fragments of a text T, all starting at the same position, and being extended by runs R_1, R_2 and R_3, respectively. If $\lfloor \log |U_1| \rfloor = \lfloor \log |U_2| \rfloor = \lfloor \log |U_3| \rfloor$, then the three runs R_1, R_2 and R_3 cannot be all distinct.*

For every set $\mathsf{runs}(R_1, R_2)$, we add the point (s_1, e_2, p) in an initially empty 3-d grid – we use the same notation as above. We report all relevant points using 3-d dynamic orthogonal range reporting queries, again employing [14]. In particular, we first retrieve the points in the range $[1, i] \times [j, n] \times [1, |j-i|/4]$. There are $\mathcal{O}(\log n)$ of them due to the above lemma. Then, for each point, corresponding say to $\mathsf{runs}(R_1, R_2)$, we compute in $\tilde{\mathcal{O}}(1)$ time the run of smallest period in $\mathsf{runs}(R_1, R_2)$ containing $S[i\,..\,j]$. In particular it is the run of minimum length in $\mathsf{runs}(R_1, R_2)$ containing $S[i\,..\,j]$ by Lemma 25.

6 Concluding remarks

We believe that, with due care, our algorithm can be adapted to handle insertions and deletions – the details are omitted due to space constraints. We leave open the questions of whether the runs of a string (or other information sufficient for answering 2-Period Queries in $\tilde{\mathcal{O}}(1)$ time) can be maintained with $\tilde{\mathcal{O}}(1)$ time per update and whether period queries for aperiodic substrings can be answered efficiently in a dynamic string.

References

1. Paniz Abedin, Sahar Hooshmand, Arnab Ganguly, and Sharma V. Thankachan. The Heaviest Induced Ancestors Problem Revisited. In *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China*, pages 20:1–20:13, 2018. doi:10.4230/LIPIcs.CPM.2018.20.
2. Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. New Data Structures for Orthogonal Range Searching. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 198–207. IEEE Computer Society, 2000. doi:10.1109/SFCS.2000.892088.
3. Amihood Amir and Itai Boneh. Locally Maximal Common Factors as a Tool for Efficient Dynamic String Algorithms. In *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China*, pages 11:1–11:13, 2018. doi:10.4230/LIPIcs.CPM.2018.11.
4. Amihood Amir and Itai Boneh. Dynamic Palindrome Detection. *CoRR*, abs/1906.09732, 2019. arXiv:1906.09732.
5. Amihood Amir, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest Common Factor After One Edit Operation. In Gabriele Fici, Marinella Sciortino, and Rossano Venturini, editors, *String Processing and Information Retrieval: 24th International Symposium, SPIRE 2017, Proceedings*, volume 10508 of *Lecture Notes in Computer Science*, pages 14–26, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-67428-5_2.

6 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest Common Substring Made Fully Dynamic. In Ola Svensson Michael A. Bender and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA 2019, Munich/Garching, Germany, September 9-11, 2019*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

7 Amihood Amir and Martin Farach. Adaptive Dictionary Matching. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 760–766. IEEE Computer Society, 1991. doi:10.1109/SFCS.1991.185445.

8 Amihood Amir, Martin Farach, Ramana M. Idury, Johannes A. La Poutré, and Alejandro A. Schäffer. Improved Dynamic Dictionary Matching. *Inf. Comput.*, 119(2):258–282, 1995. doi:10.1006/inco.1995.1090.

9 Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Trans. Algorithms*, 3(2):19, 2007. doi:10.1145/1240233.1240242.

10 Alberto Apostolico and Franco P. Preparata. Optimal Off-Line Detection of Repetitions in a String. *Theor. Comput. Sci.*, 22:297–315, 1983. doi:10.1016/0304-3975(83)90109-3.

11 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "Runs" Theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.

12 Gary Benson. Tandem repeats finder: a program to analyze DNA sequences. *Nucleic Acids Research*, 27(2):573–580, January 1999. doi:10.1093/nar/27.2.573.

13 Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj, and Søren Vind. Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation. *Algorithmica*, 80(11):3207–3224, 2018. doi:10.1007/s00453-017-0380-7.

14 Timothy M. Chan and Konstantinos Tsakalidis. Dynamic Orthogonal Range Searching on the RAM, Revisited. In Boris Aronov and Matthew J. Katz, editors, *33rd International Symposium on Computational Geometry, SoCG 2017, July 4-7, 2017, Brisbane, Australia*, volume 77 of *LIPIcs*, pages 28:1–28:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.SoCG.2017.28.

15 Maxime Crochemore. An Optimal Algorithm for Computing the Repetitions in a Word. *Inf. Process. Lett.*, 12(5):244–250, 1981. doi:10.1016/0020-0190(81)90024-7.

16 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

17 Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Extracting powers and periods in a word from its runs structure. *Theor. Comput. Sci.*, 521:29–41, 2014. doi:10.1016/j.tcs.2013.11.018.

18 Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Dynamic Graph Algorithms. In Mikhail J. Atallah and Marina Blanton, editors, *Algorithms and Theory of Computation Handbook: General Concepts and Techniques*, chapter 9. Chapman & Hall/CRC, 2010. URL: http://dl.acm.org/citation.cfm?id=1882757.1882766.

19 Nevzat Onur Domaniç and Franco P. Preparata. A Novel Approach to the Detection of Genomic Approximate Tandem Repeats in the Levenshtein Metric. *Journal of Computational Biology*, 14(7):873–891, 2007. doi:10.1089/cmb.2007.0018.

20 Paolo Ferragina. Dynamic Text Indexing under String Updates. *J. Algorithms*, 22(2):296–328, 1997. doi:10.1006/jagm.1996.0814.

21 Paolo Ferragina and Fabrizio Luccio. Dynamic Dictionary Matching in External Memory. *Inf. Comput.*, 146(2):85–99, 1998. doi:10.1006/inco.1998.2733.

22 N. J. Fine and H. S. Wilf. Uniqueness Theorems for Periodic Functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. URL: http://www.jstor.org/stable/2034009.

23 Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Longest substring palindrome after edit. In *Annual Symposium on Combinatorial Pattern Matching, CPM 2018*, pages 12:1–12:14, 2018. doi:10.4230/LIPIcs.CPM.2018.12.

24 Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster Queries for Longest Substring Palindrome After Block Edit. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPIcs*, pages 27:1–27:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.CPM.2019.27.

25 Pawel Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. Optimal Dynamic Strings. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1509–1528. SIAM, 2018. doi:10.1137/1.9781611975031.99.

26 Ming Gu, Martin Farach, and Richard Beigel. An Efficient Algorithm for Dynamic Text Indexing. In Daniel Dominic Sleator, editor, *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. 23-25 January 1994, Arlington, Virginia.*, pages 697–704. ACM/SIAM, 1994. URL: http://dl.acm.org/citation.cfm?id=314464.314675.

27 Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, 2004.

28 Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, 2004.

29 Ramana M. Idury and Alejandro A. Schäffer. Dynamic Dictionary Matching with Failure Functions. *Theor. Comput. Sci.*, 131(2):295–310, 1994. doi:10.1016/0304-3975(94)90176-7.

30 Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. Rapid Identification of Repeated Patterns in Strings, Trees and Arrays. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing, May 1-3, 1972, Denver, Colorado, USA*, pages 125–136, 1972. doi:10.1145/800152.804905.

31 Richard M. Karp and Michael O. Rabin. Efficient Randomized Pattern-Matching Algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.

32 Tomasz Kociumaka. *Efficient Data Structures for Internal Queries in Texts.* PhD thesis, University of Warsaw, 2018. URL: https://mimuw.edu.pl/~kociumaka/files/phd.pdf.

33 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Efficient Data Structures for the Factor Periodicity Problem. In Liliana Calderón-Benavides, Cristina N. González-Caro, Edgar Chávez, and Nivio Ziviani, editors, *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings*, volume 7608 of *Lecture Notes in Computer Science*, pages 284–294. Springer, 2012. doi:10.1007/978-3-642-34109-0_30.

34 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal Pattern Matching Queries in a Text and Applications. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.

35 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. doi:10.1109/SFFCS.1999.814634.

36 Gad M. Landau, Jeanette P. Schmidt, and Dina Sokol. An Algorithm for Approximate Tandem Repeats. *Journal of Computational Biology*, 8(1):1–18, 2001. doi:10.1089/106652701300099038.

37 Michael G. Main and Richard J. Lorentz. An $O(n \log n)$ Algorithm for Finding All Repetitions in a String. *J. Algorithms*, 5(3):422–432, 1984. doi:10.1016/0196-6774(84)90021-X.

38 Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining Dynamic Sequences under Equality Tests in Polylogarithmic Time. *Algorithmica*, 17(2):183–198, 1997. doi:10.1007/BF02522825.

39 Süleyman Cenk Sahinalp and Uzi Vishkin. Efficient Approximate and Dynamic Matching of Patterns Using a Labeling Paradigm (extended abstract). In *37th Annual Symposium on Foundations of Computer Science, FOCS '96, Burlington, Vermont, USA, 14-16 October, 1996*, pages 320–328. IEEE Computer Society, 1996. doi:10.1109/SFCS.1996.548491.

40 Dina Sokol, Gary Benson, and Justin Tojeira. Tandem repeats over the edit distance. *Bioinformatics*, 23(2):30–35, 2007. doi:10.1093/bioinformatics/btl309.

41 Ydo Wexler, Zohar Yakhini, Yechezkel Kashi, and Dan Geiger. Finding Approximate Tandem Repeats in Genomic Sequences. *Journal of Computational Biology*, 12(7):928–942, 2005. doi:10.1089/cmb.2005.12.928.