Clifford, R., Kociumaka, T., & Porat, E. (2019). The streaming k-mismatch problem. In *30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2019)* (pp. 1106-1125). Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9781611975482.68

The streaming k-mismatch problem

Raphaël Clifford* Tomasz Kociumaka† Ely Porat‡

Abstract

We consider the streaming complexity of a fundamental task in approximate pattern matching: the k-mismatch problem. In this problem, we must compute Hamming distances between a pattern of length n and all length-n substrings of a text for which the Hamming distance does not exceed a given threshold k. In our problem formulation, we report not only the Hamming distance but also, on demand, the full *mismatch information*, that is the list of mismatched pairs of symbols and their indices. The twin challenges of streaming pattern matching derive from the need both to achieve small working space and also to guarantee that every arriving input symbol is processed quickly.

We present a streaming algorithm for the k-mismatch problem which uses $\mathcal{O}(k \log n \log \frac{n}{k})$ bits of space and spends $\mathcal{O}(\log \frac{n}{k}(\sqrt{k \log k}+\log^3 n))$ time on each symbol of the input stream. In our formulation, the pattern is also in the stream, arriving directly before the text. The running time almost matches the classic offline solution [5] and the space usage is within a logarithmic factor of optimal. Our new algorithm therefore effectively resolves and also extends a problem first introduced in FOCS'09 [38]. En 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 of a length-n pattern within a text of length $\mathcal{O}(n)$. This secondary result provides an optimal solution to a natural encoding 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 has been a recent resurgence of interest in the complexity of the most basic problems in the field. This has been partly been fuelled by the discovery of multiple lower bounds conditioned on the hardness of a small set of well-known problems with naive solutions notoriously resistant to any significant improvement [2, 3, 8, 6, 1, 9, 18]. Pattern matching has also proved to be a rich ground for exploring the time and space complexity of streaming algorithms [38, 19, 23, 29, 7, 16, 20, 17, 22], and it is this line of research that we follow.

We consider the most basic similarity measure between a pattern and substrings of a longer text: that of computing all Hamming distances between the pattern and equal-length substrings of the text. In the streaming k-mismatch problem, the input strings arrive one symbol at a time and the task is to output the Hamming distance between the length-n pattern and the latest length-n suffix of the text, provided that it does not exceed a threshold k specified in advance.

The problem of computing the exact Hamming distances between a pattern and every length-n substring of a text of length $\mathcal{O}(n)$ has been studied in the standard offline model for over 30 years. In 1987, $\mathcal{O}(n\sqrt{n \log n})$-time solutions were first developed [4, 33]. Motivated by the need to find close matches quickly, from there the focus moved to the bounded k-mismatch version of the problem. For many years, the fastest solution ran in $\mathcal{O}(nk)$ time [35]. It was not until SODA'00, when a breakthrough result gave $\mathcal{O}(n\sqrt{k \log k})$ time [5]. Much later, an $\mathcal{O}(k^2 \log k + n \operatorname{polylog} n)$-time solution was developed [17], and in parallel to our work the running time was subsequently improved to $\mathcal{O}((n + k\sqrt{n}) \operatorname{polylog} n)$ [25]. The latter paper also provides some evidence that further progress in the offline model may be difficult to achieve.

Considered as an online or streaming problem with one text symbol arriving at a time, the k-mismatch problem admits a *linear-space* solution running in $\mathcal{O}(\sqrt{k} \log k + \log n)$ worst-case time per arriving symbol, as shown in 2010 [19]. Porat and Porat at FOCS'09 [38] 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. Prior to our work, the state-of-the-art complexity for streaming k-mismatch had two different time-space trade-offs. The problem can be solved in either $\mathcal{O}(k^2 \operatorname{polylog} n)$ space and $\mathcal{O}(\sqrt{k} \log k + \operatorname{polylog} n)$ time per arriving symbol [17] or, as shown very recently, in $\mathcal{O}(k \operatorname{polylog} n)$ time and space [26]. As we describe below, our solution not only achieves the best of both of these complexities[1] but it also tackles a harder version of the k-mismatch problem.

In our problem formulation, we add two generalisations. First, our algorithm reports not only the Hamming distance but also the full *mismatch information*— the list of mismatched pairs of symbols and their in-

*Department of Computer Science, University of Bristol, UK.
†Institute of Informatics, University of Warsaw, Poland.
‡Department of Computer Science, Bar-Ilan University, Israel.

[1] Our algorithm also improves the complexities of both by $\operatorname{polylog} n$ factors

dices. The best previous streaming k-mismatch algorithm with this extra feature is an $\mathcal{O}(k^2 \operatorname{polylog} n)$-space and $\mathcal{O}(k \operatorname{polylog} n)$-time solution given at DCC'17 [39]. Second, and uniquely amongst existing streaming approximate pattern matching algorithms, our algorithm requires no offline and potentially expensive preprocessing of the pattern. We assume that the pattern precedes the text in the input stream and we process it in a streaming fashion.

The twin challenges of streaming pattern matching stem from the need to optimise both working space and worst-case running time for every arriving symbol of the text. An important feature of the streaming model is that we must account for all the space used and cannot, for example, store a copy of the pattern. The question therefore naturally arises of what the minimum space is that one needs to solve the problem.

One can derive a space lower bound for any streaming problem by looking at a related encoding or one-way communication problem. The randomised one-way communication complexity of determining if the Hamming distance between two strings is greater than k is known to be $\Omega(k)$ bits with an upper bound of $\mathcal{O}(k \log k)$ bits [28]. In our problem formulation, however, we report not only the Hamming distance but also the full *mismatch information*—the list of mismatched pairs of symbols and their indices. In this situation, one can derive a slightly higher space lower bound of $\Omega(k(\log \frac{n}{k} + \log |\Sigma|))$ bits[2].

1.1 Our Result In this paper, we take a significant step towards resolving the time and space complexity of the streaming k-mismatch pattern matching problem.

PROBLEM 1.1. (STREAMING k-MISMATCH PROBLEM) *Consider a pattern of length n and a longer text which arrive in a stream one symbol at a time. The streaming k-mismatch problem asks after each arriving symbol of the text whether the current suffix of the text has Hamming distance at most k with the pattern and if so, it also asks to return the corresponding mismatch information.*

Our main result is an algorithm for the streaming k-mismatch problem which uses nearly optimal working space and matches within log factors the running time of the fastest online linear-space solution as well as the $\mathcal{O}(\sqrt{k} \operatorname{polylog} n)$ complexity from SODA'00 [5]. As mentioned previously and unlike the previous solutions for streaming k-mismatch (see e.g. [38, 17]), the algorithm we describe also allows us to report the mismatched symbols (and their indices) at each k-mismatch alignment.

THEOREM 1.2. *There exists a streaming k-mismatch algorithm which uses $\mathcal{O}(k \log n \log \frac{n}{k})$ bits of space and takes $\mathcal{O}(\log \frac{n}{k}(\sqrt{k \log k} + \log^3 n))$ time per arriving symbol. The algorithm is randomised and errs with probability inverse polynomial in n, i.e., its answers are correct with high probability. For each reported occurrence, the mismatch information can be reported on demand in $\mathcal{O}(k)$ time.*

While processing the pattern, our algorithm works under the same restrictions on space consumption and per-symbol running time as when processing the text.

1.2 Overview of the Techniques The overall approach we take refers back to the original streaming exact matching algorithms of [38, 7]. In particular, we do not follow the model of the existing streaming k-mismatch algorithms which all rely on concurrently solving several instances of the 1-mismatch problem. We therefore avoid the overheads and complexity associated with performing multiple stream matching in limited space. In order to achieve our time and space improvements, we develop a number of new ideas and techniques which we believe may have applications more broadly. The first is a randomised $\mathcal{O}(k \log n)$-bit 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 the related mismatch information. Our sketch has a number of desirable algorithmic properties, including the ability to be efficiently maintained subject to concatenation and prefix removal.

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 sliding window at each new alignment. This is the central obstacle in streaming pattern matching which has to be overcome.

Following the model of previous work on streaming exact pattern matching (see [38, 7]), we introduce a family of $\mathcal{O}(\log n)$ prefixes P_ℓ of the pattern with exponentially increasing lengths. We organise the algorithm into several *levels*, with the ℓth level responsible for finding the k-mismatch occurrences of P_ℓ. The task of the

[2]For a single alignment with Hamming distance k, $\log_2 \binom{n}{k} = \Omega(k \log \frac{n}{k})$ bits are required to represent the set of mismatch indices, and each of the k mismatched symbols requires $\Omega(\log |\Sigma|)$ bits to be represented, where Σ denotes the input alphabet. From this, we derive the same lower bound for the space required by any streaming k-mismatch algorithm. We assume throughout that $|\Sigma|$ is bounded by a polynomial in n.

next level is therefore to check, after $|P_{\ell+1}| - |P_\ell|$ subsequent symbols are read, which of these occurrences can be extended to k-mismatch occurrences of $P_{\ell+1}$. The key challenge is that the k-mismatch occurrences of P_ℓ have to be stored in a space-efficient way. In the exact setting, their starting positions form an arithmetic progression, but the presence of mismatches leads to a highly irregular structure making this challenge considerably more difficult.

The task of storing k-mismatch occurrences of a pattern in a space-efficient representation can be expressed in terms of a natural encoding problem which may be of independent interest. Our solution for this problem is both deterministic and asymptotically optimal. One striking property of this result is that the encoding upper bound matches the lower bound we gave earlier for two strings of exactly the same length. In other words, we require no more space to report the mismatch information at all k-mismatch alignments than we do to report it at only one such alignment.

THEOREM 1.3. *Given a pattern P of length n and a text T of length $\mathcal{O}(n)$, the alignments of the pattern and the text with at most k mismatches, as well as the applicable mismatch information, can be encoded using $\mathcal{O}(k(\log \frac{n}{k} + \log|\Sigma|))$ bits, where Σ is the input alphabet.*

As the main conceptual step in our proof of Theorem 1.3, we introduce modified versions of the pattern and text which are highly compressible but still contain sufficient information. More specifically, we place sentinel symbols at some positions of the text and the pattern, making sure that no new k-mismatch occurrences are introduced and that the mismatch information for the existing occurrences does not change. We then develop a key data structure (specified in Lemma 5.3) which lets us store the modified or *proxy* pattern in a space-efficient way. On the other hand, the relevant part of the text is covered by a constant number of k-mismatch occurrences of the pattern, so the modified text can be retrieved based on the proxy pattern and the mismatch information for these $\mathcal{O}(1)$ occurrences. We use this data to find all the k-mismatch occurrences of the pattern in the text along with their mismatch information.

We go on to show that both the encoding and decoding steps can be implemented quickly and in small space. The key tool behind the efficient decoding is a new algorithm for the k-mismatch problem in a setting with read-only random access to the input strings. This is potentially easier compared to the streaming model as maintaining the sketch of a sliding window now requires just $\mathcal{O}(k \log n)$ bits. This simple method would, however, be too slow for our purposes, and so in Theorem 5.6 we give a faster solution for the read-only model. Then in Proposition 3.3 we use this to give an algorithmic counterpart of Theorem 1.3, which is not only time- and space-efficient but which also lets us store the k-mismatch occurrences of prefixes of the pattern and retrieve them later on when they are going to be necessary. In our main algorithm, we apply it to store the k-mismatch occurrences of P_ℓ until we can try extending them to k-mismatch occurrences of $P_{\ell+1}$.

In order to guarantee that we can always process every symbol of the text in $\mathcal{O}(\sqrt{k}\operatorname{polylog} n)$ time rather than in $\mathcal{O}(k \operatorname{polylog} n)$ time, we develop a new procedure for matching strings with small approximate periods. The difficulty with such strings is that their k-mismatch occurrences may appear very frequently. Given in Theorem 4.2, our solution is based on a novel adaptation of Abrahamson's algorithm [4] designed for space-efficient convolution of sparse vectors. We apply it at the lowest level of our streaming algorithm so that at the higher levels we can guarantee that the k-mismatch occurrences of P_ℓ start at least k positions apart.

In summary, our main contribution is given by Theorem 1.2, but in order to achieve this, we have developed a number of new tools and techniques to improve the time and space of previous approaches. These include a new sliding window sketch in Proposition 3.1, a new small approximate period algorithm in Theorem 4.2, the small space encoding of Theorem 1.3 and, most importantly, our key technical innovation given by Proposition 3.3. This last result demonstrates that despite few structural properties, overlapping k-mismatch occurrences admit a very space-efficient representation with a convenient algorithmic interface.

2 Exact Streaming Pattern Matching

As a warm-up, we recall a streaming algorithm for the exact pattern matching problem and formulate it using our own notation. The input stream in this problem consists of a pattern P (of length n) followed by a text T, with a separator symbol between the two. While scanning T, we need to decide if P matches the current length-n suffix of T. In other words, having read the character $T[i]$, we verify whether $P = T[i-n+1\,..\,i]$, that is, if P *occurs in* T *at position* $i-n+1$.

The algorithm described below uses $\mathcal{O}(\log^2 n)$ bits of space and takes $\mathcal{O}(\log n)$ time per arriving symbol, which matches the original complexity by Porat and Porat [38]. However, we rely on the newer implementation by Breslauer and Galil [7, Section 3], which supports efficient preprocessing of the pattern; its running time can be improved to $\mathcal{O}(1)$, but this is out of our focus.

The constraints on the working space do not allow storing the pattern P, so we have to use hashing

even to to decide if $T = P$. The standard scheme is based on polynomial hashes known as *Karp–Rabin fingerprints* [31]. Below, we formally state some basic properties of this scheme: the fingerprints guarantee low collision probability and they can be efficiently maintained subject to concatenation and the inverse operations of prefix and suffix removal.

FACT 2.1. ([7, THEOREM 2.1, LEMMA 2.2, AND COROLLARY 2.3]) *Consider positive integers n and σ such that $\sigma = n^{\mathcal{O}(1)}$, and a family $\mathcal{U} = \{0, \ldots, \sigma - 1\}^{\leq n}$ of strings of length at most n. One can assign $\mathcal{O}(\log n)$-bit* fingerprints *$\Psi(U)$ to all $U \in \mathcal{U}$ so that:*

(a) if $U, V \in \mathcal{U}$, then $\Psi(U) = \Psi(V)$ with high probability implies $U = V$ (for any constant c fixed in advance, $\Pr[\Psi(U) = \Psi(V)] \leq \frac{1}{n^c}$ holds if $U \neq V$);

(b) if $U, V, UV \in \mathcal{U}$, then each of the fingerprints $\Psi(U)$, $\Psi(V)$, or $\Psi(UV)$, can be constructed in $\mathcal{O}(1)$ time given the other two fingerprints.

A naive streaming exact matching algorithm using Karp–Rabin fingerprints stores $\Psi(P)$ and maintains the fingerprint $\Psi(T[i-n+1\,..\,i])$ while scanning the text T. This approach (the Karp–Rabin algorithm [31]) takes $\mathcal{O}(\log n)$ bits of working space if random access to T is allowed, but it is not suitable for the streaming model because it essentially requires storing $T[i-n+1\,..\,i]$. It would be better if we were able to maintain $\Psi(T[0\,..\,i])$ and make sure that $\Psi(T[0\,..\,i-n])$ is available unless we already know that $P \neq T[i-n+1\,..\,i]$. From those two fingerprints, we could compute the fingerprint $\Psi(T[i-n+1\,..\,i])$, which could then be compared to the fingerprint of the pattern. The challenge, of course, is to do this quickly and in small space.

To implement this idea, the authors of [38, 7] introduce a family of prefixes $P_0, \ldots, P_L$ of P so that $L = \lceil \log n \rceil$, $P_L = P$, and $|P_\ell| = 2^\ell$ for $\ell < L$. The algorithm is then organised into $L + 1$ levels, with the ℓth level responsible for locating the occurrences of P_ℓ. The subsequent $(\ell + 1)$th level then verifies which of these occurrences can be extended to occurrences of $P_{\ell+1}$. For convenience, we assume that each occurrence of $P_\ell = T[i\,..\,i + |P_\ell| - 1]$ is reported along with the fingerprint $\Psi(T[0\,..\,i-1])$. In other words, the output of the ℓth level simply contains the relevant fingerprints needed to search in the next level up. This is how we define the *stream of occurrences* of P_ℓ in T:

DEFINITION 2.2. *The* stream of occurrences *of a pattern Q in a text T is a sequence Occ_Q such that $\mathsf{Occ}_Q[i] = (i, \Psi(T[0\,..\,i-1]))$ if $Q = T[i\,..\,i + |Q| - 1]$ and $\mathsf{Occ}_Q[i]$ is* empty *(denoted $\mathsf{Occ}_Q[i] = \perp$) otherwise.*

The value $\mathsf{Occ}_{P_\ell}[i]$ is very useful while processing $T[i + |P_{\ell+1}| - 1]$, but the ℓth level reports it earlier, as soon as $T[i + |P_\ell| - 1]$ is read. Hence, central to the algorithm is a sliding-window *buffer* that can store any δ subsequent entries of the stream of occurrences. The value δ is the *length* of the buffer and the only operation supported by the buffer is called a *push*. If the buffer stores $\mathsf{Occ}_Q[i], \ldots, \mathsf{Occ}_Q[i + \delta - 1]$, then the push takes $\mathsf{Occ}_Q[i + \delta]$ from the input and reports $\mathsf{Occ}_Q[i]$ to the output. The following fact is based on the observation that exact match locations that are not too far apart occur in an arithmetic sequence and hence the buffer can be encoded efficiently.

FACT 2.3. ([7, LEMMA 3.3]) *The stream of exact occurrences of Q in T can be stored in a sliding-window buffer of fixed length $\delta = \mathcal{O}(|Q|)$ which uses $\mathcal{O}(\log n)$ bits and takes $\mathcal{O}(1)$ time per push.*

Having described all the building blocks, we conclude with a complete description of the resulting algorithm. In the preprocessing phase, we compute the fingerprints $\Psi(P_\ell)$ for $\ell = 0, \ldots, L$ with $L = \lceil \log n \rceil$. Each level ℓ of the algorithm receives fingerprints from the start of the text to the locations of the occurrences of $P_{\ell-1}$ and attempts to extend those occurrences to occurrences of the longer prefix P_ℓ.

In more detail, each arriving character $T[i]$ of the text is processed as follows. First, the fingerprint $\Psi(T[0\,..\,i-1])$ is extended to $\Psi(T[0\,..\,i])$ using Fact 2.1(b). Next, the 0th level compares $T[i]$ with $P_0 = P[0]$ and reports $\mathsf{Occ}_{P_0}[i] \in \{\perp, (i, \Psi(T[0\,..\,i-1])\}$ based on the outcome. The character $T[i]$ is then processed by subsequent levels for $\ell = 1$ to L. Each such level receives $\mathsf{Occ}_{P_{\ell-1}}[i - |P_{\ell-1}| + 1]$ and puts this value into a buffer using the push operation of Fact 2.3. The buffer has length $|P_\ell| - |P_{\ell-1}| = \mathcal{O}(|P_{\ell-1}|)$ so that it returns $\mathsf{Occ}_{P_{\ell-1}}[i - |P_\ell| + 1]$ in exchange. This tells us if $P_{\ell-1}$ occurs in T at position $i - |P_\ell| + 1$ and if so, gives us the relevant fingerprint.

If $\mathsf{Occ}_{P_{\ell-1}}[i - |P_\ell| + 1]$ is empty, then so is $\mathsf{Occ}_{P_\ell}[i - |P_\ell| + 1]$ because $P_{\ell-1}$ is a prefix of P_ℓ. Otherwise, we use the fingerprint arithmetic of Fact 2.1(b) to determine $\Psi(T[i - |P_\ell| + 1\,..\,i])$ from $\mathsf{Occ}_{P_{\ell-1}}[i - |P_\ell| + 1] = (i - |P_\ell| + 1, \Psi(T[0\,..\,i - |P_\ell|])$ and $\Psi(T[0\,..\,i])$. Depending on the result, either $\perp$ or $\mathsf{Occ}_{P_{\ell-1}}[i - |P_\ell| + 1]$ is reported as $\mathsf{Occ}_{P_\ell}[i - |P_\ell| + 1]$ and so the computation for level ℓ is completed. By Fact 2.1(a), the algorithm is correct with high probability; it uses $\mathcal{O}(\log^2 n)$ bits ($\mathcal{O}(\log n)$ bits per level) and takes $\mathcal{O}(\log n)$ time to process any position of the text.

3 Streaming k-Mismatch in $\mathcal{O}(k \log^2 n)$ bits and $\mathcal{O}(k \log^4 n)$ time per symbol

In this section, we describe our first streaming algorithm for the k-mismatch problem. It uses $\mathcal{O}(k \log^2 n)$ bits of

space, which is optimal up to log factors, but the per-symbol running time of $\mathcal{O}(k \log^4 n)$ is far from that of the linear-space procedures. In contrast to the previous streaming k-mismatch algorithms, our approach shares its high-level structure with the classic procedures for exact pattern matching, as described in Section 2. This is possible due to novel tools generalising Facts 2.1 and 2.3.

Our first contribution is a rolling k-mismatch sketch, which can be applied to check whether two given strings are at Hamming distance k or less and, if so, to retrieve the mismatches. We formally define the mismatch information between two equal-length strings U and V as $\mathrm{MI}(U,V) = \{(i, U[i], V[i]) : U[i] \neq V[i]\}$. Their Hamming distance is $\mathrm{HD}(U,V) = |\,\mathrm{MI}(U,V)\,|$.

PROPOSITION 3.1. *Consider positive integers k, n, σ such that $k \leq n$ and $\sigma = n^{\mathcal{O}(1)}$, and a family $\mathcal{U} = \{0, \ldots, \sigma-1\}^{\leq n}$ of strings of length at most n. One can assign $\mathcal{O}(k \log n)$-bit* sketches $\mathrm{sk}_k(U)$ *to all $U \in \mathcal{U}$ so that:*

(a) if $U, V \in \mathcal{U}$, then given sketches $\mathrm{sk}_k(U)$ and $\mathrm{sk}_k(V)$, in $\mathcal{O}(k \log^3 n)$ time one can decide with high probability whether $\mathrm{HD}(U,V) \leq k$ and report $\mathrm{MI}(U,V)$ in case of a positive answer;

(b) if $U, V, UV \in \mathcal{U}$, then each of the sketches $\mathrm{sk}_k(U)$, $\mathrm{sk}_k(V)$, or $\mathrm{sk}_k(UV)$, can be constructed in $\mathcal{O}(k \log n)$ time given the other two.

These procedures use $\mathcal{O}(k \log n)$ bits of working space.

Our construction is based on the ideas behind Reed–Solomon error correcting codes [40] and resembles earlier sketches of Clifford et al. [15]. However, the latter are not *rolling sketches*—they fail to satisfy condition (b). It turns out that to implement the underlying procedure, we need to perform computations in a field with a characteristic at least n, whereas the sketches of [15] operate on fields with characteristic two. This change also causes one extra difficulty in that we have to use a randomised polynomial factorisation algorithm in the implementation of operation (a). The construction of the k-mismatch sketches, including the proof of Proposition 3.1, is deferred to Section 6.

With sketches as a generalisation of fingerprints to the k-mismatch setting, it is natural to define a *stream of k-mismatch occurrences* in analogy with Definition 2.2. Since we require the mismatch information to be present in the output, we include it in this stream.

DEFINITION 3.2. *The* stream of k-mismatch occurrences *of a pattern Q in a text T is a sequence Occ^k_Q such that $\mathsf{Occ}^k_Q[i] = (i, \mathrm{MI}(Q, T[i\,..\,i+|Q|-1]), \Psi(T[0\,..\,i-1]))$ if $\mathrm{HD}(Q, T[i\,..\,i+|Q|-1]) \leq k$ and $\mathsf{Occ}^k_Q[i]$ is* empty *(denoted $\mathsf{Occ}^k_Q[i] = \bot$) otherwise.*

Our central technical contribution is a counterpart of Fact 2.3: a small sliding-window buffer for the stream of k-mismatch occurrences. The overview of the ideas behind this result is given in Section 5, with technical details deferred to Sections 8 and 9.

PROPOSITION 3.3. *The stream of k-mismatch information of Q in T can be stored in a sliding-window buffer of fixed length $\delta = \Theta(|Q|)$ which uses $\mathcal{O}(k \log n)$ bits. The push operation takes $\mathcal{O}(k \log^2 n + \log^3 n)$ time if the pushed entry $\mathsf{Occ}^k_Q[i+\delta]$ or the retrieved entry $\mathsf{Occ}^k_Q[i]$ is non-empty and $\mathcal{O}(\sqrt{k \log k} + \log^3 n)$ time otherwise. Initialisation, given $\mathrm{sk}_k(Q)$ and δ, takes $\mathcal{O}(k)$ time.*

The structure of the algorithm of Section 2 can now be reused to solve the streaming k-mismatch problem. A minor difference is that the stream of k-mismatch occurrences also contains the mismatch information, which needs to be retrieved from Proposition 3.1(a). Moreover, for reasons explained at the end of Section 5, the buffer of Proposition 3.3 does not support $\delta = o(|Q|)$, which could happen if the whole pattern P is only slightly longer than the penultimate prefix P_{L-1}. We simply remove the penultimate level so that $|P_\ell| - |P_{\ell-1}| = \Theta(|P_{\ell-1}|)$ holds for each level $\ell > 0$. At each of the $\mathcal{O}(\log n)$ levels, the complexity is dominated by the cost of the sketch arithmetic of Proposition 3.1.

COROLLARY 3.4. *There exists a streaming k-mismatch algorithm which uses $\mathcal{O}(k \log^2 n)$ bits of space and takes $\mathcal{O}(k \log^4 n)$ time per arriving symbol. The algorithm is randomised and its answers are correct with high probability. It reports all the k-mismatch occurrences with the relevant mismatch information.*

4 Streaming k-Mismatch in $\mathcal{O}(k \log n \log \frac{n}{k})$ bits and $\mathcal{O}(\log \frac{n}{k}(\sqrt{k \log k} + \log^3 n))$ time per symbol

In this section, we improve the running time of the algorithm developed in Section 3. As a side effect, the space consumption is also slightly decreased. First, let us discuss the bottleneck of our base solution. Observe that when the ℓth level processes $T[i]$, then it already takes $\tilde{\mathcal{O}}(\sqrt{k})$ time if $\mathsf{Occ}^k_{P_{\ell-1}}[i - |P_{\ell-1}| + 1] = \mathsf{Occ}^k_{P_{\ell-1}}[i - |P_\ell| + 1] = \bot$, that is when there is no matching prefix from the level below. Hence, the amortised running time of the ℓth level is $\tilde{\mathcal{O}}(\sqrt{k})$ as long as the k-mismatch occurrences of $P_{\ell-1}$ are located sparsely enough. Unfortunately, this condition is never satisfied for $|P_{\ell-1}| \leq k$ and it does not need to be satisfied even for $P_L = P$.

However, if a pattern Q has two nearby k-mismatch occurrences, then we can encode it in small space as described below. Recall that a string Q of length m has a *period* $p > 0$ if $Q[0\,..\,m-p-1] =$

$Q[p\,..\,m-1]$. The d-periods describe analogous structure for the setting with mismatches: We say that a positive integer p is a *d-period* of the string Q if $\mathrm{HD}(Q[0\,..\,m-p-1], Q[p\,..\,m-1]) \leq d$.

Observation 4.1. *If a pattern Q has k-mismatch occurrences at positions ℓ, ℓ' of T satisfying $\ell < \ell' < \ell + |Q|$, then $\ell' - \ell$ is a $2k$-period of Q.*

A string Q with a d-period p can be stored using an $\mathcal{O}(d \log \frac{m}{d} + (d+p)\log|\Sigma|)$-bit *periodic representation with respect to p*, which consists of $\mathrm{MI}(Q[0\,..\,m-p-1], Q[p\,..\,m-1])$ and $Q[0\,..\,p-1]$.

Already in the algorithm of [17], patterns with a small approximate period require special treatment in order to guarantee $\tilde{\mathcal{O}}(\sqrt{k})$ per-character processing time; they also constitute the bottleneck of the solution by Golan et al. [26]. We develop a new deterministic procedure to deal with patterns with a d-period p such that both $d = \mathcal{O}(k)$ and $p = \mathcal{O}(k)$. Its running time essentially matches that of [17], but the space consumption is improved to $\mathcal{O}(k)$ machine words from $\mathcal{O}(k^2)$. Our subroutine can also be plugged into [26] to improve their amortised running time from $\tilde{\mathcal{O}}(k)$ to $\tilde{\mathcal{O}}(\sqrt{k})$. The following result gives our streaming algorithm for the small approximate-period case.

Note that the algorithm is extended to allow for outputting the stream Occ_P^k with a delay $\delta = \mathcal{O}(|P|)$ so that $\mathsf{Occ}_P^k[i-|P|+1]$ is reported when the algorithm reads $T[i+\delta]$. For $\delta = \Theta(|P|)$, we could achieve this using Proposition 3.3, but in certain special cases we need $\delta = o(|P|)$ as well.

Theorem 4.2. (Streaming algorithm for patterns with a small approximate period) *Suppose that we are given an integer k and the periodic representation of a pattern P with respect to a d-period p such that $d = \mathcal{O}(k)$ and $p = \mathcal{O}(k)$. There exists a deterministic algorithm, which uses $\mathcal{O}(k \log n)$ bits of space and takes $\mathcal{O}(\sqrt{k \log k} + \log^2 n)$ time per symbol to report the k-mismatch occurrences of P in the streamed text T.*

The stream Occ_P^k of k-mismatch occurrences can be outputted in $\mathcal{O}(k \log^2 n)$ extra time for each k-mismatch occurrence. Moreover, the algorithm may report it with any prescribed delay $\delta = \mathcal{O}(|P|)$.

The new procedure, developed in Section 7, relies on the observation that a function $\Delta_p[P]$, defined as $P[i+p] - P[i]$ for each index i, has $\mathcal{O}(p+d)$ non-zero values. Moreover, $\Delta_p[T]$ locally enjoys a similar property for fragments containing k-mismatch occurrences of P. It also turns out that the Hamming distances between the subsequent alignments of P in T can be expressed in terms of the analogous values for $\Delta_p[P]$ and $\Delta_p[T]$. We compute the latter based on a novel adaptation of Abrahamson's algorithm [4] designed for space-efficient convolution of sparse vectors.

If the pattern P has a small approximate period, then Theorem 4.2 solves the streaming k-mismatch problem. Otherwise, we use it to replace the lowest levels of the general-purpose algorithm of Section 3: we apply it for a prefix of P which does not have any $2k$-period $p \leq k$, but has an k'-period $p' \leq k$ for $2k+1 \leq k' = \mathcal{O}(k)$. Such a prefix needs to be computed in the preprocessing, which motivates the following result. Its relatively straightforward proof is also given in Section 7.

Lemma 4.3. *There exists a deterministic streaming algorithm that, given positive integers p and $d = \mathcal{O}(p)$, finds the longest prefix Y of the input string X which has a d-period $p' \leq p$. It reports the periodic representation of Y with respect to p', uses $\mathcal{O}(p)$ words of space, and takes $\mathcal{O}(\sqrt{p \log p})$ per-symbol processing time plus $\mathcal{O}(p\sqrt{p \log p})$ post-processing time.*

With k-mismatch occurrences of each prefix P_ℓ at least k-positions apart, each level now runs in $\tilde{\mathcal{O}}(\sqrt{k})$ amortised time. To achieve this bound for the whole algorithm, we need to make sure that the sketch $\mathrm{sk}_k(T[0\,..\,i])$ is updated only when we need to verify a potential k-mismatch occurrence of P_ℓ. We use the following result based on efficient computation of the sketch $\mathrm{sk}_k(V)$ for $|V| \leq k$.

Fact 4.4. *There exists a streaming algorithm that processes a string U in $\mathcal{O}(k \log n)$ bits of space using $\mathcal{O}(\log^2 n)$ time per character so that the sketch $\mathrm{sk}_k(U)$ can be retrieved on demand in $\mathcal{O}(k \log^2 n)$ time.*

The aforementioned tools let us transform the algorithm of Section 3 so that the amortised running time is brought down to $\tilde{\mathcal{O}}(\sqrt{k})$. Processing some individual symbols still requires $\tilde{\mathcal{O}}(k)$ time, but processing any subsequent k symbols takes $\tilde{\mathcal{O}}(k\sqrt{k})$ time, so we can think of the algorithm as processing the input with a varying delay between 0 and k. To achieve worst-case time per symbol, we set the last prefix P_L to be of length $n-2k$ and naively check which k-mismatch occurrences of P_L extend to k-mismatch occurrences of P. The delay is cancelled during this verification.

We can now give a complete description of our main result, an $\mathcal{O}(\log \frac{n}{k}(\sqrt{k \log k} + \log^3 n))$-time and $\mathcal{O}(k \log n \log \frac{n}{k})$-bit streaming algorithm for the k-mismatch problem.

4.1 Processing the Pattern Let us first describe the information about the pattern that we gather. If we discover that the pattern P has an $\mathcal{O}(k)$-period

$p \le k$, then we store P using the periodic representation with respect to p. For the sub-case where $|P| \le 2k$, we can trivially construct the periodic representation of P with respect to $p = 1$ in order the satisfy the requirements of Theorem 4.2. In either case this finishes the processing of the pattern.

Otherwise, we introduce a family of $\mathcal{O}(\log \frac{n}{k})$ prefixes $P_0, \ldots, P_L$ chosen so that P_0 is the longest prefix of P with a $(2k+1)$-period $p \le k$, $|P_L| = n - 2k$, and $|P_1|, \ldots, |P_{L-1}|$ are the subsequent powers of two between $|P_0|$ and $\frac{1}{2}|P_L|$ (exclusive). We store the periodic representation of P_0, the sketches $\mathrm{sk}_k(P_\ell)$ for $1 \le \ell \le L$, and $P[n-2k\,..\,n-1]$.

Because $|P| > 2k$ no output is required for the first $2k$ symbols of the text. As a result, we can start processing the text with a delay of k symbols and catch up while processing the subsequent k symbols of the text. This allows for some post-processing of the pattern before we read the text.

LEMMA 4.5. *The pattern P can be processed by a deterministic streaming algorithm which uses $\mathcal{O}(k \log n \log \frac{n}{k})$ bits of space and takes $\mathcal{O}(\sqrt{k \log k} + \log^2 n)$ time per symbol plus $\mathcal{O}(k\sqrt{k \log k} + k \log^2 n)$ post-processing time.*

Proof. While scanning the pattern, we store a buffer of $2k$ trailing symbols and run the algorithm of Lemma 4.3 to compute the longest prefix P_0 of P with a $(2k+1)$-period $p \le k$. If $|P_0| > n - 2k$, we extend the periodic representation of P_0 to the periodic representation of P, for which p must be a $4k$-period. Otherwise, we set $P_1, \ldots, P_L$ as specified above. To have $\mathrm{sk}_k(P_\ell)$ available, we use Fact 4.4 while scanning P and extract the sketches of every prefix of P whose length is a power of 2 larger than $3k$ (note that $|P_0| \ge 3k$). We run this subroutine with a delay of $2k$ symbols so that $\mathrm{sk}_k(P_L)$ can be retrieved as soon as the whole pattern P is read.

4.2 Processing the Text In the small approximate period case, we simply use Theorem 4.2 (with no delay).

Otherwise, the main component outputs the stream $\mathsf{Occ}^k_{P_L}$ of k-mismatch occurrences of P_L. It consists of levels $\ell = 0, \ldots, L$; as in Section 3, the ℓth level reports the stream $\mathsf{Occ}^k_{P_\ell}$ of k-mismatch occurrences of P_ℓ in T.

As far as the implementation is concerned, the main difference is that level 0 uses Theorem 4.2. The subsequent levels process $T[i]$ as in Section 3: Each such level receives $\mathsf{Occ}^k_{P_{\ell-1}}[i - |P_{\ell-1}| + 1]$, puts this value into a buffer of Proposition 3.3 (with length $|P_\ell| - |P_{\ell-1}|$), and receives $\mathsf{Occ}^k_{P_{\ell-1}}[i - |P_\ell| + 1]$ in exchange. If the latter value is $\perp$, then also $\mathsf{Occ}^k_{P_\ell}[i - |P_\ell| + 1] = \perp$. Otherwise, $\mathrm{sk}_k(T[0\,..\,i])$ is retrieved using Fact 4.4 and $\mathrm{sk}_k(T[0\,..\,i - |P_\ell|])$ is obtained from $\mathsf{Occ}^k_{P_{\ell-1}}[i - |P_\ell| + 1]$. The sketch $\mathrm{sk}_k(T[i - |P_\ell| + 1\,..\,i])$ is computed using Proposition 3.1(b) and compared to $\mathrm{sk}_k(P_\ell)$ using Proposition 3.1(a) to determine $\mathsf{Occ}^k_{P_\ell}[i - |P_\ell| + 1]$.

At level 1 we may have $|P_1| - |P_0| = o(|P_0|)$ (only if $L = 1$), so instead of using a buffer of Proposition 3.3, we actually delay the output of level 0 exploiting the feature built into Theorem 4.2.

The space usage of the main component is $\mathcal{O}(k \log n)$ bits per level and the per-symbol running time is $\mathcal{O}(\sqrt{k \log k} + \log^2 n)$ if $\mathsf{Occ}^k_{P_{\ell-1}}[i - |P_{\ell-1}| + 1] = \mathsf{Occ}^k_{P_{\ell-1}}[i - |P_\ell| + 1] = \perp$, and $\mathcal{O}(k \log^3 n)$ otherwise. By Observation 4.1, the latter happens $\mathcal{O}(1)$ times across every k consecutive positions, so the average per-symbol processing time is $\mathcal{O}(\log \frac{n}{k}(\sqrt{k \log k} + \log^3 n))$ across all levels. Moreover, a matching worst-case complexity can be achieved if we allow for a delay at most k, i.e., if we store a buffer of at most k trailing characters of T which are not yet processed by the main component.

Whenever a k-mismatch occurrence of P_L at position i is reported (along with the mismatch information), we must check if it extends to a k-mismatch occurrence of P. It arrives with delay at most k, so we can naively compare $P[n-2k\,..\,n-1]$ with $T[i+n-2k\,..\,i+n-1]$ before the algorithm completes processing $T[i+n-1]$. By Observation 4.1, at most one instance of this procedure needs to be run in parallel, so the extra space complexity is $\mathcal{O}(k \log n)$ bits and the additional per-symbol running time is constant. This completes the proof of Theorem 1.2, our main result.

THEOREM 1.2. *There exists a streaming k-mismatch algorithm which uses $\mathcal{O}(k \log n \log \frac{n}{k})$ bits of space and takes $\mathcal{O}(\log \frac{n}{k}(\sqrt{k \log k} + \log^3 n))$ time per arriving symbol. The algorithm is randomised and errs with probability inverse polynomial in n, i.e., its answers are correct with high probability. For each reported occurrence, the mismatch information can be reported on demand in $\mathcal{O}(k)$ time.*

5 Encoding Nearby k-Mismatch Occurrences

In this section, we discuss the key proof ideas for our central technical contribution, Proposition 3.3. We focus on the space complexity claim and argue for its correctness by proving Theorem 1.3. We conclude with a glimpse into how we establish the running time in Proposition 3.3, whose proof is deferred to Section 9.

5.1 Proof of Theorem 1.3

THEOREM 1.3. *Given a pattern P of length n and a text T of length $\mathcal{O}(n)$, the alignments of the pattern and the text with at most k mismatches, as well as the applicable mismatch information, can be encoded using $\mathcal{O}(k(\log \frac{n}{k} + \log |\Sigma|))$ bits, where Σ is the input alphabet.*

Let us first recall the corresponding encoding for $k = 0$. If $t < t'$ are subsequent starting positions of occurrences of P in T, then the difference $t' - t$ either exceeds $\frac{1}{2}|P|$ or is equal to *the shortest period* $\mathrm{per}(P)$ of the pattern P; see [7, Lemma 3.1]. Consequently, if $|T| \leq \frac{3}{2}|P|$, then the starting positions of the exact occurrences of P in T form an arithmetic progression whose difference is $\mathrm{per}(P)$. Hence, one only needs to store the positions ℓ, ℓ' of the leftmost and rightmost occurrences as well as the period $\mathrm{per}(P)$. If the actual text is longer, it is split into substrings of length $\frac{3}{2}|P|$ with overlaps of length $|P|$; thus, we apply the encoding a constant number of times.

In the presence of mismatches, it is actually more convenient to make an even stronger assumption that $|T| \leq \frac{5}{4}|P|$. We analogously define ℓ and ℓ' as the positions of the first and the last k-mismatch occurrence of P in T. Mismatches make the structure of occurrences much more complicated, but we may still consider the smallest arithmetic sequence containing all the positions of k-mismatch occurrences of P in T'. Its difference d is the greatest common divisor of the distances between the subsequent starting positions. We can use this arithmetic sequence to filter out alignments which certainly do not yield k-mismatch occurrences of P in T.

We also observe that the prefix $T[0\,..\,\ell-1]$ and the suffix $T[\ell'+n\,..\,|T|-1]$ are irrelevant, so we may focus on $T' = T[\ell\,..\,\ell'+n-1]$. The pattern P may only occur in T' at multiples of d, so we only attempt aligning $P[i]$ with $T[i']$ if $i \equiv i' \pmod{d}$. This motivates the following definition.

Definition 5.1. *Let X be a fixed string. For integers i and $d \geq 0$, the* ith class modulo d *(in X) is defined as a multiset $\mathcal{C}_d(X,i) = \{X[i'] : 1 \leq i' \leq |X|$ and $i' \equiv i \pmod{d}\}$. For $d = 0$, we assume that $i' \equiv i \pmod{0}$ holds if and only if $i = i'$.*

If the multiset $\mathcal{C}_d(P,i) \cup \mathcal{C}_d(T',i)$ is *uniform*, i.e., it contains just one character with positive multiplicity, then these two classes do not participate in any mismatches within the alignments we consider. Thus, we do not need to remember the unique character of $\mathcal{C}_d(P,i) \cup \mathcal{C}_d(T',i)$. In other words, we may replace P with a *proxy pattern* $P_\#$ and T' with a *proxy text* $T'_\#$ so that $P[i]$ and $T'[i]$ are replaced by a sentinel symbol $\#_{i \bmod d}$ if $\mathcal{C}_d(P,i) \cup \mathcal{C}_d(T',i)$ is uniform. The main property of these strings is that in any occurrence of P in T', we have not altered the symbols involved in a mismatch, while matching symbols could only be replaced by sentinels in a consistent way.

Fact 5.2. *The pattern P has a k-mismatch occurrence at position j of T if and only if $P_\#$ has a k-mismatch occurrence at position $j - \ell$ of $T'_\#$. Moreover, in that case we have*

$$\mathrm{MI}(P, T[j\,..\,j+n-1]) = \mathrm{MI}(P_\#, T'_\#[j-\ell\,..\,j-\ell+n-1]).$$

Proof. Suppose that P has a k-mismatch occurrence at position j of T. By definition of $\mathcal{P}$, we have $\ell \leq j \leq \ell'$ and $j \equiv \ell \pmod{d}$. Consequently, P has a k-mismatch occurrence at position $j' := j - \ell$ of T' satisfying $d \mid j'$. If $P[i] \neq T'[j'+i]$, then the class $\mathcal{C}_d(P,i) \cup \mathcal{C}_d(T',i)$ contains at least two distinct elements, so $P_\#[i] = P[i] \neq T'[j'+i] = T'_\#[j'+i]$. Otherwise, $P_\#[i] = T'_\#[j'+i]$, with both symbols equal to $\#_{i \bmod d}$ or $P[i] = T'[j'+i]$. Thus, $\mathrm{MI}(P, T[j\,..\,j+n-1]) = \mathrm{MI}(P_\#, T'_\#[j-\ell\,..\,j-\ell+n-1])$ and thus $P_\#$ has a k-mismatch occurrence at position $j' = j - \ell$ of $T'_\#$.

For a proof of the converse implication, suppose $P_\#$ has a k-mismatch occurrence at position j' of $T'_\#$, with $j' = j - \ell$. If $P_\#[i] = T'_\#[j'+i]$, then either $P[i] = P_\#[i] = T'_\#[j'+i] = T'[j'+i]$, or $P_\#[i] = T'_\#[j'+i] = \#_{i \bmod d}$. In the latter case, we conclude that $d \mid j'$ and $\mathcal{C}_d(P,i) \cup \mathcal{C}_d(T',i)$ is uniform, so $P[i] = T'[j'+i]$. Hence, P indeed has a k-mismatch occurrence at position j' of T', i.e., at position j of T.

Next, we show that $P_\#$ and $T'_\#$ can be encoded in $\mathcal{O}(k(\log \frac{n}{k} + \log|\Sigma|))$ bits. We start with encoding the non-uniform classes $\mathcal{C}_d(P,i)$. Here, we rely on the fact that Observation 4.1 yields $2k$-periods of P. Moreover, the assumption $|T| \leq \frac{5}{4}|P|$ lets us focus on the family $\mathrm{Per}_{\leq n/4}(P, 2k)$ of all $2k$-periods p of P such that $p \leq \frac{n}{4}$.

Lemma 5.3. *For a string X of length n and an integer k, let $\mathcal{P} \subseteq \mathrm{Per}_{\leq n/4}(X,k)$ and $d = \gcd(\mathcal{P})$. The characters $X[i]$ in non-uniform classes $\mathcal{C}_d(X,i)$ can be encoded in $\mathcal{O}(k(\log \frac{n}{k} + \log|\Sigma|))$ bits in total.*

In Section 8, we prove a claim which subsumes Lemma 5.3, so here we only illustrate the main points.

Sketch of a proof. We build a sequence $0 = d_0, \ldots, d_s = d$ of $\mathcal{O}(\log n)$ integers such that $d_\ell = \gcd(d_{\ell-1}, p_\ell)$ and $p_\ell \in \mathcal{P}$ for $1 \leq \ell \leq s$. Next, we observe that classes modulo d_ℓ for $\ell = 0, \ldots, s$ form a sequence of partitions of $\{X[i] : 0 \leq i < n\}$, with each partition coarser than the previous one. We keep the *majority*[3] of a class modulo d_ℓ whenever it differs from the majority of the enclosing class modulo $d_{\ell+1}$. Due to $p_\ell \leq \frac{n}{4}$, the majority of $\mathcal{C}_{d_\ell}(i)$ is likely to match the majority of $\mathcal{C}_{d_\ell}(i + p_\ell)$, which lets us store these characters using *run-length encoding*. We also prove that the number non-uniform classes modulo d_s is $\mathcal{O}(k)$, which lets us explicitly store their majorities.

[3] The *majority* element of a multiset S is an element with multiplicity strictly greater than $\frac{1}{2}|S|$. We keep a sentinel character if there is no majority. See Section 8 for details.

We are now ready to describe the encoding behind Theorem 1.3. If P does not occur in T with up to k-mismatches, then we need not encode any information. Otherwise, the encoding consists of the following data:

- the locations ℓ and ℓ' of the leftmost and the rightmost k-mismatch occurrence of P in T, along with the mismatch information $\mathrm{MI}(P, T[\ell\,..\,\ell+n-1])$ and $\mathrm{MI}(P, T[\ell'\,..\,\ell'+n-1])$ for both occurrences;
- the value $d = \gcd(\mathcal{P})$ and the data structure of Lemma 5.3 for $\mathcal{P}$ consisting of distances between locations of k-mismatch occurrences of P in T; $\mathcal{P} \subseteq \mathrm{Per}_{\leq n/4}(P, 2k)$ due to Observation 4.1.

By Lemma 5.3 and due to the size of the mismatch information, the encoding takes $\mathcal{O}(k(\log \frac{n}{k} + \log|\Sigma|))$ bits.

It remains to describe how to decode the k-mismatch occurrences of P in T as well as the applicable mismatch information. We first prove that one can retrieve any symbol of $P_\#$ based on Lemma 5.3 (if $\mathcal{C}_d(P, i)$ is non-uniform) or the mismatch information for the k-mismatch occurrences of P as a prefix and as a suffix of T' (otherwise). Similarly, one can retrieve $T'_\#$ because T' is covered by these two k-mismatch occurrences of P.

FACT 5.4. *Strings $P_\#$ and $T'_\#$ can be retrieved from the described encoding.*

Proof. First, note that $\mathcal{C}_d(P, i) \cup \mathcal{C}_d(T', i)$ is uniform if and only if $\mathcal{C}_d(P, i)$ is uniform and the mismatch information for neither stored k-mismatch occurrence contains $(i', P[i'], T[i''])$ with $i' \equiv i \pmod d$. This lets us retrieve where sentinel symbols occur in $P_\#$ and $T'_\#$.

Next, we shall prove that one can retrieve $P_\#[i] = P[i]$ if $\mathcal{C}_d(P, i) \cup \mathcal{C}_d(T', i)$ is non-uniform. If $\mathcal{C}_d(P, i)$ is non-uniform, we can simply use the data structure of Lemma 5.3. Otherwise, mismatch information for the k-mismatch occurrence of P as a prefix or as a suffix of T' contains $(i', P[i'], T[i''])$ for some $i' \equiv i \pmod d$. The class $\mathcal{C}_d(P, i)$ is uniform, so $P[i] = P[i']$.

Finally, consider retrieving $T'_\#[i] = T'[i]$. If $i < n$, then the decoding procedure can use $P[i]$ and the mismatch information of the k-mismatch occurrence of P as a prefix of T', which might contain $(i, P[i], T'[i])$. On the other hand, for $i' \geq n$ we can use $P[i - \ell' + \ell]$ and the mismatch information $\mathrm{MI}(P, T[\ell'\,..\,\ell'+n-1])$ of the k-mismatch occurrence of P as a suffix of T'.

Consequently, the decoding procedure computes the mismatch information for all the alignments of $P_\#$ in $T'_\#$ and outputs it (with the position shifted by ℓ) whenever there are at most k mismatches. By Fact 5.2, such output coincides with the k-mismatch occurrences of P in T. This concludes the proof of Theorem 1.3.

5.2 Overview of the Proof of Proposition 3.3

Theorem 1.3 is sufficient to establish the buffer size in Proposition 3.3 except that the stream of k-mismatch information also includes the relevant sketches. This increases the space complexity to $\mathcal{O}(k \log n)$ bits.

To implement the buffer's push operation, we need to strengthen Lemma 5.3 by providing efficient encoding and decoding procedures. Note the buffer receives k-mismatch occurrences of P in T one by one. By Observation 4.1, every two overlapping k-mismatch occurrences yield a $2k$-period of P; combining the mismatch information for subsequent occurrences, we can retrieve $\mathrm{MI}(P[1\,..\,n-p], P[p+1\,..\,n])$ for this $2k$-period p. This way, our encoding algorithm extends the family $\mathcal{P} \subseteq \mathrm{Per}_{\leq n/4}(P, 2k)$. As for decoding, we simply make sure that any character $P[i]$ in a non-uniform class $\mathcal{C}_d(P, 2k)$ can be retrieved in $\mathcal{O}(\log n)$ time. This yields a generalisation of Lemma 5.3 proved in Section 8.

LEMMA 5.5. *For a string X and an integer k, let $\mathcal{P} \subseteq \mathrm{Per}_{\leq n/4}(X, k)$ and $d = \gcd(\mathcal{P})$. There is a data structure of size $\mathcal{O}(k(\log \frac{n}{k} + \log|\Sigma|))$ bits which given an index i retrieves $X[i]$ in $\mathcal{O}(\log n)$ time if $\mathcal{C}_d(X, i)$ is non-uniform, and returns a sentinel symbol if the class is uniform. The data structure can be initialised in $\mathcal{O}(1)$ time with $\mathcal{P} = \emptyset$ and updated in $\mathcal{O}(k \log n)$ time subject to adding a k-period $p \in \mathrm{Per}_{\leq n/4}(X, k)$ to $\mathcal{P}$ given $\mathrm{MI}(X[0\,..\,n-p-1], X[p\,..\,n-1])$.*

Nevertheless, we still need to retrieve the k-mismatch occurrences of $P_\#$ in $T'_\#$. For this, we observe that any character of $P_\#$ and $T'_\#$ can be accessed in $\mathcal{O}(\log n)$ time. Thus, it suffices to solve the k-mismatch problem in the *read-only random-access model*. Such an algorithm is not hard to obtain from Proposition 3.1 and Theorem 4.2, but we state is a black box due to potential applicability.

THEOREM 5.6. *In the read-only random-access model, the streaming k-mismatch problem can be solved online with a Monte-Carlo algorithm using $\mathcal{O}(k \log n)$ bits of working space and $\mathcal{O}(\sqrt{k \log k} + \log^3 n)$ time per symbol, including $\mathcal{O}(1)$ symbol reads. For any reported k-mismatch occurrence, the mismatch information can be retrieved on demand in $\mathcal{O}(k)$ time.*

Apart from the mismatch information, the buffer of Proposition 3.3 also requires retrieving sketches $\mathrm{sk}_k(T[0\,..\,i-1])$ for each k-mismatch occurrence $T[i\,..\,i+|P|-1]$ of P. While locating the k-mismatch occurrences can be performed on the proxy text $T'_\#$, the sketches need to be based on the original text T. Resolving this discrepancy requires careful manipulation of sketches, for which we generalise both Proposition 3.1(b) and Fact 4.4. Moreover, we need to implement a buffer as a series of components, each of which

enters a decoding phase only after it completes encoding and performs some relatively costly auxiliary computations. The latter is why we forbid $\delta = o(|P|)$.

6 A Rolling k-mismatch Sketch

In this section, we develop our rolling k-mismatch sketches, thereby proving Proposition 3.1.

We also describe further efficient procedures for efficient manipulation of our sketches, which we use in Section 9 to prove Proposition 3.3. As announced in Section 3, our approach extends the deterministic sketches developed in [15] for the offline k-mismatch with wildcards problem and combines it with the classic Karp–Rabin fingerprints for exact pattern matching [31]; see Fact 2.1.

We fix an upper bound n on the length of the compared strings and a prime number $q > \max(\sigma, n^{c+1})$ for a sufficiently large constant exponent c (which is used to control error probability). We will assume throughout that all the input symbols can be treated as elements of $\mathbb{F}_q$. Recall that the Karp–Rabin fingerprints are defined as $\psi_r(U) = \sum_{i=0}^{|U|-1} U[i] \cdot r^i$ for a uniformly random $r \in \mathbb{F}_q$, and this value is included in our k-mismatch sketches. The fingerprints $\Psi(U)$ of Fact 2.1 beyond $\psi_r(U)$ also contain $r^{|U|}$ and $r^{-|U|}$ to make sure that the running time in Fact 2.1(b) is $\mathcal{O}(1)$ rather than $\mathcal{O}(\log n)$. This is not needed due to the $\mathcal{O}(k \log n)$ time allowed in Proposition 3.1(b).

Definition 6.1. (k-mismatch sketch) *For a fixed prime number q and for $r \in \mathbb{F}_q$ chosen uniformly at random, the sketch $\mathrm{sk}_k(S)$ of a string $S \in \mathbb{F}_q^\ell$ is*

$$\mathrm{sk}_k(S) = (\phi_0(S), \ldots, \phi_{2k}(S), \phi'_0(S), \ldots, \phi'_k(S), \psi_r(S)),$$

where $\psi_r(S) = \sum_{i=0}^{\ell-1} S[i] r^i$, $\phi_j(S) = \sum_{i=0}^{\ell-1} S[i] i^j$ and $\phi'_j(S) = \sum_{i=0}^{\ell-1} S[i]^2 i^j$ for $j \geq 0$.

Observe that the sketch is a sequence of $3k + 3$ elements of $\mathbb{F}_q$, so it takes $\mathcal{O}(k \log q) = \mathcal{O}(k \log n)$ 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. The following lemma proves Proposition 3.1(a).

Lemma 6.2. *Given the sketches $\mathrm{sk}_k(S)$ and $\mathrm{sk}_k(T)$ of two strings of length $\ell \leq n$, in $\mathcal{O}(k \log^3 n)$ time we can decide (with high probability) whether $\mathrm{HD}(S,T) \leq k$. If so, the mismatch information $\mathrm{MI}(S,T)$ is reported. The algorithm uses $\mathcal{O}(k \log n)$ bits of space.*

Proof. First, suppose that $\mathrm{HD}(S,T) = k' < k$. 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:

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

This set of equations is similar to those appearing in [15] and in the decoding procedures for Reed–Solomon codes. We use the standard Peterson–Gorenstein–Zierler procedure [37, 27], with subsequent efficiency improvements. This method consists of the following main steps:

1. Compute the *error locator polynomial* $P(X) = \prod_{i=1}^{k'}(1 - x_i X)$ from the $2k + 1$ *syndromes* $\phi_j(S) - \phi_j(T)$ with $0 \leq j \leq 2k$.
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 the efficient *key equation* solver by Pan [36]. 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 [12], 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 the Kaltofen–Lakshman algorithm [30] 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 $r'_i = S^2[x_i] - T^2[x_i]$:

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

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

If $\mathrm{HD}(S,T) > k$, then we may still run the procedure above, but its behaviour is undefined. This issue is resolved by using the Karp–Rabin fingerprints to help us check if we have found all the mismatches or not.

If the algorithm fails, we may assume that $\mathrm{HD}(S,T) > k$; otherwise, the failure probability is in-

verse 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

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

This verification takes $\mathcal{O}(k' \log n)$ time and its error probability is at most $\frac{\ell}{q}$ which is the collision probability of Karp–Rabin fingerprints for strings of length ℓ.

Unlike in [15], we also need to efficiently maintain sketches subject to operations listed in Proposition 3.1(b). The first claim of the lemma below implements the required procedure, while the second claim is useful in Section 9 to prove Proposition 3.3.

LEMMA 6.3. *The following operations can be implemented in $\mathcal{O}(k \log n)$ time using $\mathcal{O}(k \log n)$ bits of space, provided that all the processed strings belong to $\mathbb{F}_q^*$ and are of length at most n.*

(a) Construct one of the sketches $\mathrm{sk}_k(U)$, $\mathrm{sk}_k(V)$, or $\mathrm{sk}_k(UV)$ given the other two sketches.

(b) Construct $\mathrm{sk}_k(U)$ or $\mathrm{sk}_k(U^m)$ given the other sketch and the integer m.

Proof. **(a)** 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 $j' \le j$:

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

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 $e^{rX} = \sum_{j=0}^{\infty} r^j \frac{X^j}{j!}$. Now, the equality above can be succinctly written as $\Phi(UV) - \Phi(U) = \Phi(V) \cdot e^{|U|X}$. Consequently, the 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 generating functions in $\mathcal{O}(k \log n)$ time using efficient polynomial multiplication over $\mathbb{F}_q$ [42, 32, 34]. The coefficients $\phi'_j(U)$, $\phi'_j(V)$, and $\phi'_j(UV)$, can be computed in the same way.

(b) Observe that $\psi_r(U^m) = \sum_{i=0}^{m-1} r^{i|U|}\psi_r(U) = \frac{r^{m|U|}-1}{r^{|U|}-1}\psi_r(U)$. Thus, $\psi_r(U)$ and $\psi_r(U^m)$ are easy to compute from each other in $\mathcal{O}(\log n)$ time. Next, recall that the exponential generating function $\Phi(S) = \sum_{j=0}^{\infty} \phi_j(S)\frac{X^j}{j!}$ satisfies $\Phi(UV) = \Phi(U) + \Phi(V) \cdot e^{|U|X}$. Consequently, $\Phi(U^m) = \Phi(U) \cdot \sum_{i=0}^{m-1} e^{i|U|X} = \Phi(U^m) \cdot \frac{e^{m|U|X}-1}{e^{|U|X}-1}$. Thus, $\Phi(U)$ and $\Phi(U^m)$ can be computed from each other in $\mathcal{O}(k \log n)$ time using polynomial multiplication over $\mathbb{F}_q$. The first $\mathcal{O}(k)$ terms of the inverse of the power series $(e^{\ell X} - 1)/X$ are also retrieved in $\mathcal{O}(k \log n)$ time using polynomial multiplication and Newton's method for polynomial division; see [43].

Next, we consider the efficiency of updating a sketch given the mismatch information.

LEMMA 6.4. *Let $S, T \in \mathbb{F}_q^*$ be of the same length $\ell < n$. If $\mathrm{HD}(S,T) = \mathcal{O}(k)$, then $\mathrm{sk}_k(T)$ can be constructed in $\mathcal{O}(k \log^2 n)$ time and $\mathcal{O}(k \log n)$ bits of space given $\mathrm{MI}(S,T)$ and $\mathrm{sk}_k(S)$.*

Proof. 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. Hence, this task can be accomplished in $\mathcal{O}((d+k)\log^2 n)$ time using $\mathcal{O}((d+k)\log n)$ bits of space using the Canny–Kaltofen–Lakshman algorithm [11]. The values $\phi'_j(T) - \phi'_j(S)$ are computed analogously.

As a result, the sketch can be efficiently updated subject substituting characters.

COROLLARY 6.5. *A string $X \in \mathbb{F}_q^*$ with $|X| \le n$ can be stored in $\mathcal{O}(k \log n)$ bits so that $\mathrm{sk}_k(X)$ can be retrieved in $\mathcal{O}(k \log^2 n)$ time and the following updates are handled in $\mathcal{O}(\log^2 n)$ time: substitute $X[i] = a$ for $X[i] = b$ given the index i and the symbols $a, b \in \mathbb{F}_q$.*

Proof. We maintain the sketch $\mathrm{sk}_k(Y)$ of a previous version Y of X and a buffer of up to k substitutions required to transform Y into X, i.e., the mismatch list $\mathrm{MI}(X,Y)$. If there is room in the buffer, we simply append a new entry to $\mathrm{MI}(X,Y)$ to handle a substitution. When the buffer becomes full, we apply Lemma 6.4 to compute $\mathrm{sk}_k(X)$ based on $\mathrm{sk}_k(Y)$ and

$\mathrm{MI}(X, Y)$. This computation takes $\mathcal{O}(k \log^2 n)$ time, so we run in it parallel to the subsequent k updates (so that the results are ready before the buffer is full again).

To implement a query, we complete the ongoing computation of $\mathrm{sk}_k(Y)$ and use Lemma 6.4 again to determine $\mathrm{sk}_k(X)$ (and clear the buffer as a side effect). This takes $\mathcal{O}(k \log^2 n)$ time.

In particular, this proves Fact 4.4, where we append characters instead of substituting them.

FACT 4.4. *There exists a streaming algorithm that processes a string U in $\mathcal{O}(k \log n)$ bits of space using $\mathcal{O}(\log^2 n)$ time per character so that the sketch $\mathrm{sk}_k(U)$ can be retrieved on demand in $\mathcal{O}(k \log^2 n)$ time.*

Proof. Observe that appending 0 to X does not change the sketch, so appending a symbol a is equivalent to substituting 0 at position $|X|$ by a. Thus, the claim follows from Corollary 6.5.

7 Patterns with a Small Approximate Period

In this section, we prove Theorem 4.2 and Lemma 4.3, thereby showing how the streaming k-mismatch problem can be solved deterministically when the pattern P has a (potentially unknown) small approximate period. We start with the much simpler proof of Lemma 4.3.

LEMMA 4.3. *There exists a deterministic streaming algorithm that, given positive integers p and $d = \mathcal{O}(p)$, finds the longest prefix Y of the input string X which has a d-period $p' \le p$. It reports the periodic representation of Y with respect to p', uses $\mathcal{O}(p)$ words of space, and takes $\mathcal{O}(\sqrt{p \log p})$ per-symbol processing time plus $\mathcal{O}(p\sqrt{p \log p})$ post-processing time.*

Proof. The constraints on the time and space complexity let us process X in blocks of up to p symbols, spending $\mathcal{O}(p\sqrt{p \log p})$ time on each block.

After reading a block, we compute the longest prefix Y of X with a d-period $p' \le p$, the periodic representation of Y with respect to p', and the values

$$\mathrm{HD}(Y[0\,..\,|Y| - p'' - 1], Y[p''\,..\,|Y| - 1])$$

for $0 < p'' \le p$. When at some iteration we discover that Y is a proper prefix of X, then Y cannot change anymore, so we can ignore the forthcoming blocks.

Thus, below we assume that $X = Y$ before a block B is appended to X. In this case, we need to check if B can be appended to Y as well, i.e., if YB has any d-period $p'' \le p$. We rely on the formula

$$\begin{aligned}\mathrm{HD}((YB)[0\,..\,|YB| - p'' - 1], (YB)[p''\,..\,|YB| - 1]) &= \\ = \mathrm{HD}(Y[0\,..\,|Y| - p'' - 1], Y[p''\,..\,|Y| - 1]) &+ \\ + \mathrm{HD}((YB)[|Y| - p''\,..\,|YB| - p'' - 1], B).&\end{aligned}$$

The left summand is already available, while to determine the right summand for each p'', we use Abrahamson's algorithm [4] to compute the Hamming distance of every alignment of B within the suffix of YB of length $p + |B|$. This procedure takes $\mathcal{O}(p\sqrt{|B| \log |B|})$ time.

If we find out that YB has a d-period $p'' \le p$, we compute the periodic representation of Y with respect to p''. For this, we observe that $Y[i] \neq Y[i - p'']$ may only hold if $i < p' + p''$, $Y[i] \neq Y[i - p']$, $Y[i - p'] \neq Y[i - p' - p'']$, or $Y[i - p' - p''] \neq Y[i - p'']$. Thus, it takes $\mathcal{O}(d + p) = \mathcal{O}(p)$ time to transform the periodic representation of Y with respect to p' to the one with respect to p''. Finally, we append B to Y and update its periodic representation.

Otherwise, we partition B into two halves $B = B_L B_R$ and try appending the left half B_L to Y using the procedure above. We recurse on B_R or B_L depending on whether the algorithm succeeds.

The running time of the ith iteration is $\mathcal{O}\big(p + p\sqrt{\frac{p}{2^i} \log \frac{p}{2^i}}\big)$, because we attempt appending a block of length at most $\lceil \frac{p}{2^i} \rceil$. Consequently, the overall processing time is $\mathcal{O}(p\sqrt{p \log p})$.

The proof of Theorem 4.2 is based on novel ideas and requires combining several notions. In [17], it was proved that any k^2 consecutive values $\mathrm{Ham}_{P,T}[i] := \mathrm{HD}(P, T[i - |P| + 1\,..\,i])$ can be generated in $\mathcal{O}(k^2 \log k)$ time using $\mathcal{O}(k^2)$ words of space if the pattern P and the text T share a d-period p satisfying $d = \mathcal{O}(k)$ and $p = \mathcal{O}(k)$. Below, we show how to compute k subsequent Hamming distances in $\mathcal{O}(k\sqrt{k \log k})$ time using $\mathcal{O}(k)$ words of space, under an additional assumption that the preceding $2p$ Hamming distances are already available.

Our approach resembles Abrahamson's algorithm [4], so let us first recall how the values $\mathrm{Ham}_{P,T}[i]$ can be expressed in terms of convolutions. The *convolution* of two functions $f, g : \mathbb{Z} \to \mathbb{Z}$ is a function $f * g : \mathbb{Z} \to \mathbb{Z}$ such that

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

For a string X and a symbol $a \in \Sigma$, we define the *characteristic function* $X_a : \mathbb{Z} \to \{0, 1\}$ of positions where a occurs in X. In other words, $X_a(i) = 1$ if and only if $0 \le i < |X|$ and $X[i] = a$. The *cross-correlation* of strings T and P is a function $T \otimes P : \mathbb{Z} \to \mathbb{Z}$ defined as $T \otimes P = \sum_{a \in \Sigma} T_a * P_a^R$, where P^R denotes the reverse of P.

FACT 7.1. *We have $(T \otimes P)(i) = |P| - \mathrm{Ham}_{P,T}[i]$ for $|P| - 1 \le i < |T|$ and $(T \otimes P)(i) = 0$ for $i < 0$ and $i \ge |P| + |T|$.*

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

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

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

If a string X has a d-period p, then $X_a(i)$ is typically equal to $X_a(i+p)$. This property can be conveniently formalised using a notion of *finite differences*. For a function $f : \mathbb{Z} \to \mathbb{Z}$ and a positive integer $p \in \mathbb{Z}_+$, we define the *forward difference* $\Delta_p[f] : \mathbb{Z} \to \mathbb{Z}$ as

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

Observation 7.2. *If a string X has a d-period p, then the functions $\Delta_p[X_a]$ have at most $2(d+p)$ non-zero entries in total across all $a \in \Sigma$.*

The following lemma reuses the idea behind the Abrahamson's algorithm to compute the convolution of functions with few non-zero entries.

Lemma 7.3. *Consider functions $f, g : \mathbb{Z} \to \mathbb{Z}$ with n non-zero entries in total. The non-zero entries among 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 linear working space (with respect to the input size plus the output size).*

Proof. We define the *restriction* of a function $h : \mathbb{Z} \to \mathbb{Z}$ to a set $A \subseteq \mathbb{Z}$ so that $h|_A(j) = h(j)$ if $j \in A$ and $h|_A(j) = 0$ otherwise. Let us partition $\mathbb{Z}$ into blocks $B_k = \{\delta k, \ldots, \delta k + \delta - 1\}$ for $k \in \mathbb{Z}$. Moreover, we define $B'_k = \{i - (k+1)\delta + 1, \ldots, i - (k-1)\delta\}$ and observe that $(f*g)(j) = \sum_k (f|_{B_k} * g|_{B'_k})(j)$ for $i \le j < i + \delta$.

We say that a block B_k is *heavy* if $f|_{B_k}$ has at least $\sqrt{\delta \log \delta}$ non-zero entries. 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, which is $\mathcal{O}(n\sqrt{\delta \log \delta})$ in total because there are $\mathcal{O}(n/\sqrt{\delta \log \delta})$ heavy blocks.

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 $\mathcal{O}(n\sqrt{\delta \log \delta})$.

This is very useful because the forward difference operator commutes with the convolution:

Fact 7.4. *Consider functions $f, g : \mathbb{Z} \to \mathbb{Z}$ with finite support and a positive integer p. We have $\Delta_p[f*g] = f * \Delta_p[g] = \Delta_p[g] * f$. Consequently, $\Delta_p[f] * \Delta_p[g] = \Delta_p[\Delta_p[f*g]]$.*

Proof. Note that

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

By symmetry, we also have $\Delta_p[f*g] = \Delta_p[f]*g$. Hence, $\Delta_p[f] * \Delta_p[g] = \Delta_p[\Delta_p[f] * g] = \Delta_p[\Delta_p[f*g]]$.

The function $\Delta_p[\Delta_p[h]]$, called the *second forward difference* of $h : \mathbb{Z} \to \mathbb{Z}$, is denoted $\Delta_p^2[h]$; observe that $\Delta_p^2[h](i) = h(i+2p) - 2h(i+p) + h(i)$.

Combining Lemma 7.3, Fact 7.4, and the notions introduced above, we can compute the second forward differences of the cross-correlation between P and T efficiently and in small space:

Corollary 7.5. *Suppose that p is a d-period of 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 $\mathcal{O}(\delta + (d+p)\sqrt{\delta \log \delta})$ time using $\mathcal{O}(d+p+\delta)$ words of space.*

Proof. The functions $\Delta_p[P_a^R]$ have $2(d+p)$ non-zero entries in total, and the functions $\Delta_p[T_a]$ enjoy the same property. Hence, it takes $\mathcal{O}((d+p)\sqrt{\delta \log \delta})$ time in total to compute all the non-zero entries among $(\Delta_p[T_a] * \Delta_p[P_a^R])(j)$ for $a \in \Sigma$ and $i \le j < i + \delta$ using Lemma 7.3. Finally, we observe that $\Delta_p^2[T \otimes P] = \sum_{a\in\Sigma} (\Delta_p[T_a] * \Delta_p[P_a^R])$ by Fact 7.4.

Fact 7.1 and Corollary 7.5 can be applied to compute the subsequent Hamming distances $\mathrm{Ham}_{P,T}[i]$ provided that P and T share a common d-period p. These values can be generated in $\mathcal{O}(\sqrt{(d+p)\log(d+p)})$ amortised time using $\mathcal{O}(d+p)$ words of space, with $\Theta(d+p)$ consecutive Hamming distances actually computed in every iteration. In Lemma 7.6, we adapt this approach to the streaming setting, where $\mathrm{Ham}_{P,T}[i]$

needs to be known before $T[i+1]$ is revealed. To deal with this, we use a two-part partitioning known as the *tail trick*. Similar ideas have already been used to deamortise streaming pattern matching algorithms; see [7, 16, 17, 19].

LEMMA 7.6. *Let P be a pattern with a d-period p. Suppose that p is also an $\mathcal{O}(d+p)$-period of the text T. There exists a deterministic streaming algorithm which processes T using $\mathcal{O}(d+p)$ words of space and $\mathcal{O}(\sqrt{(d+p)\log(d+p)})$ time per symbol and reports $\mathrm{Ham}_{T,P}[i]$ for each position $i \geq |P|-1$.*

Proof. First, we assume that blocks of $d+p$ symbols $T[i\,..\,i+d+p-1]$ 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 $i-2p \leq j < i$ (initialised as zeroes for $i=0$; this is valid due to Fact 7.1). The space consumption is $\mathcal{O}(d+p)$.

When a block arrives, we update the periodic representation of T and apply Corollary 7.5 to compute $\Delta_p^2[T \otimes P](j)$ for $i \leq j < i+d+p$. Based on the stored values of $T \otimes P$, this lets us retrieve $(T \otimes P)(j)$ for $i \leq j < i+d+p$. 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 $j < i+d-p$. Such an iteration takes $\mathcal{O}((d+p)\sqrt{(d+p)\log(d+p)})$ time and uses $\mathcal{O}(d+p)$ working space.

Below, we apply this procedure in a streaming algorithm which processes T symbol by symbol. The first step is to observe that if the pattern length is $\mathcal{O}(d+p)$, we can compute the Hamming distance online using $\mathcal{O}(d+p)$ words of space and $\mathcal{O}(\sqrt{(d+p)\log(d+p)})$ worst-case time per arriving symbol [14]. We proceed under the assumption that $|P| > 2(d+p)$.

We partition the pattern into two parts: the *tail* P_T—the suffix of P of length $2(d+p)$, and the *head* P_H—the prefix of P length $|P|-2(d+p)$. Observe that $\mathrm{Ham}_{P,T}[j] = \mathrm{Ham}_{P_T,T}[j] + \mathrm{Ham}_{P_H,T}[j-2(d+p)]$. Moreover, we can compute $\mathrm{Ham}_{P_T,T}[j]$ using the online algorithm of [14]; this takes $\mathcal{O}(d+p)$ words of space and $\mathcal{O}(\sqrt{(d+p)\log(d+p)})$ time per symbol.

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

The overall space usage is $\mathcal{O}(d+p)$ words and the worst-case time per arriving symbol is $\mathcal{O}(\sqrt{(d+p)\log(d+p)})$, dominated by the online procedure of [14] and by processing blocks in the background.

To prove Theorem 4.2, we need to waive the assumption that p is an approximate period of the text. Nevertheless, we note that p must be a $(d+k)$-period of any fragment matching P with k mismatches. Thus, our strategy is to identify approximately periodic fragments of T guaranteed to contain all k-mismatch occurrences of P; Lemma 7.6 is then called for each such fragment.

THEOREM 4.2. (STREAMING ALGORITHM FOR PATTERNS WITH A SMALL APPROXIMATE PERIOD) *Suppose that we are given an integer k and the periodic representation of a pattern P with respect to a d-period p such that $d = \mathcal{O}(k)$ and $p = \mathcal{O}(k)$. There exists a deterministic algorithm, which uses $\mathcal{O}(k \log n)$ bits of space and takes $\mathcal{O}(\sqrt{k \log k} + \log^2 n)$ time per symbol to report the k-mismatch occurrences of P in the streamed text T.*

The stream Occ_P^k of k-mismatch occurrences can be outputted in $\mathcal{O}(k \log^2 n)$ extra time for each k-mismatch occurrence. Moreover, the algorithm may report it with any prescribed delay $\delta = \mathcal{O}(|P|)$.

Proof. Our strategy is to partition T into overlapping blocks for which p is an $\mathcal{O}(k)$-period, making sure that any k-mismatch occurrence of P is fully contained within a block. Then, we shall run Lemma 7.6 for each block to find these occurrences.

Let us first build the partition into blocks. We shall make sure that every position of T belongs to exactly two blocks and that p is a $(4k+2d+p)$-period of each block. While processing T, we maintain two current blocks B, B' (assume $|B| \geq |B'|$) along with their periodic representations. Let us denote the number of mismatches (with respect to the approximate period p) in B and B' by m and m', respectively. These values shall always satisfy $m' \leq d+2k$ and $m \leq m' + \min(|B'|, p) + d + 2k$, which clearly guarantees the claimed bound $m \leq 4k+2d+p$. Moreover, we shall make sure that $d+2k < m$ unless B is a prefix of T. This way, if a k-mismatch occurrence of P ends at the currently processed position, then it must be fully contained in B, because $\mathrm{HD}(P,Q) \leq k$ implies that p is a $(d+2k)$-period of Q, for any string Q.

We start with $B = B' = \varepsilon$ before reading $T[0]$. Next, suppose that we read a symbol $T[i]$. If $m' < d+2k$ or $T[i] = T[i-p]$, we simply extend B and B' with $T[i]$. In this case, m' might increase but it will not exceed $d+2k$, whereas m increases only if $m' + \min(|B'|, p)$ increases, so the inequality $m \leq m' + \min(|B'|, p) + d + 2k$ remains satisfied. On the other hand, if $m' = d+2k$ and $T[i] \neq T[i-p]$, then we set $B := B'T[i]$ and $B' := T[i]$. In this case, $m = d+2k+1$ and $m' = 0$, which satisfies the invariants as one can easily verify.

The procedure described above outputs the blocks as streams, which we pass to Lemma 7.6 with a delay δ. In order to save some space, when the construction of a block terminates and the block turns out to be shorter than $|P|$, we immediately launch a garbage collector to get rid of this block. The number of remaining blocks contained in $T[i-\delta+1\,..\,i]$ is therefore bounded by $2\lfloor\frac{\delta}{|P|}\rfloor$, because each such block is of length at least $|P|$ and each position is located within at most two such blocks. Accounting for the two blocks currently in construction and the two blocks currently processed by Lemma 7.6, this implies that at any time we store $\mathcal{O}(1+\frac{\delta}{|P|})=\mathcal{O}(1)$ blocks in total, which means that the overall space consumption is bounded by $\mathcal{O}(k)$ words.

Lemma 7.6 reports k-mismatch occurrences of P with no delay, so the overall delay of the algorithm is precisely δ. Requests for mismatch information are handled in $\mathcal{O}(k)$ time using the periodic representations of the pattern P and the currently processed block.

To allow for computing sketches, we also maintain an instance of Fact 4.4 and for every block $T[b\,..\,e]$, we store the sketch $\mathrm{sk}_k(T[0\,..\,b-1])$. As we stream the block to Lemma 7.6, we process it using another instance of Fact 4.4, with an extra delay $|P|$ (compared to Lemma 7.6). This way, whenever Lemma 7.6 reports a k-mismatch occurrence $T[j\,..\,j+|P|-1]$, the sketch $T[0\,..\,j-1]$ can be retrieved (on demand) in $\mathcal{O}(k\log^2 n)$ time (by combining $\mathrm{sk}_k(T[0\,..\,b-1])$ and $\mathrm{sk}_k(T[b\,..\,j-1])$ based on Proposition 3.1(b)). The use of Fact 4.4 increases the processing time of each position by an additive $\mathcal{O}(\log^2 n)$ term.

Note that a combination of Lemma 4.3 and Theorem 4.2 lets us give a deterministic streaming k-mismatch algorithm when the pattern has a d-period p with $d=\mathcal{O}(k)$ and $p=\mathcal{O}(k)$. The procedure of Lemma 4.3 can be called to find an $\mathcal{O}(k)$-period $p'=\mathcal{O}(k)$ of P (along with the periodic representation of P), and then T can be processed using Theorem 4.2.

8 Proof of Lemma 5.5

In this section, we prove Lemma 5.5. We define $\|\mathcal{C}_p(i)\|$ as the number of distinct elements in $\mathcal{C}_p(i)$; note that a class is *uniform* if $\|\mathcal{C}_p(i)\|=1$. 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 symbols of a string in an analogous way.

The remaining part of this section constitutes a proof of Lemma 5.5. 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. In Section 8.5 we address answering queries, and in Section 8.6 we discuss updates.

8.1 Overview Observe that if $d=\gcd(\mathcal{P})$ does not change as we insert an approximate period p to $\mathcal{P}$, then we do not need to update the data structure. Hence, let us introduce a sequence $d_0,\ldots,d_s$ of *distinct* values $\gcd(\mathcal{P})$ arising as we inserted subsequent approximate periods to $\mathcal{P}$. Moreover, for $1\le i\le s$, let $p_i\in\mathcal{P}$ be the period which caused the transition from d_{i-1} to d_i.

Fact 8.1. *The sequences $d_0,\ldots,d_s$ and $p_1,\ldots,p_s$ satisfy $d_0=0$, $d_\ell=\gcd(d_{\ell-1},p_\ell)$ for $1\le\ell\le s$, and $d_s=\gcd(\mathcal{P})$. Moreover, $d_\ell\mid d_{\ell-1}$ and $d_\ell\le\frac{n}{2^{\ell+1}}$ for $1\le\ell\le s$, and therefore $s=\mathcal{O}(\log n)$.*

Proof. We start with $\mathcal{P}=\emptyset$, so $d_0=\gcd(\emptyset)=0$. If $\gcd(\mathcal{P})\mid p$ for a newly inserted element p, we do not update the sequence. Otherwise, we append $p_\ell:=p$ and $d_\ell:=\gcd(d_{\ell-1},p_\ell)$. Note that d_ℓ is a proper divisor of $d_{\ell-1}$, so $d_\ell\le\frac{1}{2}d_{\ell-1}$ which yields $d_\ell\le\frac{d_1}{2^{\ell-1}}\le\frac{n}{2^{\ell+1}}$ by induction.

Let $\mathbf{C}_\ell$ be the partition of the symbols of X into classes $\mathcal{C}_{d_\ell}(i)$ modulo d_ℓ. Fact 8.1 lets us characterise the sequence $\mathbf{C}_0,\ldots,\mathbf{C}_s$: the first partition, $\mathbf{C}_0$, consists of singletons, i.e., it is 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_s.

Consequently, the classes modulo $\mathcal{C}_{d_\ell}(i)$ for $0\le\ell\le s$ form a laminar family, which can be represented as a forest of depth $s+1=\mathcal{O}(\log n)$; its leaves are single symbols (classes modulo $d_0=0$), while the roots are classes modulo d_s. Let us imagine that each class stores its majority element (or a sentinel # if there is no majority). 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 classes $\mathcal{C}_{d_{\ell-1}}(i')$ 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 $\mathcal{C}_{d_s}(i)$, because our procedure shall return # when $\mathcal{C}_{d_s}(i)$ is uniform. On the other hand, we explicitly store the non-uniform classes $\mathcal{C}_{d_s}(i)$ so that updates can be implemented efficiently.

In order to encode the majority symbols 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 in more detail.

OBSERVATION 8.2. *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 $\ell \geq 1$ and $0 \leq i < d_\ell$, we define the majority string $M_{\ell,i}$ of length $|M_{\ell,i}| = \frac{d_{\ell-1}}{d_\ell}$ for $\ell > 1$ and $|M_{\ell,i}| = \lceil \frac{n-i}{d_\ell} \rceil$ for $\ell = 1$. Its jth symbol $M_{\ell,i}[j]$ is defined as the majority of $\mathcal{C}_{d_{\ell-1}}(i+jp_\ell)$, or # if the class has no majority. We think of $M_{\ell,i}$ as a cyclic string for $\ell > 1$ and a linear string for $\ell = 1$.

Since $p_\ell \in \mathrm{Per}(X, k)$, we expect that the adjacent symbols of the majority strings $M_{\ell,i}$ are almost always equal. In the next section, we shall prove that the total number of mismatches between adjacent symbols is $\mathcal{O}(k)$ across all the majority strings.

8.2 Combinatorial Bounds For $1 \leq \ell \leq s$, let $\mathbf{N}_\ell \subseteq \mathbf{C}_\ell$ consist of non-uniform classes. Moreover, for $0 \leq \ell < s$, 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 $\mathcal{C}_{d_\ell}(i + p_{\ell+1})$ differ.

FACT 8.3. *Consider the decomposition of a class $\mathcal{C}_{d_\ell}(i) \in \mathbf{N}_\ell$ into classes $C \in \mathbf{C}_{\ell-1}$. At least one of these classes satisfies $C \in \mathbf{N}_{\ell-1} \cup \mathbf{K}_{\ell-1}$. Moreover, if there is just one such class, then $\ell = 1$ or this class C satisfies $C \in \mathbf{N}_{\ell-1} \setminus \mathbf{K}_{\ell-1}$.*

Proof. If the majority string $M_{\ell,i}$ is uniform, then one of the classes C must contain a symbol other than its majority; otherwise, $\mathcal{C}_{d_\ell}(i)$ would be uniform. Such a class C clearly belongs to $\mathbf{N}_{\ell-1} \setminus \mathbf{K}_{\ell-1}$.

Next, suppose that the majority string $M_{\ell,i}$ is non-uniform. Each mismatch between consecutive symbols of $M_{\ell,i}$ corresponds to a class $C \in \mathbf{K}_{\ell-1}$. If $\ell = 1$, then $M_{1,i}$ is a linear string and it may have one mismatch. Otherwise, $M_{\ell,i}$ is circular, so there are at least two mismatches between consecutive symbols.

FACT 8.4. *If $\mathcal{C}_{d_\ell}(i) \in \mathbf{K}_\ell \setminus \mathbf{N}_\ell$ for some $0 \leq \ell < s$, then there are at least $2^{\ell-1}$ positions $i' \equiv i \pmod{d_\ell}$ such that $0 \leq i' < n - p_{\ell+1}$ and $X[i'] \neq X[i' + p_{\ell+1}]$.*

Proof. If $\ell = 0$, then we just have $\mathcal{C}_0(i) \in \mathbf{K}_0$ if and only if $0 \leq i < n - p_1$ and $X[i] \neq X[i + p_1]$.

Consider an alignment between $X[0\,..\,n - p_{\ell+1} - 1]$ and $X[p_{\ell+1}\,..\,n-1]$ and let $k_{\ell,i}$ be the number of positions i' specified above. Observe that exactly $\lfloor \frac{i+p_{\ell+1}}{d_\ell} \rfloor$ symbols in $\mathcal{C}_{d_\ell}(i + p_{\ell+1})$ are at indices smaller than $p_{\ell+1}$ (and they are not aligned with any symbol of $\mathcal{C}_{d_\ell}(i)$), while exactly $k_{\ell,i}$ symbols are aligned with mismatching symbols. The remaining symbols of $\mathcal{C}_{d_\ell}(i+p_{\ell+1})$ are aligned with matching symbols of $\mathcal{C}_{d_\ell}(i)$. The class $\mathcal{C}_{d_\ell}(i)$ is uniform, so at most $k_{\ell,i} + \lfloor \frac{i+p_{\ell+1}}{d_\ell} \rfloor$ symbols of $\mathcal{C}_{d_\ell}(i + p_{\ell+1})$ are not equal to the majority of $\mathcal{C}_{d_\ell}(i)$. Since $\mathcal{C}_{d_\ell}(i + p_{\ell+1})$ does not share the majority with $\mathcal{C}_{d_\ell}(i)$, we must have $k_{\ell,i} + \lfloor \frac{i+p_{\ell+1}}{d_\ell} \rfloor \geq \frac{1}{2}|\mathcal{C}_{d_\ell}(i+p_{\ell+1})| = \frac{1}{2}\lfloor \frac{i+p_{\ell+1}}{d_\ell} \rfloor + \frac{1}{2}\lceil \frac{n-i-p_{\ell+1}}{d_\ell} \rceil$. Due to $p_{\ell+1} \leq \frac{n}{4}$ and $d_\ell \leq \frac{n}{2^{\ell+1}}$ (by Fact 8.1), this yields

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

In short, $k_{\ell,i} > 2^{\ell-1} - 1$, and thus $k_{\ell,i} \geq 2^{\ell-1}$.

LEMMA 8.5. *We have $2|\mathbf{N}_s| + \sum_{\ell=0}^{s-1} |\mathbf{K}_\ell| \leq 8k$. Consequently, the majority strings $M_{\ell,i}$ contain in total at most $8k$ mismatches between adjacent symbols and $\sum_{C \in \mathbf{N}_{d_s}} \|C\| \leq 16k$.*

Proof. We apply a charging argument. In the charging phase, each mismatch $X[i] \neq X[i + p_\ell]$ (for $1 \leq \ell \leq s$ and $0 \leq i < n - p_\ell$) receives a charge of $2^{3-\ell}$ units. The total charge is therefore at most $\sum_{\ell=1}^{s} k \cdot 2^{3-\ell} < 8k$.

Next, each such mismatch passes its charge to the class $\mathcal{C}_{d_\ell}(i)$. By Fact 8.4, each class $\mathcal{C}_{d_\ell}(i) \in \mathbf{K}_\ell \setminus \mathbf{N}_\ell$ receives at least 2 units of charge. Moreover, each class $\mathcal{C}_{d_0}(i) \in \mathbf{K}_0 \setminus \mathbf{N}_0$ receives exactly 4 units.

Finally, in subsequent iterations for $\ell = 0$ to $s - 1$, the classes modulo d_ℓ pass some charge to the enclosing classes modulo $d_{\ell+1}$: each $\mathcal{C}_{d_\ell}(i) \notin \mathbf{K}_\ell$ passes all its charge to $\mathcal{C}_{d_{\ell+1}}(i)$, whereas each $\mathcal{C}_{d_\ell}(i) \in \mathbf{K}_\ell$ leaves one unit for itself and passes the remaining charge.

We inductively prove that prior to the iteration ℓ, each class $\mathcal{C}_{d_\ell}(i) \in \mathbf{N}_\ell$ had at least two units of charge. Let us fix such a class. If $\ell = 1$, then Fact 8.3 implies that it contains a class $C \in \mathbf{K}_0$ (as $\mathbf{N}_0 = \emptyset$). As we have observed, it obtained 4 units of charge and passed 3 of them to $\mathcal{C}_{d_\ell}(i)$. Similarly, if $\mathcal{C}_{d_\ell}(i)$ contains a class $C \in \mathbf{N}_{\ell-1} \setminus \mathbf{K}_{\ell-1}$, then this class obtained at least 2 units of charge (by the inductive assumption), and passed them all to $\mathcal{C}_{d_\ell}(i)$. Otherwise, Fact 8.3 tells us that $\mathcal{C}_{d_\ell}(i)$ contains at least two classes $C \in \mathbf{N}_{\ell-1} \cup \mathbf{K}_{\ell-1}$. Each of them received at least 2 units of charge (directly from the mismatches or due to the inductive assumption) and passed at least 1 unit to $\mathcal{C}_{d_\ell}(i)$.

In the end, each class $C \in \mathbf{K}_\ell$ has therefore at least one unit of charge and each class $C \in \mathbf{N}_s$ has at least 2 units. This completes the proof of the inequality.

For the remaining two claims, observe that mismatches in the majority strings correspond to classes $C \in \mathbf{K}_\ell$, and that if $a \in C$ for $C \in \mathbf{N}_s$, then there exists $\mathcal{C}_{d_\ell}(i) \subseteq \mathcal{C}$ such that the majority string $M_{\ell,i}$ is non-uniform and contains a. Consequently, $\|C\|$ is bounded by twice the number of mismatches in the corresponding majority strings. The classes modulo d_s are disjoint, so no mismatch is counted twice.

8.3 Algorithmic Tools A *run* in a string S is a maximal uniform fragment of S, i.e., a maximal fragment of S composed of equal letters; we denote the number of runs in S by $\mathrm{rle}(S)$.

FACT 8.6. (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 symbol $S[i]$ can be retrieved in $\mathcal{O}(\log r)$ time. This representation can be constructed in $\mathcal{O}(r \log n)$ time from the run-length encoding of S.*

Proof. Let $0 = x_1 < \ldots < x_r < n$ be starting position of each run. We store the sequence $x_1, \ldots, x_r$ using the Elias–Fano representation [21, 24] (with $\mathcal{O}(1)$-time data structure for selection queries in a bitmask; see e.g. [13]). It 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 8.7. (MEMBERSHIP QUERIES [10]) *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 $i \in \{0, \ldots, n-1\}$. The construction time is $\mathcal{O}(m \log n)$.*

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 $1 \le \ell \le s$ and $0 \le i < d_\ell$) and all non-uniform classes $\mathcal{C}_{d_s}(i)$. We represent them as non-overlapping factors of a single string $\mathbf{M}$ of length $2n$, constructed as follows: Initially, $\mathbf{M}$ consists of blank symbols $\diamond$. Each non-uniform majority string $M_{\ell,i}$ is placed in $\mathbf{M}$ at position $2d_\ell + i\frac{d_{\ell-1}}{d_\ell}$ for $\ell > 1$ and $2d_\ell + i\lceil \frac{n}{d_\ell} \rceil$ for $\ell = 1$. Note that the positions occupied by strings $M_{\ell,i}$ for a fixed level ℓ belong to the range $[2d_\ell, 2d_\ell + d_{\ell-1} - 1] \subseteq [2d_\ell, 2d_{\ell-1} - 1]$ for $\ell > 1$ and $[2d_1, \ldots, 2d_1 + d_1\lceil \frac{n}{d_\ell} \rceil - 1] \subseteq [2d_1, 3d_1 + n - 1] \subseteq [2d_1, 2n - 1]$ for $\ell = 1$. These ranges are clearly disjoint for distinct values ℓ, so the majority strings $M_{\ell,i}$ indeed do not overlap. Additionally, we exploit the fact that positions within $[0, \ldots, d_s - 1]$ are free, and if $\mathcal{C}_{d_s}(i) \in \mathbf{N}_s$, we store its majority symbol at $\mathbf{M}[i]$. Lemma 8.5 yields that the total number of mismatches between subsequent symbols and the number of non-uniform classes modulo d_s are both $\mathcal{O}(k)$. Hence, $\mathrm{rle}(\mathbf{M}) = \mathcal{O}(k)$ and the space required to store $\mathbf{M}$ using Fact 8.6 is $\mathcal{O}(k(\log \frac{n+k}{k} + \log|\Sigma|))$ bits. On top of that, we also store a data structure of Fact 8.7 marking the positions in $\mathbf{M}$ where non-uniform majority strings start; this component takes $\mathcal{O}(k \log \frac{n+k}{k})$ bits.

Additionally, we keep the contents of each non-uniform class modulo d_s. We do not need to access this data efficiently, so for each such class, we simply store the symbols and their multiplicities using variable-length encoding. This takes $\mathcal{O}(\|\mathcal{C}\|(\log|\Sigma| + \log\frac{|C|+\|C\|}{\|C\|}))$ bits for each $C \in \mathbf{N}_s$, which is $\mathcal{O}(k(\log\frac{n}{k} + \log|\Sigma|))$ in total because $\sum_{C\in\mathbf{N}_s}\|C\| = \mathcal{O}(k)$ (by Lemma 8.5) and $\sum_{C\in\mathbf{N}_s}|C| \le n$.

Finally, we store integers n, d_1, d_s, as well as $\frac{d_\ell}{d_{\ell+1}}$ and $r_\ell := (\frac{p_{\ell+1}}{d_{\ell+1}})^{-1} \bmod \frac{d_\ell}{d_{\ell+1}}$ for $1 \le \ell < s$. 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 values $\frac{d_\ell}{d_{\ell+1}}$ using $\mathcal{O}(\sum_{\ell=1}^{s}\log\frac{d_\ell}{d_{\ell+1}}) = \mathcal{O}(\log n) = \mathcal{O}(k \log\frac{n+k}{k})$ bits in total. Similarly, the integers r_ℓ can be stored in $\mathcal{O}(\log n)$ bits because $1 \le r_\ell < \frac{d_\ell}{d_{\ell+1}}$.

This completes the description of our data structure; it takes $\mathcal{O}(k(\log\frac{n+k}{k} + \log|\Sigma|))$ bits.

8.5 Queries In this section, we describe the query algorithm for a given index i. We are going to iterate for $\ell = 0$ to s, and for each ℓ we will either learn $X[i]$ or find out that $X[i]$ is the majority symbol 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 of Section 8.4. Next, we query the data structure of Fact 8.7 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 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 8.6 to retrieve $X[i]$ in $\mathcal{O}(\log k)$ time.

If the query algorithm completes all the s iterations without exiting, then $X[i]$ must be the majority symbol of $\mathcal{C}_{d_s}(i)$. Thus, we retrieve $\mathbf{M}[i \bmod d_s]$, which takes $\mathcal{O}(\log k)$ time due to Fact 8.6. This symbol is either the majority of $\mathcal{C}_{d_s}(i)$ (guaranteed to be equal to $X[i]$) or

a blank symbol. In the latter case, we know that the class $\mathcal{C}_{d_s}(i)$ is uniform, so we return a sentinel #. The overall running time is $\mathcal{O}(s + \log k) = \mathcal{O}(\log n)$.

8.6 Updates If $d_s \mid p$, then the we do not need to update. Otherwise, we extend our data structure with $p_{s+1} := p$ and $d_{s+1} := \gcd(p_{s+1}, d_s)$.

First, we detect non-uniform classes modulo d_{s+1}. Facts 8.3 and 8.4 imply that a class $\mathcal{C}_{d_{s+1}}(i)$ is non-uniform if and only if for some $i' \equiv i \pmod{d_{s+1}}$ there is a class $\mathcal{C}_{d_s}(i') \in \mathbf{N}_s$ or a position $0 \le i' < n - p_{s+1}$ with $X[i'] \neq X[i' + p_{s+1}]$. Hence, we scan the non-uniform classes modulo d_s and the mismatch information for p_{s+1} grouping the entries by $i' \bmod d_{s+1}$.

For each $\mathcal{C}_{d_{s+1}}(i) \in \mathbf{N}_{s+1}$ our goal is to construct the underlying multiset and the corresponding majority string $M_{s+1,i}$. For every enclosed $C \in \mathbf{N}_s$, we use the multiset to deduce the majority symbol and store it at an appropriate position of $M_{s+1,i}$. Next, for every mismatch $X[i'] \neq X[i' + p_{s+1}]$ we store $X[i']$ in $M_{s+1,i}$ as the majority symbol of $\mathcal{C}_{d_s}(i')$ if the class is uniform. Symmetrically, if $\mathcal{C}_{d_s}(i' + p_{s+1})$ is uniform, its majority symbol $X[i' + p_{s+1}]$ is placed in $M_{s+1,i}$. The remaining symbols of $M_{s+1,i}$ are guaranteed to be equal to their both neighbours. This lets us retrieve the run-length encoding of $M_{s+1,i}$. To compute the multiset of $\mathcal{C}_{d_{s+1}}(i)$, we aggregate the data from the enclosed non-uniform classes, and use the majority string to retrieve for each symbol a the total size of enclosed classes uniform in a.

Finally, we note that the data structures of Facts 8.6 and 8.7 can be reconstructed in $\mathcal{O}(k \log n)$ time (which is sufficient to build them from scratch).

9 Proof of Proposition 3.3

In order to prove Proposition 3.3, we first give a new efficient algorithm for the streaming k-mismatch problem, assuming we can maintain a read-only copy of (the latest n symbols in) the text. Up to the additive $\mathcal{O}(\log^3 n)$ term, the running time of our algorithm matches that of the classic offline solution [5] despite having guaranteed worst-case performance per arriving symbol and using small additional space on top of the read-only access to the last n symbols of the text.

THEOREM 5.6. *In the read-only random-access model, the streaming k-mismatch problem can be solved online with a Monte-Carlo algorithm using $\mathcal{O}(k \log n)$ bits of working space and $\mathcal{O}(\sqrt{k \log k} + \log^3 n)$ time per symbol, including $\mathcal{O}(1)$ symbol reads. For any reported k-mismatch occurrence, the mismatch information can be retrieved on demand in $\mathcal{O}(k)$ time.*

Proof. First, we briefly sketch our strategy. If the pattern has an $\mathcal{O}(k)$-period $\mathcal{O}(k)$, then it suffices to apply Lemma 4.3 and Theorem 4.2. Otherwise, we can still use these results to filter the set of positions where a P has k-mismatch occurrence in T, leaving at most one candidate for each k subsequent positions. We use sketches to verify candidates, with the *tail trick* (see Lemma 7.6) employed to avoid reporting occurrences with a delay.

More formally, while processing the pattern, we also construct a decomposition $P = P_H P_T$ into the *head* P_H and the *tail* P_T with $|P_T| = 2k$, and we compute the sketch $\mathrm{sk}_k(P_H)$ (using Fact 4.4). We also apply Lemma 4.3 with $p = k$ and $d = 2k + 1$, which results in a prefix Q of P. This way, P is processed in $\mathcal{O}(\sqrt{k \log k} + \log^2 n)$ time per symbol using $\mathcal{O}(k \log n)$ bits of working space.

If $|Q| > |P_H|$, then P has a $4k$-period $p' \le k$, and we may just use Theorem 4.2 to report the k-mismatch occurrences of P in T. Otherwise, Q has a $(2k+1)$-period $p' \le k$, but no $2k$-period $p'' \le k$. In particular, due to Observation 4.1, the k-mismatch occurrences of Q are located more than k positions apart.

Processing the text T, we apply Fact 4.4 so that $\mathrm{sk}_k(T[0\,..\,i])$ and $\mathrm{sk}_k(T[0\,..\,i - |P_H|])$ can be efficiently retrieved when $T[i]$ is handled. Additionally, we run the streaming algorithm of Theorem 4.2, delayed so that a k-mismatch occurrence of Q starting at position $i - |P_H| + 1$ is reported while $T[i]$ is revealed. These components take $\mathcal{O}(k \log n)$ bits of space and use $\mathcal{O}(\sqrt{k \log k} + \log n)$ time per symbol of T.

If Theorem 4.2 reports a k-mismatch occurrence of Q at position $i - |P_H| + 1$, we shall check if P_H also has a k-mismatch occurrence there. For this, we retrieve $\mathrm{sk}_k(T[0\,..\,i])$ and $\mathrm{sk}_k(T[0\,..\,i - |P_H|])$ (using Fact 4.4), compute $\mathrm{sk}_k(T[i - |P_H| + 1\,..\,i])$ (using Proposition 3.1(b)), and compare it to $\mathrm{sk}_k(P_H)$ (using Proposition 3.1(a)). This process takes $\mathcal{O}(k \log^3 n)$ time in total, and it can be executed while the subsequent k symbols of T are revealed. If P_H has a k-mismatch occurrence at position $i - |P_H| + 1$, it results in $\mathrm{MI}(P_H, T[i - |P_H| + 1\,..\,i])$. Then, we naively compute $\mathrm{MI}(P_T, T[i + 1\,..\,i + 2k])$. Thus, as soon as $T[i + 2k]$ is revealed, we know if $\mathrm{HD}(P, T[i - |P_H| + 1\,..\,i + 2k]) \le k$, and we can retrieve the mismatch information in $\mathcal{O}(k)$ time upon request.

Since Q does not have any $2k$-period $p'' \le k$, we are guaranteed that at most two k-mismatch occurrences of Q are processed in parallel (one is extended to P_H and another one to P).

We are now able to prove Proposition 3.3.

PROPOSITION 3.3. *The stream of k-mismatch information of Q in T can be stored in a sliding-window buffer of fixed length $\delta = \Theta(|Q|)$ which uses $\mathcal{O}(k \log n)$ bits.*

The push operation takes $\mathcal{O}(k\log^2 n+\log^3 n)$ time if the pushed entry $\mathsf{Occ}^k_Q[i+\delta]$ or the retrieved entry $\mathsf{Occ}^k_Q[i]$ is non-empty and $\mathcal{O}(\sqrt{k\log k}+\log^3 n)$ time otherwise. Initialisation, given $\mathrm{sk}_k(Q)$ and δ, takes $\mathcal{O}(k)$ time.

Proof. We partition T into consecutive blocks of length $b=\frac{1}{4}\min(\delta,|Q|)$. The buffer shall be implemented as an *assembly line* of components, each responsible for k-mismatch occurrences of Q starting within a single block, called the *relevant occurrences* in what follows.

The choice of b guarantees that storing $\mathcal{O}(1)$ components suffices at any time. Moreover, the component needs to output the k-mismatch occurrences $\Theta(|Q|)$ pushes after it is fed with the last relevant occurrence, which leaves plenty of time for reorganisation. Consequently, its lifetime shall consist of three phases:

- *encoding*, when it is fed with relevant occurrences of Q in T,
- *reorganisation*, when it performs some computations to change its structure,
- *decoding*, when it retrieves the stream of k-mismatch information of Q in T.

In the encoding phase, we essentially construct the message as described in the proof of Theorem 1.3, encoding the relevant occurrences of Q, i.e., the k-mismatch occurrences of Q in the appropriate fragment of T. The only difference is that we also store the sketches $\mathrm{sk}_k(T[0\,..\,\ell-1])$ and $\mathrm{sk}_k(T[0\,..\,\ell'-1])$ corresponding to the leftmost and the rightmost relevant occurrence.

A single relevant occurrence can be processed in $\mathcal{O}(k\log n)$ time, dominated by updating the data structure of Lemma 5.5, which may need to account for a new element of $\mathcal{P}$. Apart from that, we only need to replace the rightmost relevant occurrence, along with the associated mismatch information and the sketch.

In the decoding phase, we apply Theorem 5.6 to find k-mismatch occurrences of $Q_\#$ in $T'_\#$, as defined in the proof of Theorem 1.3. To provide random access to these strings, we just need the data structure of Lemma 5.5 and the mismatch information for the two extremal relevant occurrences of Q; see Fact 5.4. To allow for $\mathcal{O}(\log n)$-time access, we organise the mismatch information in two dictionaries: for each mismatch $(i,Q[i],T[i'])$, we store $T[i']$ in a dictionary indexed by i', and $Q[i]$ in a dictionary indexed by $i \bmod d$. These dictionaries can be constructed in $\mathcal{O}(k\log^2\log k)$ time (during the reorganisation phase) [41]. As a result, we can use Theorem 5.6 to report the occurrences of Q in T in the claimed running time, along with the mismatch information. The reorganisation phase is also used to process the first n symbols of $T'_\#$.

Retrieving the corresponding sketches $T[0\,..\,i-1]$ is more involved, and here is where the reorganisation phase is crucially needed. Based on the sketches $\mathrm{sk}_k(T[0\,..\,\ell-1])$ and $\mathrm{sk}_k(T[0\,..\,\ell'-1])$, we can compute $\mathrm{sk}_k(T[\ell\,..\,\ell'-1]) = \mathrm{sk}_k(T'[0\,..\,\ell'-\ell-1])$ applying Proposition 3.1(b). Consider the point-wise difference D of strings T' and $T'_\#$. Observe that Corollary 6.5 lets us transform $\mathrm{sk}_k(T'[0\,..\,\ell'-\ell-1])$ into $\mathrm{sk}_k(D[0\,..\,\ell'-\ell-1])$ using random access to $T'_\#$ for listing mismatches. Next, we observe that D is an integer power of a string of length d, so Lemma 6.3 can be used to retrieve the sketch $\mathrm{sk}_k(D[0\,..\,d-1])$ of its root.

During the decoding phase, we maintain the data structure of Corollary 6.5 transforming $\mathrm{sk}_k(D[0\,..\,\ell'-\ell-1])$ back to $\mathrm{sk}_k(T'[0\,..\,\ell'-\ell-1])$. When a k-mismatch occurrence of $Q_\#$ is reported at position i of $T'_\#$, we report a k-mismatch occurrence of Q at position $\ell+i$. At the same time, we compute $\mathrm{sk}_k(D[i\,..\,\ell'-\ell-1])$ (using Lemma 6.3; we are guaranteed that $d \mid i$) and retrieve $\mathrm{sk}_k(T'[0\,..\,i-1]D[i\,..\,\ell'-\ell-1])$ (from Corollary 6.5). Finally, we construct $\mathrm{sk}_k(T[\ell\,..\,\ell+i-1])$ and $\mathrm{sk}_k(T[0\,..\,\ell+i-1])$ using Proposition 3.1(b).

In the encoding phase, we need to update anything only when the component is fed with a relevant occurrence, and processing such an occurrence takes $\mathcal{O}(k\log n)$ time. The reorganisation time is $\mathcal{O}(n(\sqrt{k\log k}+\log^3 n))$, which is $\mathcal{O}(\sqrt{k\log k}+\log^3 n)$ time per push. In the decoding phase, the running time is $\mathcal{O}(\sqrt{k\log k}+\log^3 n)$ per push (due to Theorem 5.6 and Corollary 6.5) plus $\mathcal{O}(k\log^2 n)$ time whenever a k-mismatch occurrence of Q is reported.

References

[1] A. Abboud, A. Backurs, and V. V. Williams. Tight hardness results for LCS and other sequence similarity measures. In *FOCS 2015*, pages 59–78. IEEE, 2015.

[2] A. Abboud and V. V. Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *FOCS 2014*, pages 434–443. IEEE, 2014.

[3] A. Abboud, V. V. Williams, and O. Weimann. Consequences of faster alignment of sequences. In *ICALP 2014*, pages 39–51. Springer, 2014.

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

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

[6] A. Backurs and P. Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *STOC 2015*, pages 51–58. ACM, 2015.

[7] D. Breslauer and Z. Galil. Real-time streaming string-matching. *ACM Transactions on Algorithms*, 10(4):22:1–22:12, 2014.

[8] K. Bringmann. Why walking the dog takes time: Frechet distance has no strongly subquadratic algorithms unless SETH fails. In *FOCS 2014*, pages 661–670. IEEE, 2014.

[9] K. Bringmann and M. Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *FOCS 2015*, pages 79–97. IEEE, 2015.

[10] A. Brodnik and J. I. Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.

[11] J. F. Canny, E. Kaltofen, and Y. N. Lakshman. Solving systems of nonlinear polynomial equations faster. In *ISSAC 1989*, pages 121–128. ACM, 1989.

[12] D. G. Cantor and H. Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation*, 36(154):587–587, 1981.

[13] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage (extended abstract). In *SODA 1996*, pages 383–391. ACM/SIAM, 1996.

[14] R. Clifford, K. Efremenko, B. Porat, and E. Porat. A black box for online approximate pattern matching. *Information and Computation*, 209(4):731–736, 2011.

[15] R. Clifford, K. Efremenko, E. Porat, and A. Rothschild. From coding theory to efficient pattern matching. In *SODA 2009*, pages 778–784. SIAM, 2009.

[16] R. Clifford, A. Fontaine, E. Porat, B. Sach, and T. Starikovskaya. Dictionary matching in a stream. In *ESA 2015*, pages 361–372. Springer, 2015.

[17] R. Clifford, A. Fontaine, E. Porat, B. Sach, and T. Starikovskaya. The k-mismatch problem revisited. In *SODA 2016*, pages 2039–2052. SIAM, 2016.

[18] R. Clifford, A. Grønlund, K. G. Larsen, and T. A. Starikovskaya. Upper and lower bounds for dynamic data structures on strings. In *STACS 2018*, pages 22:1–22:14. Schloss Dagstuhl, 2018.

[19] R. Clifford and B. Sach. Pseudo-realtime pattern matching: Closing the gap. In *CPM 2010*, pages 101–111. Springer, 2010.

[20] R. Clifford and T. A. Starikovskaya. Approximate Hamming distance in a stream. In *ICALP 2016*, pages 20:1–20:14. Schloss Dagstuhl, 2016.

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

[22] F. Ergün, E. Grigorescu, E. S. Azer, and S. Zhou. Streaming Periodicity with Mismatches. In *APPROX/RANDOM 2017*, pages 42:1–42:21. Schloss Dagstuhl, 2017.

[23] F. Ergün, H. Jowhari, and M. Saglam. Periodicity in streams. In *APPROX/RANDOM 2010*, pages 545–559. Springer, 2010.

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

[25] P. Gawrychowski and P. Uznański. Towards unified approximate pattern matching for Hamming and L_1 distance. In *ICALP 2018*, pages 62:1–62:13. Schloss Dagstuhl, 2018.

[26] S. Golan, T. Kopelowitz, and E. Porat. Towards optimal approximate streaming pattern matching by matching multiple patterns in multiple streams. In *ICALP 2018*, pages 65:1–65:16. Schloss Dagstuhl, 2018.

[27] D. Gorenstein and N. 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.

[28] W. Huang, Y. Shi, S. Zhang, and Y. Zhu. The communication complexity of the Hamming distance problem. *Information Processing Letters*, 99(4):149–153, 2006.

[29] M. Jalsenius, B. Porat, and B. Sach. Parameterized matching in the streaming model. In *STACS 2013*, pages 400–411. Schloss Dagstuhl, 2013.

[30] E. Kaltofen and Y. N. Lakshman. Improved sparse multivariate polynomial interpolation algorithms. In *Symbolic and Algebraic Computation, ISSAC 1988*, pages 467–474. Springer, 1988.

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

[32] D. E. Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition.* Addison-Wesley, 1998.

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

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

[35] G. M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

[36] V. Y. Pan. Faster solution of the key equation for decoding BCH error-correcting codes. In *STOC 1997*, pages 168–175. ACM, 1997.

[37] W. W. Peterson. Encoding and error-correction procedures for the bose–chaudhuri codes. *IRE Transactions on Information Theory*, 6(4):459–470, 1960.

[38] B. Porat and E. Porat. Exact and approximate pattern matching in the streaming model. In *FOCS 2009*, pages 315–323. IEEE, 2009.

[39] J. Radoszewski and T. A. Starikovskaya. Streaming k-mismatch with error correcting and applications. In *DCC 2017*, pages 290–299. IEEE, 2017.

[40] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, jun 1960.

[41] M. Ružić. Constructing efficient dictionaries in close to sorting time. In *ICALP 2008, Part I*, pages 84–95. Springer, 2008.

[42] A. Schönhage and V. Strassen. Schnelle multiplikation großer Zahlen. *Computing*, 7(3-4):281–292, 1971.

[43] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra (3rd ed.)*. Cambridge University Press, 2013.