The streaming k-mismatch problem

Raphaël Clifford[1], Tomasz Kociumaka[2], and Ely Porat[3]

[1]Department of Computer Science, University of Bristol, United Kingdom
raphael.clifford@bristol.ac.uk
[2]Institute of Informatics, University of Warsaw, Poland
kociumaka@mimuw.edu.pl
[3]Department of Computer Science, Bar-Ilan University, Israel
porately@cs.biu.ac.il

Abstract

We consider the problem of approximate pattern matching in a stream. In the streaming k-mismatch problem, we must compute all Hamming distances between a pattern of length n and successive n-length substrings of a longer text, as long as the Hamming distance is at most k. The twin challenges of streaming pattern matching derive from the need both to achieve small and typically sublinear working space and also to have fast guaranteed running time for every arriving symbol of the text.

We present an $\mathcal{O}(\log \frac{n}{k}(\sqrt{k \log k} + \log^3 n))$-time algorithm for the k-mismatch streaming problem which uses only $\mathcal{O}(k \log n \log \frac{n}{k})$ bits of space. The running time almost matches the fastest known offline algorithm and the space usage is within a logarithmic factor of optimal. Our new algorithm therefore effectively resolves an open problem first posed in FOCS '09 [32]. On route to this solution we also give a deterministic $\mathcal{O}(k(\log \frac{n}{k} + \log |\Sigma|))$-bit encoding of all the alignments with Hamming distance at most k between a pattern and a text of length $\mathcal{O}(n)$. The input alphabet is denoted by the symbol Σ. This secondary result provides an optimal solution to a natural communication complexity problem which may be of independent interest.

1 Introduction

Combinatorial pattern matching has formed a cornerstone of both the theory and practice of algorithm design over a number of decades. Despite this long history, there are still a number of basic questions that remain unanswered particularly in the context of small space streaming algorithms. Recently there has been renewed interest in one of the simplest and most fundamental pattern matching problems, that of computing the Hamming distance between a pattern and a longer text.

The problem of computing the exact Hamming distance between a pattern and every n-length substring of a text of length $\mathcal{O}(n)$ has been studied in the standard offline model for at least 30 years. In 1987 $\mathcal{O}(n\sqrt{n \log n})$-time solutions were first developed [1, 28]. Motivated by the need to find close matches quickly, from there the focus moved to the bounded k-mismatch version of the problem. The k-mismatch problem asks us to report all alignments of a pattern P of length n within a longer text T, wherever the Hamming distance is at most k. It was not until 2004 when the current record was given running in $\mathcal{O}(n\sqrt{k \log k})$ time [2].

Considered as an online or streaming problem with one text symbol arriving at a time, a linear space solution running in $\mathcal{O}(\sqrt{k} \log k + \log n)$ worst case time per arriving symbol was later shown in 2010 [14]. As this running time is very close to the fastest known offline algorithm, the main remaining challenge was to reduce the space usage. The breakthrough paper of Porat and Porat in FOCS '09 gave an $\mathcal{O}(k^3 \operatorname{polylog} n)$-space and $\mathcal{O}(k^2 \operatorname{polylog} n)$-time streaming solution, showing for the first time that the k-mismatch problem could be solved in sublinear space for particular ranges of k [32]. In SODA '16 this was subsequently improved to $\mathcal{O}(k^2 \operatorname{polylog} n)$ space and $\mathcal{O}(\sqrt{k} \log k + \operatorname{polylog} n)$ time per arriving symbol [13].

One can derive a space lower bound for any streaming problem by looking at a related one-way communication complexity problem. The randomised one-way communication complexity of determining if the Hamming distance between two n bits strings is greater than k is known to be $\Omega(k)$ bits with an upper bound of $\mathcal{O}(k \log k)$ [22]. In our problem formulation, however, we report not only the Hamming distance but also the full list of mismatched pairs of symbols and their indices. In this situation one can derive a slightly higher bound of $\Omega(k(\log \frac{n}{k} + \log |\Sigma|))$ bits. This follows directly from the observation that for a single alignment with Hamming distance k, there are $\binom{n}{k}$ different possible sets of mismatch indices and each of the symbols in the set of k mismatches requires $\Omega(\log |\Sigma|)$ bits to be represented, where Σ denotes the input alphabet. From this we can derive the same lower bound for the space required by any streaming k-mismatch algorithm. Prior to the work we present here, this simple lower bound for a single output combined with the $\mathcal{O}(k^2 \operatorname{polylog} n)$ upper bound from SODA'16 represented the limits of our understanding of the space complexity of this basic problem.

In this paper we almost completely resolve both the time and space complexity of the streaming k-mismatch pattern matching problem.

Problem 1.1. *Consider a pattern of length n and a text which arrives in a stream, one symbol at a time. The streaming k-mismatch problems asks after each arriving symbol whether the current suffix of the text has Hamming distance at most k with the pattern.*

Our main result is an algorithm for the streaming k-mismatch problem which almost matches the running time of the fastest known offline algorithm while using nearly optimal working space. Unlike the previous work of [32, 13] the algorithm we describe also allows us to report the full set of mismatched symbols (and their indices) at each k-mismatch alignment. This gives a remarkable resolution to the complexity of the streaming k-mismatch problem first posed by Porat and Porat in FOCS '09 [32].

Let us define the *mismatch information* at an alignment of the pattern and text to be the set of mismatch indices at this alignment as well as the list of pairs of mismatched symbols.

Theorem 1.2. *There exists a solution for Problem 1.1 which, after preprocessing the pattern, uses $\mathcal{O}(\log \frac{n}{k}(\sqrt{k \log k} + \log^3 n))$ time per arriving symbol and $\mathcal{O}(k \log n \log \frac{n}{k})$ bits of space. The algorithm is randomised and its answers are correct with high probability, that is it errs with probability inverse polynomial in n. For each reported occurrence, the mismatch information can be reported on demand in $\mathcal{O}(k)$ time.*

An important feature of Theorem 1.2 is that we must account for all the space used and cannot, for example, store a copy of the pattern after the preprocessing stage. The streaming model we use is the same as the one from the original streaming pattern matching paper of Porat and Porat, where we first preprocess the pattern offline before the text symbols arrive [32].

In order to achieve our time and space improvements we develop a number of new ideas and techniques. The first is a randomised $\mathcal{O}(k \log n)$-bit rolling sketch which allows us not only to detect if two strings of the same length have Hamming distance at most k but if they do, also to report all the related mismatch information. The sketch we give will have a number of desirable arithmetic properties, including the ability efficiently to maintain the sketch of a sliding window.

Armed with such a rolling sketch, one approach to the k-mismatch streaming problem could simply be to maintain the sketch of the length-n suffix of the text, and to compare it to the sketch of the whole pattern. Although this takes $\mathcal{O}(k \operatorname{polylog} n)$ time per arriving symbol, it would also require $\mathcal{O}(n \log |\Sigma|)$ bits of space to retrieve the leftmost symbol that has to be removed from the sketch at each new alignment. The central challenge therefore becomes how to maintain such a sliding window using small space and in running time that is close as possible to the offline algorithm.

Before explaining our solution to the k-mismatch streaming problem in more detail, we set out a natural communication problem which is of independent interest but which also contains within it a key element needed for our fast and space efficient streaming algorithm.

Problem 1.3. *Alice has a pattern P of length n and a text T of length $\mathcal{O}(n)$. She sends one message to Bob, who holds neither the pattern nor text. Bob must output all the alignments of the pattern and the text with at most k mismatches, as well as the applicable mismatch information.*

The solution we give for this problem is both deterministic and optimal.

Theorem 1.4. *There exists an deterministic one-way communication complexity protocol for Problem 1.3 that sends $\mathcal{O}(k(\log \frac{n}{k} + \log |\Sigma|))$ bits, where Σ denotes the input alphabet.*

One striking property of this result is that the communication complexity upper bound matches the lower bound we gave earlier for two strings of exactly the same length. That is, we require no more information to report all the mismatch information at all k-mismatch alignments than we do to report it at only one such alignment.

Our communication protocol uses the concept of a $2k$-period of a string, defined as the index of any alignment of a string against itself with Hamming distance at most $2k$ [13]. A particularly important property of the $2k$-periods is that the distance between any overlapping k-mismatch occurrences of the pattern must be a $2k$-period of the pattern. If d is the greatest common divisor of all $2k$-periods of the pattern, then we therefore know that all the k-mismatch occurrences must be a multiple of d positions apart and only these locations have to be checked for a potential k-mismatch occurrence. Our first key technical innovation, summarised in Lemma 4.2, is a new space and time efficient data structure that encodes the differences between the subsequent length

d blocks of the pattern. We go on to show that if two k-mismatch occurrences are sufficiently close to each other, then, combined with our new data structure, this is sufficient information to report all the k-mismatch alignments between the two occurrences. Bob can use these two facts to find all the k-mismatch occurrence of the pattern in the text along with the mismatch information. Our solution to the streaming k-mismatch problem will the techniques developed for this communication protocol, and Lemma 4.2 in particular, as a key ingredient.

For our streaming algorithm and the proof of our main result in Theorem 1.2 we consider two cases. The first is where at least one of the $2k$-periods of the pattern P is small. In this case we give a small space encoding the input strings and develop a new algorithm for convolution on sparse strings which can be applied to these encodings. As a result we show that it is possible to develop an algorithm that runs in $\mathcal{O}(\sqrt{k \log k})$ time for this small approximate period case. This result is given by Theorem 5.6.

In the second case, where all the $2k$-periods of the pattern P are large our approach will require considerably more sophistication. The information the method computes includes sketches of $\mathcal{O}(\log \frac{n}{k})$ prefixes of the pattern and a constant number of k-mismatch occurrences of each prefix along with the associated mismatch information. This is done in such a way that we are always able to compute the sketch of an n-length suffix of the text when needed. To do this we use the information we have computed so far to infer the sketches of the substrings of the text which are to be removed from the left end of the sliding window as a new symbol arrives and the sketch is rolled forwards. Our proof will make crucial use of the data structure given by Lemma 4.2 and the new small approximate period streaming algorithm as well as arithmetic properties of the rolling k-mismatch sketch.

As a final step we deamortise the full algorithm in order to give guaranteed worst case running time per arriving symbol.

1.1 A conjectured space lower bound

The space upper bound we give for our streaming algorithm, although close to optimal, is still an $\mathcal{O}(\log n)$-factor away from the known lower bound. As a final contribution we give a higher conjectured space lower bound for Problem 1.1, which partially closes this gap. We do this by observing that a particularly natural way to tackle the streaming problem is first to encode the k-mismatch alignments and mismatch information for all prefixes of the pattern against all suffixes of a substring of the text of the same length and then use this information to start processing new symbols as they arrive. Our streaming algorithm, for example, effectively does exactly this, as do the earlier streaming k-mismatch algorithms of [32, 13] and the exact matching streaming algorithms of [32, 3]. In Section 9 we show a space lower bound for Problem 1.1 for any streaming algorithm that takes this approach. We further conjecture that this lower bound is in fact tight in general.

Conjecture 1.5. *Any solution for Problem 1.1 must use at least* $\Omega(k \log \frac{n}{k}(\log \frac{n}{k} + \log |\Sigma|))$ *bits of space.*

We will argue in favour of this conjecture by giving an explicit set of patterns for which encoding the mismatch information for all alignments of the pattern against itself will require the stated number of bits. If the text includes a copy of the pattern as a substring, then the result follows. It is interesting to note that a similar conjecture can be made for the space complexity of exact pattern matching in a stream. In this case encoding the alignments of all exact matches between the prefixes and suffixes of a pattern is known to require $\Omega(\log^2 n)$ bits, matching the best known space upper bounds of [32, 3].

2 Related work

In terms of the original offline k-mismatch pattern matching problem, faster solutions have recently been given depending on the exact relationship between the pattern length, the text length, and the bound k [13, 20]. The only previous work for the streaming k-mismatch problem which can also report the mismatch information at each k-mismatch alignment is that of Radoszewski and Starikovskaya [33]. In this paper they proposed an algorithm that uses $\mathcal{O}(k^2 \log^{10} n / \log\log n)$ bits of space and achieves $\mathcal{O}(k \log^8 n / \log\log n)$ running time per arriving symbol. This algorithm therefore solves essentially the same problem we consider but with an $\mathcal{O}(k \operatorname{polylog} n)$ increase in space and an $\mathcal{O}(\sqrt{k} \operatorname{polylog} n)$ increase in time.

Research into small-space streaming pattern matching algorithms started in earnest in 2009 with the discovery of an algorithm which uses $\mathcal{O}(\log^2 n)$ bits of space and takes $\mathcal{O}(\log n)$ time per arriving symbol for exact matching [32]. This was later slightly simplified in [18] and then improved to run in constant time per arriving symbol in [3]. Since that time, other small-space streaming algorithms have been proposed for approximate pattern matching problems. For example, in 2013 a small space streaming pattern matching algorithm was shown for parameterised matching [23] and in [15] an $\mathcal{O}(\sqrt{n} \operatorname{polylog} n/\epsilon^2)$-space streaming algorithm was given for $(1+\epsilon)$-approximate Hamming distance. Most work on the one-way communication complexity of distance measures between strings has assumed that Alice and Bob hold strings of the same length. Work in this traditional communication complexity setting has included $(1+\epsilon)$-approximation [24], the so called gap Hamming problem [7], and the k-mismatch problem [22]. However, in [9] it was shown that for a large range of the most popular pattern matching problems, including L_1, L_2, L_∞-distance and edit distance, space proportional to the pattern length is required for any randomised online algorithm.

The interesting and related problem of computing all k-periods in a stream has also been tackled in [17]. In this work they show an $\Omega(n)$ space lower bound for one-pass streaming algorithms and an $\mathcal{O}(k^4 \log^9 n)$ bit one-pass streaming algorithm to compute all k-periods of length at most $n/2$.

3 Overview of the rest of the paper

Theorem 1.4 of Section 4 gives an optimal deterministic communication protocol for Problem 1.3. This result, which is of independent interest, relies on Lemma 4.2, which also encapsulates one of the key ideas of our streaming algorithm. In particular, Lemma 4.2 shows that a string of length n can be encoded using $\mathcal{O}(d(\log \frac{n}{d} + \log|\Sigma|))$ bits so that one can retrieve all its d-periods $p \leq \frac{1}{4}n$, along with the underlying mismatch information. We will also use this encoding later on in the streaming algorithm of Section 7.

As explained earlier in the introduction, our streaming k-mismatch algorithm considers two cases. The first is when the pattern has at least one $2k$-period which is $\mathcal{O}(k)$. For this case we show in Theorem 5.6 in Section 5 that the problem can be solved in $\mathcal{O}(\sqrt{k \log k})$ time per arriving character in the worst case.

In order to tackle the second case, where the pattern has no $2k$-period of size $\mathcal{O}(k)$ we will require some further machinery. In Section 6 we show a rolling k-mismatch sketch which will be used to maintain sketches of windows of the text. In Section 7 we present the main algorithm, proving Theorem 1.2 and deriving the overall complexity of $\mathcal{O}(\log \frac{n}{k}(\sqrt{k \log k} + \log^3 n))$ worst case time per arriving character.

Finally, in Section 8 we give the full proof of Lemma 4.2.

4 A Deterministic Communication Protocol for Problem 1.3

In this section, we develop an optimal deterministic communication complexity protocol for Problem 1.3. We start with a few elementary definitions.

Recall that the *Hamming distance* of equal-length strings X and Y is defined as $\mathrm{HD}(X,Y) = |\{i : X[i] \neq Y[i]\}|$. The corresponding *mismatch information* is formally defined as $\mathrm{MI}(X,Y) = \{(i, X[i], Y[i]) : X[i] \neq Y[i]\}$. Mismatch information can be encoded compactly as stated below:

Observation 4.1. *Let $X, Y \in \Sigma^n$ be two strings with Hamming distance $h = \mathrm{HD}(X,Y)$. The mismatch information $\mathrm{MI}(X,Y)$ can be encoded in $\mathcal{O}(h(\log \frac{n}{h} + \log|\Sigma|))$ bits.*

Proof. By definition, the mismatch information consist of h tuples. Each tuple consists of two characters (which we store explicitly in $\mathcal{O}(\log|\Sigma|)$ bits each), and a position. The positions form a subset of $\{1,\ldots,n\}$ of size h. Thus, they can be encoded using $\mathcal{O}(h(1+\log\frac{n}{h}))$ bits in total. □

A string X may have overlapping exact occurrences only if it is periodic, i.e., if $X[0,\ldots,n-p-1] = X[p,\ldots,n-1]$ for some $p > 0$. The k-periods, defined below, describe analogous structure for overlapping occurrences with few mismatches.

We say that an integer p is a k-period of a string X of length n if $\mathrm{HD}(X[0,\ldots,n-p-1], X[p,\ldots,n-1]) \leq k$. The set of k-periods of X not exceeding m is denoted by $\mathrm{Per}_{\leq m}(X,k)$.

The following oracle, implemented in Lemma 4.2 lets us retrieve the mismatch information $\mathrm{MI}(X[0,\ldots,n-p-1], X[p,\ldots,n-1])$ for all short k-periods. The asymptotic size of the data structure matches the bound of Observation 4.1 for mismatch information of just a single k-period despite encoding considerably more information. In particular, Lemma 4.2 shows that a string of length n can be encoded using $\mathcal{O}(d(\log\frac{n}{d} + \log|\Sigma|))$ bits so that one can quickly retrieve all its d-periods $p \leq \frac{1}{4}n$, along with the underlying mismatch information. We will also use this encoding later on in the main streaming algorithm of Section 7 .

Lemma 4.2. *Let $d = \gcd(\mathrm{Per}_{\leq n/4}(X,k))$ for a string $X \in \Sigma^n$ and an integer k. There is a data structure of size $\mathcal{O}(k(\log\frac{n}{k} + \log|\Sigma|))$ bits which, given indices i, i' such that $i \equiv i' \pmod d$, in $\mathcal{O}(\log n)$ time decides whether $X[i] = X[i']$ and retrieves both characters if they are distinct. Moreover, if $d \leq k$, any character $X[i]$ can be retrieved in $\mathcal{O}(\log n)$ time.*

Using this data structure, which is also central to our derivation of Theorem 1.2, we can now prove Theorem 1.4.

Theorem 1.4. *There exists an deterministic one-way communication complexity protocol for Problem 1.3 that sends $\mathcal{O}(k(\log\frac{n}{k} + \log|\Sigma|))$ bits, where Σ denotes the input alphabet.*

Proof sketch of Theorem 1.4. We shall assume that the text T is of length at most $5n/4$. If the actual text is longer, the full protocol splits it into substrings of length $5n/4$ with overlaps of length n, this enabling us to find all k-mismatches by repeating the protocol a constant number of times.

If P does not occur in T, Alice may send an empty message to Bob. Otherwise, her message consists of the following data:

- the location of the leftmost k-mismatch occurrence of P in T, along with the mismatch information;
- the location of the rightmost k-mismatch occurrence of P in T, along with the mismatch information;
- the value $d = \mathrm{Per}_{<n/4}(P, 2k)$ and the corresponding data structure of Lemma 4.2.

By Observation 4.1 and Lemma 4.2, the message takes $\mathcal{O}(k(\log \frac{n}{k} + \log|\Sigma|))$ bits.

Now, it suffices to describe how Bob uses the message to retrieve all the k-mismatch occurrences of P in T, as well as the corresponding mismatch information.

Let ℓ be the starting position of the leftmost k-mismatch occurrence of P in T and suppose that some other occurrence starts at position ℓ'. Observe that

$$\begin{aligned}&\mathrm{HD}(P[0,\ldots,n-\ell+\ell'-1], P[\ell-\ell',\ldots,n-1]) \leq \\ &\mathrm{HD}(P[0,\ldots,n-\ell+\ell'-1], T[\ell',\ldots,\ell+n-1]) + \mathrm{HD}(T[\ell',\ldots,\ell+n-1], P[\ell-\ell',\ldots,n-1]) \leq 2k.\end{aligned}$$

Moreover, $4(\ell'-\ell) < 4(|T|-|P|) \leq n$. Consequently, $\ell - \ell' \in \mathrm{Per}_{<n/4}(X,k)$ and thus $d \mid (\ell-\ell')$.

In particular, the rightmost occurrence must be at position $\ell + Md$ for some non-negative integer M, and each occurrence is at position $\ell + md$ for some $0 \leq m \leq M$. We shall prove that Bob can retrieve the mismatch information for the alignment of P at each of these positions. Suppose that he wants to compare $P[i]$ with $T[i+\ell+md]$. The position $\ell+md+i$ satisfies $\ell \leq i+\ell+md < \ell+Md+n$, so it must be located within the leftmost or the rightmost occurrence of P. By symmetry, we may assume (w.l.o.g.) that this is the leftmost occurrence.

Bob can use the data structure of Lemma 4.2 to retrieve $P[i]$ and $P[i+md]$ unless $P[i] = P[i+md]$. Similarly, the mismatch information for the leftmost occurrence of P lets Bob retrieve $P[i+md]$ and $T[i+\ell+md]$ unless these characters are equal. It is easy see that this is sufficient to decide whether $P[i] = T[i+\ell+md]$ and to retrieve both characters if they are not equal.

Consequently, Bob is indeed able to compute the mismatch information for the alignments of P at all positions $\ell + md$ for $0 \leq m \leq M$. He outputs the positions with at most k mismatches along with the corresponding mismatch information. □

5 Pattern Matching for Patterns with Small Approximate Period

We describe how the streaming k-mismatch problem can be solved quickly and in small space when the pattern P has at least one d-period which is of size $\mathcal{O}(k)$ when d itself is of size $\mathcal{O}(k)$. Our solution combines a new algorithm for fast convolution in sparse strings with a space efficient encoding of strings with small approximate period.

5.1 Fast Convolution of Sparse Strings

We first develop an algorithm which will allow us quickly to compute the convolution between two sparse strings.

Recall that the *convolution* of two functions $f, g : \mathbb{Z} \to \mathbb{Z}$ is a function $f * g : \mathbb{Z} \to \mathbb{Z}$ such that

$$(f*g)[i] = \sum_{j\in\mathbb{Z}} f(j)\cdot g(i-j).$$

Lemma 5.1. *Consider function $f, g : \mathbb{Z} \to \mathbb{Z}$ with supports of size at most n, Any δ consecutive values $(f*g)(i), \ldots, (f*g)(i+\delta-1)$ can be computed in $\mathcal{O}(n\sqrt{\delta \log \delta})$ time using $\mathcal{O}(n+\delta)$ words of working space.*

Proof. Let us partition $\mathbb{Z}$ into blocks $B_k = [\delta k, \delta k + \delta)$ for $k \in \mathbb{Z}$. Moreover, we define $B'_k = (i-(k+1)\delta, i-(k-1)\delta]$ and observe that $f*g(j) = \sum_k (f|_{B_k} * g|_{B'_k})(j)$ for $i \leq j < i+\delta$.

We say that a block is *heavy* if $|B_k \cap \mathrm{supp}(f)| \geq \sqrt{\delta \log \delta}$. For each heavy block B_k, we compute the convolution of $f|_{B_k} * g|_{B'_k}$ using the Fast Fourier Transform. This takes $\mathcal{O}(\delta \log \delta)$ time per heavy block and $\mathcal{O}(n\sqrt{\delta \log \delta})$ in total.

The light blocks B_k are processed naively: we iterate over non-zero entries of $f|_{B_k}$ and of $g|_{B'_k}$. Observe that each integer belongs to at most two blocks B'_k, so each non-zero entry of g is considered for at most $2\sqrt{\delta \log \delta}$ non-zero entries of f. Hence, the running time of this phase is also $\mathcal{O}(n\sqrt{\delta \log \delta})$. □

5.2 Representation of Strings with Approximate Period p

For a pattern P with small approximate period, a natural and space efficient representation is to encode the differences between characters separated by this period. This observation was first made in [13] in the context of the k-mismatch problem. This encoding may however make it more difficult to perform pattern matching quickly. In particular, we cannot apply our new fast convolution algorithm directly to find the mismatches between such encoded strings but will instead show that we can compute second order differences efficiently and that this is sufficient for our needs. We first define the first and second order differences for a sparse string.

For a function $f : \mathbb{Z} \to \mathbb{Z}$ and $p \in \mathbb{Z}_+$, we define the *forward difference* $\Delta_p[f] : \mathbb{Z} \to \mathbb{Z}$ and the *second order* forward difference $\Delta_p^2[f] : \mathbb{Z} \to \mathbb{Z}$ so that

$$\Delta_p[f](i) = f(i+p) - f(i),$$
$$\Delta_p^2[f](i) = \Delta_p[f](i+p) - \Delta_p[f](i) = f(i+2p) - 2f(i+p) + f(i).$$

Fact 5.2. *Consider functions $f, g : \mathbb{Z} \to \mathbb{Z}$ with finite support and a positive integer p. We have $\Delta_p[f * g] = f * \Delta_p[g] = \Delta_p[g] * f$ and $\Delta_p^2[f * g] = \Delta_p[f] * \Delta_p[g]$.*

Proof. Note that

$$\Delta_p[f*g](i) = \sum_{j\in\mathbb{Z}} f(j)g(i+p-j) - \sum_{j\in\mathbb{Z}} f(j)\cdot g(i-j) = \sum_{j\in\mathbb{Z}} f(j)\cdot \Delta_p[g](i-j) = (f * \Delta_p[g])(i).$$

By symmetry, we also have $\Delta_p[f * g] = \Delta_p[f] * g$. Consequently, $\Delta_p^2[f * g] = \Delta_p[\Delta_p[f] * g] = \Delta_p[f] * \Delta_p[g]$. □

We can now define the efficient representation of strings with approximate period p which we will use. For a string X and a character $a \in \Sigma$, we define a characteristic function $X_a : \mathbb{Z} \to \{0, 1\}$ so that $X_a[j] = 1$ if and only if $X[j] = a$. Observe that if X has a d-period p, then the vectors $\Delta_p[X_a]$ have $\mathcal{O}(p + d)$ non-zero entries in total (across all $a \in \Sigma$). This way, we can store X using $\mathcal{O}(d + p)$ machine words; we call it a *periodic representation* of X with respect to p.

For strings P and T, let us define a *cross-correlation* function $T \otimes P : \mathbb{Z} \to \mathbb{Z}$ so that $T \otimes P = \sum_{a\in\Sigma} T_a * P_a^R$, where $P_a^R = Q_a$ for the reverse string $Q = P^R$.

Fact 5.3. *We have $(T \otimes P)(i) = |P| - \mathrm{Ham}_{P,T}[i]$ if $|P| - 1 \le i < |T|$ and $(T \otimes P)(i) = 0$ if $i < 0$ or $i \ge |P| + |T|$.*

Proof. If $|P| - 1 \le i < |T|$, then:

$$|P| - \mathrm{Ham}_{P,T}[i] = |P| - \sum_{j=0}^{|P|-1} [T[i-j] \neq P[|P|-1-j]] = \sum_{j=0}^{|P|-1} [T[i-j] = P[|P|-1-j]] =$$
$$= \sum_{a\in\Sigma}\sum_{j=0}^{|P|-1} T_a(i-j)P_a(|P|-1-j) = \sum_{a\in\Sigma}\sum_{j=0}^{|P|-1} T_a(i-j)P_a^R(j) = \sum_{a\in\Sigma}(T_a * P_a^R)(i) = (T\otimes P)(i).$$

The second claim follows from the fact that $X_a(i) = 0$ if $i < 0$ or $i \ge |X|$. □

Combining Lemma 5.1, Fact 5.2, and the notions introduced above, we can obtain the second order differences of the cross-correlation between P and T efficiently and in small space:

Corollary 5.4. *Suppose that p is a d-period of strings P and T. Given the periodic representations of P and T with respect to p, any δ consecutive values $\Delta_p^2[T \otimes P](i), \ldots, \Delta_p^2[T \otimes P](i+\delta-1)$ can be computed in $\mathcal{O}((d+p)\sqrt{\delta \log \delta})$ time using $\mathcal{O}(d+p+\delta)$ words of working space.*

5.3 Small Approximate Period Algorithm

We apply Fact 5.3 and Corollary 5.4 to design a streaming algorithm for the case when both the pattern and the text have the same approximate period p. In order to achieve worst-case guarantee on the per-character processing time, we use a two-part partitioning known as the *tail trick*. Similar ideas were already used to deamortise streaming pattern matching algorithms; see [13, 12, 14].

Lemma 5.5. *Let P be a pattern with a d-period $p = \mathcal{O}(d)$. Suppose that p is also an $\mathcal{O}(d)$-period of the text T. There exists a deterministic streaming algorithm which processes T using $\mathcal{O}(d)$ words of space and $\mathcal{O}(\sqrt{d \log d})$ time per character, and reports $\mathrm{Ham}_{T,P}[i]$ for each position $i \geq |P| - 1$.*

Proof. First, we assume that blocks of d characters $T[i, \ldots, i+d-1]$ can be processed simultaneously.

We maintain the periodic representation of both P and T with respect to p. Moreover, we store the values $(T \otimes P)(j)$ for $i - 2p \leq j < i$ (these values are initialised as zeroes for $i = 0$; this is valid due to Fact 5.3). Since $d = \mathcal{O}(p)$, the space consumption is $\mathcal{O}(d)$.

When the block arrives, we update the periodic representation of T and apply Corollary 5.4 to compute $\Delta_p^2[T \otimes P](j)$ for $i \leq j < i + d$. Based on the stored values of $T \otimes P$, this lets us retrieve $(T \otimes P)(j)$ for $i \leq j < i + d$. Next, for each position $j > |P| - 1$, we report $\mathrm{Ham}_{T,P}[j] = |P| - (T \otimes P)(j)$. Finally, we discard the values $(T \otimes P)(j)$ for $j < i + d - 2p$. Such an iteration takes $\mathcal{O}(d\sqrt{d \log d})$ time and $\mathcal{O}(d)$ working space.

Below, we apply this procedure in a streaming algorithm which processes T character by character. The first step is to observe that if the pattern length is $\mathcal{O}(d)$, we can compute the Hamming distance online using $\mathcal{O}(d)$ words of space and $\mathcal{O}(\sqrt{d \log d})$ worst-case time per arriving symbol [10]. We now proceed under the assumption that $|P| > 2d$.

We partition the pattern into two parts: the *tail*, P_t — the suffix of P of length $2d$, and the *head*, P_h — the prefix of P length $|P| - 2d$. One can observe that $\mathrm{Ham}_{P,T}[j] = \mathrm{Ham}_{P_t,T}[j] + \mathrm{Ham}_{P_h,T}[j-2d]$. Moreover, we can compute $\mathrm{Ham}_{P_t,T}[j]$ using the aforementioned online algorithm of [10]; this takes $\mathcal{O}(\sqrt{d \log d})$ time per symbol and $\mathcal{O}(d)$ words of space.

For the second summand, we need to ensure that we will have computed $\mathrm{Ham}_{P_h,T}[j-2d]$ before $T[j]$ arrives. Hence, we partition the text into blocks of length d and use our algorithm to process a block $T[i, \ldots, i+d-1]$ as soon as it is ready. This procedure takes $\mathcal{O}(d\sqrt{d \log d})$ time, so it is can be performed in the background while we read the next block $T[i+d, \ldots, i+2d-1]$. Thus, $\mathrm{Ham}_{P_h,T}[j]$ is indeed ready on time for $j \in \{i, \ldots, i+d-1\}$.

The overall space usage is $\mathcal{O}(d)$ words and the worst-case time per arriving symbol is $\mathcal{O}(\sqrt{d \log d})$, dominated by the online procedure of [10] and by processing blocks in the background. □

Theorem 5.6. *Suppose that P is a pattern with a d-period $p = \mathcal{O}(k)$ with $d = \mathcal{O}(k)$. There exists a deterministic streaming algorithm, which uses $\mathcal{O}(k)$ words of space and $\mathcal{O}(\sqrt{k \log k})$ time per character to report the k-mismatch occurrences of P in the streamed text T. For each reported occurrence, the mismatch information can be computed on demand in $\mathcal{O}(k)$ time.*

Proof. Observe that if $\mathrm{HD}(P, Q) \leq k$, then p is a $(d+2k)$-period of Q. Thus, we maintain the longest suffix S of T for which p is a $(d+2k)$-period; we store it using the periodic representation with respect to p.

We run two instances of the algorithm from Lemma 5.5 for the pattern P. The texts T' and T'' of these instances are suffixes of T satisfying $|T'| \geq |S| \geq |T''|$. When the next character arrives, we update S and append the character to T' and T''. If this results in $|T''| > |S|$, we set $T' := T''$ and $T'' := \varepsilon$. We claim that p is always an $\mathcal{O}(k)$-period of T'. In fact, we shall prove that it is an $(2d + 4k + p + 1)$-period. Indeed, consider a moment when we reset T'' for the last time. At that point, p was a $(d + 2k + 1)$-period of T', and p is now a $(d + 2k)$-period of T''. The current text T' is the concatenation of these two strings, so p is indeed an $(2d + 4k + p + 1)$-period of T'.

If $|S| < |P|$, we know that P does not have a k-mismatch occurrence as a suffix of T. Otherwise, we retrieve $\mathrm{Ham}_{T,P}[|T|] = \mathrm{Ham}_{T',P}[|T'|]$ and report an occurrence if this value does not exceed k. The overall running time is $\mathcal{O}(\sqrt{k \log k})$ per character and the space consumption is $\mathcal{O}(k)$. The mismatch information, if needed, can be easily computed in $\mathcal{O}(k)$ time from the periodic representations of P and S. □

This concludes the description of our streaming k-mismatch algorithm in the case where the pattern has at least one small approximate period.

6 A Rolling k-mismatch Sketch

In order to develop our solution for the streaming k-mismatch problem when the pattern has no small approximate periods, we will need to introduce some further techniques. In this section, we describe a rolling sketch which will not only allow us to determine if two strings have Hamming distance at most k, but if they do, will also give us all the mismatch information. Our approach uses as its basis the deterministic sketch developed in [11] for the offline k-mismatch with wildcards problem and the classic Karp–Rabin fingerprints for exact matching [26].

We fix an upper bound n on the length of the compared strings, and a prime number $p > n^c$ for sufficiently large exponent c (the parameter c can be used to control error probability). We will assume throughout that all the input symbols can be treated as elements of $\mathbb{F}_p$ by simply reading the bit representation of the symbols. If the symbols come from a larger alphabet, then we would need to hash them into $\mathbb{F}_p$, which will introduce a small extra probability of error.

Let us start by recalling the Karp–Rabin fingerprint [26].

Definition 6.1 (Karp–Rabin fingerprints ψ_r)**.** *For a string $S \in \mathbb{F}_p^\ell$ and for $r \in \mathbb{F}_p$, the Karp–Rabin fingerprint $\psi_r(S)$ is defined as $\psi_r(S) = \sum_{i=0}^{\ell-1} S[i]r^i$.*

The most important property of Karp–Rabin fingerprints is that for any two equal-length strings U and V with $U \neq V$, the equality $\psi_r(U) = \psi_r(V)$ holds for at most ℓ out of p values $r \in \mathbb{F}_p$. Therefore if r is chosen at random from $[p]$, then the probability of a false positive is less than ℓ/n^c.

In order to tackle our k-mismatch problem, we use a modified version of the sketching function from [11], inspired by Reed–Solomon codes. Whereas the original version used a finite field of characteristic 2, which enabled a fast deterministic decoding scheme to be used, we need a field with large characteristic in order to be able to build a rolling sketch. We will see later that this change will make us use a different randomised algorithm to find the indices of the mismatches.

Definition 6.2 (Fingerprint functions ϕ and ϕ')**.** *Let p be a prime number. For a string $S \in \mathbb{F}_p^\ell$ and a non-negative integer j, we define $\phi_j(S) = \sum_{i=0}^{\ell-1} S[i]i^j$ and $\phi'_j(S) = \sum_{i=0}^{\ell-1} S[i]^2 i^j$*

Definition 6.3 (Sketch sk_k for the k-mismatch problem)**.** *For a fixed prime number p and for $r \in \mathbb{F}_p$ chosen uniformly at random, the sketch function $\mathrm{sk}_k(S)$ of a string $S \in \mathbb{F}_p^*$ is defined as:*

$$\mathrm{sk}_k(S) = (\phi_0(S), \phi_1(S), \ldots, \phi_{2k}(S), \phi'_0(S), \phi'_1(S), \ldots, \phi'_{k-1}(S), \psi_r(S), |S|).$$

Observation 6.4. *For every string $S \in \mathbb{F}_p^\ell$, the sketch $\mathrm{sk}_k(S)$ takes $\mathcal{O}(k \log p)$ bits.*

The main goal of the sketches is to check whether two given strings are at Hamming distance k or less, and, if so, to retrieve the mismatches. First, we deal with the latter task.

Lemma 6.5. *Consider strings $S, T \in \mathbb{F}_p^\ell$ such that $\mathrm{HD}(S,T) \le k$ and $\ell \le n$. Given $\mathrm{sk}_k(S)$ and $\mathrm{sk}_k(T)$, the mismatch information $\mathrm{MI}(S,T)$ can be computed in $\mathcal{O}(k \log^3 n)$ time using $\mathcal{O}(k \log n)$ bits of space. The algorithm may fail (report an error) with probability inverse polynomial in n.*

Proof. Define $k' = \mathrm{HD}(S,T)$. Let $x_1, \ldots, x_{k'}$ be the mismatch positions of S and T, and let $r_i = S[x_i] - T[x_i]$ be the corresponding numerical differences. We have:

$$\begin{array}{ccccccccc}
r_1 & + & r_2 & + & \ldots & + & r_{k'} & = & \phi_0(S) - \phi_0(T) \\
r_1 x_1 & + & r_2 x_2 & + & \ldots & + & r_{k'} x_{k'} & = & \phi_1(S) - \phi_1(T) \\
r_1 x_1^2 & + & r_2 x_2^2 & + & \ldots & + & r_{k'} x_{k'}^2 & = & \phi_2(S) - \phi_2(T) \\
 & & & \vdots & & & & & \vdots \\
r_1 x_1^{2k} & + & r_2 x_2^{2k} & + & \ldots & + & r_{k'} x_{k'}^{2k} & = & \phi_{2k}(S) - \phi_{2k}(T)
\end{array}$$

This set of equations is similar to those appearing in [11] and in the decoding procedures for Reed–Solomon codes. We use the standard Peterson–Gorenstein–Zierler procedure [31, 21], with subsequent efficiency improvements. This method consists of the following main steps:

1. Compute the *error locator polynomial* $P(z) = \prod_{i=1}^{k'} (1 - x_i z)$ from the *syndromes* $\phi_j(S) - \phi_j(T)$.
2. Find the *error locations* x_i by factoring the polynomial P.
3. Retrieve the *error values* r_i.

We implement the first step in $\mathcal{O}(k \log n)$ time using efficient *key equation* solver by Pan [30]; see Lemma A.4. The next challenge is to factorise P, taking advantage of the fact that it is a product of linear factors. As we are working over a field with large characteristic, there is no sufficiently fast deterministic algorithm for this task. Instead we use the randomised Cantor–Zassenhaus algorithm [6] (see Lemma A.3), which takes $\mathcal{O}(k \log^3 n)$ time with high probability. If the algorithm takes longer than this time then we stop the procedure and report a failure. Finally, we observe that the error values r_i can be retrieved by solving a transposed Vandermonde linear system of k' equations using Kaltofen–Lakshman algorithm [25] (see Lemma A.6) in $\mathcal{O}(k \log k \log n)$ time. Each of these subroutines uses $\mathcal{O}(k \log n)$ bits of working space.

Using the fact that we now have full knowledge of the mismatch indices x_i, a similar linear system lets us retrieve the values $r'_i = S^2[x_i] - T^2[x_i]$:

$$\begin{array}{ccccccccc}
r'_1 & + & r'_2 & + & \ldots & + & r'_{k'} & = & \phi'_0(S) - \phi'_0(T) \\
r'_1 x_1 & + & r'_2 x_2 & + & \ldots & + & r'_{k'} x_{k'} & = & \phi'_1(S) - \phi'_1(T) \\
r'_1 x_1^2 & + & r'_2 x_2^2 & + & \ldots & + & r'_{k'} x_{k'}^2 & = & \phi'_2(S) - \phi'_2(T) \\
 & & & \vdots & & & & & \vdots \\
r'_1 x_1^{k'} & + & r'_2 x_2^{k'} & + & \ldots & + & r'_{k'} x_{k'}^{k'} & = & \phi'_{k'}(S) - \phi'_{k'}(T)
\end{array} \quad (1)$$

Now, we are able to compute $S[x_i] = \frac{r'_i + r_i^2}{2 r_i}$ and $T[x_i] = \frac{r'_i - r_i^2}{2 r_i}$. □

Unfortunately, if the Hamming distance between two strings is greater than k, then we may get erroneous results from the above procedure. This issue is resolved by using the Karp–Rabin fingerprints to help us check if we have found all the mismatches or not.

Corollary 6.6. *Given the sketches $\mathrm{sk}_k(S)$ and $\mathrm{sk}_k(T)$ of two strings of the same length $\ell \le n$, in $\mathcal{O}(k \log^3 n)$ time we can decide (with high probability) whether $\mathrm{HD}(S,T) \le k$. If so, the mismatch information $\mathrm{MI}(S,T)$ is reported. The algorithm uses $\mathcal{O}(k \log n)$ bits of space.*

Proof. First, run the procedure of Lemma 6.5. If the algorithm fails, we may assume that $\mathrm{HD}(S,T) > k$; otherwise the failure probability is inverse polynomial in n.

A successful execution results in the mismatch information $\{(x_i, s_i, t_i) : 1 \le i \le k'\}$. Observe that $\mathrm{HD}(S,T) \le k$ if and only if $S[x_i]-T[x_i] = s_i-t_i$ and $S[j]-T[j] = 0$ at the remaining positions. In order to verify this condition, we compare the Karp–Rabin fingerprints, i.e., test whether

$$\psi_r(S) - \psi_r(T) = \sum_{i=1}^{k'} (s_i - t_i) r^{x_i}.$$

It is easy to see that this verification takes $\mathcal{O}(k' \log n)$ time. The error probability is at most $\frac{\ell}{p}$. □

Next, we consider efficiency of updating a sketch given the information about mismatches and we conclude that the sketch can be computed quickly and in small space.

Fact 6.7. *Consider two strings $S, T \in \mathbb{F}_p^n$ with $\mathrm{HD}(S,T) = d$. Given $\mathrm{sk}_k(S)$ and $\mathrm{MI}(S,T)$, the sketch $\mathrm{sk}_k(T)$ can be computed in $\mathcal{O}((d+k)\log^2 n)$ time using $\mathcal{O}(k \log n)$ bits of working space.*

Proof. If $d > k$, we gradually transform $\mathrm{sk}_k(S)$ into $\mathrm{sk}_k(T)$: we scan $\mathrm{MI}(S,T)$, processing k entries each time. Thus, we may assume without loss of generality that $d \le k$.

Let $\mathrm{MI}(S,T) = \{(x_i, s_i, t_i) : 0 \le i < d\}$. First, observe that the Karp–Rabin fingerprint can be updated in $\mathcal{O}(d \log n)$ time. Indeed, we have $\psi_r(T) - \psi_r(S) = \sum_{i=0}^{d-1} (t_i - s_i) r^{x_i}$, and each power r^{x_i} can be computed in $\mathcal{O}(\log n)$ time. Next, we shall compute $\phi_j(T) - \phi_j(S) = \sum_{i=0}^{d-1} (t_i - s_i) x_i^j$ for $j \le 2k$. This problem is an instance of transposed Vandermonde evaluation, see Lemma A.5. Hence, this task can be accomplished in $\mathcal{O}(k \log^2 n)$ time using $\mathcal{O}(k \log n)$ bits of space using the Canny–Kaltofen–Lakshman algorithm. The values $\phi'_j(T) - \phi'_j(S)$ are computed analogously. □

Corollary 6.8. *The sketch $\mathrm{sk}_k(S)$ of a given string $S \in \mathbb{F}_p^\ell$ can be computed in $\mathcal{O}((\ell+k)\log^2 n)$ time using $\mathcal{O}(k \log n)$ bits of working space.*

Proof. Let $\mathbf{0} \in \mathbb{F}_p^\ell$ be the all-zero string. Note that $\mathrm{sk}_k(\mathbf{0})$ can be computed trivially: the last coordinate is ℓ, whereas the remaining ones are zeroes. Moreover, $\mathrm{MI}(\mathbf{0}, S)$ can be generated online in $\mathcal{O}(\ell)$ time using $\mathcal{O}(k \log n)$ bits of working space. We plug the mismatches to Fact 6.7 so that $\mathrm{sk}_k(\mathbf{0})$ is transformed into $\mathrm{sk}_k(S)$ in $\mathcal{O}((\ell+k)\log^2 n)$ time using $\mathcal{O}(k \log n)$ bits of working space. □

Unlike in [11], we also need to update sketches subject to concatenation and prefix removal.

Lemma 6.9. *Consider sketches $\mathrm{sk}_k(U)$, $\mathrm{sk}_k(V)$, and $\mathrm{sk}_k(UV)$. Given two of these sketches, we can compute the third one in $\mathcal{O}(k \log n)$ time using $\mathcal{O}(k \log n)$ bits of space.*

Proof. First, observe that $\psi_r(UV) = \psi_r(U) + r^{|U|}\psi_r(V)$. This formula can be used to retrieve one of the Karp–Rabin fingerprints given the remaining two ones. The running time $\mathcal{O}(\log n)$ is dominated by computing $r^{|U|}$ (or $r^{-|U|}$).

Next, we express $\phi_j(UV) - \phi_j(U)$ in terms of $\phi_{j'}(V)$ for $j' \le j$:

$$\phi_j(UV) - \phi_j(U) = \sum_{i=0}^{|V|-1} V[i](|U|+i)^j = \sum_{i=0}^{|V|-1} \sum_{j'=0}^{j} V[i] \binom{j}{j'} i^{j'} |U|^{j-j'} = \sum_{j'=0}^{j} \binom{j}{j'} \phi_{j'}(V) |U|^{j-j'}.$$

Let us introduce an exponential generating function $\Phi(S) = \sum_{j=0}^{\infty} \phi_j(S) \frac{X^j}{j!}$, and recall that the exponential generating function of the geometric progression with ratio r is $e^{rX} = \sum_{j=0}^{\infty} t^j \frac{X^j}{j!}$. Now,

the equality above can be succinctly written as $\Phi(UV) - \Phi(U) = \Phi(V) \cdot e^{|U|X}$. Consequently, given first $2k+1$ coefficients of $\Phi(U)$, $\Phi(V)$, or $\Phi(UV)$, can be computed from the first $2k+1$ terms of the other two functions in $\mathcal{O}(k \log n)$ time using efficient polynomial multiplication over $\mathbb{F}_p$ [34]; see Corollary A.2. The coefficients $\phi'_j(U)$, $\phi'_j(V)$, and $\phi'_j(UV)$, can be computed in the same way. □

7 A Streaming Algorithm for k-mismatch

We now present our main result, an $\mathcal{O}(\log \frac{n}{k}(\sqrt{k \log k} + \log^3 n))$-time and $\mathcal{O}(k \log n \log \frac{n}{k})$-bits streaming algorithm for k-mismatch and thus prove Theorem 1.2. It first processes the pattern using unbounded resources to extract $\mathcal{O}(k \log n \log \frac{n}{k})$ bits of useful information. Then, it discards the pattern and starts processing the text.

If the pattern has a $2k$-period $p < k$, then we apply the streaming algorithm of Theorem 5.6, which is deterministic, uses $\mathcal{O}(k \log n)$ bits of space, and takes $\mathcal{O}(\sqrt{k \log k})$ time per character. Thus, we henceforth assume that P does not have any $2k$-period $p < k$.

We start with Section 7.1, where we describe the information stored about the pattern, along with two useful applications of this data. Next, in Section 7.2, we develop the first version of our streaming algorithm processing the text. In this implementation, each character is processed in $\mathcal{O}(\log \frac{n}{k}(\sqrt{k \log k} + \log^3 n))$ amortised time. In the final Section 7.3, we deamortise our procedure.

7.1 Preprocessing the pattern

In a similar fashion to the earlier work on streaming exact matching [32, 3], we introduce prefixes $P_1, \ldots, P_L$ of the pattern with exponentially increasing lengths $m_1, \ldots, m_L$. As the base case, we define P_1 to be the shortest prefix of P which does not have any $2k$-period $p < k$. It follows that $|P_1| > 2k$ and that P_1 has a $(2k+1)$-period $p < k$. The lengths of the subsequent prefixes are chosen so that they satisfy the following condition:

Observation 7.1. *Given integers $m_1 \leq n$, one can construct a sequence $m_2, \ldots, m_L$ with $L = \mathcal{O}(\log \frac{n}{m_1})$ such that $m_L = n$, and $m_\ell < m_{\ell+1} \leq \frac{5}{4} m_\ell + 1$ for $1 \leq \ell < L$.*

For each prefix P_ℓ, we denote $d_\ell = \gcd(\mathrm{Per}_{\leq m_\ell/4}(P_\ell, 2k))$ (see Section 4). Moreover, we distinguish $L' = \max\{\ell : d_\ell \leq k\}$. For each $\ell \in \{1, \ldots, L\}$, we store the following data:

- the sketch $\mathrm{sk}_k(P_\ell)$ and the value d_ℓ;
- if $\ell \geq L'$, the data structure of Lemma 4.2 built for the $2k$-periods of the prefix P_ℓ;
- if $\ell > L'$, the sketch $\mathrm{sk}_k(P[0, \ldots, d_\ell - 1])$.

As $m_1 = |P_1| > 2k$, the total size of this data is $\mathcal{O}(k \log n \log \frac{n}{k})$ bits. Additionally, we preprocess the prefix P_1 so the we can use the small approximate period streaming algorithm of Theorem 5.6 for P_1; the latter information takes $\mathcal{O}(k \log n)$ bits.

Before we proceed, let us describe two useful algorithmic applications of the stored information.

Fact 7.2. *A character $P_\ell[j]$ can be retrieved in $\mathcal{O}(\log n)$ time unless $\ell > L'$ and $P[j] = P[j \bmod d_\ell]$.*

Proof. First, observe that the data structure of Lemma 4.2 constructed for $P_{L'}$ lets us retrieve an arbitrary character of $P_{L'}$, and thus also of P_ℓ for $\ell \leq L'$. Similarly, Lemma 4.2 applied for P_ℓ lets us test if $P_\ell[j] = P_\ell[j \bmod d_\ell]$ and retrieve both characters if they are distinct. □

Fact 7.3. *Consider a fragment $F = P_\ell[\alpha \cdot d_\ell, \ldots, \beta \cdot d_\ell - 1]$ for $\alpha \leq \beta$. Its sketch $\mathrm{sk}_k(F)$ can be computed in $\mathcal{O}((k + |F|) \log^2 n)$ time using $\mathcal{O}(k \log n)$ bits of working space.*

Proof. We consider two cases. If $\ell > L'$, we partition F into blocks $F = F_\alpha \cdots F_{\beta-1}$ of length d_ℓ. For each block F_j, we use Fact 7.2 to generate a stream representing mismatch information $\mathrm{MI}(F_j, P[0, \ldots, d_\ell - 1])$. We plug this stream to Fact 6.7 in order to transform the stored sketch $\mathrm{sk}_k(P[0, \ldots, d_\ell - 1])$ into the sketch $\mathrm{sk}_k(F_j)$. Finally, we use Lemma 6.9 to combine the sketches of subsequent blocks into $\mathrm{sk}_k(F)$. The running time is $\mathcal{O}(d_\ell \log^2 n)$ per block and $\mathcal{O}(|F| \log^2 n)$ in total, while the working space is $\mathcal{O}(k \log n)$ bits.

If $\ell \leq L'$, then Fact 7.2 lets us retrieve the characters $P[j]$ for $\alpha d_\ell \leq j < \beta d_\ell$ in $\mathcal{O}(\log n)$ time each. Hence, Corollary 6.8 yields $\mathrm{sk}_k(F)$ in $\mathcal{O}((|F|+k) \log^2 n)$ time and $\mathcal{O}(k \log n)$ bits of space. □

7.2 Processing the text in amortised $\mathcal{O}(\log \frac{n}{k}(\sqrt{k \log k} + \log^3 n))$ time

Our algorithm consists in L levels running in parallel: each prefix P_ℓ is processed by one level of our algorithm. Level ℓ is responsible for determining when $P_{\ell+1}$ has a k-mismatch within the text. If the level ℓ finds such a k-mismatch occurrence, it passes the underlying mismatch information to the level $\ell + 1$; the topmost level $L - 1$ passes the mismatch information directly to the output. Every character $T[i]$ is processed by all the levels in the ascending order of their identifiers. Level 0, that is seeking for k-mismatch occurrences of P_1, is implemented using Theorem 5.6. In the overview below, we briefly cover levels $\ell > 0$.

The streaming algorithm always keeps the data about the pattern as described above, and the previous k characters $T[i-k, \ldots, i-1]$ of the text. Additionally, each level ℓ must maintain enough information to be able to determine which k-mismatch occurrences of P_ℓ can be extended to k-mismatch occurrences of $P_{\ell+1}$. We distinguish the *active region* A_ℓ of the text containing the starting positions of the occurrences of P_ℓ which we still need to try extending to $P_{\ell+1}$. Immediately before reading character $T[i]$, the active region is going to be $A_\ell = [i - m_{\ell+1} + 1, i - m_\ell]$. By Occ_ℓ ($\mathrm{Occ}_\ell \subseteq A_\ell$) we denote the actual starting positions of the *active* k-mismatch occurrences of P_ℓ. We shall also say that the whole level is *active* if $\mathrm{Occ}_\ell \neq \emptyset$.

In this terminology, the data stored at level ℓ needs to be sufficient to decide, for each active occurrence $j \in \mathrm{Occ}_\ell$, whether it can be extended to a k-mismatch occurrence of $P_{\ell+1}$. A naive solution could be to maintain $\mathrm{sk}_k(T[j, \ldots, i])$ for each $j \in \mathrm{Occ}_\ell$ so that $\mathrm{sk}_k(T[j, \ldots, j + m_{\ell+1} - 1])$ could be tested against $\mathrm{sk}_k(P_{\ell+1})$ while we process the character $T[j + m_{\ell+1} - 1]$. However, this is unfeasible in small space because $|\,\mathrm{Occ}_\ell\,|$ could be large, potentially of size $\Theta(|P|)$.

Nevertheless, as we have already observed in Section 4, two highly overlapping k-mismatch occurrences of P_ℓ induce much structure, which can be used to retrieve all the intermediate k-mismatch occurrences of the pattern. More precisely, based on the short $2k$-periods of the pattern, we identified an arithmetic progression of positions where all the intermediate occurrences must start, and we encoded the pattern (using Lemma 4.2) so that one can then verify on which *candidate positions* k-mismatch occurrences of P_ℓ actually start. Here, our strategy is similar: we also maintain a few overlapping occurrences of P_ℓ (along with the mismatch information) so that all the active occurrences are located in between. However, now we look for $P_{\ell+1}$ rather than P_ℓ, so we use Lemma 4.2 for a different purpose: in order to shift the sketch $\mathrm{sk}_k(T[j, \ldots, i])$ from one candidate position b to the next candidate position c. In more detail, Lemma 4.2 and the mismatch information for the stored k-mismatch occurrences let us retrieve $\mathrm{sk}_k(T[b, \ldots, c-1])$ (based on Fact 7.3) so that Lemma 6.9 can be applied to move the sketch forward.

Our choice of the shortest prefix P_1 guarantees that the k-mismatch occurrences of P_1, and thus P_ℓ for each $\ell \geq 1$, are located more than k positions apart. Consequently, we can spend much time for each occurrence, e.g., to manipulate the sketches, which takes $\mathcal{O}(k \log^{\mathcal{O}(1)} n)$ time. Nevertheless, the difference d_ℓ of the arithmetic sequence of candidate positions might be much smaller than k, in which case we cannot afford verifying every position using Corollary 6.6. However, this may

happen only for $\ell \leq L'$, and then Fact 7.2 provides efficient random access to P_ℓ. Combined with the mismatch information, it also yields random access to the region of the text covered by the k-mismatch occurrences of P_ℓ. Hence, we may re-run the streaming algorithm of Theorem 5.6 to filter out positions where $P_{\ell+1}$ cannot have a k-mismatch occurrence, because P_1 does not have one. We have to run the streaming algorithm of Theorem 5.6 using the pattern prefix P_1 separately for each level however as we cannot afford to remember answers too far in the past. This way, each level performs $\mathcal{O}(1)$ operations on sketches while processing k consecutive positions.

Below, we describe the streaming algorithm in detail, relying on the intuition provided above. In Section 7.2.1 we specify the information we store about the text using several invariants. This is followed by the detailed description of the implementation, which is provided in Section 7.2.2 along with the correctness proof. We conclude with the complexity analysis in Section 7.2.3.

7.2.1 Invariants at level $\ell > 0$

Let us now explicitly specify the information maintained by each level $\ell > 0$ of the streaming algorithm. First of all, we explicitly store the active region A_ℓ. We shall guarantee that $|A_\ell| \leq |m_{\ell+1} - m_\ell|$ at all times and $A_\ell = [i - m_{\ell+1} + 1, i - m_\ell]$ before processing $T[i]$. Further data is present only if the level is active, i.e., if $\mathrm{Occ}_\ell \neq \emptyset$. Our streaming algorithms stores the following data and ensures that each of the specified conditions on the variables holds true while the character $T[i]$ is been processed:

Invariant 1. The position $e = \max \mathrm{Occ}_\ell$.
This is the rightmost active occurrence of P_ℓ

Invariant 2. A position $b \leq \min \mathrm{Occ}_\ell$ such that $b \in A_\ell$ and $b \equiv e \pmod{d_\ell}$.
This is the leftmost candidate position where an occurrence of $P_{\ell+1}$ may start.

Invariant 3. The sketch $\mathbf{s} = \mathrm{sk}_k(T[b, \ldots, i'])$ for some $i' \in \{i - k + 1, \ldots, i\}$.

Invariant 4. The starting positions $x' \in \mathrm{Occ}_\ell$ and $x'' \in [x' - \frac{1}{4}m_\ell, b]$ of k-mismatch occurrences of P_ℓ.

Invariant 5. Mismatch information is stored for the occurrences of P_ℓ at positions e, x', and x''.

Invariant 6. If $\ell \leq L'$, a delayed instance $\mathcal{A}_\ell$ of the algorithm of Theorem 5.6, looking for k-mismatch occurrences of P_1 in T and having already processed $T[0, \ldots, b + m_1 - 1]$.

The maintained information takes $\mathcal{O}(k \log n)$ bits per level, and $\mathcal{O}(k \log n \log \frac{n}{k})$ bits in total. Before we proceed, let us state two simple yet useful properties of the stored positions. This tells occurrences of P_ℓ may occur in the vicinity of position b.

Observation 7.4. *If $j \in \mathrm{Occ}_\ell$, then $j \in \{b, \ldots, e\}$ and $j \equiv b \pmod{d_\ell}$. Moreover, we have $x' \equiv x'' \equiv e \equiv b \pmod{d_\ell}$.*

Proof. Note that $b - j \leq |A_\ell| - 1 \leq m_{\ell+1} - m_\ell - 1 \leq \frac{1}{4}m_\ell$. Hence, $b - j \in \mathrm{Per}_{\leq m_\ell/4}(P_\ell, 2k)$, and thus $j \equiv e \pmod{d_\ell}$. Now, the first claim follows from the explicit assumption $e \equiv b \pmod{d_\ell}$. Since $x' \in \mathrm{Occ}_\ell$, this also yields $x' \equiv b \pmod{d_\ell}$. To prove $x'' \equiv b \pmod{d_\ell}$, we observe that $x' - x'' \leq \frac{1}{4}m_\ell$, i.e., $x' - x'' \in \mathrm{Per}_{\leq m_\ell/4}(P_\ell, 2k)$, so we have $x'' \equiv x' \equiv b \pmod{d_\ell}$. □

7.2.2 Implementation of the streaming algorithm at level $\ell > 0$

There are three parts to be completed while processing $T[i]$. These are:

```
1  if level ℓ is active and i' = i − k then                                  ▷ Part 1.
2      s := sk_k(T[b, ..., i]);          ▷ Append T[i'+1, ..., i] using Corollary 6.8 and Lemma 6.9.
3      i' := i;

4  A_ℓ := [i − m_{ℓ+1} + 2, i − m_ℓ];                                        ▷ Part 2.
5  if b = i − m_{ℓ+1} + 1 then
6      s := sk_k(T[b, ..., i]);          ▷ Append T[i'+1, ..., i] using Corollary 6.8 and Lemma 6.9.
7      i' := i;
8      if Ham_{T,P_{ℓ+1}}[i] ≤ k then          ▷ Apply Corollary 6.6 for s and sk_k(P_{ℓ+1}).
9          Report the mismatch information;
10         if ℓ < L' then Report a copy of A_ℓ;
11     if b = e then Deactivate level ℓ;
12     else
13         if x' = b then (x'', x') := (x', e) ;          ▷ Mismatch information copied implicitly.
14         c := b + d_ℓ;                                  ▷ Advance A_ℓ by d_ℓ positions using Fact 7.5.
15         while ℓ ≤ L' and Ham_{T,P_1}[c + m_1 − 1] > k do   ▷ Reported using A_ℓ (Theorem 5.6).
16             c := c + d_ℓ;                              ▷ Advance A_ℓ by d_ℓ positions using Fact 7.5.
17         s' := sk_k(T[b, ..., c − 1]);                  ▷ Fact 7.6
18         s := sk_k(T[c, ..., i]);          ▷ Trim T[b, ..., c − 1] using Lemma 6.9 and s'.
19         b := c;

20 A_ℓ := [i − m_{ℓ+1} + 2, i − m_ℓ + 1];                                    ▷ Part 3.
21 if Ham_{T,P_ℓ}[i] ≤ k then                          ▷ Reported by level ℓ − 1.
22     if level ℓ is active. then e := i − m_ℓ + 1;   ▷ Mismatch information available from level ℓ − 1
23     else
24         Activate level ℓ;
25         b := e := x' := x'' := i − m_ℓ + 1;         ▷ Mismatch information available from level ℓ − 1.
26         if ℓ ≤ L' then Set A_ℓ;                     ▷ A copy of A_{ℓ−1} reported by level ℓ − 1.
27         s := sk_k(T[b, ..., i]);                    ▷ Fact 6.7
28         i' := i;
```

Algorithm 1: Processing symbol $T[i]$ at level $\ell > 0$.

Part 1: update the sketch $\mathbf{s}$ so that $i' > i - k$;

Part 2: advance the left endpoint of the active interval A_ℓ;

Part 3: advance the right endpoint of the active interval A_ℓ.

The implementation of Part 1 is straightforward: we do not need to do anything unless $i' = i-k$. In that case, we apply Corollary 6.8 to construct $\mathrm{sk}_k(T[i'+1, \ldots, i])$ and use Lemma 6.9 to combine it with $\mathrm{sk}_k(T[b, \ldots, i'])$. We may then set $i' := i$. As a result, Part 1 is void for the next $k-1$ iterations.

Next, consider Part 2, where we advance the left endpoint of A_ℓ from $i-m_{\ell+1}+1$ to $i-m_{\ell+1}+2$. Since $\min \mathrm{Occ}_\ell \geq b \geq i + m_{\ell+1} + 1$, one can easily verify that there is nothing to do if $b \neq i - m_{\ell+1} + 1$. Otherwise, we need to check if $P_{\ell+1}$ has a k-mismatch at position $i - m_{\ell+1} + 1$, i.e., whether $\mathrm{HD}(P_{\ell+1}, T[i - m_{\ell+1} + 1, \ldots, i]) \leq k$. For this, we apply Corollary 6.6 to $\mathrm{sk}_k(P_{\ell+1})$ and $\mathrm{sk}_k(T[b, \ldots, i])$. The former sketch is already stored, while to obtain the latter, we update $i' := i$ just like we did above (when $i' = i - k$). If Corollary 6.6 detects k-mismatch occurrence of $P_{\ell+1}$,

we pass the mismatch information to the subsequent level.

Now, we are ready to update A_ℓ (which affects Occ_ℓ as well). If $b = e$, then Occ_ℓ becomes empty and we can delete all the information stored about the text as specified in the invariants listed above. This way, the level is deactivated. Otherwise, we need to make some changes so that the invariants are valid again. We first deal with Invariant 4, which becomes false if $x' = b$. I In this case, we have $e \in \mathrm{Occ}_\ell$, $x' \leq b$, and $e - x' \leq m_{\ell+1} - m_\ell - 1 \leq \frac{1}{4} m_\ell$. Thus, we set $x'' := x'$ and $x' := e$ (copying the corresponding mismatch information) so that Invariant 4 is satisfied again.

Next, we need to increase b so that $b \in A_\ell$ and Invariant 2 becomes valid again. Note that Observation 7.4 guarantees that $\min \mathrm{Occ}_\ell \equiv e \equiv b \pmod{d_\ell}$, so $d_\ell > 0$ and $\min \mathrm{Occ}_\ell \geq b + d_\ell$. Consequently, we may increase b to $c := b + d_\ell$. However, if $\ell \leq L'$, we would like to increase b even more so that we do need to use Corollary 6.6 too often. To achieve this goal, we repeatedly increase c by d_ℓ until P_1 has a k-mismatch occurrence starting at position c. Note that $c \leq \min \mathrm{Occ}_\ell \leq x'$, because any k-mismatch occurrence of P_ℓ starts with a k-mismatch occurrence of P_1. In particular, we are guaranteed that b never exceeds x', so Invariant 2 and Invariant 4 remain valid.

To find the k-mismatch occurrences of P_1, we apply the instance $\mathcal{A}_\ell$ of the algorithm of Theorem 5.6 We feed $\mathcal{A}_\ell$ with the subsequent characters of T using the following fact; due to $x'' \leq b \leq c \leq x'$, we only need $T[j]$ for positions j within the scope of its applicability.

Fact 7.5. *If $\ell \leq L'$, any character $T[j]$ with $x'' \leq j < x' + m_\ell$ can be retrieved in $\mathcal{O}(\log n)$ time.*

Proof. Observe that $T[j]$ is located within a k-mismatch occurrence of P_ℓ at a position $p \in \{x'', x'\}$. In this occurrence, $T[j]$ is aligned with $P[j - p]$. The latter character can be retrieved in $\mathcal{O}(\log n)$ time using Fact 7.2, while the mismatch information $\mathrm{MI}(P_\ell, T[p \ldots, p + m_\ell - 1])$ (Invariant 5) lets us check if $T[j] = P[j - p]$ and, if not, learn the character $T[j]$. □

To complete the implementation of Part 2, the sketch $\mathbf{s}$ needs to be updated from $\mathrm{sk}_k(T[b, \ldots, i])$ to $\mathrm{sk}_k(T[c, \ldots, i])$. For this, we compute $\mathrm{sk}_k(T[b, \ldots, c-1])$ as described below, and then we apply Lemma 6.9 to trim the leading characters $T[b, \ldots, c-1]$ from $\mathbf{s}$.

Fact 7.6. *If $c \in \{b, \ldots, x'\}$ satisfies $c \equiv b \pmod{d_\ell}$, the sketch $\mathrm{sk}_k(T[b, \ldots, c-1])$ can be computed in $\mathcal{O}((k + c - b) \log^2 n))$ time using $\mathcal{O}(k \log n)$ bits of working space.*

Proof. Observe that $T[b, \ldots, c-1]$ is contained within $T[x'', \ldots, x'' + m_\ell - 1]$. Hence, it is aligned with a fragment $F = P_\ell[b - x'', \ldots, c - x'' - 1]$ in the k-mismatch occurrence of P_ℓ at position x''. Moreover, Observation 7.4 guarantees that $c \equiv b \equiv x'' \pmod{d_\ell}$, so Fact 7.3 lets us compute $\mathrm{sk}_k(F)$ in $\mathcal{O}((k + c - b) \log^2 n)$ time using $\mathcal{O}(k \log n)$ bits of working space. Next, we use the mismatch information for the occurrence of P_ℓ at position x'' (Invariant 5) to retrieve $\mathrm{MI}(F, T[b, \ldots, c-1])$. We plug this data to Fact 6.7 in order to transform $\mathrm{sk}_k(F)$ into $\mathrm{sk}_k(T[b, \ldots, c-1])$. The latter step takes $\mathcal{O}(k \log^2 n)$ time and $\mathcal{O}(k \log n)$ bits of working space. □

In Part 3, we advance the right endpoint of A_ℓ from $i - m_\ell$ to $i - m_\ell + 1$. Here, only a new k-mismatch occurrence of P_ℓ at position $i - m_\ell + 1$ may invalidate the invariants. We describe how to handle such an occurrence, reported by the level below along with the mismatch information. First, suppose that the level ℓ is inactive. In this case, we need to activate it because $\mathrm{Occ}_\ell = \{i - m_\ell + 1\}$ is no longer empty. It is easy to check that setting $b := e := x' := x'' := i - m_\ell + 1$ satisfies the invariants. Moreover, the required mismatch information (Invariant 5) is provided by the level $\ell - 1$. Similarly, a suitable instance of the algorithm of Theorem 5.6 (Invariant 6) is also provided if $\ell \leq L'$. Next, apply Fact 6.7 to compute the sketch $\mathrm{sk}_k(T[i - m_\ell + 1, i])$ based on $\mathrm{sk}_k(P_\ell)$ and the mismatch information provided by level $\ell - 1$.

Now, suppose that the level ℓ was active, i.e., Occ_ℓ already contained some position before inserting $i - m_\ell + 1$. Such a position is stored in the variable e, which needs to be updated to $i-m_\ell+1$ (along with the mismatch information). Both positions belong to Occ_ℓ, so Observation 7.4 guarantees that the distance between them is a multiple of d_ℓ. Hence, Invariant 2 is still satisfied after we update e. The remaining invariants are not affected by the update.

This completes the description of the algorithm, whose detailed implementation is provided in Algorithm 1. The discussion above can be summarised as follows:

Lemma 7.7. *With high probability, Algorithm 1 correctly maintains the invariants and reports the required data whenever $P_{\ell+1}$ has a k-mismatch occurrence ending at position i.*

7.2.3 Complexity Analysis

Our next task is to bound the space consumption and the running time of our streaming algorithm. The following property is crucial for efficiency of Algorithm 1:

Fact 7.8. *Prior to executing Line 19, we always have $c > b + k$.*

Proof. We consider two cases. If $\ell > L'$, then $d_\ell > k$, so $c = b + d_\ell$ trivially satisfies the inequality. Otherwise, we observe that P_1 has k-mismatch occurrences at both positions b and c. If $b < c \leq b+k$, this would yield a $2k$-period $p \leq k$ of P_1. Such a period does not exist by definition of P_1. □

We are now ready to bound the amortised running time of Algorithm 1.

Lemma 7.9. *Each level $\ell > 0$ uses $\mathcal{O}(k \log n)$ bits of space and takes $\mathcal{O}(\sqrt{k \log k}+\log^3 n)$ amortised time per character.*

Proof. We analyse the amortised cost of each line of Algorithm 1. Line 2 takes $\mathcal{O}(k \log^2 n)$ time, but it is executed at most once per k iterations, so its amortised cost is $\mathcal{O}(\log^2 n)$. By Fact 7.8, the condition in Line 5 is also satisfied at most once per k iterations. Consequently, the amortised cost of Lines 6–13 and 18–19 is $\mathcal{O}(\log^3 n)$, dominated by the application of Corollary 6.6. We amortise the cost of Lines 14–17 using the increase of b, i.e., $c-b$. The algorithm of Theorem 5.6 takes $\mathcal{O}(\sqrt{k \log k})$ per character, so advancing $\mathcal{A}_\ell$ takes $\mathcal{O}((c-b)\sqrt{k \log k})$ time. We use Fact 7.5 to feed $\mathcal{A}_\ell$, which takes $\mathcal{O}((c-b)\log n)$. Moreover, Fact 7.6 computes $\mathrm{sk}_k(T[b, \ldots, c-1])$ in $\mathcal{O}((k+(c-b))\log^2 n) = \mathcal{O}((c-b)\log^2 n)$ time. Consequently, the amortised cost is $\mathcal{O}(\sqrt{k \log k} + \log^2 n)$.

In Part 3, the condition in Line 20 is satisfied at most once per k consecutive positions because a k-mismatch occurrence of P_ℓ yields a k-mismatch occurrence of P_1. Consequently, the amortised cost of Part 3 is $\mathcal{O}(\log^2 n)$, dominated by the application of Fact 6.7 in Line 28.

The overall amortised running time is therefore $\mathcal{O}(\sqrt{k \log k} + \log^3 n)$ per character. The bound of $\mathcal{O}(k \log n)$ bits on the working space follows from space guarantees of each of the subroutines. □

We are now ready to prove an amortised version of Theorem 1.2.

Theorem 7.10. *There exists a solution for Problem 1.1 which, after preprocessing the pattern, uses $\mathcal{O}(\log \frac{n}{k}(\sqrt{k \log k} + \log^3 n))$ amortised time per character and $\mathcal{O}(k \log n \log \frac{n}{k})$ bits of space. The algorithm is randomised and its answers are correct with high probability, that is it errs with probability inverse polynomial in n. For each reported occurrence, the mismatch information can be reported on demand in $\mathcal{O}(k)$ time.*

Proof. The space consumption is $\mathcal{O}(k \log n \log \frac{n}{k})$ bits for the pattern, and $\mathcal{O}(k \log n)$ bits for each level of the streaming algorithm. The number of levels is $\mathcal{O}(\log \frac{n}{k})$, so this is $\mathcal{O}(k \log n \log \frac{n}{k})$ bits in total. As far as time complexity is concerned, level 0 takes $\mathcal{O}(\sqrt{k \log k})$ time per symbol and, by Lemma 7.9, the subsequent $\mathcal{O}(\log \frac{n}{k})$ levels take $\mathcal{O}(\sqrt{k \log k} + \log^3 n)$ amortised time per character. □

7.3 Deamortisation

We will now describe how to deamortise our online k-mismatch algorithm. First, we show that some computations in Algorithm 1 can be delayed so that the amortisation is needed only to smooth out the running time over the subsequent k iterations.

Lemma 7.11. *The algorithm of Theorem 7.10 can be implemented so that the worst-case processing time of k consecutive characters is $\mathcal{O}(k \log \frac{n}{k}(\sqrt{k \log k} + \log^3 n))$.*

Proof. The algorithm of Theorem 5.6 already has a $\mathcal{O}(\sqrt{k \log k})$ worst-case time per character, so it suffices to update the implementation of levels $\ell > 0$, provided in Algorithm 1. The analysis in Lemma 7.9 reveals that the only obstacle to overcome is the cost $\mathcal{O}((c-b)(\sqrt{k \log k} + \log^2 n))$ of Lines 14–17. In a single iteration we may even have $c - b = \Theta(m_\ell)$, but one can observe that the condition in Line 5 will not be satisfied during the next time $c - b - 1$ iterations. Moreover, in the meantime the sketch $\mathbf{s}$ is only accessed in Line 2, where subsequent characters are appended to the sketch. These updates are independent of the trimming of $T[b, \ldots, c-1]$ in Line 18. Consequently, we may execute Lines 14–17 in parallel with the subsequent iterations, allocating $\mathcal{O}(\sqrt{k \log k} + \log^2 n)$ time per character (for a single layer). Once this thread terminates, we update $\mathbf{s}$, running Line 18 in a single iteration (so that it is not intermixed with Line 2). This way, each layer takes $\mathcal{O}(k\sqrt{k \log k} + k \log^3 n)$ time to process any subsequent k characters, and the claim follows. □

Further deamortisation is based on a simple variant of the *tail trick*, a technique which we already used in Section 5.

Theorem 1.2. *There exists a solution for Problem 1.1 which, after preprocessing the pattern, uses $\mathcal{O}(\log \frac{n}{k}(\sqrt{k \log k} + \log^3 n))$ time per arriving symbol and $\mathcal{O}(k \log n \log \frac{n}{k})$ bits of space. The algorithm is randomised and its answers are correct with high probability, that is it errs with probability inverse polynomial in n. For each reported occurrence, the mismatch information can be reported on demand in $\mathcal{O}(k)$ time.*

Proof. We assume that the pattern P does not have any $4k$-period $p \leq k$. Otherwise, Theorem 5.6 can be used to process each character in $\mathcal{O}(\sqrt{k \log k})$ worst-case time. Let us partition the pattern P into two parts, the *head* P_h and the *tail* P_t, so that $P = P_h P_t$ and $|P_t| = 2k$. Our assumption on P guarantees that $|P| > 4k$ (so the partition is valid) and that P_h does not have any $2k$-period $p \leq k$.

We shall run the amortised algorithm of Lemma 7.11 for the head P_h and, for each k-mismatch occurrence reported, naively check if it extends to a k-mismatch occurrence of P. More specifically, we store a buffer of the previous $2k$ characters $T[i-2k, \ldots, i-1]$ and we make sure that the algorithm matching P_h processes the character $T[j]$ before we read $T[j+k]$. The worst-case guarantee of Lemma 7.11 shows that this is possible while the number of steps it performs is $\mathcal{O}(\log \frac{n}{k}(\sqrt{k \log k} + \log^3 n))$ per each read symbol. This way, whenever $\mathrm{Ham}_{P_h,T}[j-2k] \leq k$, we get notified before we process $T[j-k]$. The forthcoming k iterations can be used to naively compute $\mathrm{Ham}_{P_t,T}[j]$ so that $\mathrm{HD}_{P,T}[j]$ can be determined as $\mathrm{Ham}_{P_h,T}[j-2k] + \mathrm{Ham}_{P_t,T}[j]$. At any time we have $\mathcal{O}(1)$ such pending occurrences of P_h because the k-mismatch occurrences of P_h are located at least k positions apart (otherwise, P_h would have a $2k$-period $p \leq k$). Hence, processing the tail takes $\mathcal{O}(k \log n)$ bits of extra space and $\mathcal{O}(1)$ time per character. The space usage and the time complexity are therefore dominated by the processing of the head P_h using Lemma 7.11. □

8 Encoding Strings with Many Approximate Periods (Proof of Lemma 4.2)

In this section, we prove Lemma 4.2; its statement is repeated below for completeness.

Lemma 4.2. *Let $d = \gcd(\mathrm{Per}_{\le n/4}(X,k))$ for a string $X \in \Sigma^n$ and an integer k. There is a data structure of size $\mathcal{O}(k(\log \frac{n}{k} + \log|\Sigma|))$ bits which, given indices i, i' such that $i \equiv i' \pmod d$, in $\mathcal{O}(\log n)$ time decides whether $X[i] = X[i']$ and retrieves both characters if they are distinct. Moreover, if $d \le k$, any character $X[i]$ can be retrieved in $\mathcal{O}(\log n)$ time.*

Our encoding is primarily based on the following concept of classes modulo p.

Definition 8.1. *Let X be a fixed string of length n. For integers i, p with $p \ge 0$, the* i-th class modulo p *(in X) is defined as a multiset:*

$$\mathcal{C}_p(i) = \{X[i'] : 1 \le i \le n \text{ and } i' \equiv i \pmod p\}.$$

For $p = 0$ we assume that $i' \equiv i \pmod 0$ if and only if $i = i'$.

We say that a multiset is *uniform* if all its elements are equal. The *majority* element of a multiset S is an element with multiplicity strictly greater than $\frac{1}{2}|S|$. We define uniform strings and majority characters of a string in an analogous way.

Using these notions, a stronger version of Lemma 4.2 can be formulated:

Lemma 8.2. *Let $d = \gcd(\mathrm{Per}_{\le n/4}(X,k))$ for a string X of length n and a positive integer k. There is a data structure of size $\mathcal{O}(k(\log \frac{n}{k} + \log|\Sigma|))$ bits which, given an index i, in $\mathcal{O}(\log n)$ time can retrieve $X[i]$ unless $d > k$ and the class $\mathcal{C}_d(i)$ is uniform.*

The reduction is as follows: Suppose that we are asked to compare $X[i]$ and $X[i']$ for $i \equiv i' \pmod d$. We try retrieving both characters in $\mathcal{O}(\log n)$ time each. This procedure succeeds unless $\mathcal{C}_d(i) = \mathcal{C}_d(i')$ is uniform. In this case, we are guaranteed that $X[i] = X[i']$.

The remaining part of this section constitutes a proof of Lemma 8.2. We start with Section 8.1, where we introduce the main ideas, which rely on the structure of classes and their majorities. The subsequent Section 8.2 provides further combinatorial insight necessary to bound the size of our encoding. Section 8.3 presents two abstract building blocks based on well-known compact data structures. Next, in Section 8.4 we give a complete description of our encoding, and in Section 8.5 we address answering queries.

8.1 Overview

First, we observe that it suffices to focus on an $\mathcal{O}(\log n)$-element subset of $\mathrm{Per}_{\le n/4}(X,k)$.

Fact 8.3. *There is a subset $\mathcal{P} = \{p_1, \ldots, p_s\} \subseteq \mathrm{Per}_{\le n/4}(X,k)$ such that the partial greatest common divisors $d_\ell = \gcd(p_1, \ldots, p_\ell)$ satisfy the following properties:*

- *$d_0 = 0$, $d_1 = p_1$, and $d_s = d$;*
- *$d_\ell \mid d_{\ell-1}$ for $1 \le \ell \le s$;*
- *$d_\ell \le \frac{n}{2^{\ell+1}}$ for $1 \le \ell \le s$, and consequently $|\mathcal{P}| = \mathcal{O}(\log n)$.*

Proof. We construct the set $\mathcal{P}$ inductively, adding subsequent elements p_ℓ one by one. We maintain the value $d_\ell = \gcd(p_1, \ldots, p_\ell)$, which is always a multiple of d. We start with an empty set with $d_0 = \gcd \emptyset = 0$, and the construction terminates as soon as $d_\ell = d$. Otherwise, d_ℓ is a common divisor of $\mathrm{Per}_{\le n/4}(X,k)$, so we choose $p_{\ell+1} \in \mathrm{Per}_{\le n/4}(X,k)$ so that $d_\ell \nmid p_{\ell+1}$, and consequently $d_{\ell+1}$ is a proper divisor of d_ℓ. The latter condition guarantees that the construction stops. Moreover, $d_{\ell+1} < \frac{1}{2} d_\ell$ for $\ell \ge 1$, so $d_\ell \le \frac{d_1}{2^{\ell-1}} \le \frac{n}{2^{\ell+1}}$. In particular, $d_s \le \frac{n}{2^{s+1}}$, so $s = \mathcal{O}(\log n)$. □

Let $\mathbf{C}_\ell$ be the partition of the characters of X into classes $\mathcal{C}_{d_\ell}(i)$ modulo d_ℓ. Fact 8.3 lets us characterise the sequence $\mathbf{C}_0, \ldots, \mathbf{C}_\ell$: the first partition, $\mathbf{C}_0$, is the partition into singletons, i.e., the finest possible partition. Then, each partition is coarser than the previous one, and finally $\mathbf{C}_s$ is the partition into classes modulo d.

Consequently, the classes modulo $\mathcal{C}_{d_\ell}(i)$ for $0 \leq \ell \leq s$ form a laminar family, which can be represented as a forest of depth $s+1 = \mathcal{O}(\log n)$; its leaves are single characters (classes modulo $d_0 = 0$), while the roots are classes modulo d. Let us imagine that each class stores its majority element (if there is one). Observe that if all the classes $\mathcal{C}_{d_{\ell-1}}(i')$ contained in a given class $\mathcal{C}_{d_\ell}(i)$ share a common majority element, then this value is also the majority of $\mathcal{C}_{d_\ell}(i)$. Consequently, storing the majority elements of all the contained $\mathcal{C}_{d_{\ell-1}}(i')$ is redundant. Now, in order to retrieve $X[i]$, it suffices to start at the leaf $\mathcal{C}_{d_0}(i)$, walk up the tree until we reach a class storing its majority, and return the majority, which is guaranteed to be equal to $X[i]$. This is basically the strategy of our query algorithm. A minor difference is that we do not store the majority element of uniform classes $\mathcal{C}_d(i)$, because our query procedure is allowed to fail if $\mathcal{C}_d(i)$ is uniform.

In order to efficiently store the majority characters of classes $\mathcal{C}_{d_{\ell-1}}(i')$ contained in a given class $\mathcal{C}_{d_\ell}(i)$, let us study the structure of these classes is more detail

Observation 8.4. *Each class modulo d_ℓ can be decomposed as follows into non-empty classes modulo $d_{\ell-1}$;*

$$\mathcal{C}_{d_\ell}(i) = \bigcup_{j=0}^{\frac{d_{\ell-1}}{d_\ell}} \mathcal{C}_{d_{\ell-1}}(i + jp_\ell) \qquad \text{if } \ell > 1$$

$$\mathcal{C}_{d_\ell}(i) = \bigcup_{j=0}^{\lceil \frac{n-i}{d_\ell} \rceil} \mathcal{C}_{d_{\ell-1}}(i + jp_\ell) \qquad \text{if } \ell = 1$$

Motivated by this decomposition, for each class $\mathcal{C}_{d_\ell}(i)$ with $\ell \geq 1$, we define the majority string $M_{\ell,i}$ so that $M_{\ell,i}[j]$ is the majority character of $\mathcal{C}_{d_{\ell-1}}(i + jp_\ell)$, or \$ if the class has no majority. We set $|M_{\ell,i}| = \frac{d_{\ell-1}}{d_\ell}$ for $\ell > 1$ and $|M_{\ell,i}| = \lceil \frac{n-i}{d_\ell} \rceil$ for $\ell = 1$.

Since $p_\ell \in \mathrm{Per}(X, k)$, we expect that the adjacent characters of the majority strings $M_{\ell,i}$ are almost always equal. In fact, in the next section we shall prove that the total number of mismatches (across all majority strings) is $\mathcal{O}(k)$. The number of non-uniform classes modulo d (for which we also store the majority element) is also shown to be $\mathcal{O}(k)$.

8.2 Combinatorial Bounds

For $1 \leq \ell \leq s$, let $\mathbf{N}_\ell \subseteq \mathbf{C}_\ell$ consist of non-uniform classes modulo d_ℓ. Moreover, let $\mathbf{K}_\ell \subseteq \mathbf{C}_\ell$ consist of classes $\mathcal{C}_{d_\ell}(i)$ such that the majority elements of $\mathcal{C}_{d_\ell}(i)$ and $\mathcal{C}_{d_\ell}(i + p_{\ell+1})$ are distinct. The following two inequalities are crucial to bound the size of our data structure.

Lemma 8.5. *The following inequalities are satisfied for $1 \leq \ell \leq s$:*

1. $2|\mathbf{N}_{\ell+1}| + |\mathbf{K}_\ell| \leq 2|\mathbf{N}_\ell| + 2|\mathbf{K}_\ell \setminus \mathbf{N}_\ell|$ *for* $1 \leq \ell < s$,
2. $|\mathbf{K}_\ell \setminus \mathbf{N}_\ell| \leq 2^{2-\ell} \cdot k$.

Proof. **1.** We provide a discharging argument. In the charging phase, each class in $\mathcal{C}_{d_\ell}(i) \in \mathbf{N}_\ell \cup \mathbf{K}_\ell$ receives two tokens. The number of assigned tokens is clearly $2|\mathbf{N}_\ell| + 2|\mathbf{K}_\ell \setminus \mathbf{N}_\ell|$. Next, each class

$\mathcal{C}_{d_\ell}(i) \in \mathbf{K}_\ell$ passes one token to class $\mathcal{C}_{d_{\ell+1}}(i)$ containing it, while each class $\mathcal{C}_{d_\ell}(i) \in \mathbf{N}_\ell \setminus \mathbf{K}_\ell$ passes two tokens to $\mathcal{C}_{d_{\ell+1}}(i)$.

Every class $\mathcal{C}_{d_\ell}(i) \in \mathbf{K}_\ell$ is left with one token, so we only need to prove that each class $\mathcal{C}_{d_{\ell+1}}(i) \in \mathbf{N}_{\ell+1}$ received at least two tokens. Suppose that $\mathcal{C}_{d_{\ell+1}}(i)$ is such a class. Observation 8.4 yields a decomposition

$$\mathcal{C}_{d_{\ell+1}}(i) = \bigcup_{j=0}^{\frac{d_\ell}{d_{\ell+1}}} \mathcal{C}_{d_\ell}(i + jp_{\ell+1}).$$

If the majority string $M_{\ell+1,i}$ is uniform, then one of these classes satisfies $\mathcal{C}_{d_\ell}(i+jp_{\ell+1}) \in \mathbf{N}_\ell \setminus \mathbf{K}_\ell$ and passes two tokens to $\mathcal{C}_{d_{\ell+1}}(i)$. Otherwise, there are at least two classes $\mathcal{C}_{d_\ell}(i + jp_{\ell+1}) \in \mathbf{K}_\ell$, each of which passes two tokens to $\mathcal{C}_{d_{\ell+1}}(i)$. This completes the proof.

2. Suppose that $\mathcal{C}_{d_\ell}(i) \in \mathbf{K}_\ell \setminus \mathbf{N}_\ell$ satisfies the condition listed above, and let $k_{\ell,i}$ be the total number of mismatches between $X[i + jd_\ell]$ and $X[i + jd_\ell + p_{\ell+1}]$ for integers j such that $0 \leq i + jd_\ell < i + jd_\ell + p_{\ell+1} < n$.

Observe that at most $\lceil \frac{p_{\ell+1}}{d_\ell} \rceil$ characters in $\mathcal{C}_{d_\ell}(i + p_{\ell+1})$ are at indices smaller than $p_{\ell+1}$ (and they are not aligned with any character of $\mathcal{C}_{d_\ell}(i)$), and exactly $k_{\ell,i}$ characters are aligned with mismatching characters. The remaining characters are aligned with matching characters of $\mathcal{C}_{d_\ell}(i)$. The class $\mathcal{C}_{d_\ell}(i)$ is uniform, so this means that at most $k_i + \lceil \frac{p_{\ell+1}}{d_\ell} \rceil$ characters of $\mathcal{C}_{d_\ell}(i+p_{\ell+1})$ are not equal to the majority character of $\mathcal{C}_{d_\ell}(i)$. Since $\mathcal{C}_{d_\ell}(i+p_{\ell+1})$ does not share the majority character with $\mathcal{C}_{d_\ell}(i)$, we must have $k_i + \lceil \frac{p_{\ell+1}}{d_\ell} \rceil > \frac{1}{2}|\mathcal{C}_{d_\ell}(i + p_{\ell+1})|$. Since $p_{\ell+1} \leq \frac{n}{4}$, this yields

$$k_{\ell,i} > \tfrac{1}{2}|\mathcal{C}_{d_\ell}(i + p_{\ell+1})| - \left\lceil \tfrac{p_{\ell+1}}{d_\ell} \right\rceil \geq \tfrac{1}{2} \left\lfloor \tfrac{n}{d_\ell} \right\rfloor - \left\lceil \tfrac{n}{4d_\ell} \right\rceil > 2 \left\lfloor \tfrac{n}{4d_\ell} \right\rfloor - \left\lfloor \tfrac{n}{4d_\ell} \right\rfloor - 1 \geq \left\lfloor \tfrac{n-4d_\ell}{4d_\ell} \right\rfloor.$$

Consequently,

$$k_{\ell,i} \geq \left\lceil \tfrac{n-4d_\ell}{4d_\ell} \right\rceil \geq \max(1, \tfrac{n-4d_\ell}{4d_\ell}) \geq \tfrac{1}{2}(1 + \tfrac{n-4d_\ell}{4d_\ell}) = \tfrac{n}{8d_\ell} \geq \tfrac{n2^{\ell+1}}{8n} = 2^{\ell-2},$$

with the last inequality due to Fact 8.3. The fact that $p_{\ell+1} \in \mathrm{Per}(X, k)$ yields

$$k \geq \sum_{i=0}^{d_\ell - 1} k_{\ell,i} \geq \sum_{i:\mathcal{C}_q(i) \in \mathbf{K}_\ell \setminus \mathbf{N}_\ell} k_{\ell,i} \geq |\mathbf{K}_\ell \setminus \mathbf{N}_\ell| \cdot 2^{\ell-2},$$

so $|\mathbf{K}_\ell \setminus \mathbf{N}_\ell| \leq 2^{2-\ell} \cdot k$, as claimed. □

We are now ready to conclude the bounds essential for the size of our data structure.

Corollary 8.6. *1. The number of non-uniform classes modulo d is at most $5k$.*

2. The majority strings $M_{\ell,i}$ contain in total at most $10k$ mismatches between adjacent characters.

Proof. First, observe that $M_{1,i}[j] = X[i + jp_1]$, so each mismatch between $M_{1,i}[j]$ and $M_{1,i}[j + 1]$ corresponds to a mismatch between $X[i + jp_1]$ and $X[i + jp_1 + p_1]$. The total number of such mismatches is k. Moreover, this also yields a bound $|\mathbf{N}_1| \leq k$.

Next, note that for $\ell > 1$, a mismatch between $M_{\ell,i}[j]$ and $M_{\ell,i}[j + 1]$ corresponds to a class $\mathcal{C}_{d_{\ell-1}}(i + jp_\ell) \in \mathbf{K}_{\ell-1}$. These values can be bounded using Lemma 8.5, which states that $2|\mathbf{N}_{\ell+1}| + |\mathbf{K}_\ell| \leq 2|\mathbf{N}_\ell| + 2^{3-\ell} \cdot k$ for $1 \leq \ell < s$. Summing up the inequalities, we obtain:

$$2|\mathbf{N}_s| + \sum_{\ell=1}^{s-1} |\mathbf{K}_\ell| \leq 2|\mathbf{N}_1| + k \sum_{\ell=1}^{s-1} 2^{3-\ell} \leq 2k + 8k = 10k.$$

Consequently, $|\mathbf{N}_s| \leq 5k$ and the total number of mismatches adjacent characters of mismatch strings $M_{\ell,i}$ for $\ell > 1$ is at most $10k$. □

8.3 Algorithmic Tools

A maximal uniform fragment of a string S is called a *run*; we denote the number of runs by $\mathrm{rle}(S)$.

Fact 8.7 (Run-length encoding)**.** *A string S of length n with $r = \mathrm{rle}(S)$ can be encoded using $\mathcal{O}(r(\log \frac{n+r}{r} + \log|\Sigma|))$ bits so that any given character $S[i]$ can be retrieved in $\mathcal{O}(\log r)$ time.*

Proof. Let $1 = x_1 < \ldots < x_r \leq n$ be starting position of each run. We store the sequence $x_1, \ldots, x_r$ using the Elias–Fano representation [16, 19] (with $\mathcal{O}(1)$-time data structure for selection queries in a bitmask; see e.g. [8]). This representation takes $\mathcal{O}(r \log \frac{n+r}{r} + r)$ bits and allows $\mathcal{O}(1)$-time access. In particular, in $\mathcal{O}(\log r)$ time we can binary search for the run containing a given position i. The values $X[x_1], \ldots, X[x_r]$ are stored using $\mathcal{O}(r \log|\Sigma|)$ bits with $\mathcal{O}(1)$-time access. □

Fact 8.8 (Membership queries, [4])**.** *A set $A \subseteq \{0, \ldots, n-1\}$ of size at most m can be encoded in $\mathcal{O}(m \log \frac{n+m}{m})$ bits so that one can check in $\mathcal{O}(1)$ time whether $i \in A$ for a given $i \in \{0, \ldots, n-1\}$.*

8.4 Data Structure

Following the intuitive description in Section 8.1, we shall store all the non-uniform majority strings $M_{\ell,i}$ (for $1 \leq \ell \leq s$ and $0 \leq i < d_\ell$). Corollary 8.6 yields that the total number of mismatches between subsequent characters is $\mathcal{O}(k)$, so the number of non-uniform majority strings is also $\mathcal{O}(k)$. If $d \leq k$, we also store the majority character of each class modulo d. Otherwise, we store this character for non-uniform classes only; note that there are $\mathcal{O}(k)$ such classes due to Corollary 8.6.

We store all this information in an efficient encoding of a single string $\mathbf{M}$. Let $\mathbf{M}$ be a string of length $2n$, initially consisting of blank characters $\diamond$. At $\mathbf{M}[i]$ we store the majority character of $\mathcal{C}_d(i)$ unless $d > k$ and the class $\mathcal{C}_d(i)$ is uniform. Moreover, each non-uniform majority string $M_{\ell,i}$ is placed in $\mathbf{M}$ at position $2d_\ell + i\frac{d_{\ell-1}}{d_\ell}$ if $\ell > 1$ and $2d_\ell + i\lceil nd_\ell \rceil$ if $\ell = 1$.

It is easy to see that for each ℓ the target fragments do not overlap. Moreover, they are contained within range $[2d_\ell, 2d_\ell + d_{\ell-1} - 1] \subseteq [2d_\ell, 2d_{\ell-1} - 1]$ for $\ell > 1$ and $[2d_1, \ldots, 2d_1 + d_1\lceil nd_1 \rceil - 1] \subseteq [2d_1, 3d_1 + n - 1] \subseteq [2d_1, 2n - 1]$ for $\ell = 1$. These ranges are clearly disjoint for distinct values ℓ.

We store the string $\mathbf{M}$ using Fact 8.7. Note that $|\mathbf{M}| \leq 2n$ and $\mathrm{rle}(M) = \mathcal{O}(k)$, so this takes $\mathcal{O}(k(\log \frac{n+k}{k} + \log|\Sigma|))$ bits. Additionally, we for each non-uniform majority string $M_{\ell,i}$, we store its starting position. This set is implemented using Fact 8.8, so the space complexity is $\mathcal{O}(k(\log \frac{n+k}{k}))$.

Finally, we store integers n, $p_1 = d_1$, $d = d_s$, as well as $\frac{d_\ell}{d_{\ell+1}}$ and $r_\ell := (\frac{p_{\ell+1}}{d_{\ell+1}})^{-1} \mod \frac{d_\ell}{d_{\ell+1}}$ for $1 \leq \ell < s$. A naive estimation of the required space is $\mathcal{O}(s \log n) = \mathcal{O}(\log^2 n)$ bits, but variable-length encoding lets us store the value $\frac{d_\ell}{d_{\ell+1}}$ using $\mathcal{O}(\sum_{\ell=1}^{s} \log \frac{d_\ell}{d_{\ell+1}}) = \mathcal{O}(\log n)$ bits. Similarly, the integers r_ℓ can be stored in $\mathcal{O}(\log n)$ bits, because $1 \leq r_\ell < \frac{d_\ell}{d_{\ell+1}}$.

This completes the description of our data structure. Its overall size in bits is bounded by $\mathcal{O}(k(\log \frac{n+k}{k} + \log|\Sigma|))$, due to $\log n \leq k \log \frac{n+k}{k}$.

8.5 Queries

In this section, we describe the query algorithm for a given index i.

We are going to iterate for $\ell = 1$ to s, and for each ℓ we will either learn $X[i]$ or find out that $X[i]$ is the majority character of $\mathcal{C}_{d_\ell}(i)$. Consequently, entering iteration ℓ, we already know that $X[i]$ is the majority of $\mathcal{C}_{d_{\ell-1}}(i)$. We also assume that d_ℓ is available at that time.

We compute the starting position of $M_{\ell,i \bmod d_\ell}$ in $\mathbf{M}$ according to the formulae given in Section 8.4. Next, we query the data structure of Fact 8.8 to find out if the majority string is uniform. If so, we conclude that $X[i]$ is the majority of $\mathcal{C}_{d_\ell}(i)$ and we may proceed to the next level. Before this, we need to compute $d_{\ell+1} = d_\ell \cdot (\frac{d_\ell}{d_{\ell+1}})^{-1}$, An iteration takes $\mathcal{O}(1)$ time in this case.

Otherwise, we need to learn the majority of $\mathcal{C}_{d_{\ell-1}}(i)$, which is guaranteed to be equal to $X[i]$. This value is $M_{\ell,i \bmod d_\ell}[j]$ where $j = \lfloor \frac{i}{d_1} \rfloor$ for $\ell = 1$, and $j = r_\ell \lfloor \frac{i}{d_\ell} \rfloor \bmod \frac{d_{\ell-1}}{d_\ell}$ for $\ell > 1$. We know the starting position of $M_{\ell,i \bmod d_\ell}$ in $\mathbf{M}$, so we just use Fact 8.7 to retrieve $X[i]$ in $\mathcal{O}(\log k)$ time using Fact 8.7.

If the query algorithm completes all the s iterations without exiting, then $X[i]$ is guaranteed to be the majority character of $\mathcal{C}_d(i)$. Thus, we retrieve $\mathbf{M}[i \bmod d]$ using Fact 8.7. This character is either the majority of $\mathcal{C}_d(i)$ (guaranteed to be equal to $X[i]$) or a blank character. In the latter case, we know that $d > k$ and the class $\mathcal{C}_d(i)$ is uniform.

Thus, the algorithms always returns $X[i]$ or states that $d > k$ and $\mathcal{C}_d(i)$ is uniform. The overall running time is $\mathcal{O}(s + \log k) = \mathcal{O}(\log n)$. This completes the proof of Lemma 8.2.

9 An $\Omega(k \log \frac{n}{k}(\log \frac{n}{k} + \log |\Sigma|))$ bits conjectured lower bound for the streaming k-mismatch problem

In order to support Conjecture 1.5 we show that the lower bound holds under the assumption that, at some point in its execution, the algorithm computes all alignments with Hamming distance at most k of the pattern and a substring of the text of equal length along with the associated mismatch information.

Lemma 9.1. *Any solution for Problem 1.1 that computes all alignments with Hamming distance at most k of the pattern and a substring of the text of equal length along with the associated mismatch information must use at least $\Omega(k \log \frac{n}{k}(\log \frac{n}{k} + \log |\Sigma|))$ bits of space.*

Proof. We prove the space lower bound by showing an explicit set of patterns for which any encoding of all the k-mismatch alignments between prefixes of the pattern P and suffixes of the text T, along with the mismatch information, must use $\Omega(k \log \frac{n}{k}(\log \frac{n}{k} + \log |\Sigma|))$ bits. We define our string recursively. Consider a base string $S_0 = 0^k$. To create S_{i+1} we make three copies of S_i and concatenate them to each other to make $S_i S_i S_i$. We then choose $\lfloor \frac{1}{2}k \rfloor$ indices at random from the middle copy of S_i and randomly change the symbols at those indices. Let S_i' be this modified middle third so that $S_{i+1} = S_i S_i' S_i$. There are $\Theta(\log \frac{n}{k})$ levels to the recursion, so for the final string there are therefore $\Omega(\log \frac{n}{k})$ alignments at which the Hamming distance is at most k. Mismatch information for each of them contains $\lfloor \frac{1}{2}k \rfloor$ randomly chosen indices and $\lfloor \frac{1}{2}k \rfloor$ random symbols which need to be reported. This gives a lower bound of $\Omega(k \log \frac{n}{k}(\log \frac{n}{k} + \log |\Sigma|))$ bits in total needed for any encoding. □

A Appendix: Algebraic Algorithms on $\mathbb{F}_p$

In this appendix, we recall several classic problems in computer algebra involving a prime field $\mathbb{F}_p$, which arise in Section 6. The time and space complexities of their solutions depend on the relation between the field size p and the input size, as well as on the model of computation. Below, we state these complexities for the setting used throughout the paper, which is as follows: We assume the word RAM model with word size w, which supports constant-time arithmetic and bitwise operations on w-bit integers.

Lemma A.1 (Integer multiplication; Schönhage and Strassen [34, 27])**.** *The product of two n-bit integers can be computed in $\mathcal{O}(n)$ time using $\mathcal{O}(n)$ bits of space provided that $w = \Omega(\log n)$.*

Next, we consider operations in the field $\mathbb{F}_p$ where p is a prime number with $\log p = \Theta(w)$, and the corresponding ring of polynomials $\mathbb{F}_p[X]$. We always assume that the degrees of input polynomials are bounded by $n = 2^{\mathcal{O}(w)}$. Polynomial multiplication is a basic building block of almost all efficient algebraic algorithms on $\mathbb{F}_p$. Our model of computation allows for the following efficient solution:

Corollary A.2 (Polynomial multiplication)**.** *Given two polynomials $A, B \in \mathbb{F}_p[X]$ of degree at most n, the product $A \cdot B$ can be computed in $\mathcal{O}(n \log p)$ time using $\mathcal{O}(n \log p)$ bits of space.*

Proof. Polynomial multiplication in $\mathbb{Z}[X]$ can be reduced to integer multiplication via the Kronecker substitution [29]. More precisely, a polynomial $P(X) = \sum_{i=0}^{n} p_i X^i$ of degree n with $0 \le p_i < N$ is represented as an integer with $n+1$ blocks of $1 + 2 \log N + \log n$ bits each so the binary encoding of p_i is stored in the i-th least significant block. To multiply polynomials in $\mathbb{F}_p[X]$, we can compute the product in $\mathbb{Z}[X]$ and then replace each coefficient by its remainder modulo p. □

In Section 6, we use efficient solutions to three classic problems listed below. The original papers refer provide the space complexity in terms of the time $M(n)$ of polynomial multiplication in $\mathbb{F}_p$. The space complexity is not specified explicitly; one can retrieve it by analysing the structure of the original algorithms and their subroutines, such as multi-point evaluation, division, and gcd computation of polynomials. We refer to a textbook [35] for detailed descriptions of these auxiliary procedures as well as of the Cantor–Zassenhaus algorithm.

Lemma A.3 (Polynomial factorization; Cantor–Zassenhaus [6])**.** *Given a polynomial $A \in \mathbb{F}_p[X]$ of degree n with n distinct roots, all the roots of A can be identified in $\mathcal{O}(n \log^3 p)$ time using $\mathcal{O}(n \log p)$ bits of space. The algorithm may fail (report an error) with probability inverse polynomial in p.*

Proof. The Cantor–Zassenhaus algorithm proceeds in several iterations; see [35]. Each iteration involves $\log p$ multiplications and divisions of degree-$\mathcal{O}(n)$ polynomials, as well as several gcd computations involving polynomials of total degree $\mathcal{O}(n)$. The overall time of these operations is $\mathcal{O}(n \log^2 p + n \log n \log p) = \mathcal{O}(n \log^2 p)$. Each step can be interpreted as a random partition of the set of roots of A into two subsets. The computation terminates when every two roots are separated by at least one partition. After $\Omega(\log p)$ phases this condition is not satisfied with probability inverse polynomial in p. □

Lemma A.4 (BCH Decoding; Pan [30])**.** *Consider a sequence $s_j = \sum_{i=0}^{n-1} \alpha_i \cdot \beta_i^j \in \mathbb{F}_p$ with distinct values β_i and coefficients $\alpha_i \neq 0$ for $n < p$. Given the values $s_0, s_1, \ldots, s_{2n}$, the polynomial $\prod_{i=1}^{n}(1 - X\beta_i)$ can be computed in $\mathcal{O}(n \log p)$ time using $\mathcal{O}(n \log p)$ bits of space.*

Lemma A.5 (Transposed Vandermonde matrix-vector multiplication; Canny–Kaltofen–Lakshman [5])**.** *Consider a sequence $s_j = \sum_{i=0}^{n-1} \alpha_i \cdot \beta_i^j \in \mathbb{F}_p$ with distinct values $\beta_i \in \mathbb{F}_p$ and arbitrary coefficients $\alpha_i \in \mathbb{F}_p$. Given the values $\alpha_0, \ldots, \alpha_{n-1}$ and $\beta_0, \ldots, \beta_{n-1}$, the coefficients $s_0, \ldots, s_{n-1}$ can be computed in $\mathcal{O}(n \log n \log p)$ time using $\mathcal{O}(n \log p)$ bits of space.*

Lemma A.6 (Solving transposed Vandermonde systems; Kaltofen–Lakshman [25])**.** *Consider a sequence $s_j = \sum_{i=0}^{n-1} \alpha_i \cdot \beta_i^j \in \mathbb{F}_p$ with distinct values $\beta_i \in \mathbb{F}_p$ and arbitrary coefficients $\alpha_i \in \mathbb{F}_p$. Given the values $s_0, \ldots, s_{n-1}$ and $\beta_0, \ldots, \beta_{n-1}$, the coefficients $\alpha_0, \ldots, \alpha_{n-1}$ can be retrieved in $\mathcal{O}(n \log n \log p)$ time using $\mathcal{O}(n \log p)$ bits of space.*

References

[1] K. Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, 1987.

[2] Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *Journal of Algorithms*, 50(2):257–275, 2004.

[3] Danny Breslauer and Zvi Galil. Real-time streaming string-matching. In *CPM '11: Proc. 22nd Annual Symp. on Combinatorial Pattern Matching*, pages 162–172, 2011.

[4] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640, 1999.

[5] John F. Canny, Erich Kaltofen, and Yagati N. Lakshman. Solving systems of nonlinear polynomial equations faster. In Gaston H. Gonnet, editor, *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation, ISSAC '89, Portland, Oregon, USA, July 17-19, 1989*, pages 121–128. ACM, 1989.

[6] David G Cantor and Hans Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation*, pages 587–592, 1981.

[7] Amit Chakrabarti and Oded Regev. An optimal lower bound on the communication complexity of gap-hamming-distance. *SIAM Journal on Computing*, 41(5):1299–1317, 2012.

[8] D. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *SODA '96: Proc. 7th ACM-SIAM Symp. on Discrete Algorithms*, pages 383–391, 1996.

[9] R. Clifford, M. Jalsenius, E. Porat, and B. Sach. Space lower bounds for online pattern matching. *Theoretical Computer Science*, 483:58–74, 2013.

[10] Raphaël Clifford, Klim Efremenko, Benny Porat, and Ely Porat. A black box for online approximate pattern matching. In *CPM '08: Proc. 19th Annual Symp. on Combinatorial Pattern Matching*, pages 143–151, 2008.

[11] Raphaël Clifford, Klim Efremenko, Ely Porat, and Amir Rothschild. From coding theory to efficient pattern matching. In *SODA '09: Proc. 20th ACM-SIAM Symp. on Discrete Algorithms*, pages 778–784, 2009.

[12] Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. Dictionary matching in a stream. In *ESA '15: Proc. 23rd Annual European Symp. on Algorithms*, pages 361–372, 2015.

[13] Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. The k-mismatch problem revisited. In *SODA '16: Proc. 27th ACM-SIAM Symp. on Discrete Algorithms*, pages 2039–2052, 2016.

[14] Raphaël Clifford and Benjamin Sach. Pseudo-realtime pattern matching: Closing the gap. In *CPM '10: Proc. 21st Annual Symp. on Combinatorial Pattern Matching*, pages 101–111, 2010.

[15] Raphaël Clifford and Tatiana A. Starikovskaya. Approximate hamming distance in a stream. In *ICALP '16: Proc. 43rd International Colloquium on Automata, Languages and Programming*, pages 20:1–20:14, 2016.

[16] Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974.

[17] F. Ergun, Elena Grigorescu, Erfan Sadeqi Azer, and Samson Zhou. Streaming periodicity with mismatches. In *RANDOM '17: Proc. 21st Intl. Workshop on Randomization and Computation*, 2017.

[18] F. Ergun, H. Jowhari, and M. Sağlam. Periodicity in streams. In *RANDOM '10: Proc. 14th Intl. Workshop on Randomization and Computation*, pages 545–559, 2010.

[19] Robert M. Fano. On the number of bits required to implement an associative memory. Computation Structures Group Memo 61, Massachusetts Institute of Technology, August 1971.

[20] Paweł Gawrychowski and Przemysław Uznański. Optimal trade-offs for pattern matching with k mismatches. *arXiv preprint arXiv:1704.01311*, 2017.

[21] Daniel Gorenstein and Neal Zierler. A class of error-correcting codes in p^m symbols. *Journal of the Society for Industrial and Applied Mathematics*, 9(2):207–214, 1961.

[22] Wei Huang, Yaoyun Shi, Shengyu Zhang, and Yufan Zhu. The communication complexity of the Hamming distance problem. *Information Processing Letters*, 99(4):149–153, 2006.

[23] Markus Jalsenius, Benny Porat, and Benjamin Sach. Parameterized matching in the streaming model. In *STACS '13: Proc. 30th Annual Symp. on Theoretical Aspects of Computer Science*, pages 400–411, 2013.

[24] Thathachar S. Jayram and David P. Woodruff. Optimal bounds for Johnson–Lindenstrauss transforms and streaming problems with subconstant error. *ACM Transactions on Algorithms (TALG)*, 9(3):26, 2013.

[25] Erich Kaltofen and Yagati N. Lakshman. Improved sparse multivariate polynomial interpolation algorithms. In Patrizia M. Gianni, editor, *Symbolic and Algebraic Computation, International Symposium ISSAC'88, Rome, Italy, July 4-8, 1988, Proceedings*, volume 358 of *Lecture Notes in Computer Science*, pages 467–474. Springer, 1988.

[26] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249 –260, 1987.

[27] D. E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms.* Addison-Wesley Longman Publishing Co., Inc., 1997.

[28] S. R. Kosaraju. Efficient string matching. Manuscript, 1987.

[29] L. Kronecker. Grundzüge einer arithmetischen Theorie der algebraischen Grössen. *Journal für die reine und angewandte Mathematik*, 92:1–122, 1882.

[30] Victor Y. Pan. Faster solution of the key equation for decoding bch error-correcting codes. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 168–175, New York, NY, USA, 1997. ACM Press.

[31] W. Wesley Peterson. Encoding and error-correction procedures for the Bose–Chaudhuri codes. *IRE Trans. Information Theory*, 6(4):459–470, 1960.

[32] Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *FOCS '09: Proc. 50th Annual Symp. Foundations of Computer Science*, pages 315–323, 2009.

[33] Jakub Radoszewski and Tatiana Starikovskaya. Streaming k-mismatch with error correcting and applications. In *Data Compression Conference (DCC), 2017*, pages 290–299, 2017.

[34] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3-4):281–292, 1971.

[35] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra (3. ed.)*. Cambridge University Press, 2013.