Dynamic Suffix Array with Sub-linear update time and Poly-logarithmic Lookup Time

Amihood Amir∗ Itai Boneh†
Bar-Ilan University Bar-Ilan University
and
Georgia Tech

Abstract

The suffix array $SA_S[1 \ldots n]$ of an $n$-length string $S$ is a lexicographically sorted array of the suffixes of $S$. The suffix array is one of the most well known and widely used data structures in string algorithms. We present a data structure for maintaining a representation of the suffix array of a dynamic string which undergoes symbol substitutions, deletions, and insertions.

For every string manipulation, our data structure can be updated in $\tilde{O}(n^{3/2})$ time with $n$ being the current length of the string. For an input query $i \in [1 \ldots n]$, our data structure reports $SA_S[i]$ in $O(\log^5(n))$ time.

We also present a faster data structure, with $\tilde{O}(\sqrt{n})$ update time, for maintaining the Inverted Suffix Array of a dynamic string undergoing symbol substitutions updates. For an input query $i \in [1 \ldots n]$, our data structure reports the $i$’th entry in the inverted suffix array in $O(\log^4(n))$ time.

Our data structures can be used to obtain sub-linear dynamic algorithms for several classical string problems for which efficient dynamic solutions were not previously known.

∗Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel, +972 3 531-8770; amir@cs.biu.ac.il
†Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel, email: itai.bone@biu.live.ac.il.
1 Introduction

Recently, there has been a growing interest in dynamic strings algorithms. Many classical stringology problems have been lately investigated in dynamic settings. Perhaps the most natural problem of stringology is the one of constructing an index for a given string. That is, preprocessing a string \( S \) in order to obtain a data structure that given an input word \( p \), outputs the set \( \text{occ} \) of indexes in \( S \) in which the word \( p \) occurs. In the dynamic settings, we wish to maintain an index for a string that undergoes edit operations (substitution, deletion, or insertion of a single symbol in any location in the text). The current state of the art for dynamic indexing is the data structure of \([24]\) that requires \( O(p + \log^2(n) + |\text{occ}|) \) time per update and supports \( O(p + \log^2(n) + |\text{occ}|) \) time queries.

In the \textbf{Longest Common Factor} problem, we are given two strings \( S \) and \( T \). We are interested in finding the largest words that is a substring of both \( S \) and \( T \). Charalampopoulos et al. \([15]\) show how to maintain the LCF of two dynamic strings \( S \) and \( T \) that are going through edit operation updates with polylogarithmic time per update. In the \textbf{Longest Increasing Subsequence} problem, we are given an input string \( S[1 \ldots n] \) that consists of numbers. We are interested in finding the longest increasing sequence of indexes \( i_1, i_2 \ldots i_x \subseteq \{1 \ldots n\} \) such that \( S[i_j] < S[i_{j+1}] \) for every \( j \in \{1 \ldots x-1\} \). Gawrychowski et al. \([23]\) provided an algorithm for maintaining an approximation of the LIS of a dynamic string in polylogarithmic time per update. Kociumaka and Seddighin \([28]\) provided an algorithm for maintaining the exact LIS of a dynamic string with polynomial sublinear time per update. In addition to the above, the problems of finding the longest palindrome substring in a text \([3, 22]\), finding all the runs in a text \([3, 22]\), and computing the edit distance between two strings \([16]\) have all been recently investigated in dynamic settings.

The natural interest in dynamic string algorithms grows from the fact that the biggest digital library in the world - the web - is constantly changing, as well as from the fact that other big digital libraries - genomes and astrophysical data, are also subject to change through mutation and time, respectively.

The \textbf{suffix tree} \([42]\) and \textbf{suffix array} \([36]\) have been, arguably, the most powerful and heavily used tools in Stringology. The \textbf{suffix tree} of string \( S \) is a compressed trie of all suffixes of \( S \), and the \textbf{suffix array} of \( S \) corresponds to a pre-order traversal of all the leaves of the suffix tree of \( S \). The natural application of the suffix tree is for indexing, but it has been used for many purposes. An incomplete list includes approximate matching \([32, 33]\), parameterized matching \([9, 11, 25, 34]\), efficient compression \([1, 2, 7, 12, 19, 37, 40, 43, 45, 46]\), finding syntactic regularities in strings \([8, 13, 18, 25, 27, 29, 31, 35, 41, 44]\), and much more.

It was not until recently that an algorithm for maintaining the suffix tree or suffix array of a dynamically changing text had been sought. The difficulty is that even a single change in the text may cause a change in linear number of entries in the suffix array. Thus, although a dynamic suffix array algorithm would be extremely useful to automatically adapt many static pattern matching algorithms to a dynamic setting, other techniques had to be sought.

The key to most recent efficient dynamic solutions that appeared in the Stringology literature is renaming. However, renaming is not a panacea for dynamic algorithms to all the problems that the suffix tree or array solved in the static setting. Perhaps the key property of the suffix array is that the suffixes are sorted lexicographically. The powerful renaming and locally persistent parsing techniques developed thus far do not maintain lexicographic ordering. It is, thus, no surprise that problems like maintaining the Burrows-Wheeler transform, or finding the Lyndon root of a substring, do not hitherto have an efficient dynamic version.

The current state of the art for maintaining the suffix array of a dynamic text is the flexible data structure of Amir and Boneh \([4]\). For every parameter \( k \in \{1 \ldots n\} \), they provide a data structure with \( \tilde{O}(k) \) update time supporting look up queries to \( SA[i] \) in \( \tilde{O}(\sqrt{k}) \) time.

1.1 Our Contribution

We enhance the data structure of Amir and Boneh \([4]\) to obtain the following.

1. A data structure for maintaining the suffix array of a dynamic string in \( \tilde{O}(n^{\frac{1}{2}}) \) time per text update with \( O(\log^3(n)) \) time for a look up query \( SA[i] \). This data structure supports symbol substitution, symbol insertion and symbol deletion updates.

2. A data structure for maintaining the \textit{inverted} suffix array of a dynamic string in \( \tilde{O}(\sqrt{n}) \) time per text update with \( O(\log^3(n)) \) time for a look up query \( SA[i] \). This data structure only supports symbol substitution updates.
Both data structures use $\tilde{O}(n)$ space. Note that our data structures maintain the sublinear polynomial update time of $O(\sqrt{n})$ and exponentially improve the query time from polynomial to polylogarithmic. Additionally, our data structure for dynamic suffix array can be used as a black box, alongside previously existing results, to obtain the following.

1. A dynamic text $S[1 \ldots n]$ can be maintained with $\tilde{O}(n^{3/4})$ time per edit operation update to support random access queries to $BWT[i]$, the burrows wheeler transform of $S$, in polylogarithmic time.

2. A dynamic text $S[1 \ldots n]$ can be maintained with $\tilde{O}(n^{3/4})$ time per edit operation update to enable random access queries to the $LCP$ array of $S$ in polylogarithmic time.

3. A dynamic text $S[1 \ldots n]$ can be maintained with $\tilde{O}(n^{3/4})$ time per edit operation update to enable a simulation of the suffix tree of $S$ with polylogarithmic time for traversing a suffix tree edge.

In [4], summarized in Section 6, the above results are presented with a polynomial query time, rather then polylogarithmic. Replacing the data structure of [4] with ours yields the above. We refer the interested reader to [4] for an overview of how a dynamic suffix array is utilized to achieve these results.

1.2 Technical Overview

**Dynamic Inverse Suffix Array** Similarly to [4], we partition the suffixes of the dynamic string $S[1 \ldots n]$ into clusters. Every cluster $C$ corresponds to a word $w_C$ with length $|w_C| = \sqrt{n}$. The cluster $C$ contains all the suffixes starting with the prefix $w_C$. We say that two suffixes are close if they are located within the same cluster, and far otherwise. It is shown in [4] that these clusters can be maintained in $\tilde{O}(\sqrt{n})$ time when a substitution update is applied to $S$.

Given a query $i$ for $iSA[i]$, we are interested in finding the lexicographical rank of $S[i \ldots n]$ among the suffixes of $S$. Equivalently, we want to count the number of suffixes of $S[1 \ldots n]$ that are lexicographically smaller than $S[i \ldots n]$. The inverse suffix array algorithm of [4] counts the number of suffixes far from $S[i \ldots n]$ that are smaller than $S[i \ldots n]$. The bottleneck that causes for the polynomial query time in [4] is counting the number of close suffixes that are smaller than $S[i \ldots n]$. This task can be equivalently described as evaluating the lexicographical rank of $S[i \ldots n]$ within the suffixes in its cluster.

In order to overcome this bottleneck, we wish to maintain an 'array' $R[1 \ldots n]$. For every suffix $S[i \ldots n]$, $R[i]$ stores the rank of $S[i \ldots n]$ within the suffixes in its cluster. In order to update $R$ in sublinear time, we provide a deep analysis of the nature in which the ranks in $R$ change as a result of a substitution update. The conclusion of our analysis is that while $R$ may require updates in $\Omega(n)$ entries as a result in a single substitution, these updates can be compactly described as a sequence of $O(\sqrt{n})$ special updates called stairs updates.

**Definition 1.** A decreasing stairs update (See Figure 2) is denoted by $S = (i, j, p)$. Applying $S$ to $R$ increases the values stored in $R[i \ldots j]$ as follows: the rightmost $p$ indexes $R[j - p + 1 \ldots j]$ are increased by 1, the $p$ indexes to their left are increased by 2, and so on until we finally exceed $R[i \ldots j]$.

Formally, for every $t \in [1 \ldots \lceil \frac{j - i + 1}{p} \rceil]$, the counters $R[a]$ with $a \in [\min(j - t \cdot p + 1, i) \ldots j - (t - 1) \cdot p]$ are increased by $t$.

We show how to obtain the sequence of $O(\sqrt{n})$ stairs updates that represent the changes to $R$ in $\tilde{O}(\sqrt{n})$ time. We then provide a data structure that maintains an ordered set of integers that is applied as a sequence of stairs updates. The utility of provided by our data structure for stairs updates is described by Theorem 2.

**Theorem 2.** There is a data structure for maintaining an indexed set of integers $R$ to support both applying a stairs update to $R$ and look-up queries $R[i]$ in $O(\log^2(U))$ time, provided that all the updates $(i, j, p)$ have the same value of $p$. $U$ is the overall amount of updates. The data structure consumes $O(U \log^2(U))$ space. Initializing the data structure (all counters are set to 0) takes $O(1)$ time.

The restriction that every stair update that is applied to the ranks has the same width $p$ does not necessarily hold in our dynamic algorithm. It may be the case that the stairs updates within the sequence have various step widths, or that the stairs updates applied in the most recent update have different step width from the updates that were applied in previous update. This denies us from using our data structure for stairs update in a straightforward manner to maintain $R$.

In order to solve that problem, we identify a unique property of the stairs updates that need to be applied to $R$.  

3
Observation 3. Every stairs update \((i, j, p)\) that is applied to \(R\) corresponds to a run \(S[i \ldots j]\) in \(S\) with period \(p\).

Observation 3 allows us to exploit existing insights on the structure of runs within a string to bound the number of different values of \(p\) for which a stairs update with step width \(p\) was applied to an index \(i\) in \(R\). Specifically, we show that there are \(O(\log(n))\) such values of \(p\). With this property, our problem can be reduced to maintaining a sequence \(\{St_1, St_2 \ldots St_n\}\) of restricted stairs updates data structure described in Theorem 2. Every stairs update data structure \(St_p\) is responsible for updates with a step width \(p\). When we wish to evaluate \(R[i]\), we sum the values stored in \(St_p\) over the \(p\) values such that \(St_p[i]\) was affected by a stairs update. Since there are \(O(\log(n))\) such values, the query time remains polylogarithmic.

**Dynamic Suffix Array** As in our solution for inverse suffix arrays, we build upon the data structure presented in [4]. For an integer parameter \(k\), the dynamic data structure of [4] provides the following utility.

**Fact 4.** A query \(i\) for \(SA_S[i]\) of a dynamic \(S\) can be reduced to a Close Suffix Selection query (defined below) in \(O(\log^2(n))\) time. The data structure for executing this reduction can be maintained in \(O(k)\) per edit operation update.

| Close Suffixes Select |
|------------------------|
| Given a subword of size \(k\) of \(S\) represented by its starting index \(i \in [1 \ldots n - k + 1]\) and a rank \(r\), return the suffix \(S[i \ldots n]\) with lexicographic rank \(r\) among the suffixes starting with \(S[i \ldots i + k - 1]\) |

In Section 4, we prove the following:

**Theorem 5.** Given a dynamic text \(S[1 \ldots n]\), there is a data structure that supports close suffixes selection queries in \(O(\log^2(n))\) time. The data structure can be maintained in \(\tilde{O}(\frac{n^2}{k} + k)\) time when an edit operation is applied to \(S\). The data structure uses \(\tilde{O}(n + \binom{n}{2})\) space.

By setting \(k = n^{\frac{1}{4}}\) and applying the above reduction, we obtain the following:

**Theorem 6.** For a dynamic string \(S[1 \ldots n]\), there is a data structure that supports look up queries to \(SA_S[i]\) in \(O(\log^2(n))\) time. When an edit operation update is applied to \(S\), the data structure can be maintained in \(\tilde{O}(n^{\frac{5}{4}})\) time. The data structure uses \(\tilde{O}(n)\) space.

We present the main idea for proving Theorem 23. Let \(w\) be a word of size \(t = \frac{k}{p}\). By exploiting periodic structures in strings, we can represent all the occurrences of \(w\) in \(S[1 \ldots n]\) with \(O(\frac{n}{k})\) arithmetic progressions. We denote this compact set of arithmetic progressions representing the occurrences of \(w\) as \(POR_w\). The data structure of [4] allows the evaluation of \(POR_w\) from an index \(i\) representing a starting index of \(w\) in \(O(\frac{n}{k})\) time. We denote the explicit set of occurrences of \(w\) in \(S\) as \(Occ_w\). We denote as \(A_w[1 \ldots |Occ_w|]\) the array consisting of the starting indexes of occurrences of \(w\) sorted by lexicographic order of the suffixes starting in these indexes. While the size of \(POR_w\) is bounded by \(O(\frac{n}{k})\), the size of \(A_w\) may be \(\Omega(n)\). Note that a Close suffix selection query is equivalent to a random access query in \(A_w\).

Let \(C = (a, b, p) \in POR_w\) be an arithmetic progression in the representation of the occurrences of \(w\). We denote the size of the cluster as \(|C| = \frac{k-p}{p} + 1\). The arithmetic progression \(C\) corresponds to occurrences of \(w\) starting in \(a, a+p, a+2\cdot p \ldots a+|C|\cdot p = b\). We say that \(C\) implies these occurrences of \(w\). We define the following properties of clusters and of occurrences implied by clusters.

**Definition 7.** Let \(w_i = S[a + i \cdot p \ldots a + i \cdot p + t - 1]\) be an occurrence implied by \(C\) of \(w\). The period rank of \(w_i\) is defined to be \(r_p(w_i) = |C| - i\).

**Definition 8 (Cluster Head, Cluster Tail, Increasing and Decreasing Clusters).** Let \(w_{|C|} = [b \ldots b + t - 1]\) be the rightmost occurrence of \(w\) represented by \(C\) (with respect to starting index). Let \(w_p = w[e-p+1 \ldots e]\) be the suffix of size \(p\) of \(w\). We call the suffix \(S[b + t \ldots n]\) the tail of \(C\) denoted as \(Tail(C)\). We call \(C\) an increasing cluster if \(Tail(C) >_L w_{|C|}\) or a decreasing cluster if \(Tail(C) <_L w_{|C|}\).

See Figure 1 for an example of a cluster and its properties. In Section 7, we prove that \(A_w\) has the following structure (derived from Lemma 74 and Lemma 75):

\(A_w\) can be described as a concatenation \(A_w = D[1 \ldots |D|], I[1 \ldots |I|]\) with \(D\) being the sub array containing the set of occurrences implied by decreasing clusters, and \(I\) being the set of occurrences implied by increasing clusters.
Therefore, restricting selection queries data structure of $w$ for every $r \in [0 \ldots m]$ contains the occurrences of $w$ from decreasing clusters with $r_p(w') = r$ sorted by lexicographic order of the tails of their clusters.

Similarly, $I = I_0, I_1, \ldots, I_{t-1}, I_t$. $t$ is the maximal period rank of an occurrence in an increasing cluster. For every $r \in [0 \ldots t]$, $I_r$ contains the occurrences $s$ from increasing clusters with $r_p(s) = r$ sorted by lexicographic order of the tails of their clusters (See Figure 8).

Let $w = S[s \ldots e]$ be a subword of $S$. Let $l$ and $r$ be two non-negative integers.

**Definition 9.** An occurrence $w_1 = S[s_1 \ldots e_1]$ of $w$ is an $(l, r)$-extendable occurrence of $w$ if $lcs(s - 1, s_1 - 1) \geq l$ and $lcp(e + 1, e_1 + 1) \geq r$. We define the array $A_1^{w(r)}$ as the lexicographically sorted array of suffixes of $S$ starting with an $(l, r)$-extendable occurrence of $w$.

It is easy to see that given a word $w^c = S[s - l \ldots e + r]$ containing $w$, an occurrence of $w^c$ in index $i$ is equivalent to an $(l, r)$-extendable occurrence of $w$ in $i + l$. Therefore, $A_{w^c}[i] = A_{w^{(r)}}[i] - l$ for every $i \in [1 \ldots |A_{w^c}|]$. We present the following problem:

**Extension Restricting Selection**

**Preprocess:** A word $w = S[s \ldots e]$ and POR$_w$

**Query:** upon $l, r, i$. Report $A_{w}^{(r)}[i]$

In Section 7, we prove the following:

**Theorem 10.** If LCE queries can be evaluated in $O(\log(n))$ time, the periodic occurrences representation POR$_w$ of a word $w = S[s \ldots e]$ can be preprocessed in time $\mathcal{O}(|POR_w|)$ to answer Extension Restricting Selection queries in $O(\log^5(n))$ time. The data structure uses $\mathcal{O}(|POR_w|)$ space.

LCE queries are formally defined in Section 2. Existing results in dynamic stringology can be used in a straightforward manner to enable $O(\log(n))$ LCE queries in dynamic settings.

Constructing the data structure described in Theorem 28 is the most technically involved part in the construction of our dynamic suffix array data structure. It is achieved by exploiting the characterization of the structure of $A_w$. Our insights on the structure of $A_w$ allow us to reduce an Extension Restricting Selection query to a small set of multidimensional range queries.

With the data structure of Theorem 28, a dynamic data structure for Close Suffix selection queries can be constructed as follows. We initialize a data structure for supporting dynamic LCE queries on $S$ by employing Lemma 11. We also initialize the dynamic data structure of Fact 4 for obtaining periodic occurrences representations and reducing suffix array queries to close selection queries. Let $t = \lceil \frac{n}{\log(n)} \rceil$. We partition $S$ into words of size $t$ starting in indexes $i = 0 \mod t$. For every word $w_x = S[x \cdot t \ldots (x + 1) \cdot t - 1]$ for $x \in [0 \ldots \lceil \frac{n}{t} \rceil]$, we obtain POR$_{w_x}$ in $\mathcal{O}(\frac{n}{t})$ time. We preprocess POR$_{w_x}$ for Extension Restricting Selection queries using Theorem 28. We keep an array ERSQ[1 \ldots \lceil \frac{n}{t} \rceil] with ERSQ$x$ containing a pointer to the Extension Restricting Selection queries data structure of $w_x$.

Upon an edit operation update, we update the data structures of Fact 4. We then partition the updated $S$ in the same manner as in the initialization step. Note that the words in the partition may change, as the size of $S$ may change due to an insertion or a deletion update, and $k$ may be a function of $|S| = n$. For every word $w_x$ in the updated partition, we extract POR$_{w_x}$ and evaluate the extension restricting selection queries data structures for every word $w_x$ from scratch. We also construct the array ERSQ from scratch as in the initialization step.

Upon a Close Suffix Selection query $(i, r)$, the word $w_{[i]}$ is completely contained within $w = S[i \ldots i + k - 1]$. Let $L$ be the distance between the starting indexes of $w_{[i]}$ and $w$ and let $R$ be the distance between the ending indexes. We execute an Extension Restricting Selection query $(L, R, r)$ on the Extension Restricting Selection queries data structure of $w_{[i]}$. Let the output of the query be $z$. We return $z - L$.

**Correctness:** A Close Suffix Selection query $(i, r)$ is equivalent to reporting $A_w[r]$. Denote the indexes of $w_{[i]} = S[s \ldots e]$. The word $w = S[i \ldots i + k - 1]$ can be written as $w = S[s - L, e + R]$. Therefore, $A_w[r] = A_{w_{[i]}}^z[r] - L$.

**Preprocessing time:** The preprocessing times of the LCE data structure and the periodic occurrences data structure are both bounded by $\mathcal{O}(n)$. The remaining operations are, in fact, an initial update. The complexity analysis for these is provided below.

**Update time:** Evaluating the partition can be executed in $O(\frac{n}{t})$ time in a straightforward manner. There are $\frac{n}{t}$ words in our partition. For every word $w$, we find POR$_w$ in $\mathcal{O}(\frac{n}{t})$ time. Since $|w| = \frac{n}{t}$, we have $|POR_w| \in O(\frac{n}{t})$. So constructing the Extension Restricting Selection data structure
from \(POR_w\) takes \(\tilde{O}(\frac{n}{T})\). The update time for the \(LCE\) data structure is polylogarithmic and the update time for the periodic occurrences representation data structure is \(\tilde{O}(k)\). The overall update complexity is dominated by \(\tilde{O}((\frac{n}{T})^2 + k)\).

**Space:** The space consumed by the Extension Restricting Selection data structures is bounded by \(\tilde{O}((\frac{n}{T})^2)\). The space consumed by the array \(ERSQ\) is bounded by \(O(\frac{n}{T})\). The space consumed by the data structure for periodic occurrences representation is bounded by \(O(n)\). The space consumed by the \(LCE\) data structure is bounded by \(\tilde{O}(n)\). The overall space complexity is bounded by \(\tilde{O}(n + (\frac{n}{T})^2)\).

**Query time:** The query consists of a constant number of basic arithmetic operations, a single array lookup and a single Extension Restricting Selection query. The query time is dominated by \(O(\log_2^*(n))\).

### 1.3 Roadmap

This paper is organized as follows: Section 2 gives basic definitions and terminology. Section 3 provides an overview of [4], and presents the bottleneck in the complexity of the algorithms presented there. This identifies the difficulty that the current paper solves. In Section 4 we present a static data structure problem, and show that, surprisingly, an efficient data structure for solving this problem, enables us to construct an efficient dynamic suffix array data structure. We provide an efficient solution for this problem in Section 5. For dynamic inverted suffix array queries, our results appear in Section 5. Section 6 provides some complementary proofs for the result of Section 5. Section 8 provides two data structures that are required for the construction of Section 6.

## 2 Preliminaries

### 2.1 Strings

Let \(S = S[1]S[2]...S[n]\) be a string (or a word) of length \(|S| = n\) over an ordered alphabet \(\Sigma\) of size \(|\Sigma| = \sigma\). By \(\varepsilon\) we denote an empty string. For two positions \(i\) and \(j\) in \(S\), we denote by \(S[i...j] = S[i]...S[j]\) the factor (sometimes called substring) of \(S\) that starts at position \(i\) and ends at position \(j\) (it equals \(\varepsilon\) if \(j < i\)). We recall that a prefix of \(S\) is a factor that starts at position 1 \((S[1...j]\) and a suffix is a factor that ends at position \(n\) \((S[i...n])\). The prefix ending in index \(i\) is denoted as \(S_i\) and the suffix starting in index \(i\) is denoted as \(S^i\).

An integer \(p\) is a period of \(S\) if \(S[i] = S[i+p]\) for every \(i \in [1...n-p]\). \(p\) is called the period of \(S\) if it is the minimal period of \(S\). We denote the period of a string \(S\) as \(\text{per}(S)\). If \(\text{per}(S) < \frac{|S|}{2}\), we say that \(S\) is periodic. A substring \(R = S[i...j]\) of \(S\) is called a run if it is periodic with period \(p\), and every substring of \(S\) containing \(R\) does not have a period \(p\). Equivalently, \(R\) is a run if it is periodic with period \(p\), and \(S[i-1] \neq S[i-1+p]\) and \(S[j+1] \neq S[j+1-p]\). If \(R\) is a prefix (resp. suffix) of \(S\), the first (resp. second) constraint is not required.

Let \(w[1...m]\) be a string. We say that \(w\) occurs in position \(i \in [1...n-m+1]\) in \(S\) if \(S[i+j-1] = w[j]\) for every \(j \in [1...m]\). Every substring \(S[i...i+m-1] = w\) is an occurrence of \(w\) starting in index \(i\).

The longest common prefix of two strings \(S[1...n], T[1...m]\), denoted as \(LCP(S, T)\), is the maximal integer \(l \in [1...\min(n, m)]\) such that \(S[1...l] = T[1...l]\). The longest common suffix of \(S\) and \(T\), denoted as \(LCS(S, T)\), is the maximal integer \(l \in [1...\min(n, m)]\) such that \(S[n-l+1...n] = T[m-l+1...m]\).

Given a string \(S\), a Longest Common Extension (often shortened to \(LCE\)) data structure for \(S\) supports the following two queries:

1. \(LCP(i,j)\) - return the longest common prefix of \(S[i...n]\) and \(S[j...n]\).
2. \(LCS(i,j)\) - return the longest common suffix of \(S[1...i]\) and \(S[1...j]\).

The results of \((24)\) or \((33)\) can be applied to obtain the following:

**Lemma 11.** There is a deterministic (resp. randomized) data structure for maintaining a dynamic string with \(O(\log(n) \log^*(n))\) (resp. \(O(\log(n))\) expected) time per edit operation (deletion, insertion, or substitution of a symbol) so that an \(LCE\) query can be answered in \(O(\log(n) \log^*(n))\) (resp. \(O(\log(n))\) expected) time. The data structure uses \(\tilde{O}(n)\) space.

**Remark:** Throughout the paper, we use the randomized variant of Lemma 11. Thus, we refer to the application of an \(LCE\) query as an \(O(\log(n))\) time operation. Every such reference to Lemma 11 can be
made deterministic by applying the deterministic variant of the lemma with the cost of an $O(\log^* (n))$ multiplicative slowdown.

We point out that other than LCE queries, the algorithms presented in this paper do not use randomization. Therefore, our results can be derandomized by adding a multiplicative $\log^* (n)$ factor to the complexity of an LCE query.

Let $S[1 \ldots n]$ and $T[1 \ldots m]$ be two strings over an ordered alphabet $\Sigma$. $S$ is lexicographically smaller than $T$, denoted as $S <_L T$, if $S[LCP(S, T) + 1] < T[LCP(S, T) + 1]$. In the case in which $LCP(S, T) = \min(|S|, |T|)$, $S <_L T$ if $|S| < |T|$. Note that given two indexes $i$ and $j$ representing two suffixes of a string $S$, Lemma [11] enables to lexicographically compare $S_i$ and $S_j$ in $O(\log (n))$ time.

The suffix array $SA_S$ is an array of size $n$ containing the suffixes of $S$ sorted lexicographically. A suffix $S[i \ldots n]$ is represented in $SA_S$ by its starting index $i$, so the size of $SA_S$ is linear. The Inverse Suffix Array $iSA_S$ is the inverse permutation of the suffix array. Namely, $iSA_S[S[i]] = x$ for every $x \in [1 \ldots n]$. In this paper, we handle a dynamic string that is undergoing substitution, deletion, and insertion updates. A substitution update is given as a pair $(i, \sigma)$.

In this paper, we handle a dynamic string that is undergoing substitution updates, it can be easily generalized to edit operations. The following is a basic observation regarding the occurrences of a word within $S$.

**Observation 13.** Given a word $w$ with $|w| = x$ and period $p$, starting in index $i$ in $S$, let $j$ be the index closest to $i$ that is also a starting index of an occurrence of $w$ in $S$. Either $|j - i| = p$ or $|j - i| \geq \frac{x}{p}$.

With Observation [13] we are ready to introduce the periodic occurrences representation.

**Fact 14.** The set of occurrences of a word $w$ having length $x$ and with period $p$ in a string $S[1 \ldots n]$ can be represented by $O(\frac{x}{p})$ arithmetic progressions (often referred to as clusters) of the form $C = (a, b, p)$. The cluster $C = (a, b, p)$ represents a set of occurrences of $w$ in $S$ starting in indexes $i = a + p \cdot t$ with $t \in [0, \frac{b - a}{p}]$. The size of a cluster, denoted as $|C| = \frac{b - a}{p} + 1$ is the number of occurrences represented by $C$. Every cluster $C$ is associated with a run $RC$ in $S$ with period $p$. The occurrences represented by $C$ are all the occurrences of $w$ contained within $RC$. It follows that every cluster is locally maximal, in the sense that there are no occurrences of $w$ starting in $a - p$ or in $b + p$.

We call the representation of the occurrences of $w$ described in Fact [14] the **periodic occurrences representation** of $w$.

**Lemma 15.** For every parameter $x \in [1 \ldots n]$, there is a data structure that, given integer $i$, outputs the periodic occurrences representation of $S[i \ldots i + x - 1]$ in a dynamic text $S[1 \ldots n]$ in $\tilde{O}(\frac{x}{p})$ time. This data structure can be maintained in $\tilde{O}(x)$ time after an edit operation update is applied to $S[1 \ldots n]$.

Lemma 3 of [4] and the discussion following Observation 1 there, serve as a proof for Lemma [15]. Specifically, the discussion following Observation 1 presents a method for obtaining the periodic occurrences representation of $w$ in $\tilde{O}(\frac{x}{p})$ time using, what they called, the k-Words tree data structure. Lemma 3 ensures that the k-Words tree can be maintained over edit operations updates in $\tilde{O}(x)$ time. While the lemma in [4] refers to a dynamic text undergoing substitution updates, it can be easily generalized to edit operations.
2.3 Dynamic multidimensional range queries

Dynamic multidimensional range queries is an algorithmic tool enabling the maintenance of a set of d-dimensional points. A data structure for d-dimensional range queries supports queries concerning the points within an input d-dimensional range. A d-dimensional point is given by the values of its d coordinates. A d-dimensional range $R$ is given by a set of $d$ ranges $R = [X_1 \ldots Y_1] \times [X_2 \ldots Y_2] \times \cdots \times [X_d \ldots Y_d]$. The point $p = (x_1, x_2 \ldots x_d)$ is in the range $R (p \in R)$ if $X_i \leq x_i \leq Y_i$ for every $i \in [1 \ldots d]$. Every point $p$ is assigned a numeric value $v(p)$. The result below can be achieved by using the range tree data structure $[\text{14}].$

Fact 16. There is a data structure for maintaining a set $D$ of $d$-dimensional points and supports the following queries:

1. Add a new point $p = (x_1, x_2, \ldots x_d)$ with value $v(p) = v$ to $D$.
2. Remove an existing point $p = (x_1, x_2, \ldots x_d)$ from $D$.
3. $\text{COUNT}(R)$ Given a $d$-dimensional range $R = [X_1 \ldots Y_1] \times [X_2 \ldots Y_2] \times \cdots \times [X_d \ldots Y_d]$, report the number of points in $D \cap R$.
4. $\text{SUM}(R)$: Given a $d$-dimensional range $R = [X_1 \ldots Y_1] \times [X_2 \ldots Y_2] \times [X_d \ldots Y_d]$, report the sum of $v(p)$ taken over the points in $D \cap R$. The sum of an empty set is defined to be $0$.
5. $\text{REPORT}(R)$: Given a $d$-dimensional range $R = [X_1 \ldots Y_1] \times [X_2 \ldots Y_2] \times [X_d \ldots Y_d]$, return all the points in $D \cap R$.

Queries 1 - 4 can be supported in $O(\log^d(n))$ and range report queries can be supported in $O(\log^d(n) + k)$ with $k$ being the amount of reported points and $n$ being the number of points in $D$. The data structure consumes $O(\log^{d-1}(n))$ space.

3 Close Suffix Selection and Close Suffix Rank

Our algorithm improves the results of $[\text{4}].$ For self-containment, this section provides an overview of the terminology, techniques, and results of $[\text{4}]$ that are relevant to our work. This section also identifies the sub-tasks that need to be efficiently implemented in order to improve the previous work.

3.1 Close suffixes, far suffixes, and the $k$-words tree

Let $k$ be an integer parameter. We start by defining a simple yet useful data structure.

Definition 17. For a string $S[1 \ldots n]$, the $k$-words tree of $S$ is a balanced search tree with the following components:

1. For every distinct $k$-length subword $w$ of $S$, the $k$-words tree contains a node $v = N(w)$ corresponding to $w$.
2. The nodes of the $k$-words tree are sorted by lexicographical order of their corresponding words.
3. $N(w)$ stores a balanced search tree containing the starting indexes of every occurrence of $w$ in $S$ in increasing order. $N(w)$ also stores the number of words in the subtree rooted in $N(w)$.

Let $v$ be a node in the $k$-words tree. We call the word $w$ such that $N(w) = v$ the word of $v$. We denote the word of $v$ as $W(v)$. We point out the distinction between nodes and words: every node in the $k$-words tree represents a set of occurrences of certain sub-words. The number of words in a subtree of the $k$-words tree is the accumulated number of occurrences represented by the nodes of the sub-tree. The following theorem is proven in $[\text{4}]$.

Theorem 18 ($[\text{4}]$, Lemma 3). Given a string $S[1 \ldots n]$, a data structure for dynamically maintaining the $k$-words tree of $S$ can be constructed in $\tilde{O}(n)$ time using $O(n)$ space. When applying an edit operation update to $S$, the $k$-words tree can be updated in $\tilde{O}(k)$ time.

In $[\text{4}]$, Lemma 3 is proved with respect to substitution updates. It can be easily modified to support deletion and insertions as well.

We proceed to define far and close suffixes.
Definition 19. Let $S^i = S[i...n]$ and $S^j = S[j...n]$ be two suffixes of a text $S[1...n]$. $S[i...n]$ and $S[j...n]$ are called close suffixes (or $k$-close suffixes) if $LCP(i, j) \geq k$, and far suffixes (or $k$-far suffixes) otherwise.

When we discuss $k$-close and $k$-far suffixes throughout the paper, the value of $k$ is clear from context. We therefore use the notations ‘close’ and ‘far’ suffixes, omitting the specific value of $k$.

The primary role of the $k$-words tree is to apply a reduction from a query about the suffix array to a query about close suffixes. Note that the starting indexes of close suffixes are stored within the same node of the $k$-words tree while the starting indexes of far suffixes are stored within different nodes.

3.2 Dynamic Inverted Suffix Array

Reporting $iSA[i]$ is equivalent to counting the number of suffixes that are lexicographically smaller than $S[i...n]$. Upon a query integer $i$ for $iSA[i]$, we can find the node $v$ in the $k$-words tree with $W(v) = S[i...i+k−1]$ in $O(\log^2(n))$ time by applying a classic binary tree search. The lexicographical comparison required in every node can be executed in $O(\log(n))$ time by employing Lemma 1.

By counting the words to the left of the path from the root to $v$, we obtain the amount of suffixes of $S$ that are lexicographically smaller than $S[i...n]$ and are far from $S[i...n]$. Given the path, this can be easily executed in logarithmic time. The remaining problem is to find the number of suffixes close to $S[i...n]$ that are lexicographically smaller than $S[i...n]$.

We formalize the problem of finding the close smaller suffixes as follows:

| Close Suffixes Rank |
|---------------------|
| Given an index $i$ representing a suffix $S[i...n]$, return lexicographic rank of $S[i...n]$ among the suffixes close to $S[i...n]$. |

In this paper, we present a dynamic data structures for Close Suffix Rank queries. Specifically, we prove the following in Section 5.

Theorem 20. Given a dynamic text $S[1...n]$, there is a data structure that supports Close Suffixes Rank queries in $O(\log^3(n))$ time for $k = \sqrt{n}$. The data structure can be maintained in $O(\sqrt{n})$ when a substitution update is applied to $S$.

With Theorem 20 and the above reduction using the $k$-words tree, our main result for inverted suffix array is trivially achieved:

Theorem 21. Given a dynamic string $S[1...n]$ there is a data structure that reports $iSA_S[i]$ in $O(\log^4(n))$ time. The data structure can be maintained in $O(\sqrt{n})$ time when a substitution update is applied to $S$. The data structure uses $O(n)$ space.

3.3 Dynamic Suffix Array

A query $i$ for $SA_S[i]$ can also be reduced to a query about close suffixes using the $k$-words tree by applying the following recursive algorithm:

$Findvr(Root, i)$:

Input: An integer $i$ and a node $Root$ in the $k$-words tree. Initially the root.

Output: A node $v$ and an integer $r$.

Let $R$ and $L$ be the right and the left children of $Root$, respectively. Let $|T(R)|$, $|T(L)|$ be the number of words in the trees rooted in $R$ and in $L$, respectively. Let $|Root|$ be the number of words contained in the root node.

1. If $|T(L)| \geq i$ return $Findvr(L, i)$.
2. If $|T(L)| + |Root| \leq i$ return the tuple $(Root, i - |T(L)|)$.
3. If $|T(L)| + |Root| < i$ return $Findvr(R, i - |T(L)| - |Root|)$

The following can be easily verified:

Fact 22. Applying Algorithm $Findvr(R, i)$ with $R$ the root of the $k$-words tree of $S$ yields the node $v$ containing $SA[i]$ and the lexicographic rank $r$ of $SA_S[i]$ within the suffixes close to $SA_S[i]$. The time complexity of Algorithm $Findvr$ is $O(\log(n))$. 

9
$w = abaaba$

Figure 1: Periodic cluster of occurrences and its properties. The lines above the string segment denote the occurrences implied by the cluster. Note that in this example, $Tail(C) = bab\ldots > baa = w_s$. Therefore, the cluster is increasing.

With that, reporting $SA[i]$ is reduced to a Close Suffix Selection Query defined below.

### Close Suffixes Select

Given a subword of size $k$ represented as an index $i \in [1\ldots n - k + 1]$ and a rank $r$, return the suffix $S[i\ldots n]$ with lexicographic rank $r$ among the suffixes starting with $S[i\ldots i + k - 1]$.

In Section 4, we prove the following:

**Theorem 23.** Given a dynamic text $S[1\ldots n]$, there is a data structure that supports close suffixes selection queries in $O(\log^2(n))$ time. The data structure can be maintained in $\tilde{O}(\frac{n}{k}^2 + k)$ time when an edit operation is applied to $S$. The data structure uses $\tilde{O}(n + \frac{n}{k}^2)$ space.

By setting $k = n^\frac{2}{3}$ and applying the above reduction with a $k$-words tree, we obtain the following:

**Theorem 24.** For a dynamic string $S[1\ldots n]$, there is a data structure that supports look up queries to $SA_S[i]$ in $O(\log^2(n))$ time. When an edit operation update is applied to $S$, the data structure can be maintained in $\tilde{O}(n^{\frac{2}{3}})$ time. The data structure uses $\tilde{O}(n)$ space.

### 4 Dynamic Close Suffix Select Queries

In this section, we present the main idea for proving Theorem 23. Let $w$ be a word of size $t = \frac{k}{2}$. We denote as $POR_w$ the periodic occurrences representation of $w$ in $S[1\ldots n]$ and $Occ_w$ the set of occurrences of $w$. We denote as $A_w[i\ldots |Occ_w|]$ the array consisting of the starting indexes of occurrences of $w$ sorted by lexicographic order of the suffixes starting in these indexes. The size of $POR_w$ is bounded by $O(\frac{n}{k})$ while the size of $A_w$ may be $\Omega(n)$.

Let $C = (a, b, p) \in POR_w$ be a periodic cluster of occurrences of $w$ with period $per(w) = p$.

**Definition 25.** Let $w_t = S[s_t\ldots e_t]$ be an occurrence implied by $C$ of $w = S[s\ldots e]$ and let $w_s = w[e - p + 1\ldots e]$ be the suffix of size $p$ of $w$. The period rank of $w_t$, denoted as $r_p(w_t)$, is the maximal integer $x$ such that $S[e_t + 1\ldots e_t + p - x] = (w_s)^x$. Equivalently, $r_p(w_t)$ is the maximal number of consecutive occurrences of $w_s$ following $w_t$.

One can easily observe that the $t$'th occurrence in $C$ has $r_p(w_t) = |C| - t - 1$. (Recall that the occurrences within a cluster are indexed with zero based indexes)

**Definition 26 (Cluster Head, Cluster Tail, Increasing and Decreasing Clusters).** Let $w_0 = S[s_0\ldots e_0]$ and $w_{|C| - 1} = S[e_{|C| - 1}\ldots e_{|C| - 1}]$ be the leftmost and the rightmost occurrence of $w$ represented by $C$, respectively (with respect to starting index). Let $w_s = w[e - p + 1\ldots e]$ be the suffix of size $p$ of $w$. We call the prefix $S[1\ldots s_0 - 1]$ the head of $C$ denoted as $Head(C)$ and the suffix $S[e_{|C| - 1} + 1\ldots n]$ the tail of $C$ denoted as $Tail(C)$. We call $C$ an increasing cluster if $Tail(C) >_L w_s$, or a decreasing cluster if $Tail(C) <_L w_s$.

See Figure 1 for an example of a cluster and its properties. In Section 7 we prove that $A_w$ has the following structure (derived from Lemma 74 and Lemma 75):
For every occurrence containing the set of occurrences implied by decreasing clusters, and \( r \) being the set of occurrences implied by increasing clusters.

\( D \) can be written as the concatenation \( D = D_0, D_1 \ldots D_{m-1}, D_m \) where \( m \) is the maximal period rank of an occurrence in a decreasing cluster. For every \( r \in [0 \ldots m] \), \( D_r \) contains the occurrences \( w' \) from decreasing clusters with \( r_p(w') = r \) sorted by lexicographic order of the tails of their clusters.

Similarly, \( I = I_t, I_{t-1} \ldots I_1, I_0 \) where \( t \) is the maximal period rank of an occurrence in an increasing cluster. For every \( r \in [0 \ldots t] \), \( I_r \) contains the occurrences \( s \) from increasing clusters with \( r_p(s) = r \) sorted by lexicographic order of the tails of their clusters (See Figure 5).

Let \( w = S[s \ldots e] \) be a subword of \( S \). Let \( l \) and \( r \) be two non-negative integers.

**Definition 27.** An occurrence \( w_1 = S[s_1 \ldots e_1] \) of \( w \) is an \((l, r)\)-extendable occurrence of \( w \) if \( lcs(s - 1, s_1 - 1) \geq l \) and \( lcp(e + 1, e_1 + 1) \geq r \). We define the array \( A^{(l, r)}_w \) as the lexicographically sorted array of suffixes of \( S \) starting with an \((l, r)\)-extendable occurrence of \( w \).

It is easy to see that given a word \( w^e = S[s - e \ldots e + r] \) containing \( w \), an occurrence \( w^e \) in index \( i \) is equivalent to an \((l, r)\)-extendable occurrence of \( w \) in \( i + l \). Therefore, \( A^{(l, r)}_w[i] = A^{(l, r)}_w[i] - l \) for every \( i \in [1 \ldots |A^{(l, r)}_w|] \). We consider the following problem:

**Extension Restricting Selection**

**Preprocess:** A word \( w = S[s \ldots e] \) and \( POR_w \)

**Query:** upon input \( l, r, i \). Report \( A^{(l, r)}_w[i] \)

In Section 7 we prove the following:

**Theorem 28.** If LCE queries can be evaluated in \( O(\log(n)) \) time, the periodic occurrences representation \( POR_w \) of a word \( w = S[s \ldots e] \) can be preprocessed in time \( \widetilde{O}(|POR_w|) \) to answer Extension Restricting Selection queries in \( O(\log^3(n)) \) time. The data structure uses \( \widetilde{O}(|POR_w|) \) space.

Constructing the data structure described in Theorem 28 is the most technically involved part in the construction of our dynamic suffix array data structure. It is achieved by exploiting the characterization of the structure of \( A_w \). Our insights on the structure of \( A_w \) allow us to reduce an Extension Restricting Selection query to a small set of multidimensional range queries.

We are now ready to present the algorithm for dynamic Close Suffix selection queries. We initialize a data structure for supporting dynamic LCE queries on \( S \) by employing Lemma 11. We also initialize a dynamic data structure for obtaining periodic occurrences representations by employing Fact 15. Let \( t = \lfloor \frac{|S|}{2} \rfloor \). We partition \( S \) into words of size \( t \) starting in indexes \( i = 0 \mod t \). For every word \( w_x = S[x \cdot t \ldots (x + 1) \cdot t - 1] \) for \( x \in [0 \ldots \lfloor \frac{|S|}{t} \rfloor] \), we obtain \( POR_{w_x} \) in \( \widetilde{O}(\frac{n}{t}) \) time. We preprocess \( POR_{w_x} \) for Extension Restricting Selection queries using Theorem 28. We keep an array \( ERSQ[1 \ldots \lfloor \frac{n}{t} \rfloor] \) with \( ERSQ[x] \) containing a pointer to the Extension Restricting Selection queries data structure of \( w_x \).

Upon an edit operation update, we update the data structures for LCE queries and for obtaining the periodic occurrence representation. We then partition the updated \( S \) in the same manner as in the initialization step. Note that the words in the partition may change, as the size of \( S \) may change due to an insertion or a deletion update, and \( k \) may be a function of \( |S| = n \). For every word \( w_x \) in the updated partition, we extract \( POR_{w_x} \) and evaluate the extension restricting selection queries data structures for every word \( w_x \) from scratch. We also construct the array \( ERSQ \) from scratch as in the initialization step.

Upon a Close Suffix Selection query \((i, r)\), the word \( w_{[i \ldots i + k - 1]} \) is completely contained within \( w = S[i \ldots i + k - 1] \). Let \( L \) be the distance between the starting indexes of \( w_{[i \ldots i + k - 1]} \) and \( w \) and let \( R \) be the distance between the ending indexes. We execute an Extension Restricting Selection query \((L, R)\) on the Extension Restricting Selection queries data structure of \( w_{[i \ldots i + k - 1]} \). Let the output of the query be \( z \). We return \( z - L \).

**Correctness:** A Close Suffix Selection query \((i, r)\) is equivalent to reporting \( A_w[r] \). Denote the indexes of \( w_{[i \ldots i + k - 1]} = S[i \ldots e] \). The word \( w = S[i \ldots i + k - 1] \) can be written as \( w = S[s - L, e + R] \). Therefore, \( A_w[r] = A^{(L, R)}_{w_{[i \ldots i + k - 1]}}[r] - L \).

**Preprocessing time:** The preprocessing times of the LCE data structure and the periodic occurrences data structure are both bounded by \( \widetilde{O}(n) \). The remaining operations are, in fact, an initial update. The complexity analysis for these is provided below.

**Update time:** Evaluating the partition can be executed in \( O(\frac{n}{t}) \) time in a straightforward manner. There are \( \frac{n}{t} \) words in our partition. For every word \( w \), we find \( POR_w \) in \( \widetilde{O}(\frac{n}{t}) \) time. Since
In Section 8, we prove the following:

We call the interval increased by exactly \( r(i) \) close suffixes rank queries an ‘array’ of integers \( R[1 \ldots n] \), initialized as the initial close suffix ranks calculated by Lemma 29 \((R[i] = r(i))\). When a substitution update is applied to \( S \) the ranks \( r(i) \) of the suffixes change. We wish to change \( R[i] \) in a corresponding manner.

We provide an analysis of the updates that are need to be applied to \( R \) when a symbol substitution is applied to \( S \). Unsurprisingly, these changes are too wild to be efficiently applied to an ordinary array (e.g. - \( \Omega(\sqrt{n}) \) indexes may have to be updated). However, the updates to the ranks can be described as a small set of interval increment and stairs updates on \( R \).

**Definition 30.** An interval increment update is denoted by \( I = (i, j, x) \). Applying \( I \) to \( R \) increases the value of \( R[a] \) by \( x \) for \( a \in [i \ldots j] \).

**Definition 31.** A decreasing stairs update \( (See \ Figure 3\) is denoted by \( S = (i, j, p) \). Applying \( S \) to \( R \) increases the value of the indexes in \( R[i \ldots j] \) as follows: the rightmost \( p \) indexes \( R[j - p + 1 \ldots j] \) are increased by \( 1 \), the \( p \) indexes to their left are increased by \( 2 \), and so on until we finally exceed \( R[i-2 \ldots j] \).

Formally, for every \( t \in [1 \ldots \lfloor \frac{j - i + p - 1}{p} \rfloor] \), the counters \( R[a] \) with \( a \in [\max(j - t \cdot p + 1, i) \ldots j - (t - 1) \cdot p] \) are increased by \( t \).

We call \( p \) the step width of the stairs update \( S \). We call the interval increased by exactly \( t \) the \( t \)th step of the update.

An increasing stairs update is defined symmetrically, with the ‘stairs’ having increasing (rather than decreasing) values from left to right. A stairs update can also be a negative update, decreasing the values of the \( t \)th step by \( t \) rather than increasing it by \( t \). A formal definition of these variants can be found in Section 8.

**Example 32.** Let \( R = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0) \). Applying a decreasing stairs update \( (3, 8, 2) \) on \( R \) will result in \( R' = (0, 0, 0, 3, 3, 2, 2, 1, 1, 0, 0) \). Applying an increasing stairs update \( (2, 8, 3) \) instead will result in \( R' = (0, 1, 1, 1, 2, 2, 2, 3, 3, 0, 0) \). Applying a negative decreasing stairs update \( (4, 10, 2) \) on \( R' \) will result in \( R'' = (0, 1, 1, -3, -2, -1, -1, 1, 1, -1, -1) \).

In section 8 we prove the following:
Figure 2: A visualization of an application of a decreasing stairs update and its properties. The update $S$ is applied to the array $R$ to obtain $R'$. 

**Theorem 2.** There is a data structure for maintaining an indexed set of integers $R$ to support both applying a stairs update to $R$ and look-up queries $R[i]$ in $O(\log^3(U))$ time, provided that all the updates $(i, j, p)$ have the same value of $p$. $U$ is the overall amount of updates. The data structure consumes $O(U \log^2(U))$ space. Initializing the data structure (all counters are set to 0) takes $O(1)$ time.

For the sake of self containment, we provide a proof for the following (simple, possibly folklore) claim:

**Claim 33.** There is a data structure for maintaining an indexed set of integers $R$ to support both applying an interval increment update to $R$ and look-up queries $R[i]$ in time $O(\log(n))$.

Our algorithm may require stairs updates with varying values of $p$. We will eventually describe the details for how the restricted data structure of Theorem 2 is utilized. Until then, we refer to stairs update application and query as an $O(\log^3(n))$ time procedure. We want stairs updates to be executed in $O(\log(n))$ rather than $O(\log^3(U))$. This can be achieved in amortized time by rebuilding our data structure from scratch after every $O(n)$ updates. The running time can be deamortized using standard techniques.

From now on, we fix $x \in [1 \ldots n]$ to be the index within $S[1 \ldots n]$ such that $S[x]$ was substituted. By $S$ we refer to the text after the update was applied, and by $T$ we refer to the text prior to the substitution. For an index $i \in [1 \ldots n]$, we denote the word $w_{i,old}^n = S[i \ldots i + k - 1]$ and $w_{i,old}^n = T[i \ldots i + k - 1]$. For $w_{i,new}^n$ and $w_{i,old}^n$ to be defined for every $i \in [1 \ldots n]$, we process $S$ as if it has the string $\$^{k}$ appended to its end for some symbol $\not\in \Sigma$ that is lexicographically smaller than every symbol $\sigma \in \Sigma$. Note that this does not affect the lexicographical order between the suffixes of $S$.

Consider the following clustering of the suffixes of $S$: two suffixes $T^i$ and $T^j$ are in the same cluster if and only if they are closed. Equivalently, every cluster $C$ corresponds to a word $w_C$ and a suffix $T^i$ is in $C$ if and only if $w_{i,old}^n = w_C$. Even though we do not explicitly maintain such a clustering in our data structure, it is helpful to think of the suffixes of $S$ as if they are stored in this manner.

Observe that $R[i]$ is the lexicographic rank of $T^i$ among the suffixes in $C_{w_{i,new}^n}$. After an update is applied to $T$, $R[i]$ needs to be modified to be the rank of $S^i$ among the suffixes in a possibly different cluster $C_{w_{i,new}^n}$. We distinguish between two types of suffixes:

1. **Dynamic suffixes:** $S^i$ with $x \in [i \ldots i + k - 1]$: For these suffixes, $w_{i,old}^n \neq w_{i,new}^n$.
2. **Static suffixes:** $S^i$ with $x \not\in [i \ldots i + k - 1]$. For these suffixes, $w_{i,old}^n = w_{i,new}^n$. 


Figure 3: A visualization of a shift update. The (mostly) red rectangles resemble $w^\text{old}_i$ and $w^\text{old}_j$. In $T$, the words $w^\text{old}_i$ and $w^\text{old}_j$ are identical, except for the marked non-red rectangle, which represents a ‘blue’ symbol in $w^\text{old}_i$ and a ‘green’ symbol in the corresponding location in $w^\text{old}_j$. The blue arrows to the right of $w^\text{old}_i$ and $w^\text{old}_j$ resemble the LCP between $i + k - 1$ and $j + k - 1$. In this example, the green symbol in $w^\text{old}_i$ is modified by an update to a blue symbol, so we have $w^\text{new}_i = w^\text{new}_j$. The symbol following the LCP after $i$ is 1, and the symbol following the LCP after $j$ is 0, so we have $S^i < L S^j$. The suffix $S^j$ became close to $S^i$ as a result of the last substitution, so $r(i)$ is increased by 1. The reason is that we have a new suffix close to $S^i$ that is lexicographically smaller than $S^i$.

Figure 4: A visualization of an overtake update. The red rectangles resemble $w^\text{old}_i$ and $w^\text{old}_j$, and we have $w^\text{old}_i = w^\text{old}_j$, so $T^i$ and $T^j$ are close suffixes. The substitution occurs in an index that is not contained either in $w^\text{old}_i$ or in $w^\text{old}_j$, therefore they are both static suffixes. The lcp between $i + k - 1$ and $j + k - 1$ is denoted by the blue arrows. Prior to the update, the lcp terminates in a 1 following $S^i$ and in a 0 following $j$. Therefore, $T^j > L T^i$. After the update is applied, the lcp is extended by 1 symbol and we have $S^i < L S^j$, so $S^j$ overtake $S^i$. The rank of $S^i$ should be decreased by 1 and the rank of $S^j$ should be increased by 1.

Note that dynamic suffixes are suffixes that are moved to a different cluster as a result of the update, and static suffixes are the suffixes that stay in the same cluster after the update is applied. We partition the updates applied to $R$ into 3 types:

1. **Evaluation Updates** - Evaluating $R[i]$ for every dynamic suffix $S^i$.

2. **Shift Updates** - Increasing (resp. decreasing) by 1 the values of $R[i]$ for a static suffixes $S^i$ as a result of a dynamic suffix $S^j$ that has $w^\text{new}_j = w^\text{new}_i$ (resp. $w^\text{old}_j = w^\text{old}_i$) and $S^j < L S^i$ (resp. $T^j < L T^i$) (See Figure 3).

3. **Overtakes** - For a pair of close static suffixes $S^i, S^j$ such that $S^i > L S^j$ and $T^i < L T^j$, increasing the value of $R[i]$ by 1 and decreasing the value of $R[j]$ by 1 (See Figure 4).

An evaluation update corresponds to evaluating the rank of a dynamic suffix within the new cluster for which it was recently inserted. A shift update corresponds to an increase (resp. decrease) in the rank of a static suffix $S^i$ as a result of a dynamic suffix that is lexicographically smaller than $S^i$ being inserted (resp. deleted) from the cluster of $S^i$. An overtake corresponds to a change in lexicographical order between two static suffixes within the same cluster.

It should be clear that if we apply all the evaluation updates, all the shift updates, and all the overtake updates to $R$, then $R$ is properly set and $R[i] = r(i)$ for every $i \in [1 \ldots n]$ with respect to the close suffix ranks $r(i)$ in the updated text $S$. In the following sections we show how to efficiently apply these types of updates to $R$. 

14
5.1 Evaluation updates

In this section, we present an $O(\sqrt{n})$ time algorithm for evaluating $r(i)$ for $i \in [x - k + 1 \ldots x - \frac{k}{2} + 1]$. These are only half of the dynamic suffixes. We conclude by explaining how the same techniques can be used to construct a similar algorithm to evaluate $r(i)$ for $i \in [x - \frac{k}{2} \ldots x]$. The task of evaluating $r(i)$ can be equivalently described as a counting problem. Specifically, $r(i)$ is the number of suffixes $S^j$ starting with $w_{i}^{\text{new}}$ such that $S^j <_L S^i$. With that characteristic in mind, a naive approach would be the following: Iterate the values $i \in [x - k + 1 \ldots x]$ and for every value, evaluate the set of close suffixes and count how many of these are lexicographically smaller than $S^i$. An explicit straightforward execution of this idea would result in an algorithm with running time $\tilde{O}(n^{1.5})$, since the number of dynamic suffixes is $O(k) = O(\sqrt{n})$ and the number of suffixes close to a certain dynamic suffix can not be bounded by anything smaller than $O(n)$. Of course, this is not a satisfactory running time. We present an implementation of the naive algorithm that exploits the structure of the problem to achieve a time complexity of $O(\sqrt{n})$.

Our enhanced implementation of the naive approach relies on exploiting the proximity of the starting indexes of the words $w_{i}^{\text{new}}$ corresponding to dynamic suffixes $S^j$. Instead of evaluating the set of occurrences of $w_{i}^{\text{new}}$ for every $i \in [x - k + 1 \ldots x - \frac{k}{2} + 1]$, we consider the occurrences of the word $w_{i}^{\text{old}} = S[x - \frac{k}{2} + 1 \ldots x]$. Note that $w_{i}^{\text{old}}$ is a subword of all the words $w_{i}^{\text{new}}$ with $i \in [x - k + 1 \ldots x - \frac{k}{2} + 1]$. We use an occurrence of $w_{i}^{\text{old}}$ as an anchor that may be extendable to an occurrence of $w_{i}^{\text{new}}$ for some values of $i$. Since $w_{i}^{\text{old}}$ has length $\Theta(k)$, the occurrences of $w_{i}^{\text{old}}$ in $S$ can be represented as a set of $O(\frac{k}{\sqrt{n}}) = O(\sqrt{n})$ clusters (Fact 14). We provide a method to efficiently process a cluster $C$ of occurrences of $w_{i}^{\text{old}}$ to deduce the modifications to $r(i)$ that are required as a result of occurrences represented by $C$.

To simplify notations, we define the function $D(\delta) = x - \frac{k}{2} + 1 - \delta$. Upon input integer $\delta$, the function $D(\delta)$ outputs the index in $S$ that $\delta$ indexes to the left from the starting index of $w_{i}^{\text{old}}$. The following fact captures the relation between the occurrences of $w_{i}^{\text{old}}$ and the suffixes that should be counted by our algorithm.

(The indexes of $w_{i}^{\text{old}}$ are renamed to $S[s \ldots e]$ for clearer presentation)

Fact 34. Let $S^j$ be a dynamic suffix with $i = D(\delta)$ for some $\delta \in [0 \ldots \frac{k}{2}]$. For an index $j \in [1 \ldots n]$, the following are equivalent:

1. $S^j$ is close to $S^i$ (starts with an occurrence of $w_{i}^{\text{new}}$) and $S^j <_L S^i$.

2. There is an occurrence $w_1 = S[s_1 \ldots e_1]$ of $w_{i}^{\text{old}} = S[s \ldots e]$ in index $s_1 = j + \delta$ such that $S^{s_1} <_L S^s$,

\[
\text{lcs}(s_1 - 1, s - 1) \geq \delta \quad \text{and} \quad \text{lcp}(e_1 + 1, e + 1) \geq \frac{k}{2} - \delta.
\]

Fact 34 holds due to the fact that $w_{i}^{\text{old}}$ is a subsequence of $w_{i}^{\text{new}}$ for these values of $i$. An occurrence of $w_{i}^{\text{new}}$ is equivalent to an occurrence of $w_{i}^{\text{old}}$ that has a sufficiently long common extension with $w_{i}^{\text{old}}$. See Figure 5 for a visualization.

For $i = D(\delta)$ with $\delta \in [0 \ldots \frac{k}{2}]$, we say that an occurrence of $w_{i}^{\text{new}}$ in index $j$ is induced by the occurrence of $w_{i}^{\text{old}}$ in index $j + \delta$.

We use Fact 34 to evaluate $r(i)$ for every $i \in [x - k + 1 \ldots x - \frac{k}{2} + 1]$ in a parallel manner. Instead of processing occurrences of $w_{i}^{\text{new}}$ for every dynamic suffix $S^j$ with $i \in [D(\frac{k}{2}) \ldots D(0)]$, we process the occurrences of $w_{i}^{\text{old}} = S[s \ldots e]$. For every occurrence of $w_1 = S[s_1 \ldots e_1]$, we identify values of $i$ for which $w_{i}^{\text{new}}$ is induced by $w_1$ according to Fact 34. If $S^{s_1} <_L S^s$, we count the induced suffixes as smaller close suffixes of their corresponding dynamic suffixes.

A technical implementation of the above approach is given as Algorithm 1.

Algorithm 1

```plaintext
Initialize an array of counters $Re[0 \ldots \frac{k}{2}]$ with zeroes.

For every occurrence $w = S[s_1 \ldots e_1]$ of $w_{i}^{\text{old}}$:

1. Check if $S^{s_1} <_L S^s$. If it is not - stop processing this occurrence.

2. Compute $l = \text{lcs}(s_1 - 1, s - 1)$ and $r = \text{lcp}(e_1 + 1, e + 1)$.

3. Increase the counters of $Re$ with indexes in $[\max(0, \frac{l}{2} - r), \min(l, \frac{k}{2})]$ by 1.

For every $\delta \in [0 \ldots \frac{k}{2}]$, set $R[D(\delta)] \leftarrow Re[\delta]$.
```

Claim 35. By the end of the inner loop of Algorithm 1, $Re[\delta] = r(D(\delta))$.
occurrences of \( w \) and \( r \) contain a constant number of occurrences for which Lemma 36 does not provide a representation of.

The values of \( \text{lcs} \) to the right of \( \text{Ex} \) and \( \text{Ex} \) and the run with period \( p \) that satisfies the following conditions is processed:

Let \( l \) be the lengths of the extension of the run with period \( p \) containing \( w \) to the left of \( s - 1 \) and to the right of \( e + 1 \), respectively. Let \( \text{Ex}_l \) and \( \text{Ex}_r \) be the lengths of the extension of the run with period \( p \) containing \( w \) to the left of \( s_0 - 1 \) and to the right of \( e_0 + 1 \), respectively.

1. If \( \text{Ex}_l \neq \text{Ex}_l^C + p \cdot t \), we have \( l_t = \min(\text{Ex}_l, \text{Ex}_l^C + p \cdot t) \).
2. If \( \text{Ex}_r \neq \text{Ex}_r^C - p \cdot t \), we have \( r_t = \min(\text{Ex}_r, \text{Ex}_r^C - p \cdot t) \).

The values of \( \text{Ex}_l \), \( \text{Ex}_r \), \( \text{Ex}_l^C \) and \( \text{Ex}_r^C \) can be evaluated in \( O(\log(n)) \) time given \( C \), the indexes of \( w \), and a data structure for \( O(\log(n)) \)-time LCE queries.

For most of the occurrences \( w \) represented by a cluster \( C \), Lemma 36 provides a compact, efficiently computable representation of the common extensions of \( w \) and \( w \). We note that every cluster may contain a constant number of occurrences for which Lemma 36 does not provide a representation of \( l_t \) and \( r_t \). We formally define these occurrences as follows.
**Definition 37.** Let \( C = (a, b, p) \) be a cluster of occurrences of a word \( w = S[s \ldots e] \) with \( \text{per}(w) = p \) in \( S[1 \ldots n] \). Let \( E_x, E_y, Ex_x \) and \( Ex_y \) be the lengths of the extension of the runs as in Lemma 36. \( w_2 \) is an aligned occurrence if \( Ex_x = p \cdot t \) or \( Ex_y = Ex_y - p \cdot t \).

The aligned occurrences within a cluster \( C \) can be easily found in constant time given \( C, w \) and the runs extension lengths. The two constraints defining aligned occurrences are linear equations with respect to \( t \). Therefore, there is at most one value \( t \) satisfying each equation and it can be found in constant time.

For evaluation updates, we are only interested in the LCE values that correspond to occurrences \( w_2 \) with \( S^{w_2} \preceq_S S^x \). To extract these values, we employ the following fact proven in Section 6.

**Claim 38.** Let \( s \) be an index and let \( C = (a, b, p) \) be a periodic cluster with \( s_t = a + t \cdot p \). There are consecutive intervals \( I_\prec, I_\succ \subseteq [0 \ldots |C| - 1] \) such that \( S^{w_2} \preceq_S S^x \) iff \( t \in I_\prec \) and \( S^{w_2} >_S S^x \) iff \( t \in I_\succ \). The intervals \( I_\prec \) and \( I_\succ \) can be evaluated in \( O(\log(n)) \) time given \( C, s \) and a data structure for \( O(\log(n)) \) LCE queries.

Let \( l_t = \text{lcs}(s - 1, s_t - 1) \) and \( r_t = \text{lcp}(e + 1, e_t + 1) \). We evaluate a compact representation of the list \( L^x = l_0, l_1, l_2, \ldots l_{|C| - 1} \) by applying Lemma 36. We can obtain \( I_\prec = [i \ldots j] \) by applying Claim 38. Recall that we are interested in occurrences \( w_1 \) such that \( S^{w_1} <_S S^x \). So the relevant LCS values are \( l_t \) with \( t \in I_\prec \). We proceed to work on the reduced list \( L_\prec = l_i, l_{i+1}, \ldots l_j \). For similar arguments, we proceed to work on the reduced list \( R_\succ = r_t, r_{t+1}, \ldots r_r \) corresponding to the relevant values of \( r_t \).

So far, we characterized the structure of \( \text{lcp} \) and \( \text{lcs} \) between a word and a set of its occurrences represented by a cluster. In what follows, we provide a method to efficiently apply the updates that are derived from these values to \( R \).

**Definition 39 (p-Normal Sequence).** Let \( B = b_0, b_1, \ldots b_t \) be a sequence of integers. \( B \) is a \( p \)-normal sequence for some integer \( p \) if:

1. \( B \) is a fixed sequence: \( b_i = b_0 \) for every \( i \in [1 \ldots t] \), or
2. \( B \) is an arithmetic progression with difference \( p \): \( b_i = b_0 + p \cdot t \) for every \( i \in [1 \ldots t] \), or
3. \( B \) is the maximum of two \( p \)-normal sequences: \( b_i = (\max(a_i^1, a_i^2)) \) for two \( p \)-normal sequences \( \{a_i^1\} \) and \( \{a_i^2\} \), or
4. \( B \) is the minimum of two \( p \)-normal sequences: \( b_i = (\min(a_i^1, a_i^2)) \) for two \( p \)-normal sequences \( \{a_i^1\} \) and \( \{a_i^2\} \).

The degree of a \( p \)-normal sequence is denoted as \( \text{deg}(B) \). If \( B \) is either fixed or an arithmetic progression then \( \text{deg}(B) = 1 \). If \( B \) is a maximum sequence with \( b_i = (\max(a_i^1, a_i^2)) \), or a minimum sequence with \( b_i = (\min(a_i^1, a_i^2)) \) then \( \text{Deg}(B) = \text{Deg}(\{a_i^1\}) + \text{Deg}(\{a_i^2\}) \).

Finally, the construction tree of a \( p \)-normal sequence \( B \) is a full binary tree denoted as \( \text{CT}(B) \). It aims to represent the structure of \( B \). If \( B \) has \( \text{deg}(B) = 1 \) then \( \text{CT}(B) \) consists of a single node. If \( B \) is a maximum sequence with \( b_i = (\max(a_i^1, a_i^2)) \), or a minimum sequence with \( b_i = (\min(a_i^1, a_i^2)) \) then \( \text{CS}(B) \) consists of a root with the left child being \( \text{CT}(\{a_i^1\}) \) and the right child being \( \text{CT}(\{a_i^2\}) \).

**Example 40.** The sequence \( A = 4, 7, 10, 13, 16 \) is a 3-normal sequence with degree 1, as it is an arithmetic progression with difference \( p = 3 \). \( \text{CT}(A) \) is a single root node \( \{A\} \). The sequence \( B = b_0, b_1, \ldots b_t \) with \( b_i = \max(13, 4 + 7 \cdot t) \) is a 7-normal sequence. The left input in the max expression is a fixed sequence with respect to \( t \), which is a \( p \)-normal sequence with degree 1. The right sequence is an arithmetic progression with difference 7 with respect to \( t \), which is a 7-normal sequence with degree 1. Therefore, \( \text{deg}(B) = 1 + 1 = 2 \). \( \text{CS}(B) \) is a root node with two leaf children corresponding to the construction trees of \( [7, 7, 7 \ldots] \) and \( [4, 11, 18 \ldots] \).

The following lemma is proven in Section 6.

**Lemma 41.** Let \( U = u_1, u_2, \ldots, u_{|U| - 1} \) be a set of interval increment updates with \( u_i = (i_t, j_t, 1) \) and both \( \{i_t\} \) and \( \{j_t\} \) being \( p \)-normal sequences for some integer \( p \) with degrees \( \text{Deg}(\{i_t\}), \text{Deg}(\{j_t\}) \in O(1) \). Applying all the interval updates in \( U \) is equivalent to applying a constant number of interval increment updates and stairs updates with stair width \( |p| \) to \( R \). The set of equivalent stairs and interval updates can be evaluated in constant time given the representation of \( i_t \) and \( j_t \) as \( p \)-normal sequence (The input can be given, for example, as the construction trees of \( \{i_t\} \) and \( \{j_t\} \)).
Consider the occurrence \( w_t \) of \( w^e \) starting in index \( s_i \) represented by \( C \) for some \( t \in I_\prec = [i_\prec \ldots j_\prec] \). When \( w_t \) is iterated in Algorithm 1, the interval increment \((\max(0, r_t - \frac{k}{2}), \min(l_t, \frac{k}{2}))\) is applied to \( Re \). Denote the left border of the interval increment applied when \( w_t \) is iterated as \( i_t = \max(0, r_t - \frac{k}{2}) \) and the right border as \( j_t = \min(l_t, \frac{k}{2}) \). We make the following claim:

**Claim 42.** Given \( C, w = S[s \ldots e] \) and \( I_\prec \), all the updates \( u_t = (i_t, j_t, 1) \) for \( t \in I_\prec \) can be applied to \( Re \) in time \( O(\log^3(n)) \).

**Proof:** Let \( Al \) be the set of aligned occurrences within \( C \). It is known that \( |Al| \leq 2 \). Denote the set of aligned occurrences as \( Al = \{t_1, t_2\} \) with \( t_1 < t_2 \). If \( |Al| \leq 1 \), we set \( t_2 = \infty \). If \( |Al| = 0 \), we set \( t_1 = -\infty \). For \( t \notin Al \) we have \( Ex_t \neq Ex_t^C \) and \( Ex_t \neq Ex_t^C - p \cdot t \). Therefore, \( i_t = \max(0, \min(Ex_t, Ex_t^C - p \cdot t)) \) and \( j_t = \min(\min(Ex_t, Ex_t^C - p \cdot t), \frac{k}{2}) \) according to Lemma 36. These are \((p)\)-normal sequence with degree \( d_t = 3 \). For the integers in \( I_1 = [i_\prec \ldots t_1] \), both \( \{i_t\} \) and \( \{j_t\} \) are \( p\)-normal sequences. The same applies for the intervals \( I_2 = (t_1 \ldots t_2) \) and \( I_3 = (t_2 \ldots j_\prec) \). All the updates \( u_t \) for \( t \in I_1 \cup I_2 \cup I_3 \) can be expressed as a constant number of stairs updates by applying Lemma 41 for every interval. Interval updates corresponding to aligned occurrences can be applied in \( O(\log(n)) \) time each by employing Claim 33. Applying a constant number of stairs and interval updates can be executed in \( O(\log^3(n)) \) time by Theorem 2. The representations of \( r_t \) and \( l_t \) can be obtained in \( O(\log(n)) \) time by employing Lemma 36.

We are now ready to present an efficient implementation of Algorithm 1.

**Algorithm 2**

1. Initialize an array of counters \( Re[0 \ldots \frac{k}{2}] \) with zeroes.
2. Obtain the periodic occurrences representation \( por \) of \( w^e \).
3. For every singular occurrence \( w = S[s \ldots e] \) in \( por \):
   1. Check if \( S^{s_1} < \leq S^s \). If it is not, then stop processing this occurrence.
   2. Compute \( l = \text{lcs}(s_1 - 1, s - 1) \) and \( r = \text{lcp}(e_1 + 1, e + 1) \).
   3. Increase the counters of \( Re \) with indexes \( \{\max(0, r - \frac{k}{2}), \min(l, \frac{k}{2})\} \) by 1.
4. For every cluster \( C = (a, b, c) \) of occurrences in \( por \):
   1. Obtain \( I_\prec \) using Claim 35.
   2. Apply all the updates corresponding to occurrences \( w_t \) with \( t \in I_\prec \) implied by \( C \) by employing Claim 42.
5. For every \( \delta \in [0 \ldots \frac{k}{2}] \), set \( R[D(\delta)] \leftarrow Re[\delta] \).

It is straightforward that the updates applied to \( Re \) in Algorithm 2 are equivalent to the updates applied to \( Re \) in Algorithm 1. Therefore, the close suffix ranks evaluated in \( Re \) are correct directly from Claim 33.

**Claim 43.** Algorithm 2 runs in \( \tilde{O}(\sqrt{n}) \) time.

**Proof:** Since \( |w^e| = \frac{k}{2} \), the size of the periodic occurrences representation is \( \frac{2n}{k} \in O(\sqrt{n}) \) and it can be obtained in time \( O(\sqrt{n}) \) by applying Lemma 15.

For a singular occurrence in the representation, we process in logarithmic time just as in Algorithm 1. For a cluster, we use Claim 36 and Claim 42 to apply all the updates corresponding to occurrences implied by \( C \). The bottleneck of handling a cluster is the application of Claim 2 in \( O(\log^3(n)) \) time. The overall time to process an element, either a singular occurrence or a cluster, is polylogarithmic. Since there are \( O(\sqrt{n}) \) elements in the representation. Algorithm 2 runs in \( \tilde{O}(\sqrt{n}) \) time.

Note that Algorithm 2 initializes the data structure \( Re \) for stairs updates with a constant stair size \( p \), uses it to evaluate the ranks and inserts its values to the matching indexes in \( R \). After the execution of Algorithm 2, \( Re \) can be discarded, since it is not used in future updates. The restricted result of Theorem 2 is sufficient for this case.

This concludes the evaluation of the ranks for dynamic suffixes starting in \( [x - k + 1 \ldots x - \frac{k}{2}] \). We still need to handle the suffixes starting in \( [x - \frac{k}{2} + 1 \ldots x] \). This is done in the same manner, but instead of \( w^e = S[x - \frac{k}{2} + 1 \ldots x] \) we consider the occurrences of \( w^e = S[x \ldots x + \frac{k}{2} - 1] \). One can easily verify that \( w^e_i \) is a subword of every \( w^e_i^\ast w \) with \( i \in [x - \frac{k}{2} + 1 \ldots x] \).
5.2 Shift updates

Shift updates are handled by applying the same observations that we used for evaluation updates. We partition shift updates into two subtypes:

1. **Right shifts**: A static suffix $S^i$ whose rank is increased by 1 since $w_j^{new} = w_i^{new}$ for some dynamic suffix $S^j <_L S^i$.

2. **Left shifts**: A static suffix $S^i$ whose rank is decreased by 1 since $w_j^{old} = w_i^{old}$ for some dynamic suffix $S^j$ with $T^j <_L T^i$.

We start by considering right shifts. For every static suffix $S^i$, we wish to count the dynamic suffixes $S^j$ with $w_j^{new} = w_i^{new}$ and $S^j <_L S^i$, and increase $R[i]$ by this number. Similarly to Section 5.1, we show how to increase the value of $R[i]$ for static suffixes $S^i$ by the amount of dynamic suffixes $S^j$ with $w_i^{new} = w_j^{new}$ such that $S^j < S^i$ and $j \in [x - k + 1 \ldots x - \frac{k}{2}]$. The task of increasing $R[i]$ by the amount of dynamic suffixes $S^j$ with $w_i^{new} = w_j^{new}$ and $S^j < S^i$ for some $j \in [x - \frac{k}{2} + 1 \ldots x]$ can be handled in a similar way by replacing $w_j^r = S[x - \frac{k}{2} + 1 \ldots x]$ with $w_i^r = S[x \ldots x + \frac{k}{2} - 1]$.

We consider the following variation of Fact 44:

**Fact 44.** Let $S^i$ be a dynamic suffix with $i = D(\delta)$ for some $\delta \in [0 \ldots \frac{k}{2}]$. For an index $j \in [1 \ldots n]$, the following are equivalent:

1. $S^j$ is close to $S^i$ (starts with an occurrence of $w_i^{new}$) and $S^j >_L S^i$.
2. There is an occurrence $w_i = S[s_1 \ldots e_1]$ of $w_i^r = S[s \ldots e]$ in index $s_1 = j + \delta$ such that $S^{s_1} >_L S^i$, lcs($s_1 - 1, s - 1$) $\geq \delta$ and lcp($e_1 + 1, e + 1$) $\geq \frac{k}{2} - \delta$.

Similarly to Algorithm 1, we can construct an algorithm for increasing the ranks of suffixes smaller than $S^i$ for every $i \in [1 \ldots n]$ by processing the occurrences of $w_i^r$ according to Fact 44.

**Algorithm 3**

For every occurrence $w = S[s_1 \ldots e_1]$ of $w_i^r$:

1. Check if $S^{s_1} >_L S^i$. If it is not - stop processing this occurrence.
2. Compute $l = \text{lcs}(s_1 - 1, s - 1)$ and $r = \text{lcp}(e_1 + 1, e + 1)$.
3. Increase the counters $R[i]$ with $i \in [\text{max}(s_1 - l, s_1 - \frac{k}{2}), \text{min}(e_1 + r - k + 1, s_1)]$ by 1.

Algorithm 3 is very similar to Algorithm 1. The main difference between these two algorithms is that Algorithm 3 applies updates directly to $R$, unlike Algorithm 1 that applies updates to a temporary array $Re$ and inserts the final values to the matching indexes in $R$. This change is required because shift updates may affect all the suffixes in $S$ while evaluation updates only affects a small set of suffixes. We prove the following claim:

**Claim 45.** For every $i \in [1 \ldots n]$, the value of $R[i]$ is increased in the execution of Algorithm 3 by the amount of dynamic suffixes $S^j$ with $w_i^{new} = w_j^{new}$ and $j \in [x - k + 1 \ldots x - \frac{k}{2}]$ such that $S^j <_L S^i$.

**Proof:** Let $i \in [1 \ldots n]$ be an index in $S$. Algorithm 3 increases the value of $R[i]$ by 1 every time it processes an occurrence $w_1 = S[s_1 \ldots e_1]$ of $w_i^r$ with the following properties:

1. $S^{s_1} >_L S^i$
2. $\text{max}(s_1 - l, s_1 - \frac{k}{2}) \leq i$
3. $\text{min}(e_1 + r - k + 1, s_1) \geq i$

Recall that Fact 44 suggests that for every dynamic suffix $S^j$ with $j = D(\delta)$ for some $\delta \in [0 \ldots \frac{k}{2}]$ such that $w_i^{new} = w_j^{new}$ and $S^j <_L S^i$, we have a unique occurrence of $w_i^r$ starting in index $i + \delta$. We show that $i$ is increased by Algorithm 3 when iterating an occurrence $w_1$ if $w_1$ is an occurrence corresponding to one of these dynamic suffixes.

From the second component in the max expression in 2 and 3 we get $i \in [s_1 - \frac{k}{2}, s_1]$. Therefore, we have $s_1 = i + \delta$ for some $\delta \in [0 \ldots \frac{k}{2}]$. The first expression in 2 yields $l \geq s_1 - i$ and therefore
\[ \delta \leq \text{lcs}(s_1-1, s-1). \] The first expression in (3) yields \( r \geq i + k - e_1 - 1. \) Since \( e_1 = s_1 + \frac{k}{2} - 1, \) we have \( \text{lcp}(e_1 + 1, e + 1) = r \geq \frac{k}{2} - \delta. \) We have shown that \( i \) is increased by Algorithm 3 when processing \( w_1 \) iff an occurrence \( w_1 \) satisfies Proposition 2 of Fact 44 with respect to \( i. \) This concludes the proof.

Similarly to Algorithm 1, Algorithm 3 can be easily implemented in time \( O(|\text{occ}(w^*_1)|) . \) Again, this is not sufficiently efficient. The solution is the same - we process the periodic occurrences representation of \( w^*_1 \) rather than the explicit occurrences.

Let \( C = (a, b, p) \) be a cluster of occurrences \( s_t = a + t \cdot p \) of \( w^*_1. \) Let \( w_t = S[s_t \ldots e_t] \) be the \( t \)’th occurrence in the cluster \( C. \) Let \( l_t = \text{lcs}(s_t-1, s-1) \) and \( r_t = \text{lcp}(e_t + 1, e). \) Let \( I_s \) be the interval of \( t \) values that satisfies \( S^{s_t}_t > I_s. \) Let \( i_t = \max(s_t - l_t, s_t - \frac{k}{2}) \) and \( j_t = \min(e_t + r_t - k + 1, s_t) \) be the ends of the interval update applied to \( r \) when the occurrence \( w_t \) is processed in Algorithm 3. We make the following claim:

**Claim 46.** Given \( C \) and \( w = S[s \ldots t], \) all the updates \( u_t = (i_t, j_t, 1) \) for \( t \in I_s \) can be applied to \( R \) in time \( O(\log^3(n)) \).

**Proof:** Similar to the proof of Claim 42. We start by employing Lemma 36 to obtain \( l_t = \min(E_x l, Ex^C_l + p \cdot t) \) and \( r_t = \min(E_x e, Ex^C_e - p \cdot t) \) for non-aligned occurrences.

With that, we can prove the following:

**Claim 47.** For non-aligned occurrences, \( \{i_t\} \) and \( \{j_t\} \) are \( p \)-normal sequences with degree 3 \( \in O(1). \)

**Proof:** By substituting Lemma 36 and \( s_t = s_0 + t \cdot p, \) we get \( i_t = \max(s_0 + p \cdot t - \min(E_x l, Ex^C_l + p \cdot t), s_0 + p \cdot t - \frac{k}{2}) \). Since \( s_0 - \frac{k}{2} \) is a constant, the second argument of this max expression is an arithmetic progression with difference \( p. \) The first argument can be rewritten as \( \max(s_0 + t \cdot p - Ex^C_t, s_0 - Ex^C_t) \), which is a \( p \)-normal sequence with degree 2. With that, we proved that \( \{i_t\} \) is a \( p \)-normal sequence with degree 3. Similar arguments can be made to show that \( \{j_t\} \) is a \( p \)-normal sequence with degree 3 as well.

It follows from the above claim that applying all the updates corresponding to non-aligned occurrences can be reduced to applying a constant number of stairs with stair width \( p \) updates and interval increment updates (Lemma 44). The (at most two) aligned occurrences can be explicitly handled by applying two interval increment updates in \( O(\log(n)) \) time. The time for applying a constant number of stairs and interval increment updates corresponding to the non-aligned occurrences is bounded by \( O(\log^3(n)) \).

With this, an efficient implementation of Algorithm 3 can be constructed in an identical manner to the construction of Algorithm 2.

**Algorithm 4**

1. **Obtain the periodic occurrences representation por of \( w^*_1. \)**
   
   For every singular occurrence \( w = S[s_1 \ldots e_1] \) in por:
   
   1. Check if \( S^{s_1}_1 \geq L S^s. \) If it is not, then stop processing this occurrence.
   2. Compute \( l = \text{lcs}(s_1-1, s-1) \) and \( r = \text{lcp}(e_1 + 1, e + 1). \)
   3. Increase the counters of \( R \) with indexes in \( [\max(s_1 - l, s_1 - \frac{k}{2}), \min(e_1 + r - k + 1, s_1)] \) by 1

   For every cluster \( C = (a, b, c) \) of occurrences in por:

   1. Obtain \( I_s \) using Claim 38.
   2. Apply to \( R \) all the interval updates corresponding to \( w_t \) with \( t \in I_s \) by employing Claim 46.

It is easy to see that Algorithm 4 applies exactly the same updates to \( R \) as Algorithm 3 does. The same arguments as in the proof of Claim 43 can be made to prove that Algorithm 4 runs in time \( \tilde{O}(\sqrt{n}). \) Recall that Algorithm 4 only counts dynamic suffixes \( S^j \) with \( j \in [x - k + 1 \ldots x - \frac{k}{2}], \) and a symmetric procedure needs to be executed for the remaining dynamic suffixes. This concludes the evaluation of right shifts.

Finally, we need to handle left shifts. For every static suffix \( S^i, \) we need to count the amount of dynamic suffixes \( S^j \) for which \( w^1_{i, j} = w^1_{i, j} \) and \( T^j < L T_i, \) and subtract that number from \( R[i]. \)

This can be executed with an algorithm almost identical to Algorithm 4 with the following two minor modifications:
1. Instead of considering the occurrences of $w_i^c = S[x - \frac{i}{2} + 1 \ldots x]$, consider occurrences of $u_i^c = T[x - \frac{i}{2} + 1 \ldots x]$

2. Replace every interval and stairs update with its negative variant.

Since we consider the same indexes of $w_i^c$ in $T$ rather than in $S$, an occurrence of $u_i^c$ is extendable to occurrences of $w_i^{old}$ in the same manner as an occurrence of $w_i^c$ is extendable to occurrences of $w_i^{new}$. Similarly to right shifts, Algorithm 4 with the above modification would only count dynamic suffixes $R$.

In order to apply overtake updates to $S$, we have an occurrence $w_i^c$, any occurrence of $w_i^c$ is extendable to occurrences of $w_i^{new}$. The updates are required for every $i \in [1 \ldots n]$, where $i$ is extendable to occurrences of $w_i^c$, and a symmetric procedure needs to be executed for the remaining dynamic suffixes.

We conclude this section with the following.

**Lemma 48.** Upon a substitution update on text $S$, all the shift updates can be applied to $R$ in time $O(\sqrt{n})$.

### 5.3 Overtake updates

In order to apply overtake updates to $R$, we need to increase the value of $R[i]$ by the amount of static suffixes $S^j$ for which $T^i <_L T^j$ and $S^i >_L S^j$ ($S^i$ overtake $S^j$) and decrease $R[i]$ by the amount of static suffixes $S^i$ for which $T^i >_L T^j$ and $S^i <_L S^j$ ($S^j$ overtake $S^i$). These updates are required for every $i \in [1 \ldots n]$ (excluding dynamic suffixes). For this purpose, we consider the word $w^{swap} = S[x-k \ldots x-1]$.

We introduce the notations $lcs_X(i, j)$ and $lcp_X(i, j)$ denoting the LCS (resp. LCP) of the prefixes (resp. suffixes) of string $X$ ending (resp. starting) in indexes $i$ and $j$. Until now, we used $lcs$ and $lcp$ to refer to $lcp_S$ and $lcs_S$. We present the following lemma:

(We rename the indexes of $w^{swap}$ to $w^{swap} = S[s \ldots e]$ for clearer presentation)

**Lemma 49.** Let $S^j$ be a static suffix. A static suffix $S^j$ overtakes $S^i$, iff one of the following conditions is satisfied.

**Condition 1:** All of the following sub-conditions are satisfied for $s_1 = j + s - i$:

1. $s - i \geq 0$
2. There is an occurrence $w_1 = S[s_1 \ldots e_1]$ of $w^{swap}$
3. $S^{s_1}$ is a static suffix
4. $S^{s_1}$ overtakes $S^s$
5. $m(s, s_1) \geq s - i$.

**Condition 2:** All of the following sub-conditions are satisfied for $s_1 = i + s - j$:

1. $s - j \geq 0$
2. There is an occurrence $w_1 = S[s_1 \ldots e_1]$ of $w^{swap}$
3. $S^{s_1}$ is a static suffix
4. $S^s$ overtakes $S^{s_1}$
5. $m(s, s_1) \geq s - j$.

In both conditions, $m(s, s_1) = \min(lcs_S(s - 1, s_1 - 1), lcs_T(s - 1, s_1 - 1))$

**Proof:**

**Lemma 49** may seem technical and involved, but it aims to describe a fairly straightforward intuition. In a high level, **Lemma 49** states that the common extension to the right of two suffixes $S^i$ and $S^j$ such that one overtake the other must reach the substitution index $x$. The common extension must therefore contain an occurrence $w_1$ of $w^{swap}$, and there must be an overtake between $w^{swap}$ and $w_1$. For an example, see Figure 3.

**Proof:** We start by proving the first direction. Namely, we show that if a static suffix $S^j$ overtake $S^i$, we have an occurrence $w_1$ satisfying one of the conditions presented in the Lemma.

Let $r = lcp_S(i, j)$ and $r' = lcp_T(i, j)$. We start by proving the following claim:

**Claim 50.** Either $x = i + \min(r', r)$ or $x = j + \min(r, r')$. 

21
We consider two cases:

1. \( r = r' \). In this case, from the definition of \( r' \) we have \( T[i...i+r-1] = T[j...j+r-1] \). And from the fact that \( S^j \) overtakes \( S^i \), we have \( T[i+r] > T[j+r] \).

2. \( r < r' \). In this case, we have \( T[i...i+r'-1] = T[j...j+r'-1] \). Since \( r < r' \), we must have \( T[i+r] = T[j+r] \).

In both cases, the lexicographic order between \( T[i+r] \) and \( T[j+r] \) is different from the lexicographic order between \( S[i+r] \) and \( S[j+r] \). Therefore, either \( T[i+r] \) or \( T[j+r] \) is the symbol modified by the substitution update.

We partition our proof into two cases, depending on which of the options for the value of \( x \) presented in Claim \([23] \) applies.

1. \( x = i + \min(r, r') \). In this case, we show that condition 1 is satisfied. Since \( S^i \) is static and \( i \leq x \), we must have \( i \leq x - k = s \) (sub-condition \([1] \)). Therefore, \( S[i...i+\min(r, r')-1] \) and \( T[i...i+\min(r, r')-1] \) both contain the substring \( S[x-k+1...x-1] = w_{swap} \).

Let \( d_s \) and \( d_e \) be two integers such that \( i + d_s = s \) and \( i + d_e = e \). Note that \( d_s = \min(r, r') - k \) and \( d_e = \min(r, r') - 1 \). Since \( r \) is the longest common prefix of \( S^i \) and \( S^j \), we have \( S[i...i+\min(r, r')-1] = S[j...j+\min(r, r')-1] \). And in particular, \( S[j + d_s...j + d_e] = S[i + d_s...i + d_e] = w_{swap} \).

Since \( \min(r, r') \leq r' \), we have equality of the substrings with the same indexes in \( T \). So we have an occurrence of \( w_{swap} \) in index \( j + d_s \) both in \( S \) and in \( T \). Denote \( s_1 = j + d_s \) and \( e_1 = j + d_e \) and \( w_1 = S[s_1...e_1] \). Note that \( s_1 = j + s - i \), \( s_1 \) is a starting index of an occurrence of \( w_{swap} \) in \( S \) (sub-condition \([2] \)), and since \( s_1 \) is a starting index of the same word \( w_{swap} \) with length \( k \) both in \( S \) and in \( T \), \( S^{s_1} \) is a static suffix (sub-condition \([3] \)).

From \( S[i...i+r-1] = S[j...j+r-1] \) we get \( S[i + d_s...i + r - 1] = S[j + d_s...j + r - 1] \). Recall that \( S[i+r] \) \(<_L S[j+r] \). Therefore, \( S^a \) \(<_L S^s \). From \( T[i...i+r'-1] = T[j...j+r'-1] \) we get \( T[i + d_s...i + r' - 1] = T[j + d_s...j + r' - 1] \) and therefore \( T[s...i+r'-1] = T[s_1...j+r'-1] \). Since \( S^j \) overtakes \( S^i \), we have \( T[i+r'] \) \(>_L T[j+r'] \) and therefore \( T^a \) \(>_L T^{s_1} \). We showed that \( T^a \) \(>_L T^{s_1} \).
and $S^* <_L S^{s_1}$. Therefore, $S^{s_1}$ overtakes $S^*$ (sub-condition 3). Since $S[i \ldots s] = S[j \ldots s_1]$ and $T[i \ldots s] = T[j \ldots s_1]$ we have $m(s, s_1) = \min(lcs_2(s - 1, s_1 - 1), lcs_T(s - 1, s_1 - 1)) \geq s - i$.

2. $x = j + \min(r, r')$. Symmetrical arguments can be made to show that in this case, condition 2 is satisfied.

We proceed to prove the second direction of the lemma. Namely, if there is an occurrence $w_1 = S[s_1 \ldots e_1]$ of $w_{\text{swap}} = S[s \ldots e]$ satisfying one of the conditions in the statement of the Lemma, $S^j$ overtakes $S^i$. We provide a proof for the first condition. A proof for the second condition can be constructed in a similar manner.

Assume that condition 1 holds, so we have $s - i \geq 0$, and an occurrence $w_1 = S[s_1 \ldots e_1]$ of $w_{\text{swap}}$ such that $s_1 = j + s - i$. We also know that $S^{s_1}$ is a static suffix overtaking $S^*$ and $m(s, s_1) \geq s - i$.

From the definition of $m(s, s_1)$ we have:

1. $S[s_1 - m(s, s_1)] = S[s - m(s, s_1)]$
2. $T[s_1 - m(s, s_1)] = T[s - m(s, s_1)]$

Since $m(s, s_1) \geq s - i \geq 0$, the above particularly implies:

1. $S[i \ldots s - 1] = S[i \ldots s_1 - 1]$
2. $T[i \ldots s - 1] = T[i \ldots s_1 - 1]$

Let $r = lcp_2(s, s_1)$ and $r' = lcp_T(s, s_1)$. From the definitions of $r$ and $r'$ we have $S[s \ldots s + r - 1] = S[s' \ldots s' + r - 1]$ and $T[s \ldots s + r' - 1] = T[s_1 \ldots s_1 + r' - 1]$. Putting it together with the above, we get:

1. $S[i \ldots s + r - 1] = S[i \ldots s + r_1 - 1]$
2. $T[i \ldots s + r' - 1] = T[i \ldots s + r_1 - 1]$

Since $S^{s_1}$ is a static suffix overtaking $S^*$, we have $S[s_1 + r] > S[s + r]$ and $T[s_1 + r'] < T[s + r']$. Therefore, $S^j$ overtakes $S^i$.

Given two static suffixes $S^i$ and $S^j$ such that $S^j$ overtakes $S^i$, we say that an occurrence of $w_{\text{swap}}$ satisfying one of the conditions of Lemma 49 with respect to $S^i$ and $S^j$ is implying the overtake between $S^j$ and $S^i$. The following Lemma is required to ensure that we count every overtake exactly once.

**Lemma 51.** Given two static suffixes $S^i$ and $S^j$ such that $S^j$ overtakes $S^i$, there is exactly one occurrence $w_1$ of $w_{\text{swap}}$ implying the overtake between $S^j$ and $S^i$.

**Proof:** The existence of at least one implying occurrence is immediate from Lemma 49. Given $i$, $j$ and $s$, there are only two possible starting indexes for the starting index of $w_1$: $s_1 = j + s - i$ and $s_2 = i + s - j$. Assume to the contrary that both $s_1$ and $s_2$ are starting indexes of an occurrence of $w_{\text{swap}}$ satisfying the first and the second conditions in Lemma 49 respectively.

Let $d = i - j$. Note that $s_1 = s - d$ and $s_2 = s + d$. Therefore, one of the indexes $s_1$, $s_2$ is to the right of $s$. We assume that $s_1 > s$, the case in which $s_2 > s$ can be treated in a similar way. Since $s_1$ is to the right of $s = x - k$, and is the starting index of a static suffix, it must be the case that $s_1 > x$.

Since both of the conditions of Lemma 49 hold, we have $s - i \geq 0$ and $j - s \geq 0$. Therefore, both $i$ and $j$ are not to the right of $s$. Since $m(s, s_1) \geq s - i$ we have the following:

1. $S[j \ldots s_1 - 1] = S[i \ldots s_1 - 1]$
2. $T[j \ldots s_1 - 1] = T[i \ldots s_1 - 1]$

Recall that $s_1 > x$ and $j < s < x$. Therefore, $x \in [j \ldots s_1 - 1]$. Since $x$ is the modified index, $T[x] \neq S[x]$. In particular, $T[j \ldots s_1 - 1] \neq S[j \ldots s_1 - 1]$ and it can not be the case that both (1) and (2) hold. Therefore, we have reached a contradiction.

Lemmas 49 and 51 enable the construction of an algorithm with the structure of Algorithms 1–4. It allows us to identify overtake updates using the occurrences of $w_{\text{swap}}$ in a similar manner to how we use Fact 44 to identify shift updates using occurrences of $w_{\text{swap}}$. As before, we start by showing an inefficient algorithm for the sake of presenting the modifications that are required to be applied to $R$ in order to represent the overtake updates corresponding to an individual occurrence of $w_{\text{swap}}$. We present Algorithm 5.
Algorithm 5

For every occurrence \( w = S[s_1 \ldots e_1] \) of \( w^{\text{swap}} \):

1. If \( S^{s_1} \) is a dynamic suffix - stop processing \( S^{s_1} \)

2. If \( S^{s_1} \prec_S S^s \) and \( T^{s_1} \succ_T T^s \) (\( S^s \) overtakes \( S^{s_1} \))
   
   (a) Compute \( l_S = \text{lcs}_S(s_1 - 1, s - 1) \) and \( l_T = \text{lcs}_T(s_1 - 1, s - 1) \).
   
   (b) Increase the counters of \( R \) with indexes in \([s - \min(l_S, l_T), s]\) by 1.
   
   (c) Decrease the counters of \( R \) with indexes in \([s_1 - \min(l_S, l_T), s_1]\) by 1.

3. If \( S^{s_1} \succ_S S^s \) and \( T^{s_1} \prec_T T^s \) (\( S^{s_1} \) overtakes \( S^s \))
   
   (a) Compute \( l_S = \text{lcs}_S(s_1 - 1, s - 1) \) and \( l_T = \text{lcs}_T(s_1 - 1, s - 1) \).
   
   (b) Decrease the counters of \( R \) with indexes in \([s - \min(l_S, l_T), s]\) by 1.
   
   (c) Increase the counters of \( R \) with indexes in \([s_1 - \min(l_S, l_T), s_1]\) by 1.

We start by proving that Algorithm 5 applies overtake updates correctly.

**Lemma 52.** For every \( i \in [1 \ldots n] \), let \( \uparrow_i \) be the number of times that \( R[i] \) is increased during the execution of Algorithm 5. Let \( \downarrow_i \) be the number of times that \( R[i] \) is decreased during the execution of Algorithm 5. \( \uparrow_i \) equals to the number of suffixes \( S^j \) that \( S^i \) is overtaking. \( \downarrow_i \) equals to the amount of suffixes \( S^j \) that are overtaking \( S^i \).

**Proof:** Let \( j \in [1 \ldots n] \) be an index (we use \( j \) instead of \( i \) in order to be consistent with the statement of Lemma 49). \( R[j] \) is increased in step 3c (second 'if') every time an iterated occurrence \( w_1 = S[s_1 \ldots e_1] \) of \( w^{\text{swap}} \) satisfies the following conditions:

1. \( S^{s_1} \) is static.
2. \( S^s \) overtakes \( S^{s_1} \).
3. \( j \leq s \).
4. \( j \geq s - \min(\text{lcs}_S(s_1 - 1, s - 1), \text{lcs}_T(s_1 - 1, s - 1)) \)

Let \( i = s - s + j \). It can be easily verified that all the sub-conditions of Condition 2 of Lemma 49 are satisfied for \( s_1 \), making \( w_1 \) a unique occurrence implying that \( S^j \) overtake \( S^i \).

An index \( j \) is increased in step 3c (third 'if') every time an iterated occurrence \( w_1 = S[s_1 \ldots e_1] \) satisfies the following conditions:

1. \( S^{s_1} \) is static.
2. \( S^{s_1} \) overtakes \( S^s \).
3. \( j \leq s_1 \).
4. \( j \geq s_1 - \min(\text{lcs}_S(s_1 - 1, s - 1), \text{lcs}_T(s_1 - 1, s - 1)) \)

Let \( i = s - s_1 + j \). It can be easily verified that all the sub-conditions of condition 1 of Lemma 49 are satisfied for \( s_1 \), making \( w_1 \) a unique occurrence implying that \( S^j \) overtakes \( S^i \).

Lemma 51 guarantees that every suffix \( S^j \) such that \( S^j \) overtakes \( S^i \) corresponds to a unique occurrence of \( w^{\text{swap}} \) satisfying one of the above conditions. It directly follows that \( \uparrow_j \) is exactly the amount of suffixes overtaken by \( S^j \).

Symmetrical arguments can be made to show the \( \downarrow_j \) is equal to the amount of suffixes that overtake \( S^j \).

As in evaluation and shift updates, a straightforward implementation of Algorithm 5 is not efficient. In order to implement Algorithm 5 efficiently, we need the following claims:

**Claim 53.** Let \( C = (a, b, p) \) be a cluster of occurrences of a word \( w = S[s \ldots e] \) with the \( t \)’th occurrence represented by \( C \) denoted as \( w_t = S[s \ldots e] \). There are consecutive intervals \( I^t \) and \( I^s \) such that \( S^{s_1} \) overtakes \( S^s \) iff \( t \in I^s \) and \( S^s \) overtakes \( S^{s_1} \) iff \( t \in I^t \). The intervals \( I^t \) and \( I^s \) can be found in \( O(\log(n)) \) time.

24
Claim 56. for non-aligned occurrences to $R$.

Proof: We can apply Claim 38 to find the intervals $I_S = [i_1, ... j_1]$ and $I_L = [i_2, ... j_2]$ such that $S^a = \{ l \}$ if $t \in I_L$ and $T^a < \frac{1}{2} t$ if $t \in I_L$. By definition, $I = I_S \cap I_L = [\min(i_1, i_2), ... \min(j_1, j_2)]$.

We can apply Claim 38 to find the intervals $I_S = [i_3, ... j_3]$ and $I_L = [i_4, ... j_4]$ such that $S^a = \{ l \}$ if $t \in I_S$ and $T^a > \frac{1}{2} t$ if $t \in I_S$. By definition, $I = I_S \cap I_L = [\max(i_3, i_4), ... \min(j_3, j_4)]$.

We need to further 'filter' the occurrences represented by a cluster to exclude dynamic suffixes. We employ the following simple claim.

Claim 54. Let $I = [i, ... j]$ be a subinterval of $[0, ... |C| - 1]$ representing a subset of the indexes of the occurrences implied by a cluster $C$. It can be processed in $O(1)$ time to evaluate at most two subintervals $I_1$ and $I_2$ such that $t \in I_1 \cup I_2$ if $S^a$ is a static suffix and $t \in I$.

Proof: We need to find the maximal value $t_1$ in $I$ such that $s_t \leq x - k$ and the minimal value $t_2$ such that $s_{t_2} \geq x + 1$. Straightforwardly, $I_1 = [i, ... t_1]$ and $I_2 = [t_2, ... j]$ satisfy the desired condition.

We adopt the notations of $I^t_1$ and $I^t_2$ from Claim 53 for the following claim. We present the notation $I_{st}^1 = [i_{st}^1, j_{st}^1]$ and $I_{st}^2 = [i_{st}^2, j_{st}^2]$ denoting the intervals of indexes within $[0, ... |C| - 1]$ corresponding to static suffixes.

Let $C = (a, b, p)$ be a cluster of occurrences of $w^{\text{swap}} = S[s, ... e]$ with the $t$th occurrence represented by $C$ denoted as $w_t = S[s_t, ... e_t]$. We employ the notations $I_{st}^1 = \text{lcp}_S(s - 1, s_t - 1)$ and $I_{st}^2 = \text{lcp}_P(e + 1, e_t + 1)$.

For $t \in I^1 \cap (I_{st}^1 \cup I_{st}^2)$, we define the following sequences of interval increment updates:

- $u_1^t = (i_{st}^1, j_{st}^1, 1) = (s_{t} - \min(I_{st}^1, I_{st}^2), s_t, 1)$ the interval increment update applied to $R$ when $w_t$ is processed in step 2a of Algorithm 5.

- $u_2^t = (i_{st}^2, j_{st}^2, -1) = (s_t - \min(I_{st}^1, I_{st}^2), s_t, -1)$ the (negative) interval increment update applied to $R$ when $w_t$ is processed in step 2a of Algorithm 5.

For $t \in I^1 \cap (I_{st}^1 \cup I_{st}^2)$, we define the following sequences of interval increment updates:

- $u_3^t = (i_{st}^3, j_{st}^3, 1) = (s_{t} - \min(I_{st}^1, I_{st}^2), s_t, 1)$ the interval increment update applied to $R$ when $w_t$ is processed in step 3a of Algorithm 5.

- $u_4^t = (i_{st}^4, j_{st}^4, -1) = (s_t - \min(I_{st}^1, I_{st}^2), s_t, -1)$ the (negative) interval increment update applied to $R$ when $w_t$ is processed in step 3a of Algorithm 5.

We make the following claim.

Claim 55. Given $C$ and $w = S[s, ... t]$, all the updates $u_y^t$ for $t \in I^1 \cap (I_{st}^1 \cup I_{st}^2)$ and $y \in \{1, 2\}$ can be applied to $R$ in time $O(\log^3(n))$. All the updates $u_y^t$ for $t \in I^1 \cap (I_{st}^1 \cup I_{st}^2)$ and $y \in \{3, 4\}$ can be applied to $R$ in time $O(\log^3(n))$.

Proof: As in the proof of Claim 52, we start by applying Lemma 36 to obtain $l_t = \min(Ex_t, Ex_t + p_t)$ for non-aligned occurrences $w_t$. We proceed to prove the following claim:

Claim 56. $f_t^2$ is a $p$-normal sequence with degree 4 for non-aligned occurrences.

Proof: By Lemma 36 both $f_t^2$ and $f_t^1$ can be written as $f_t^2 = \min(f_S, f_T + t \cdot p)$ and $f_t^1 = \min(f_T, a_T + t \cdot p)$ for some constants $f_S$, $f_T$, $a_S$ and $a_T$ for non-aligned occurrences. Therefore, $f_t^2 = s_0 + t \cdot p - \min(f_T, a_T + t \cdot p), f_S, (a_S + t \cdot p))$. It follows that $f_t^2 = \max(s_0 + t \cdot p - f_T, a_T + t \cdot p, f_S, a_S))$. This is a $p$-normal sequence with degree 4.

It can be easily proven that $f_t^1$ and $f_t^2$, excluding aligned occurrences, are $p$-normal sequences with constant degrees for every $z \in [1, 4]$ by applying the same reasoning.

We apply Lemma 11 in the same manner as in the proof of Claim 52 to obtain a constant number of interval and stairs updates that are equivalent to applying all the updates corresponding to non-aligned occurrences. Note that we need to treat $I^1 \cap I_{st}^1$ and $I^1 \cap I_{st}^2$ independently, as 11 deals with a consecutive $p$-normal sequence. Applying the constant set of stairs updates and interval increment updates can be executed in $O(\log^3(n))$ time. The (at most two) updates corresponding to aligned occurrences can be applied to $R$ in $O(\log(n))$ time by employing Claim 33.

With Claim 53, Claim 54 and Claim 55, we are ready to present the efficient variant of Algorithm 5.
Algorithm 6

Obtain the periodic occurrences representation \(p\) of \(w^{\text{swap}}\):

For every singular occurrence \(w = S[s_1 \ldots e_1]\) in \(p\): Process \(w_1\) as in Algorithm 5.

For every cluster of occurrence \(C = (a,b,p)\) in \(p\):

Obtain \(I^1_{st} = I^1 \cap (I^1_{st} \cup I^2_{st})\) and \(I^1_{st} = I^1 \cap (I^1_{st} \cup I^2_{st})\) using Claims 53 and 54.

If \(I^1_{st}\) is not empty: Efficiently apply the interval updates corresponding to \(w^i\) with \(t \in I^1\) using Claim 55.

If \(I^1_{st}\) is not empty: Efficiently apply the interval updates corresponding to \(w^i\) with \(t \in I^1\) using Claim 55.

It is straightforward that Algorithm 6 applies the same updates to \(R\) as Algorithm 5 does - as it only processes occurrences corresponding to static suffixes that either overtake \(S^\circ\) or are overtaken by \(S^\circ\). We conclude this section with the following.

Lemma 57. Upon a substitution update in index \(x\) of \(S[1 \ldots n]\). All the overtake update can be applied to \(R\) in \(O(\sqrt{n})\) time.

Proof: Applying Algorithm 6 results in the same changes to \(R\) as applying Algorithm 5. This is equivalent to applying the overtake updates according to Lemma 52. Algorithm 6 processes every singular occurrence or cluster in the periodic occurrences representation of \(w^{\text{swap}}\) in polylogarithmic time. Therefore, it runs in time \(O(\sqrt{n})\).

Recall that for shift and overtake updates we manipulate the existing ranks in \(R\) by applying interval increment and stairs updates to \(R\), but for evaluation we set the updated ranks explicitly. Therefore, shifts and overtake modifications needed to be counted and applied to \(R\) prior to the evaluation of the ranks of dynamic suffixes. This is required to avoid the case in which a rank is set for a dynamic suffix, and then modified by a stairs or an interval update.

With that, we have shown how to efficiently apply every type of updates to \(R\). Upon a substitution update to \(S\) in index \(x\), we apply algorithms 2 and 6 to update \(R[i]\). After doing so, \(R[i]\) contains the updated close suffix rank of \(S^i\) for every \(i \in [1 \ldots n]\).

5.4 Handling Stairs Updates

In this section, we show how to apply the restricted data structure provided by Theorem 2 to our settings. We recall that the data structure of Theorem 2 enables the application of a stairs update to a set of counters and an evaluation of a counter in polylogarithmic time, but only if every stairs update \((i,j,p)\) has the same stair width \(p\).

Our dynamic algorithm requires stairs update with various values of \(p\), so the data structure of Theorem 2 is not sufficient. We start by showing that for every stairs update \((i,j,p)\) that is applied by our algorithm, \(S[i \ldots j]\) is periodic. We then proceed to show how this property, alongside with the restricted stairs updates data structure, can be used to efficiently support stairs updates in our settings.

We start with the following observations concerning the structure of the interval updates applied in Algorithm 3 and in Algorithm 5.

Observation 58. Let \(C = (a,b,p)\) be a cluster of occurrences of a word \(w^C = S[s_1 \ldots e]\) with period \(p\) that is processed in Algorithm 5. Let \(R_C = S[i_C \ldots j_C]\) be the run with period \(p\) containing the occurrences implied by \(C\). For an occurrence \(w_t = S[s_t \ldots e_t]\) implied by \(C\), let \(u_t = (i_t,j_t,1)\) be the interval increment update applied to \(R\) when \(w_t\) is processed by Algorithm 5. Every occurrence \(w_t\) implied by \(C\) that is not an aligned occurrence has \([i_t \ldots j_t] \subseteq [i_C \ldots j_C]\).

Proof: In algorithm 5, the left border of the interval update applied to an occurrence \(w_t\) of \(w^C\) is \(i_t = \max(s_t - l_t, s_t - \frac{k}{2})\). The right border of the update is \(j_t = \min(e_t + r_t - k + 1, s_t)\). According to Lemma 36, for an occurrence that is not aligned we have \(i_t = \max(s_t - \min(Ex_1, Ex^C_t + p \cdot t), s_t - \frac{k}{2})\), which is at least \(s_t - \min(Ex_1, Ex^C_t + p \cdot t) \geq s_t - Ex^C_t - p \cdot t = s_0 - Ex^C_t = i_C\).
According to Lemma \ref{lem:dominant}, we have \( j_t = \min(e_t + \min(Ex_t, Ez_t) - p \cdot t) - k + 1, s_t) \leq s_t. \) Since all the occurrences implied by \( C \) are contained within \( R_C \), we have \( s_t \leq j_C \) and therefore \( j_t \leq j_C \). We showed that \( i_t \geq i_C \) and \( j_t \leq j_C \). Therefore, \( [i_t \ldots j_t] \subseteq [i_C \ldots j_C] \). \hfill \( \square \)

We proceed with a similar observation concerning Algorithm \ref{alg:stairsswap}.

**Observation 59.** Let \( C = (a, b, p) \) be a cluster of occurrences of a word \( w_{\text{swap}} = S[i \ldots e] \) with period \( p \) that is processed in Algorithm \ref{alg:stairsswap}. Let \( Run_C = S[i_C \ldots j_C] \) and \( Run_w = S[i_w \ldots j_w] \) be the runs with period \( p \) containing the occurrences implied by \( C \) and the word \( w_{\text{swap}} \), respectively. For an occurrence \( w_t = S[i_t \ldots e_t] \) implied by \( C \), let \( u_1, u_2, u_3 \) and \( u_4 \) be the interval increment update applied to \( R \) when \( w_t \) is processed by Algorithm \ref{alg:stairsswap} in step 2b, step 2c, step 3b and step 3c, respectively. Let \( u_1^2 = (i_1^2, j_1^2, b_2) \) for \( z \in \{1 \ldots 4\} \) and \( b_2 \in \{-1, 1\} \). The following holds:

1. for \( z = 1 \) and \( z = 3 \), every non-aligned occurrence \( u_t \) has \( [i_t^2, j_t^2] \subseteq [i_w \ldots j_w] \).
2. for \( z = 2 \) and \( z = 4 \), every non-aligned occurrence \( u_t \) has \( [i_t^2, j_t^2] \subseteq [i_C \ldots j_C] \).

**Proof:** For \( z = 2 \) and \( z = 4 \), the left border of the interval update applied to an occurrence \( w_t \) of \( w_{\text{swap}} \) in step \( 2b \) in both parts is \( i_t^2 = i_t^2 = s_t - \min(l_t^w, l_t) \geq s_t - l_t \) and the right border is \( j_t^2 = j_t^2 = s_t \). We showed that \( s_t - l_t \geq i_C \) and \( s_t \leq j_C \) for a non-aligned occurrence in the proof of Observation \ref{obs:dominant}. So in this case, \( [i_t \ldots j_t] \subseteq [i_C \ldots j_C] \).

For \( z = 1 \) and \( z = 3 \), the left border of the interval update applied to an occurrence \( w_t \) of \( w_{\text{swap}} \) is in step \( 2b \) in both parts is \( i_t^2 = i_t^2 = s - \min(l_t^w, l_t) \geq s - l_t \) and the right border is \( j_t^2 = j_t^2 = s \). Since the run \( Run_w \) contains \( w_{\text{swap}} \), we have \( j_w \geq e > s \). According to Lemma \ref{lem:dominant} if \( w_t \) is not an aligned occurrence we have \( i_t \geq s - \min(Ex_t, Ez_t) \geq s - \frac{p \cdot t}{2} \). Therefore, we have \( [i_t^2 \ldots j_t^2] \subseteq [i_w \ldots j_w] \). \hfill \( \square \)

**Lemma 60.** For every stairs update \((i, j, p)\) applied by Algorithm \ref{alg:stairsswap} or by Algorithm \ref{alg:stairsswap} \( S[i \ldots j] \) is contained within a run with period \( p \). Either \( Run_w \) or \( Run_C \).

**Proof:** Recall that we only use stairs updates to apply interval increment updates corresponding to non-aligned occurrences.

Let \( C \) be a cluster of occurrences of a word \( w \) with period \( p \). Let \( U \) be the set of intervals updates \( U \) that are applied to \( R \) when the occurrences implied by \( C \) are processed either in Algorithm \ref{alg:stairsswap} or in one of the steps in Algorithm \ref{alg:stairsswap} in which an interval increment update is applied. In the proofs of Claims \ref{obs:dominant} and \ref{obs:dominant} we show that for non-aligned occurrences, \( U \) is a \( p \)-normal sequence with a constant degree, and therefore can be reduced to a constant sized set \( St \) of stairs updates with stair width \( |p| = p \).

According to Observations \ref{obs:dominant} and \ref{obs:dominant} either interval update \( u = (i, j, 1) \in U \) has \( [i \ldots j] \subseteq [i_C \ldots j_C] \) or every \( u \in U \) has \( [i \ldots j] \subseteq [i_w \ldots j_w] \). Naturally, every stairs update \((i_1, j_1, p) \in St \) has \( [i_1 \ldots j_1] \subseteq [i_C \ldots j_C] \) or \( [i_1 \ldots j_1] \subseteq [i_w \ldots j_w] \). That is due to the fact that applying these stairs updates is equivalent to applying \( U \) and therefore they only affects the indexes that are affected by the updates in \( U \).

To be more precise, it is theoretically possible that \( St \) contains stairs updates that affect an index \( i \) that was not modified by the updates in \( U \), and the updates in \( St \) cancel each other in index \( i \). An examination of Lemma \ref{lem:stairsswap} in Section \ref{sec:stairsswap} (in which the method for reducing a sequence of interval increment updates to a small set of stairs updates and interval updates is presented) easily shows that this is never the case.

Recall that in every stairs update \((i, j, p)\) applied by our algorithm, \( p \) is the period of a word with length at most \( k \). The following directly follows.

**Observation 61.** Every stairs update \((i, j, p)\) applied by our algorithm has \( p \in \{1 \ldots \frac{p}{2}\} \).

We proceed to show how to use the restricted data structure to maintain stairs updates with various stair widths in our settings.

**The idea:** We only apply interval increment updates directly to \( R \). Additionally to \( R \), we maintain a restricted data structure \( St_p \) for applying and querying stairs updates with fixed stair width \( p \) for \( p \in \{1 \ldots \frac{p}{2}\} \). We also keep a data structure \( ActInt \) for 2-dimensional range searching queries. When we wish to apply a stairs update \((i, j, p)\) to \( R \), we apply it to \( St_p \) instead. Additionally, we add a point \((i, j)\) with value \( v(i, j) = p \) to \( ActInt \). This point represent the interval \([i \ldots j]\) that was affected by the stairs update. Upon a query for the close suffixes rank of \( S^i \), we extract from \( ActInt \) the intervals touching the index \( i \). The values of the extracted points form the set \( P \) of values of \( p \) such that \( St_p \) received an update affecting index \( i \). We evaluate \( \Sigma_P = \Sigma_{p \in P} St_p[i] \) and report \( \Sigma_P + R[i] \). The periodicity of the intervals
that are affected by \( S_{tp} \) ensures that \(|P| \in O(\log(n))\), so the process of reporting the rank of \( S^t \) remains polylogarithmic. However, further work needs to be dedicated to the maintenance of these intervals and their periodicity properties.

We present the notation of an extremely periodic string.

**Definition 62.** A string \( S \) is extremely periodic if \( \text{per}(S) \leq \frac{|S|}{\alpha} \)

We are interested in runs of \( S \) that are also extremely periodic. The following property of extremely periodic runs of a string \( S \) is proven in Section 6.

**Theorem 63.** Given a string \( S[1 \ldots n] \), let \( x \in [1 \ldots n] \) be an index within \( S \). There are \( O(\log(n)) \) extremely periodic runs containing \( x \).

We start by slightly modifying algorithms 3 and 5. Specifically, we replace some of the stairs updates applied in these algorithms with the set of interval update they represent. We only do that in a situation in which there are not many such interval updates, so applying them explicitly is not too costly. The details for this adjustment are as follows.

Let \( w \) be a word we process in one of our algorithms for modifying the ranks (\( w_t, w_{\text{swap}}, \ldots \)). Let \( R_{unw} = S[s_{rw} \ldots e_{rw}] \) the run with period \( p \) containing \( w \). If \( R_{unw} \) is not extremely periodic, it must be the case that \( \frac{|w|}{p} < \text{per}(w) \).

If that is the case, we can afford to process them explicitly and apply the interval increment updates corresponding to them rather than using stairs updates (as in algorithms 3 and 5).

Otherwise, assume that we process the cluster \( C \) of occurrences of \( w \). Let \( R_C = S[s_r \ldots e_r] \) be the run with period \( p \) containing \( C \). Assume that we need to apply a stairs update \((i, j, p)\) corresponding to a subset of the occurrences within \( C \) such that \( S[i \ldots j] \) is contained within the run \( R_C \). We check if \( R_C \) is extremely periodic. If it is not - there are at most 4 occurrences of \( w \) in \( C \). We apply the interval increment updates derived from these occurrences to \( R \) instead of processing them as stairs updates.

If \( R_C \) is extremely periodic, we apply the stairs update as follows. We add the point \( p_{R_C} = (s_r, e_r) \) with value \( v(p_{R_C}) = p \) to \( \text{ActInt} \) (while avoiding duplicates), and apply the stairs update \((i, j, p)\) to \( St_p \). If \( S[i \ldots j] \) is contained within \( RU_{unw} \), we add the point \( p_w = (s_{rw}, e_{rw}) \) with the same value \( v(p_w) = p \) instead of \( p_{R_c} \).

**Observation 64.** With the above modifications, the running times of algorithms 3 and 5 remains \( \tilde{O}(\sqrt{n}) \).

For every \( i \in [1 \ldots n] \), the modifications applied to \( R[i] \) by Algorithm 3 and Algorithm 5 prior to these modifications are equivalent to the updates applied to \( R[i] + \sum_{p=1}^{\frac{\sqrt{n}}{2}} St_p[i] \) with the modifications.

At this point, it may seem like we are done. We only insert extremely periodic runs to \( \text{ActInt} \), so according to Theorem 63, every index \( i \in [1 \ldots n] \) is only touched by \( O(\log(n)) \) runs in \( \text{ActInt} \). Upon a query for the close suffix rank of \( S^t \), we can find all the runs in \( \text{ActInt} \) containing \( i \), sum \( \Sigma_p \) the values of \( St_p[i] \) with the corresponding \( p \) values, and return \( \Sigma_p + R[i] \).

However, there is another subtle point to handle. The intervals \( S[s_r \ldots e_r] \) such that \( (s_r, e_r) \) is inserted to \( \text{ActInt} \) are extremely periodic runs in the time of their insertion. Future updates may change that by cutting an extremely periodic run or by extending it.

Therefore, we need to dedicate some extra work to maintain the following two invariants:

1. Every interval (represented as a point) within \( \text{ActInt} \) is an extremely periodic run
2. If \( x \) is not contained in interval (represented as point) \( p_1 = [s_1 \ldots e_1] \) with value \( p \), we have \( St_p[x] = 0 \).

**Lemma 65.** After a substitution in index \( x \), the data structures \( \text{ActInt} \) and \( \{St_p\}_{p \in [1 \ldots \frac{n}{2}]} \) can be updated such that the above invariants are preserved, and the accumulated value of \( R[i] + \sum_{p=1}^{\frac{\sqrt{n}}{2}} St_p[i] \) is not affected for every \( i \in [1 \ldots n] \). The update can be executed in \( \tilde{O}(\sqrt{n}) \) time.

**Proof:** Upon an update in index \( x \), we assume that the invariants are satisfied prior to the application of the substitution update in \( x \). We query \( \text{ActInt} \) for all the intervals containing the index \( x \) we get a set \( P' \) of points \( p_1 = (s_1, e_1) \) such that \( S[s_1 \ldots e_1] \) was an extremely periodic run with period \( v(p_1) \) in \( T \).

Therefore, according to Theorem 63, \( |P'| \in O(\log(n)) \).

For every point \( p_1 = (s_1, e_1) \in P' \) with \( v(p_1) = p \), we initialize a set \( \text{Exclude}_{p_1} = \{x\} \) of candidates for exclusion. The update in \( x \) cuts the run \( T[s_1 \ldots e_1] \) into two maximal substrings with period \( p \) -
$R_l = S[s_1 \ldots x - 1]$ and $R_r = S[x + 1 \ldots e_1]$. We start by deleting $p_1$ from ActInt. If $|R_l| \geq 5 \cdot p$, we insert $p_2 = [s_1 \ldots x - 1]$ with $v(p_2) = p$ to ActInt. Otherwise, we insert the indexes $[s_1 \ldots x - 1]$ to Exclude$_{p_2}$. (And do not add $p_1$ to ActInt). We treat $R_r$ the same.

We proceed to treat the candidates for exclusion. For every index $i \in$ Exclude$_{p_1}$, we check if there is an interval $p_2$ in ActInt containing $i$ with $v(p_2) = p$. If there is a point with that property - we simply proceed to the next index. If there is not - we evaluate the current value $a = St_p[i]$, set $R[i] = R[i] + a$ and set $St_p[i] = 0$. By doing that, we ensure that the value of $St_p$ is no longer required for computing the rank of $S^i$ (second variant), while not changing the value of $R[i] + \sum_{p=1}^\infty St_p[i]$.

We also query ActInt for intervals starting with $x + 1$ or ending with $x - 1$. The extremely periodic runs corresponding to these intervals may have been extended as a result of the update. For every point $p_1 = (s_1, x - 1)$ with $v(p_1) = p$, we query $r = \text{lcp}(x - 1, x - 1 - p)$. If $r > 0$ - the run $S[s_1, x - 1]$ was extended. In this case, we remove $p_1$ from ActInt and add $p_2 = (s_1, x - 1 + r)$ instead (while avoiding duplicates). We do the symmetric procedure for every point $p_1 = (x + 1, e_1)$.

**Complexity:** The bottleneck of maintaining the runs within ActInt is the exclusion procedure. Apart from the exclusion, we use a constant number of lcp, lcs, basic arithmetic operations and range queries to ActInt for every point. Since there are $O(\log(n))$ points touching $x, x - 1$ and $x + 1$, this is polylogarithmic. We note that the data structure for maintaining and querying stairs updates can be easily modified to allow setting a certain index to 0 without hurting its performance.

As for the exclusion, we only exclude an interval of indexes if its size is less than $5 \cdot p \leq \frac{5k}{2} \in O(k) = O(\sqrt{n})$. We exclude at most two such intervals for every contested point $p_1$, so $|\text{Exclude}_{p_2}| \in O(\sqrt{n})$. For every $i \in \text{Exclude}_{p_1}$, we query ActInt at most twice and execute a constant number of queries on $St_p$ and $R$, all polylogarithmic operations. So the overall complexity of the exclusion procedure is $O(\sqrt{n})$ per point, which is $O(\sqrt{n})$ over all the contested points.

With that, the proof of Theorem 20 is completed. Applying the stairs updates in the proofs of Claims 46 and 55 to $R$ is done implicitly by applying them to $St_p$ while maintaining the extremely periodic invariants after every update using Lemma 65.

As for the query time, for the close suffixes rank of $S^i$ we query ActInt for $\text{Report}([1 \ldots i] \times [i \ldots n])$ to obtain the set $P$ of the intervals containing $i$. The time complexity of this query is $O(\log^2(n) + |P|) = O(\log(n))$. We query every $St_p$ such that $\text{per}(p_1) = p$ for some point $p_1 \in P$ for $St_p[i]$ in $O(\log(n))$ time each and sum the results to obtain $\Sigma_p$. We query $R$ for $R[i]$ in $O(\log(n))$ time and return $R[i] + \Sigma_p$. The complexity is dominated by $O(\log^4(n))$.

### 6 Complementary Proofs for Close Suffixes Rank queries

#### 6.1 Proof of Lemma 36

**Lemma 36.** Let $C = (a, b, p)$ be a cluster of occurrences of a word $w = S[s_1 \ldots e]$ with $\text{per}(w) = p$ in $S[1 \ldots n]$. Let the occurrences represented by $C$ be $w_0, w_1, \ldots w_{|C| - 1}$ with $w_i = S[s_1 \ldots e_1]$ and $s_1 = a + t \cdot p$. Let $l_t = \text{lcs}(s_t - 1, s - 1)$ and $r_t = \text{lcp}(e_1 + 1, e + 1)$. Let $Ex_t$ and $Ex_r$ be the lengths of the extension of the run with period $p$ containing $w$ to the left of $s - 1$ and to the right of $e + 1$, respectively. Let $Ex_t^C$ and $Ex_r^C$ be the lengths of the extension of the run with period $p$ containing $w_0$ to the left of $s_0 - 1$ and to the right of $e_0 + 1$, respectively.

1. If $Ex_t \neq Ex_t^C + p \cdot t$, we have $l_t = \min(Ex_t, Ex_t^C + p \cdot t)$.
2. If $Ex_r \neq Ex_r^C - p \cdot t$, we have $r_t = \min(Ex_r, Ex_r^C - p \cdot t)$

The values of $Ex_t$, $Ex_r$, $Ex_t^C$ and $Ex_r^C$ can be evaluated in $O(\log(n))$ time given $C$, the indexes of $w$, and a data structure for $O(\log(n))$-time LCE queries.

**Proof:** For $t \in [0 \ldots |C| - 1]$, let $Ex^C_t$ and $Ex^C_r$ be the extension of the run with period $p$ containing $C$ to the left and to the right from $w_t$, respectively. Since the starting index of $w_t$ is $s_t = s_0 + t \cdot p$, we have $Ex_t = Ex_t^C + t \cdot p$ and $Ex_r = Ex_r^C - t \cdot p$. We proceed to prove the correctness of the formula for $\text{lcs}(s_1, s_t - 1)$. The formula for $\text{lcp}(s + 1, e_1 + 1)$ can be treated similarly.

The run with period $p$ extends at least $m = \max(Ex_t, Ex_r^C)$ symbols to the left of both $s$ and $s_t$. Let $\delta \in [1 \ldots m]$ be an integer. Let $r_\delta \in [0 \ldots p - 1]$ and $q_\delta$ be the two unique integers such that $\delta = q_\delta \cdot p + r_\delta$. Due to the extensions of the run, we have $S[s - \delta] = S[s - r_\delta + p] = w[p - r_\delta]$. Since the extension of the period is of size at least $m$ to the left of $s_t$ as well, we have $S[s_t - \delta] = S[s_t - r_\delta + p] = w[p - r_\delta] = w[p - r_\delta]$. From transitivity, we get $S[s - \delta] = S[s_t - \delta]$ for every $\delta \in [1 \ldots m]$ and therefore $\text{lcs}(s_1, s_t - 1) \geq m$.

29
From our assumption that $Ex_i \neq Ex_j^C + t \cdot p$ we get $Ex_i \neq Ex_j^C$. If $Ex_i < Ex_j^C$, the extension of the run with period $p$ to the left of $s$ terminates in index $s - (m + 1)$. Therefore, $S[s - (m + 1)] \neq S[s - (m + 1) + p]$. However, the extensions of the run with period $p$ extends at least $m + 1$ indexes to the left of $s_t$. Therefore we have $S[s_t - (m + 1)] = S[s_t - (m + 1) + p]$. We know from $lcs(s - 1, s_t - 1) \geq m$ that $S[s_t - (m + 1) + p] = S[s_t - (m + 1) + p]$. From substitution and transitivity we getystem and transitivity we get $S[s - (m + 1)] \neq S[s_t - (m + 1)]$ and therefore $lcp(s - 1, s_t - 1) = m$. Symmetric arguments can be made in the case in which $Ex_i > Ex_j^C$.

Evaluating $Ex_i$, $Ex_j$, $Ex_j^C$ and $Ex_j^C$ can be efficiently executed by querying $LCS(s - 1, s + p - 1)$, $LCP(e - 1, e - p - 1)$, $LCS(s_0 - 1, s_0 + p - 1)$ and $LCP(e_0 + 1, e_0 + p - 1)$ respectively.

6.2 Proof of Claim 38

Claim 38. Let $s$ be an index and let $C = (a, b, p)$ be a periodic cluster with $s_4 = a + t \cdot p$. There are consecutive intervals $I_s \subseteq [0 \ldots |C| - 1|$ such that $S^{s} \leq_S S^w$ if $t \in I_s$ and $S^{s} >_L S^w$ if $t \in I_s^C$. The intervals $I_s$ and $I_s^C$ can be evaluated in $O(log(n))$ time given $C$, $s$ and a data structure for $O(log(n))$ LCE queries.

Proof: We start by proving the following claim:

Claim 66. For every cluster $C = (a, b, p)$ of occurrences, let $Seq = S^{s_0}, S^{s_1}, \ldots S^{s_{|C| - 1}}$ be the sequence of suffixes starting in the starting indexes of occurrences of $W$ implied by $C$. Seq is either an increasing or a decreasing lexicographic sequence.

Proof: Let $s_i = a + i \cdot p$ and $s_{i+1} = s_i + p$ be two successive starting indexes of occurrences implied by $C$. Let $Ex_r$ be the extension of the run with period $p$ to the right of $s_0$. For every $j \in [s_0 + p, s_0 + Ex_r]$ we have $S[j] = S[j - p]$. In particular, $S[s_{i+1} + x] = S[s_i + x] = S[s_i + x]$ for $x \in [0, \ldots Ex_r - (i + 1) \cdot p]$. Since the run with period $p$ does not extend to the index $s_0 + Ex_r + 1$, we have $S[s_{i+1} + Ex_r - p \cdot (i + 1) + 1] = S[s_0 + Ex_r + 1] \neq S[s_0 + Ex_r + 1 - p] = S[s_i + Ex_r - (i + 1) \cdot p + 1]$. Therefore, $lcp(s_i, s_{i+1}) = Ex_r - p \cdot (i + 1)$ and the lexicographic order between $s_i$ and $s_{i+1}$ is only dependent on the lexicographic order between $s_0 + Ex_r + 1$ and $S[s_0 + Ex_r + 1 - p]$. The two characters deciding the order between $s_i$ and $s_{i+1}$ are independent from the value of $i$. Therefore, it is either the case that $S^{s_i} \leq_S S^{s_{i+1}}$ for every $i \in [0, \ldots |C| - 2]$ or $S^{s_i} >_L S^{s_{i+1}}$ for every $i \in [0, \ldots |C| - 2]$.

Claim 66 guarantees that the values $i \in [0, \ldots |C| - 1]$ such that $S^i \leq_S S^{i+1}$ (resp. $S^{i} >_L S^{i+1}$) form a consecutive sub interval $I_s$ (resp. $I_s^C$). Specifically either a suffix or a prefix of $[0, \ldots |C| - 1]$.

We proceed to show how to efficiently evaluate $I_s$. Similar technique can be used to evaluate $I_s^C$.

Let $r_0 = lcp(s_i, s)$ for $i \in [0, \ldots |C| - 1]$. We start by treating the case in which $r_0 < |w|$. Since every $s_i$ is a starting index of an occurrence of $w$, we must have $r_1 = r_0$ for every $i \in [0, \ldots |C| - 1]$. Therefore, the symbols deciding the lexicographic order between $S^i$ and $S^{s_i}$ are $S[s_0 + r_0]$ and $S[s_0 + r_0]$. Since $r_0 < p$, $S[s_0 + r_0] = w[r_0]$. It follows that the lexicographic order between $S^i$ and $S^{s_i}$ is independent of $i$, and therefore either $I_s = [1, \ldots |C| - 1]$ or $I_s = \emptyset$. We can decide which of these two options apply by comparing $S[s_0 + r_0]$ and $w[r_0]$.

We proceed to treat the case in which $r_0 \geq |w|$. If $l_0 \geq |w|$, we have that $s$ is a starting index of an occurrence of $w$. Therefore, Lemma 36 can be applied to obtain $r_1 = min(Ex_r, Ex_r^C - p \cdot i + |w|)$ for $Ex_r \neq Ex_r^C - p \cdot i$ (with the notations of $Ex_r$ and $Ex_r^C$ as in the statement of Lemma 36). Let $i' = \frac{Ex_r - Ex_r^C}{p}$. We treat three distinct sub intervals of $[1, \ldots |C| - 1]$. For every sub interval, we show that the lexicographic order between $S^i$ and $S^{s_i}$ is fixed within the sub interval.

Claim 67. The interval $[0, \ldots i']$ is either fully contained in $I_s$, or disjoint from $I_s$.

For $i \in [0, \ldots i']$, we have $Ex_r < Ex_r^C - p \cdot i$ and therefore $r_1 = Ex_r + |w|$. Therefore, the lexicographic order between $S^i$ and $S^{s_i}$ is decided by the values of $S[s + |w| + Ex_r]$ and $S[s + |w| + Ex_r^C]$. For $i < i'$, $s_i + |w| + Ex_r < s_0 + p \cdot i + |w| + Ex_r^C - p \cdot i = s_0 + |w| + Ex_r^C$. Therefore, $S[s_0 + |w| + Ex_r]$ is within the periodic run containing $w_0$. It follows that $S[s + |w| + Ex_r] = S[s_0 + p \cdot i + |w| + Ex_r^C] = S[s_0 + |w| + Ex_r^C]$. Both of the symbols that decide the lexicographic order between $S^i$ and $S^{s_i}$ are independent from $i$. Therefore, Either $[0, \ldots i'] \subseteq I_s$, or $[0, \ldots i'] \cup I_s = \emptyset$.

Claim 68. The interval $(i', \ldots |C| - 1]$ is either fully contained in $I_s$ or disjoint from $I_s$.

For $i \in (i', \ldots |C| - 1]$, we have $Ex_r > Ex_r^C - p \cdot i$ and therefore $r_1 = Ex_r^C - p \cdot i + |w|$. Therefore, the lexicographic order between $S^i$ and $S^{s_i}$ is decided by the values of $c_1 = S[s + |w| + Ex_r^C - p \cdot i]$ and $c_2 = S[s_0 + |w| + Ex_r^C - p \cdot i]$. For $i > i'$, $s + |w| + Ex_r^C - p \cdot i < s + |w| + Ex_r^C$. Therefore, the index $s + |w| + Ex_r^C - p \cdot i$ is contained within the run with period $p$ containing $w$. It follows that
$c_i^p = S_{[s+|w|+Ex_i^C-p-i]}$ is independent of $i$. Note that $c_i^p = S_{[s_0+p+i+|w|+Ex_i^C-p-i]} = S_{[s_0+|w|+Ex_i^C]}$. Both of the symbols that decide the lexicographic order between $S^*$ and $S^{p_i}$ are independent from $i$. Therefore, either \((i'...|C|-1) \subseteq I_3\) or \((i'...|C|-1) \cup I_2 = \emptyset\).

We use a single LCP query to find $r_0$. If $r_0 < |w|$, we can either report $I_2 = [0...|C|-1]$ or $I_2 = \emptyset$ according to the lexicographical order between $S^*$ and $S^{p_i}$.

If $r_0 \geq |w|$, we employ Lemma 66 to find $Ex_i^C$ and $Ex_x$ in $O(\log(n))$ time. With these in hand, \(\mathcal{U}\) can be evaluated in constant time. We lexicographically compare between $S^*$ and a single suffix from each of the intervals \([0...i'), (i'...|C|-1]\). According to Claim 67 and Claim 68, the results of these comparisons can be used to decide if \([0...i')\) and \((i'...|C|-1)\) form a part of $I_2$.

If \(i'\) is an integer, the single suffix $S^{p_i}$ needs to be compared to $S^*$ in order to decide if \(i' \in I_2\). We can evaluate \(i'\) in constant time. We report $I_2$ to be the union of the intervals that were found to be contained in $I_2$ and possibly \(i'\).

In the process of evaluating $I_2$, we executed a constant number of arithmetic operations and lcp queries, as well as a single application of Lemma 66. The time complexity is bounded by $O(\log(n))$.

**6.2.1 Proof of Theorem 63**

**Theorem 63.** Given a string $S[1...n]$, let $x \in [1...n]$ be an index within $S$. There are $O(\log(n))$ extremely periodic runs containing $x$.

The following well known fact is helpful for proving Theorem 63 and is derived directly from the periodicity lemma.

**Fact 69.** Two runs $r_1$ and $r_2$ in a string $S$ can have an overlap with length at most $\text{per}(r_1) + \text{per}(r_2) - 1$.

To prove Theorem 63, we prove the following lemma:

**Lemma 70.** For every $k \in [1...\log_2(n)]$, there can be at most two distinct runs $R$ containing the index $x$ with $(\dfrac{1}{2})^{k-1} \leq \text{Per}(R) < (\dfrac{1}{2})^k$.

**Proof:** Assume to the contrary that there are three distinct runs $R_1 = S[s_1...e_1]$, $R_2 = S[s_2...e_2]$ and $R_3 = S[s_3...e_3]$ containing the index $x$ with $\text{Per}(R_i) = p_i$ and $(\dfrac{1}{2})^{k-1} \leq p_i < (\dfrac{1}{2})^k$ for $i = 1, 2, 3$. Assume w.l.o.g that $s_1 \leq s_2 \leq s_3$. If one of the runs in our settings, $R_i$, fully contains one of the other runs, $R_j$. The overlap between them is $|R_j| \geq 5 \cdot p_j > p_j + \dfrac{3}{2} \cdot p_j > p_j + p_i$, in contradiction to Fact 69. Therefore, no run is containing the other and it must be the case that $e_1 < e_2 < e_3$. It follows that $R_2$ is completely covered by $R_1$ and $R_3$ and therefore $|R_2| \leq |R_1 \cap R_2| + |R_2 \cap R_3|$. Due to Fact 69, $|R_2 \cap R_1| < p_2 + p_3$ and $|R_2 \cap R_3| < p_2 + p_3$. Summing the two yields $|R_1 \cap R_2| + |R_2 \cap R_3| < 2 \cdot p_2 + p_1 + p_3$ and from transitivity we get $|R_2| < 2 \cdot p_2 + p_1 + p_3$. Since $5 \cdot p_2 \leq |R_2|$, we have $5 \cdot p_2 < p_1 + p_3$. Let $p_m$ be the maximal period among $p_1$ and $p_3$. We must have $3 \cdot p_2 < 2 \cdot p_m$ which finally yields $\frac{1}{2} \cdot p_2 < p_m$, which is a contradiction to the assumption that $p_m, p_2 \in [\dfrac{1}{2}^{k-1}...\dfrac{1}{2})^k]$.

With Lemma 70, the proof of Theorem 63 is straightforward. For every $k \in [1...\log_2(n)],$ there are at most 2 runs with period \(\left(\dfrac{1}{2}\right)^{k-1} \leq \text{Per}(R) < \left(\dfrac{1}{2}\right)^k\) containing $x$.

**6.3 Proof of Lemma 41**

**Lemma 41.** Let $U = u_1, u_2,..., u_{|U|-1}$ be a set of interval increment updates with $u_i = (i_t, j_t, 1)$ and both $\{i_t\}$ and $\{j_t\}$ being p-normal sequences for some integer $p$ with degrees $\text{Deg}(\{i_t\}), \text{Deg}(\{j_t\}) \in O(1)$. Applying all the interval updates in $U$ is equivalent to applying a constant number of interval increment updates and stairs updates with stair width $|p|$ to $R$. The set of equivalent stairs and interval updates can be evaluated in constant time given the representation of $i_t$ and $j_t$ as p-normal sequence (The input can be given, for example, as the construction trees of $\{i_t\}$ and $\{j_t\}$).

**Proof:** We start by proving the following claim:

**Claim 71.** Let $U = u_0, u_1,..., u_{|U|-1}$ be a sequence of interval increment updates with $u_i = (i_t, j_t, 1)$ and let $p$ be an integer. If $\{i_t\}$ and $\{j_t\}$ are both p-normal sequences with $\text{deg}(\{i_t\}) = d_i$, and $\text{deg}(\{j_t\}) = d_j$, applying all the interval updates in $U$ to an array can be reduced to applying $O(2^{d_i+d_j})$ interval increment updates and stairs updates with stair width $|p|$. These stairs updates can be obtained in time $O(2^{d_i+d_j})$ given the representation of $\{i_t\}$ and $\{j_t\}$ as p-normal sequences.

**Proof:** By strong induction on $d_i + d_j$. Our base case is when $d_i = d_j = 1$. If it is the case, one of the following must occur:
1. Both $i_t$ and $j_t$ are arithmetic progressions with difference $p$.

2. One of $i_t, j_t$ is an arithmetic progression with difference $p$ and the other is fixed.

3. Both $i_t$ and $j_t$ are fixed.

Lemma 106 in Section 8 suggests that applying all the interval updates $(i_t, j_t, 1)$ for $t \in [0 \ldots |U| - 1]$ can be reduced to applying a constant number of stairs updates. This concludes the induction base.

Assume $d_i + d_j = D > 2$. In particular, assume $d_i \geq 2$. The case in which $d_j \geq 2$ is treated symmetrically. We start with the following claim:

**Claim 72.** If $d_i \geq 2$, $CT\{i_t\}$ contains a $p\text{-normal sequence } B'$ with $\deg(B') = 2$.

**Proof:** It is easy to observe that a node corresponding to a $p\text{-normal sequence } B'$ has $\deg(B') = 2$ if its construction tree is a root with two leaf children. Every finite full binary tree of size at least 3 must contain a node with two leaf children.

Let $B'$ be a $p\text{-normal sequence in } CT\{i_t\}$ with $\deg(B') = 2$. Assume that $B'$ is a maximum sequence $b'_t = \max\{a^1_t, a^2_t\}$. The case in which $B'$ is a minimum sequence can be handled similarly. $a^1_t$ and $a^2_t$ are both $p\text{-normal sequences with degree 1.}$ So one of the following must occur:

1. Both $a^1_t$ and $a^2_t$ are fixed
2. Both $a^1_t$ and $a^2_t$ are arithmetic progression with difference $p$.
3. One of $a^1_t$ or $a^2_t$ is an arithmetic progression with difference $p$, and the other is fixed.

If $1$ occurs, $B'$ can be replaced with the fixed value $\max(a^1_0, a^2_0)$. If $2$ occurs, $B'$ can be replaced with the arithmetic progression $a^y_0 + p \cdot t$ with the maximal value of $a^y_0$ for $y \in \{0, 1\}$. In both $1$ and $2$, we replace every occurrence of $B'$ in $\{i_t\}$ with a $p\text{-normal sequence with degree 1.}$ By doing so we obtain an equivalent sequence $\{i'_t\}$ with degree $d_i < d_i$. From the induction hypothesis, the updates sequence $\{i'_t\}$ can be reduced to $O(2^{d_i+d_j}) = O(2^{d_i+d_j})$ stairs updates. Since $\{i'_t\}$ is equivalent to $\{i_t\}$, applying these stairs updates is equivalent to applying the updates of $U$. If $3$ occurs, assume w.l.o.g that $a^1_t$ is a fixed value sequence with value $a$ and $a^2_t = a^2_0 + p \cdot t$. Assume that $p \geq 0$ and let $t' = \lfloor \frac{a-a^2_0}{p} \rfloor$. It can be easily verified that $\max(a^1_t, a^2_t) = a^1_t = a$ for $t \in [0 \ldots t']$ and that $\max(a^1_t, a^2_t) = a^2 = a^2_0 + p \cdot t$ for $t \in [t' + 1 \ldots |U| - 1]$.

Therefore, in $t \in [0 \ldots t']$ we can replace every occurrence of $B'$ with a fixed sequence $a$. In $t \in [t' + 1 \ldots |U| - 1]$ we can replace every occurrence of $B'$ with an arithmetic progression $a^a_0 + p$. In both cases, we replace an expression with degree 2 with an expression with degree 1 and therefore obtain a $p\text{-normal sequence with a strictly lower degree.}$ By the recursion hypothesis, both the updates $u_0, u_1 \ldots u_{t'}$ and the updates $u_{t' + 1} \ldots u_{|U| - 1}$ can be represented by a set of $O(2^{d_i+d_j-1})$ stairs updates. Applying both of these sets is equivalent to applying all the updates in $U$. The collective size of these sets is $O(2^{2d_i+d_j-1}) = O(2^{d_i+d_j})$.

An algorithm for finding the set of stairs updates in $O(2^{d_i+d_j})$ time (ignoring polynomial factors) can be directly derived from the proof.

With Claim 71, we are finally ready to prove Lemma 11:

Let $d_i = Deg\{i_t\}$ and $d_j = Deg\{i_t\}$. According to Lemma 71, the updates of $U$ can be reduced to $O(2^{d_i+d_j})$ stairs or interval updates in $O(2^{d_i+d_j})$ time. Since $d_i, d_j \in O(1)$, we have $O(2^{d_i+d_j}) = O(1)$.

**7 Extension Restricting Selection Queries**

This section is dedicated to proving Theorem 23.

**7.1 Occurrence Selection Queries**

We start by providing data structure for the following problem:

| Occurrence Selection Query |
|----------------------------|
| **Input:** $POR_w$, the periodic occurrences representation of a word $w$. |
| **Query:** upon input $i$. Return $A_w[i]$ |
A visualization of the proof of Lemma 74. [Left] A demonstration of the first case. In the picture, \( r_p(w_1) = 3 \) and \( r_p(w_2) = 2 \). Following the first two instances of \( w_s \), the comparison between \( w_1 \) and \( w_2 \) comes down to comparing \( w_s \) and \( \text{Tail}(C^d) \). [Center] A demonstration of the second case with \( r_p(w_1) = 2 \) and \( r_p(w_2) = 3 \). Symmetrically to the previous case, the comparison between \( w_1 \) and \( w_2 \) comes down to comparing \( C^i \) and \( w_s \). [Right] A demonstration of the third case with \( r_p(w_1) = r_p(w_2) = 3 \). The comparison between \( w_1 \) and \( w_2 \) comes down to the comparison between the tails of the clusters.

The goal is to efficiently process \( \text{POR}_w \) to enable an occurrence selection query in polylogarithmic time, without explicitly constructing \( A_w \). Specifically, we wish to construct a data structure for OS queries in time \( O(|\text{POR}_w|) \).

In this section, we make an extensive use of the terms period rank, tail of a cluster, and head of a cluster as defined in Section 4. We provide the following example to better familiarize the reader with these terms. Figure 7 is another example of these terms.

**Example 73.** Let \( S = \text{BAABBABBBABBABABAB} \). The cluster \( C = (4, 13, 3) \) represents a periodic set of occurrence of \( w = BBABBAB \) with period 3. Consider the occurrence in \( w_0 = S[4 \ldots 10] \). Its period rank is \( r_p(w_0) = 3 \), since there are 3 consecutive occurrences of \( w[5 \ldots 7] = BAB \) following index 10. The period rank of the second occurrence in the cluster, \( w_1 = S[7 \ldots 13] \), is \( r_p(w_1) = 2 \). One can easily observe that the \( t \)-th occurrence has \( r_p(w_1) = |C| - t - 1 \). The tail of \( C \) is \( \text{Tail}(C) = S[20 \ldots 21] = AB \) and the head is \( \text{Head}(C) = S[1 \ldots 3] = BAA \).

The tail of a cluster can be equivalently described as the suffix following the rightmost consecutive occurrence of \( w_s \) following \( w_0 \) (or equivalently, any other \( w_i \)). Since \( w_{|C|-1} \) has the rightmost occurrence of \( w_s \) as a suffix, the tail starts right after \( w_{|C|-1} \). Notice that we must have \( \text{Tail}(C)[1 \ldots p] \neq w_s \). Otherwise, there would be another occurrence of \( w \) exactly \( p \) indexes to the right of \( w_{|C|-1} \).

**Lemma 74.** Let \( C^i = (a^i, b^i, p), C^d = (a^d, b^d, p) \in \text{POR}_w \) be an increasing cluster and a decreasing cluster respectively (as defined in Section 4). Let \( w_1 = S[s_1 \ldots e_1] \) be an occurrence of \( w \) implied by \( C^i \) and \( w_2 = [s_2 \ldots e_2] \) be an occurrence of \( w \) implied by \( C^d \). It must be the case that \( S[s_1 \ldots n] >_L S[s_2 \ldots n] \).

**Proof:** In what follows, we slightly abuse the notation of \( <_L \). We use the notation \( w_1 <_L w_2 \) between two equal words to denote that the suffixes starting in their corresponding starting indexes have \( S[s_1 \ldots n] <_L S[s_2 \ldots n] \).

We distinguish between three cases (See Figure 7 for a visualization):

1. \( r_p(w_1) > r_p(w_2) \): In this case, the first \( |w| + p \cdot r_p(w_2) \) symbols of both suffixes match. The \( p \) symbols following \( S[e_1 + p \cdot r_p(w_1)] \) are an occurrence of \( w_s \). The \( p \) symbols following \( S[e_2 + p \cdot r_p(w_2)] \) are \( \text{Tail}(C^d) \). Since \( C^d \) is decreasing, \( w_s > \text{Tail}(C^d) \) and we have \( w_1 >_L w_2 \).

2. \( r_p(w_1) < r_p(w_2) \): In this case, the first \( |w| + p \cdot r_p(w_1) \) symbols of both suffixes match. The \( p \) symbols following \( S[e_1 + p \cdot r_p(w_1)] \) are \( \text{Tail}(C^i) \) and the \( p \) symbols following \( S[e_2 + p \cdot r_p(w_2)] \) are an occurrence of \( w_s \). Since \( C^i \) is increasing, \( \text{Tail}(C^i) > w_s \) and we have \( w_1 >_L w_2 \).

3. \( r_p(w_1) = r_p(w_2) \) : In this case, the lexicographic order is determined by the lexicographic order between \( \text{Tail}(C^i) \) and \( \text{Tail}(C^d) \). We have \( \text{Tail}(C^i) >_L w_s >_L \text{Tail}(C^d) \).

**Lemma 74** suggests that \( A_w \) can be expressed as the concatenation of two arrays \( A = D[1 \ldots |D|][I[1 \ldots |I|] \) with \( D \) the array of lexicographically sorted suffixes implied by decreasing clusters and \( I \) the lexicographically sorted suffixes implied by increasing clusters. We can use this fact to reduce our task to either querying \( D \) or \( I \) for the element in a specified index. Upon a query for \( A_w[i] \) with \( i \leq |D| \), we query
D[i]. For \( i > |D| \) we query \( I[i − |D| − 1] \). We can calculate the size of \( D \) in \( O(\frac{r}{2}) \) time by accumulating the sizes of the decreasing clusters. From now on, we show how to construct a data structure for finding \( I[i] \) for a given index \( i \). A data structure for finding \( D[i] \) can be constructed in a similar manner.

Using the same arguments as in the proof of Lemma 74, the following can be proven:

**Lemma 75.** \( C_1 = (a_1, b_1, p), C_2 = (a_2, b_2, p) \) \( \in \text{Occ} \) be two increasing clusters (not necessarily distinct). Let \( w_1 = S[s_1 \ldots e_1] \) be an occurrence of \( w \) implied by \( C_1 \) and \( w_2 = [s_2 \ldots e_2] \) be an occurrence of \( w \) implied by \( C_2 \).

1. If \( r_p(w_1) > r_p(w_2) \), \( w_1 <_L w_2 \).
2. If \( r_p(w_1) = r_p(w_2) \) and \( \text{Tail}(C_1) <_L \text{Tail}(C_2) \), \( w_1 <_L w_2 \).

Lemma 75 suggests that sorting the suffixes of \( I \) by lexicographic order is equivalent to sorting them primarily by decreasing order of their rank, and secondary by increasing lexicographic order of the tails of their clusters. Namely, \( I \) can be written as the concatenation of \( m \) arrays, with \( m \) being the maximal rank of an occurrence within an increasing cluster. \( I = T_m, T_{m−1} \ldots T_1, T_0 \). For every \( r \in [0 \ldots m] \), \( T_r \) is the set of occurrences with rank \( r \) sorted by increasing lexicographic order of the tails of their clusters.

The size of \( T_r \) is the amount of clusters with size at least \( r + 1 \). Given an index \( i \), we are interested in finding \( r \) and \( i' \) such that \( I[i] \in T_r \) and in particular, \( I[i] = T_r[i'] \). See Figure 8.

We process the set of increasing clusters in \( \text{POR}_w \), denoted as \( \text{POR}_I \), to obtain the following.

1. \( \text{Ranks} = r_1, r_2, r_3 \ldots r_d \) - the list of distinct values of the maximal ranks within an increasing cluster in \( \text{POR}_I \) in decreasing order. Namely, if there is a cluster \( C \in \text{POR}_I \) with size \( x \), the number \( x − 1 \) is in \( \text{Ranks} \).
2. \( \text{Sizes} = s_1, s_2, s_3 \ldots s_d \) - a list with \( s_q \) being the amount of clusters with a maximal rank of at least \( r_q \). Note that \( \text{Sizes} \) is monotonically increasing.
3. \( \text{Tails} = t_1, t_2, t_3 \ldots t_{|\text{POR}_I|} \) - a lexicographically sorted list of the tails of the clusters in \( \text{POR}_I \).

\( \text{Tail}(C) \) is represented in \( \text{Tails} \) as a pointer to \( C \).

Note that \( r_1 = m \) and \( s_d = |\text{POR}_I| \). We define a set of \( d \) subarrays (denoted as ‘buckets’) of \( I \) such that \( I = B_1, B_2 \ldots B_d \). The bucket \( B_q \) for \( q \in [1 \ldots d − 1] \) contains the concatenation of \( T_z \) for \( z \in [r_{q+1} + 1 \ldots r_q] \) in decreasing order of \( z \) (as they appear in \( I \)). \( B_d \) is the concatenation of \( T_z \) for \( z \in [0 \ldots r_d] \) in decreasing order of \( z \). See Figure 9 for a demonstration of \( \text{Ranks} \), \( \text{Sizes} \), \( \text{Tails} \), and the buckets.

We make the following simple observation.

**Observation 76.** The size of every \( T_2 \) in the bucket \( B_q \) is \( s_q \).

The following directly follows.

**Observation 77.** The size of \( B_q \) is \( s_q \cdot (r_q − r_{q+1}) \) (with \( r_{d+1} = −1 \))
Using the above, we calculate the sizes of the buckets, and use them to evaluate the starting index $i_q$ of every bucket $B_q$ within $I$. We insert the starting indexes to a data structure for predecessor queries $B_{starts}$. For every cluster $C$, we denote the rank of Tail($C$) in Tails as $tr(C)$. We initialize a data structure $T_{lex}$ for 2-d range counting queries. For every cluster in $POR_I$, we add the point $(tr(C), |C|)$ to $T_{lex}$. This concludes the construction of our data structure.

Upon query input $i$, we query $B_{starts}$ for the predecessor of $i$ and obtain $i_z$. We deduce that $i$ is within the bucket $B_q$. Let $\ell_z = r_z - r_{z+1}$ be the number of $T$ sub-arrays contained in $B_z$. For clearer presentation, we rename the $T$ sub-arrays contained in $B_z$ and denote $B_z = T_0^z \cdot T_1^z \ldots T_{\ell_z-1}^z \cdot T_{\ell_z}^z$. It can be easily verified that $T_q^z = T_{r_q-z}^z$ for every $q \in [0 \ldots \ell_z - 1]$. Let $T_{\ell_z}^z$ be the sub array in the concatenation of $B_z$ containing $i$. Since every $T_q^z$ has the same size $s_z$, we can find the index $r'$ by evaluating $r' = \lfloor \frac{i - \ell_z}{s_z} \rfloor$. Via the aforementioned relation, we use $r'$ to obtain an index $r$ such that $T_r$ contains $i$. We also calculate the index $i' = i - i_z - s_z \cdot r'$ that satisfies $T_{i'}[i'] = I[i]$. 

Our next task is to find the $i'$th lexicographic suffix in $T_r$. Recall that $T_r$ consists of the suffixes with rank $r$ from the clusters of size at least $r + 1$ sorted by increasing lexicographic order of the tails of their corresponding clusters.

In order to find $T_r[i']$, we perform a ‘restricted’ binary search on Tails using $T_{lex}$ (See Figure 10 for a demonstration). We start by querying $T_{lex}$ for $L = Count([1 \ldots \lfloor |POR_I| \rfloor \times [r + 1 \ldots \infty])$. The condition on the second coordinate ensures that only clusters with size at least $r + 1$ are counted. These are the clusters that have an implied occurrence in $T_r$. The second coordinate condition fixes for all the queries of the binary search. If $L \geq i'$ we deduce that $T_r[i']$ is in the left half of Tails and otherwise we deduce that it is in the right half. We shrink the searched interval accordingly and proceed recursively to finally obtain a tail Tail($C$) that is the $i'$th lexicographic tail among the tails of clusters participating in $T_r$. We deduce that $T_r[i']$ is from the cluster of $t$, denoted as $C = (a, b, p)$. Now we know the rank and the source cluster of $A[i]$, so we can finally report $I[i] = a + ((|C| - r - 1) \cdot p)$

**Preprocess complexity:** Ranks, Sizes and the sizes and starting indexes of the buckets can be evaluated in $\tilde{O}(|POR_I|)$ time by sorting the sizes of the clusters. Inserting the $O(|POR_I|)$ starting indexes of buckets to $B_{starts}$ takes $\tilde{O}(|POR_I|)$ using, for example, a balanced search tree. Tails can be obtained
Figure 10: A demonstration of the binary search executed using Tlex. The clusters corresponding to circles above the \( r + 1 \) horizontal line have a represented occurrence that participates in \( T_r \). Since the occurrences appear in \( T_r \) in lexicographic order of their clusters, we are interested in finding the \( i' \) point from the left above this line.

in \( \tilde{O}(|POR|) \) time by sorting the tails of the clusters with a standard \( O(n \log n) \)-comparisons sorting algorithm using LCP queries to compare between tails. Insertions to Tlex are also polylogarithmic.

The space complexity is dominated by the space dedicated to the 2 dimensional data structure Tlex. It is bounded by \( O(|POR| \log(|POR|) = \tilde{O}(|POR|)) \). Recall that for queries on \( A_w \), we need to do a symmetric preprocessing for the set of decreasing clusters too. The overall time and space complexity are both \( \tilde{O}(|POR|) \).

Query time: In the process of finding \( r \) and \( i' \), we use a predecessor query and constant number of array lookups and basic arithmetic operations to find \( i_z \), \( r' \) and \( |B_z| \). The complexity for this part of the query is dominated by \( O(\log(|POR|)) \) \( \subseteq \tilde{O}(\log(n)) \).

To find \( T_r[i'] \), we execute a restricted binary search on Tails. We use a range query on Tlex in every iteration of the binary search. A 2-dimensional counting query is executed in \( O(\log^2(|POR|)) \), so the overall time complexity of this step is \( O(\log^3(|POR|)) \) \( \subseteq \tilde{O}(\log^3(n)) \).

We conclude this subsection with the following:

**Theorem 78.** We can process a periodic clusters of occurrences \( POR_w \) of a word \( w \) in \( \tilde{O}(|POR|) \) to support look-up queries for \( A_w[i] \) in \( O(\log^2(n)) \) time. As a side effect, our query can also output the indexes \( i' \) and \( r \) such that \( A_w[i] = T_r[i'] \) as well as the cluster \( C \) implying the occurrence starting in \( A_w[i] \).

### 7.2 Extensions Restricting Range Counting

Let \( POR_w \) be the periodic occurrences representation of a word \( w = S[s \ldots e] \). In this subsection we introduce the final ingredient for Extensions Restricting Selection data structure:

| **Extension Restricting Range Counting** |
| **Input:** POR<sub>w</sub> |
| **Query:** for input integers \( l, r, i, j \), return the amount of \((l, r)\)-extendable occurrences in \( A_w[i \ldots j] \) |

Eventually, we will show that Extension Restricting Selection query can be reduced to \( \log(n) \) Extension Restricting Range Counting queries via a binary search.

First, we recall that \( A_w = D[1 \ldots |D|][I[1 \ldots |I|] \] with \( D \) and \( I \) being the sets of occurrences of \( w \) represented by \( POR_w \) implied by a decreasing and by an increasing cluster, respectively. \( A_w[i \ldots j] \) is either a sub array of \( I \), a sub array of \( D \), or a concatenation of a suffix of \( D \) and a prefix of \( I \). Therefore, it is sufficient to count \((l, r)\)-extendable occurrences in an interval within \( D \) and an interval within \( I \).

We provide a data structure for counting \((l, r)\)-extendable occurrences in an interval within \( I \). A similar algorithm can be constructed to count the \((l, r)\)-extendable occurrences within an interval in \( D \).
7.2.1 Case 1: Periodic Extended Word

Let \( p \) be the period of \( w = S[s \ldots e] \). In this subsection, we deal with the case in which \( w' = S[s-l \ldots e+r] \) also has a period \( p \). In these settings, we can use the following rule to decide if an occurrence of \( w \) is \((l, r)\)-extendable:

**Fact 79.** Let \( C \in \text{POR}_w \) be a periodic cluster of occurrences of \( w \). Let \( R_C = S[s_C \ldots e_C] \) and let \( w_t = S[s_t \ldots e_t] \) be an occurrence represented by \( C \). Let \( Ex_t^l = s_1 - s_t \) be the length of the part of \( R_C \) to the left of \( w_t \) and \( Ex_t^r = e_r - e_1 \) be length of the part of \( R_C \) to the right of \( w_t \). \( w_t \) is \((l, r)\)-extendable iff \( Ex_t^l \geq l \) and \( Ex_t^r \geq r \).

In words - \( w_t \) is \((l, r)\)-extendable iff \( S[s_t - l \ldots e_t + r] \) is contained within \( R_C \). See Figure 11 for an illustration.

![Illustration](image)

Figure 11: An illustration of Fact 79. \( w \) has \( l \) symbols to its left and \( r \) symbols to its right contained within a run with period \( p \). An occurrence \( w_t \) must have this property as well in order to be \((l, r)\)-extendable.

Let \( q_R, q_L, r_R < p \) and \( r_L < p \) be the unique non negative integers such that \( r = q_R \cdot p + r_R \) and \( l = q_L \cdot p + r_L \). Consider a cluster \( C = (a, b, p) \in \text{POR}_t \) with implied occurrences \( w_t = S[s_t \ldots e_t] \) starting in indexes \( s_t = a + t \cdot p \) for \( t \in [0, |C| - 1] \). We aim to provide a characterization of the \((r, l)\)-extendable occurrences within \( C \). For this purpose, we consider 3 types of occurrences represented by \( C \):

1. An occurrence \( w_t \) is central if \( r_p(w_t) \in [q_R + 1 \ldots |C| - q_L - 2] \)
2. An occurrence \( w_t \) is borderline if \( |C| \geq q_R + q_L + 1 \) and \( r_p(w_t) \in \{q_R, |C| - q_L - 1\} \)
3. An occurrence \( w_t \) is marginal if it is neither central nor borderline.

Let \( R_C = S[s_C \ldots e_C] \). We present the notation \( r_L(C) = s_0 - s_C \) and \( r_R(C) = e_C - e_{|C|-1} \). \( r_L(C) \) (resp. \( r_R(C) \)) is the remainder from the left (resp. right) of the run \( R_C \) that is not contained within any of the occurrences in \( C \).

**Example 80.** In the text \( S = bbababababababababababa \), there is a cluster \( C \) of occurrences of \( w = ababababababa \) with \( R_C = S[2 \ldots 27] = bbababababababababababa \). The prefix \( bb \) precedes the leftmost occurrence represented by \( C \), therefore \( r_L(C) = 2 \). The suffix \( bab \) with length \( 3 \) of \( R_C \) follows the rightmost occurrence represented by \( C \), therefore we have \( r_R(C) = 3 \).

We make the following simple Claim.

**Claim 81.** An occurrence \( w_t \) represented by a cluster \( C \) has a part with length exactly \((|C| - 1 - r_p(w_t)) \cdot p + r_C(C) \) of \( R_C \) to its left and a part with length \( r_p(w_t) \cdot p + r_C(C) \) of \( R_C \) to its right.

**Proof:** By definition, \( w_0 \) has a part of length exactly \( r_L(C) \) of \( R_C \) to its left, which is consistent with our claim as \( r_p(w_0) = |C| - 1 \). Every consecutive occurrence to the right has a period rank lower by 1, and a part with length longer by \( p \) of \( R_C \) to its left (as it starts \( p \) indexes to the right of the previous occurrence). This inductively proves the correctness of our claim for all the occurrences represented by \( |C| \). The length of the extension to the right can be proven in a symmetric manner. □

The following holds due to Fact 79.
Lemma 82. 1. Every central occurrence is an \((l,r)\)-extendable occurrence.

2. Every marginal occurrence is not an \((l,r)\)-extendable occurrence.

3. A borderline occurrence with \(r_p(w_t) = q_R < |C| - q_L - 1\) is an \((l,r)\)-extendable occurrence iff \(r_R(C) \geq r_R\).

4. A borderline occurrence with \(r_p(w_t) = |C| - q_L - 1 > q_R\) is an \((l,r)\)-extendable occurrence iff \(r_L(C) \geq r_L\).

5. A borderline occurrence with \(r_p(s_t) = |C| - q_L - 1 = q_R\) is an \((l,r)\)-extendable occurrence iff \(r_L(C) \geq r_L\) and \(r_R(C) \geq r_R\).

Proof: Applying Claim \(\text{S1}\) and Fact \(\text{F9}\) on every case yields the corresponding result. □

Note that if \(|C| < q_L + q_R + 1\), all the occurrences are marginal occurrences. So \(C\) does not contain any \((l,r)\)-extendable occurrences.

We recall the partition \(I = T_m, T_{m-1} \ldots T_1, T_0\) as in the previous section. Let \(T_r\) be the sub array containing \(i\) and \(T_j\) be the sub array containing \(j\). And let \(i_r\) and \(j_r\) be two indexes such that \(T_r, [i_r, j_r] = I[i]\) and \(T_r, [j_r, j] = I[j]\). Every sub array \(T_r\) with \(r \in [r_j + 1 \ldots r_i - 1]\) is fully contained within \(I[i \ldots j]\). Therefore, every occurrence with rank \(r \in [r_j + 1 \ldots r_i - 1]\) is in \(I[i \ldots j]\).

We define 5 subsets of the \((l,r)\)-extendable occurrences that a cluster \(C\) may contribute to \(I[i \ldots j]\):

1. Cent\((C)\) - Central occurrences \(w_t\) with period rank \(r_p(w_t) \in [r_j + 1 \ldots r_i - 1]\).
2. Bor\(_R(C)\) - an \((l,r)\)-extendable borderline occurrence \(w_t\) with \(r_p(w_t) = q_R \in [r_j + 1 \ldots r_i - 1]\)
3. Bor\(_L(C)\) - an \((l,r)\)-extendable borderline occurrence \(w_t\) with \(r_p(w_t) = |C| - q_L - 1 \in [r_j + 1 \ldots r_i - 1]\).
4. \(T_r(C)\) - an \((l,r)\)-extendable occurrence \(w_t\) with \(r_p(w_t) = r_i\)
5. \(T_j(C)\) - an \((l,r)\)-extendable occurrence \(w_t\) with \(r_p(w_t) = r_j\)

Note that apart from Cent\((C)\), all the listed sets are either of size 0 or of size 1. Also note that the union of these sets form the set of \((l,r)\)-extendable occurrences implied by \(C\) in \(I[i \ldots j]\), so our problem is reduced to accumulating the sizes of these unions over all the clusters in \(\text{POR}_I\).

We start by considering the size of Cent\((C)\). Notice that only clusters with \(|C| \geq q_R + q_L + 3\) have central occurrences.

Fact 83. 1. If \(|C| - q_L - 2 \geq r_i - 1\), then \(|\text{Cent}(C)| = r_i - 1 - \max(r_j, q_R)\).

2. If \(q_R + q_L + 3 \leq |C|\) and \(|C| - q_L - 1 < r_i - 1\), then \(|\text{Cent}(C)| = |C| - q_L - 2 - \max(r_j, q_R)\)

Proof: The Central occurrences of \(C\) are those with ranks in \(I_{cen} = [q_R + 1 \ldots |C| - q_L - 2]\). Cent\((C)\) is the intersection of the Central occurrences with the occurrences with rank in \(I_c = [r_j + 1 \ldots r_i - 1]\). This intersection is \([\max(r_j + 1, q_R + 1) \ldots \min(r_i - 1, |C| - q_L - 2)]\). The size of the interval yields the formula. □

Next, we consider the sizes of Bor\(_R(C)\) and Bor\(_L(C)\). Note that each of these sets can contain at most 1 occurrence. Also note that only clusters with \(|C| \geq q_R + q_L + 1\) have borderline occurrences.

The following facts hold directly from Lemma \(\text{S2}\) and the definitions of Bor\(_R(C)\) and Bor\(_L(C)\).

Fact 84. If \(|C| \geq q_R + q_L + 2\):

1. Bor\(_R(C)\) and Bor\(_L(C)\) are disjoint.

2. We have \(|\text{Bor}_R(C)| = 1\) if \(r_R(C) \geq r_R\) and \(q_R \in [r_j + 1 \ldots r_i - 1]\).

3. We have \(|\text{Bor}_L(C)| = 1\) if \(r_L(C) \geq r_L\) and \(|C| - q_L - 2 \in [r_j + 1 \ldots r_i - 1]\).

Fact 85. If \(|C| = q_R + q_L + 1\):

1. Bor\(_R(C)\) = Bor\(_L(C)\)

2. \(|\text{Bor}_R(C)| = 1\) if \(r_R(C) \geq r_R, r_R(C) \geq r_L\) and \(q_R \in [r_j + 1 \ldots r_i - 1]\).
Finally, we present a useful fact for evaluating the sizes of $T_i(z)(C)$ and $T_r(z)(C)$. These are also of size at most 1. Recall that the occurrences in $T_i$ and $T_r$ are sorted by lexicographical order of the tails of their clusters. We denote as $C_i$ and $C_j$ be the clusters representing $I[i]$ and $I[j]$, respectively. The following Facts directly follow:

**Fact 86.** Let $T_i[i_z, \ldots, j_z]$ be a subarray of some $T_i$. Let $C_{i_z}$ and $C_{j_z}$ be the clusters representing $T_i[i_z]$ and $T_r[j_z]$, respectively. The cluster $C$ has an $(l, r)$ extendable occurrence in $T_i[i_z, \ldots, j_z]$ iff the occurrence $w_i$ implied by $C$ with $r_j(w_i) = z$ is $(l, r)$-extendable and $Tail(C_{i_z}) \leq L Tail(C) \leq L Tail(C_{j_z})$.

Equipped with facts 83 and 86, we can express the sum of the contributions of all the clusters to $I[i \ldots j]$ as a set of $O(1)$ range queries. We initialize the notations of $tr(C)$ - the lexicographic rank of $Tail(C)$ among the clusters in $POR_i$. We initialize a 4-dimensional range queries data structures $Tlex$. For every cluster $C$, we insert the point $p_C = (|C|, r_L(C), r_R(C), tr(C))$ with $v(p_C) = |C|$, to $Tlex$.

**Lemma 87.** The sum $\Sigma_{C \in POR_i} |Cent(C)|$ can be evaluated in $O(\log^4(n))$ time given $Tlex$.

**Proof:** We make the following range queries on $Tlex$:

1. $sum_{mid} = SM(|q_L + q_R + 3 \ldots r_i + q_L| \times \ldots)$
2. $count_{mid} = COUNT(|q_L + q_R + 3 \ldots r_i + q_L| \times \ldots)$
3. $count_{large} = COUNT(|\max(r_i + q_L + 1, q_R + q_L + 3) \ldots \infty| \times \ldots)$

The $\times \cdots \times$ symbol denotes the complete range of the corresponding coordinate and is equivalent to $[-\infty, \ldots, \infty]$. According to Fact 83, Every cluster $C$ such that $p_C$ is in the interval counted in $count_{large}$ has $|Cent(C)| = r_i - 1 - \max(r_j, q_R)$. So the overall contribution of these clusters to $\Sigma_{C \in POR_i}$ is $count_{large} = count_{large}(r_i - 1 - \max(r_j, q_R))$.

Every cluster $C$ such that the value of $p_C$ is accumulated in $sum_{mid}$ has $|Cent(C)| = |C| - q_L - 2 - \max(r_j, q_R)$. The overall contribution of these clusters to $\Sigma_{C \in POR_i}$ is $count_{mid} = sum_{mid} - count_{mid} \cdot (q_L + 2 + \max(r_j, q_R))$. We have $\Sigma_{C \in POR_i} = count_{mid} + count_{large}$ since the clusters whose contributions are considered in $count_{mid}$ are disjoint from the clusters considered in $count_{large}$, and every cluster that was not considered in either of the two is too small to have central occurrences. We use a constant number of 4-dimensional range queries so the time complexity is $O(\log^4(n))$.

**Lemma 88.** $\Sigma_{Bord} = \Sigma_{C \in POR_i} |Bord_L(C) \cup Bord_R(C)|$ can be evaluated in $O(\log^4(n))$ time given $Tlex$.

**Proof:** We make the following range queries on $Tlex$:

1. $count_L = COUNT(|\max(q_R + q_L + 2 \ldots r_i + q_L| \times \ldots)|$)
2. $count_R = COUNT(|q_R + q_L + 2 \ldots \infty| \times \ldots)$.
3. $count_M = COUNT(|q_R + q_L + 1 \ldots q_R + q_L + 1| \times \ldots|)$.

Note that according to Fact 84, the point $p_C$ is counted in $count_L$ iff $|C| > q_R + q_L + 2$ and $|Bord_L(C)| = 1$, and the point $p_C$ is counted in $count_R$ iff $|C| > q_R + q_L + 2$ and $|Bord_R(C)| = 1$. Fact 84 suggests that $Bord_L(C)$ and $Bord_R(C)$ are distinct when $|C| \geq q_R + q_L + 2$. So the overall contribution of clusters with size at least $q_R + q_L + 2$ to $\Sigma_{Bord}$ is exactly $count_L + count_R$.

According to Fact 85, a point $p_C$ is counted in $count_M$ iff $C$ has a single Borderline $(r, l)$-extendable occurrence $w_i$ with $r_j(w_i) = q_R$. It follows that the contribution of clusters with size $q_R + q_L + 1$ to $\Sigma_{Bord}$ is $count_M$. To summarize, $\Sigma_{Bord} = count_L + count_R + count_M$. We use a constant number of 4-dimensional range queries to evaluate $\Sigma_{Bord}$, so the time complexity is $O(\log^4(n))$.

**Lemma 89.** $\Sigma_T = \Sigma_{C \in POR_i} |T_r(z)(C) \cup T_r(z)(C)|$ can be evaluated in $O(\log^4(n))$ time given $Tlex$.

Specifically, we prove the following:

**Lemma 90.** Given an interval $[i_z, \ldots, j_z]$ and a period rank $z$, the amount of $(l, r)$-extendable occurrences in $T_z[i_z, \ldots, j_z]$ can be evaluated in $O(\log^4(n))$ time given $Tlex$.

**Proof:** Let $C_{i_z}$ and $C_{j_z}$ be the clusters containing $T_z[i_z]$ and $T_z[j_z]$ respectively. If $z < q_R$, all the occurrences with rank $z$ are Marginal, and therefore there are $0 (l, r)$-extendable occurrences in $T_z$. Otherwise, we execute the following range queries on $Tlex$: 39
1. \( \text{count}_{\text{cent}} = \text{COUNT}([z + q_L + 2 \ldots \infty] \times \ast \times [\text{tr}(C_i) \ldots \text{tr}(C_j)]) \)

2. \( \text{count}_{\text{bord}L} = \text{COUNT}([z + q_L + 1 \ldots z + q_L + 1] \times [r_L \ldots \infty] \times \ast \times [\text{tr}(C_i) \ldots \text{tr}(C_j)]) \)

3. \( \text{count}_{\text{bord}R} = \text{COUNT}([q_R + q_L + 2 \ldots \infty] \times \ast \times [r_R \ldots \infty] \times [\text{tr}(C_i) \ldots \text{tr}(C_j)]) \)

4. \( \text{count}_{\text{bord}M} = \text{COUNT}([q_R + q_L + 1 \ldots q_R + q_L + 1] \times [r_L \ldots \infty] \times [r_R \ldots \infty] \times [\text{tr}(C_i) \ldots \text{tr}(C_j)]) \)

In all the above queries, the constraint on the last coordinate ensures that we are only counting points that correspond to clusters \( C \) with \( \text{Tail}(C_i) \leq \text{Tail}(C) < \text{Tail}(C_j) \). According to Fact 56, we are interested in counting the clusters with this property, that also has an \((l, r)\)-extendable occurrence with degree \( z \).

If \( z = q_R \), every occurrence in \( T_i \) is a borderline occurrence. For arguments similar to the ones used in the evaluation of \( \Sigma_{\text{bord}} \), a point \( p_C \) is counted in \( \text{count}_{\text{bord}M} \) iff \( C \) has an \((l, r)\)-extendable occurrences with rank \( q_R \) and \( |C| \leq q_R + q_L + 2 \). Similarly, \( p_C \) is counted in \( \text{count}_{\text{bord}L} \) iff \( C \) has an \((l, r)\)-extendable occurrence with rank \( q_R \) and \( |C| = q_R + q_L + 1 \). Overall, there are \( \text{count}_{\text{bord}L} + \text{count}_{\text{bord}M} \) \((l, r)\)-extendable occurrences in \( T \).

We check if \( z = q_R \) and return \( \text{count}_{\text{bord}M} \) accordingly. We used a constant number of \( 4 \)-dimensional range queries, so the time complexity is \( O(\log^3(n)) \). We also need to obtain \( tr(C_i) \) and \( tr(C_j) \). We can use Theorem 78 to obtain \( C_i \) and \( C_j \) in \( O(\log^2(n)) \) time.

We store the clusters such that every cluster is associated with the auxiliary information \( tr(C) \).

We conclude this subsection with the following:

**Theorem 91.** The periodic occurrences representation \( \text{POR}_w \) of a word \( w = S[s \ldots e] \) with \( \text{per}(w) = p \) can be preprocessed in time \( O(|\text{POR}_w|) \) to answer Expression Restricting Range Counting queries of the form \( \{l, r, i, j\} \) in \( O(\log^4(n)) \) time, provided that \( \text{per}(S[s-l \ldots e+r]) = p \). The generated data structure uses \( \tilde{O}(|\text{POR}_w|) \) space.

**Proof:** We preprocess \( \text{POR}_w \) for \( A_w \) lookup queries by employing Theorem 78. We also construct \( T_{\text{lex}} \).

To process a query \( \{l, r, i, j\} \) with \( \text{per}(S[s-l \ldots e+r]) = p \), we use the data structure of Theorem 78 to obtain \( T_{\text{lex}}, T_{\text{r}}, i_{\text{lex}}, j_{\text{lex}} \), and the clusters \( C_i \) and \( C_j \) containing \( i \) and \( j \) respectively. We also evaluate \( q_L, q_R, r_L \) and \( r_R \). We now have all the data required to execute the range queries on \( T_{\text{lex}} \) to evaluate \( \Sigma_{\text{bord}}, \Sigma_T \) and \( \Sigma_{\text{Cent}} \) by employing lemmas 57, 58, 59. It can be easily verified that the sets of \((l, r)\)-extendable occurrences counted in \( \Sigma_{\text{bord}}, \Sigma_T \) and \( \Sigma_{\text{Cent}} \) are disjoint. We output the sum \( \Sigma_{\text{bord}} + \Sigma_T + \Sigma_{\text{Cent}} \).

**Preprocessing time and space:** The construction of the data structure for \( A_w \) look-ups takes \( \tilde{O}(\text{|POR}_w) \) time and space by Theorem 78. The arguments required to construct the points of \( T_{\text{lex}} \) can be evaluated by calculating \( |C|, r_L(C) \) and \( r_R(C) \) in \( O(1) \) for every cluster \( |C| \), and sorting the tails to get \( tr(C) \) for every cluster \( C \). We then insert every one of the \( |\text{POR}_w| \) points to \( T_{\text{lex}} \) in \( O(\log^4(n)) \). The overall time is \( O(|\text{POR}_w|) \), and so is the space of the generated data structure.

**Query time:** The query consists of two lookup queries in \( A_w \), a constant number of arithmetic operations to evaluate \( q_R, q_L, r_R \) and \( r_L \), and a single applications of each of lemmas 57, 58, 59. The time is bounded by \( O(\log^4(n)) \).

### 7.2.2 Case 2: An A-periodic Extended Word

In this subsection, we examine the case in which \( w' = S[s-l \ldots e+r] \) does not have a period \( \text{per}(w) = p \). We show that in this case, every cluster contributes at most a single \((l, r)\)-extendable occurrence to \( A_w[i \ldots j] \). Let \( R = S[s-E_i \ldots e + E_e] \) be the run with period \( p \) containing \( w \). Let \( q_L, q_R, r_R, r_L \) defined as in the previous section, and similarly let \( q_{E_i}, q_{E_e}, r_{E_i}, r_{E_e}, < p \) and \( r_{E_e} < p \) be non negative integers such that \( E_i = q_{E_i} \cdot p + r_{E_i} \) and \( E_e = q_{E_e} \cdot p + r_{E_e} \).
Note that if \( w' \) does not have a period \( p \), it must be the case that either \( r > E_r \) or \( l > E_l \). Let \( C_w \) be the periodic cluster containing \( w \). The following facts are the key for efficiently executing Restricted Range Counting Queries:

**Fact 92.** If \( l > E_l \) and \( r < E_r \), only an occurrence \( w_t \) implied by cluster \( C \) with period rank \( r_p(w_t) = |C| - q_{E_l} - 1 \) can be \((l, r)-extendable\). This \( w_t \) is \((l, r)-extendable\) iff one of these conditions hold:

1. \( \text{LCS}(\text{Head}(C), \text{Head}(C_w)) \geq l - p \cdot q_{E_l} \) and \( |C| \geq q_{E_l} + q_R + 2 \)
2. \( \text{LCS}(\text{Head}(C), \text{Head}(C_w)) \geq l - p \cdot q_{E_l} \) and \( |C| = q_{E_l} + q_R + 1 \) and \( r_R(C) \geq r_R \)

**Fact 93.** If \( l \leq E_l \) and \( r > E_r \), only an occurrence \( w_t \) implied by cluster \( C \) with period rank \( r_p(w_t) = q_{E_r} \) can be \((l, r)-extendable\). This \( w_t \) is \((l, r)-extendable\) iff one of these conditions hold:

1. \( \text{LCP}(\text{Tail}(C), \text{Tail}(C_w)) \geq r - p \cdot q_{E_r} \) and \( |C| \geq q_{E_r} + q_R + 2 \)
2. \( \text{LCP}(\text{Tail}(C), \text{Tail}(C_w)) \geq r - p \cdot q_{E_r} \) and \( |C| = q_{E_r} + q_R + 1 \) and \( r_L(C) \geq r_L \)

**Fact 94.** If \( l > E_l \) and \( r > E_r \), only an occurrence \( w_t \) implied by cluster \( C \) with period rank \( r_p(w_t) = q_{E_r} \) can be \((l, r)-extendable\). This \( w_t \) is \((l, r)-extendable\) iff \(|C| = q_{E_r} + q_R + 1\), \( \text{LCP}(\text{Tail}(C), \text{Tail}(C_w)) \geq r - p \cdot q_{E_r} \) and \( \text{LCS}(\text{Head}(C), \text{Head}(C_w)) \geq l - p \cdot q_{E_l} \)

![Diagram of facts 92-94](image)

The intuition behind facts 92-94 is the following. An occurrence \( w_0 = S[i_0 \ldots j_0] \) of \( w \) has period \( p \). For \( w_0 \) to be \((l, r)-extendable\), it must agree with \( w' \) on the extension of the period \( p \) around \( w_0 \) (at least within \( w_0 = S[i_0 - l \ldots j_0 + r] \)). Furthermore, there must be a sufficiently long agreement between the non-periodic 'left overs'. For an illustration, see Figure 12. We provide a proof for Fact 92, facts 93 and 94 can be proven by similar arguments.

**Proof:** Let \( w_t = S[s_t \ldots e_t] \) be an occurrence implied by \( C \), and let the run with period \( p \) containing \( C \) be \( R = S[s_C \ldots e_C] \). Recall that the size of the extension of \( R \) to the left of \( w_t \) is \((|C| - 1 - r_p(w_t)) \cdot p + r_L(C) \) (Claim 81).

We start by proving the following claim.

**Claim 95.** An \((l, r)-extendable\) occurrence \( w_t \) implied by \( C \) must have \( r_p(w_t) = |C| - q_{E_l} - 1 \).

**Proof:** If \( r_p(w_t) < |C| - q_{E_l} - 1 \), the extension of \( R \) to the left from \( w_t \) is at least \((q_{E_l} + 1) \cdot p > q_{E_l} \cdot p + r_{E_l} = E_l \) and therefore, (according to Lemma 36) \( \text{lcs}(s_t - 1, s - 1) = E_l < l \) so \( w_t \) is not \((l, r)-extendable\).

If \( r_p(w_t) > |C| - q_{E_l} - 1 \), the extension of \( R \) to the left from \( w_t \) is at most \((q_{E_l} - 1) \cdot p < q_{E_l} \cdot p + r_{E_l} = E_l \). Therefore, \( \text{lcs}(s - 1, s_t - 1) < E_l < l \) and \( w_t \) is not \((l, r)-extendable\). We get that for \( w_t \) to be \((l, r)\)extendable, it must have \( r_p(w_t) = |C| - q_{E_l} - 1 \).

We proceed with the following claim:
Claim 96. An occurrence \( w_t \) with \( r_p(w_t) = |C| - q_{E_t} - 1 \) has \( \text{lcs}(s-1, s_t-1) \geq 1 \) if \( \text{lcs}(\text{Head}(C_w), \text{Head}(C)) \geq l - p \cdot q_{E_t} \).

Proof: If \( r_p(w_t) = |C| - q_{E_t} - 1 \), \( w_t \) has a (not necessarily maximal) interval with period \( p \) of size \( q_{E_t} \cdot p \) to its left, proceeded by \( \text{Head}(C) \), \( w \) has the same structure - it has a periodic interval of size at least \( q_{E_t} \cdot p \) to its left proceeded by \( \text{Head}(C_w) \). Comparing from right to left, the suffixes \( S^{s-1} \) and \( S^{s_t-1} \) match within the periodic interval. Therefore we have \( S[s - p \cdot q_{E_t} \ldots s_t - 1] = S[s - p \cdot q_{E_t} \ldots s - 1] \). From this point, we are comparing between the heads of \( C \) and \( C_w \). So \( \text{lcs}(s - 1, s - 1) = \text{lcs}(\text{Head}(C_w), \text{Head}(C)) + p \cdot q_{E_t} \) and we need \( \text{lcs}(\text{Head}(C_w), \text{Head}(C)) \geq l - p \cdot q_{E_t} \) for \( w_t \) to have a sufficiently long agreement the left with \( w \).

Let \( w_t \) be an occurrence represented by a cluster \( C \) with \( \text{lcs}(\text{Head}(C_w), \text{Head}(C)) \geq l - p \cdot q_{E_t} \). According to the above claim, \( w_t \) is sufficiently extendable to the left to be an \((l, r)\)-extendable occurrence. We consider the following cases for the size of \( C \).

If \( |C| < q_{E_t} + q_r + 1 \), the extension of \( w_t \) with \( r_p(w_t) = |C| - q_{E_t} - 1 \) to the right is at most \( (q_r - 1) \cdot p < q_r + r_r = r \). Since \( E_r \geq r \), Lemma 56 yields \( \text{lcp}(e + 1, e_t + 1) < r \) and \( w_t \) is not \((l, r)\)-extendable.

If \( |C| \geq q_{E_t} + q_r + 2 \), the extension of \( R \) to the right of \( w_t \) with \( r_p(w_t) = |C| - q_{E_t} - 1 \) is at least \( (q_r + 1) \cdot p > q_r + r_r = r \). Since \( E_r \geq r \), the period \( p \) extends at least \( r \) indexes both to the right of \( w_t \) and to the right of \( w \). It follows that \( \text{lcp}(e + 1, e_t + 1) \geq r \) and \( w_t \) is sufficiently extendable from the right by making it an \((l, r)\)-extendable occurrence.

If \( |C| = q_{E_t} + q_r + 1 \), \( w_t \) has an extension of \( \text{Run e} \) of size exactly \( q_r \cdot p + r_p(C) \) to its right. Since \( E_c \geq r \), \( p \) extends at least \( r \) indexes to the right of \( w_t \) for it to be sufficiently extendable to the right. Therefore, in this case \( w_t \) is \((l, r)\)-extendable if \( r_p(C) \geq r_r \).

Equipped with facts 92-94, we are ready to formulate the problem as a set of range queries. We construct a data structure for 6-dimensional range queries \( T_{\text{lex}} \). For every cluster \( C \), we insert the point:

\[ p_C = (|C|, r_L(C), r_R(C), \text{lcs}(\text{Head}(C), \text{Head}(C_w)), \text{lcp}(\text{Tail}(C), \text{Tail}(C_w)), tr(C)) \]

More precisely, we will present our queries as if they are executed on the aforementioned 6-dimensional data structure. We will never use more than 4 non-trivial ranges ("\(*\)"") in a single query, so it is sufficient to store a set of 4-dimensional data structures, each containing a different subsets of the coordinates \( p_C \) that is required for a certain query.

Let our input Extension Restricting Range Query be \((l, r, i, j)\). We recall the notations of \( T_{r_i}, T_{r_j}, i_r, j_r \) and \( j_t \), such that \( T_{r_i} [i_r] = I[i] \) and \( T_{r_j} [j_r] = I[j] \), as well as \( C_l \) and \( C_j \) the clusters respectively implying the occurrences starting in \( I[i] \) and in \( I[j] \). We distinguish between three cases with each case corresponding to one of the facts 92-94.

**Case 1: \( l \geq E_t \) and \( r \leq E_r \).** We start by counting the \((l, r)\)-extendable occurrences in \( T_z \) for \( z \in [r_j + 1 \ldots r_t - 1] \). To do that, we execute the following range queries on \( T_{\text{lex}} \):

1. \( \text{count}_{mid1} = \text{COUNT}([\text{max}(q_{E_t} + r_j + 2 \ldots q_{E_t} + r_r) \ldots q_{E_t} + r_i] \times \ast \times \ast \times \times [l - q_{E_t} \cdot p] \ldots \infty] \times \ast \times \ast) \)
2. \( \text{count}_{mid2} = \text{COUNT}([q_{E_t} + q_r + 1 \ldots q_{E_t} + q_r] \times \ast \times [r_R \ldots \infty] \times [l - q_{E_t} \cdot p] \ldots \infty] \times \ast \times \ast) \)

We set \( \text{count}_{mid2} = 0 \) if \( q_r \notin [r_j + 1 \ldots r_t - 1] \).

**count_{mid1}** counts occurrences from clusters \( C \) with \( |C| \geq q_{E_t} + q_r + 2 \) and \( \text{LCS}(\text{Head}(C_w, C)) \geq l - q_{E_t} \cdot p \) - these are \((l, r)\)-extendable occurrences that correspond to the first condition in Fact 92.

The additional constraint on \( C \) ensures that \( |C| - q_{E_t} - 1 \notin [r_j + 1 \ldots r_t - 1] \). Therefore, \( \text{count}_{mid1} \) counts the amount of \((l, r)\)-extendable occurrences \( w_t \) corresponding to the first condition of Fact 92 with \( r_p(w_t) = |C| - q_{E_t} - 1 \in [r_j + 1 \ldots r_t - 1] \).

**count_{mid2}** counts \((l, r)\)-extendable occurrences that correspond to the second condition. Every \((l, r)\)-extendable occurrence \( w_t \) in this case has rank \( r_p(w_t) = |C| - q_{E_t} - 1 = q_r \). Since we nullify \( \text{count}_{mid2} \) if \( q_r \notin [r_j + 1 \ldots r_t - 1] \), we get that \( \text{count}_{mid2} \) counts the \((l, r)\)-extendable occurrences that corresponds to the second condition and are within the relevant interval of ranks.

Overall, \( \text{count}_{mid1} + \text{count}_{mid2} \) is the number of \((l, r)\)-extendable occurrences in \( T_z \) for \( z \in [r_j + 1 \ldots r_t - 1] \).

We are left with the tasks of counting the \((l, r)\)-extendable occurrences in the parts of \( T_{r_i} \) and \( T_{r_j} \) that are contained in \( I[i \ldots j] \). As in the periodic case, this is easily reducible to counting the \((l, r)\)-extendable occurrences within a given interval \( T_z[i_z \ldots j_z] \) of a sub-array \( T_z \).

As in the periodic case, we construct a range query to count the \((l, r)\)-extendable occurrences within an interval \( T_z[i_z \ldots j_z] \) in \( T_z \). Let \( C_l \) and \( C_j \) be the clusters implying \( T_z[i_z] \) and \( T_z[j_z] \), respectively. We execute the following range query on \( T_{\text{lex}} \):

42
1. \( \text{count}_1 = \text{COUNT}([z + q_L + 1 \ldots z + q_L + 1] \times \ldots \times [L - q_{E_L} \cdot p \ldots \infty] \times \ldots \times [\text{tr}(C_t) \ldots \text{tr}(C_j)]) \)

2. \( \text{count}_2 = \text{COUNT}([z + q_L + 1 \ldots z + q_L + 1] \times \ldots \times [\text{tr}(C_t) \ldots \text{tr}(C_j)]) \)

We proceed to make the following claims regarding \( \text{count}_1 \) and \( \text{count}_2 \).

**Claim 97.** A cluster \( C \) implies an \((l,r)\)-extendable occurrence in \( T_z \) iff \(|C| = q_{E_z} + z + 1 \) and it implies an \((l,r)\)-extendable occurrence.

**Proof:** We start by proving \( \rightarrow \). Let \( w_l \) be an \((l,r)\)-extendable occurrence in \( T_z \) implied by cluster \( C \). According to Lemma 92, \( w_l \) must have \( r_p(w_l) = |C| - q_{E_z} - 1 \). Since \( w_l \) is in \( T_z \), it must have \( r_p(w_l) = z \).

By transitivity, we get \(|C| - q_{E_z} - 1 = z \) which implies \(|C| = q_{E_z} + z + 1 \).

We proceed to prove \( \leftarrow \). Let \( w_l \) be an \((l,r)\)-extendable occurrence implied by \( C \), with \(|C| = q_{E_z} + z + 1 \).

Lemma 92 suggests that \( r_p(w_l) = z \).

**Claim 98.** If \( z \geq q_R + 1 \), The number of \((l,r)\)-extendable occurrences in \( T_z[i_z \ldots j_z] \) is \( \text{count}_1 \).

**Proof:** The constraint on the first coordinate ensures that the clusters counted by \( \text{count}_1 \) are with size \( q_{E_z} + z + 1 \). 

If \( q_{E_z} + z + 1 \geq q_E + q_R + 2 \), the first case of Lemma 92 applies and every cluster \( C \) with \(|C| = q_{E_z} + z + 1 \) contains an \((l,r)\)-extendable occurrence iff \( lcs(\text{Head}(C), \text{Head}(C_w)) \geq L - q_{E_L} \cdot p \) and \( r_R(C) \geq r_R(C_w) \). The constraint on the third and on the forth coordinates ensures that we only count clusters that contain an \((l,r)\)-extendable occurrence.

So far, we have shown that points satisfying the constraints on the first and on the 4th coordinates corresponds to the set of clusters with size \( q_{E_z} + z + 1 \) and containing an \((l,r)\)-extendable occurrence.

Claim 97 suggests that this is exactly the set of clusters that contains an \((l,r)\)-extendable occurrence in \( T_z \).

Finally, the constraint on \( \text{tr}(C) \) ensures that we only count clusters \( C \) with \( \text{tr}(C_t) \leq \text{tr}(C) \leq \text{tr}(C_j) \). Since the occurrences \( T_z \) are sorted by the lexicographic order of the tails of their clusters, that constraint ensures that we only count clusters with an \((l,r)\)-extendable occurrence in \( T_z[i_z \ldots j_z] \).

**Claim 99.** If \( z = q_R \), the number of \((l,r)\)-extendable occurrences in \( T_z[i_z \ldots j_z] \) is \( \text{count}_2 \).

**Proof:** The constraint on the first coordinate ensures that the clusters counted by \( \text{count}_1 \) are with size \( q_{E_z} + z + 1 \). 

If \( z = q_R \), the second case of Lemma 92 applies and every cluster \( C \) with \(|C| = q_{E_z} + z + 1 \) contains an \((l,r)\)-extendable occurrence with degree \( z \) iff \( lcs(\text{Head}(C), \text{Head}(C_w)) \geq L - q_{E_L} \cdot p \) and \( r_R(C) \geq r_R(C_w) \).

The constraints on the third and on the forth coordinates ensures that we only count clusters that imply an \((l,r)\)-extendable occurrence.

So far, we have shown that points satisfying the constraints on the first, the third and the 4th coordinates corresponds to the set of clusters with size \( q_{E_z} + z + 1 \) that contain an \((l,r)\)-extendable occurrence. Claim 97 suggests that this is exactly the set of clusters that contains an \((l,r)\)-extendable occurrence in \( T_z \).

As in the proof of Claim 98, the constraint on \( \text{tr}(C) \) ensures that we only count clusters with an \((l,r)\)-occurrence within the interval \( T_z[i_z \ldots j_z] \) in \( T_z \).

**Claim 100.** If \( z < q_R \), there are no \((l,r)\)-extendable occurrences in \( T_z[i_z \ldots j_z] \).

**Proof:** Assume to the contrary that \( T_z \) contains an \((l,r)\)-extendable occurrence \( w \) implied by a cluster \( C' \). Claim 97 suggests that that \( r_p(w_t) = |C'| - q_{E_z} - 1 \). Since \( w_t \) is in \( T_z \), we get \( z = |C'| - q_{E_z} - 1 \).

From our assumption that \( z < q_R \) we get \(|C'| < q_{E_z} + q_R + 1 \). This is a contradiction to Lemma 92 in which both cases require \(|C'| \geq q_{E_z} + q_R + 1 \).

With claims 98 [100], it is straightforward to construct an algorithm for counting \((l,r)\)-extendable occurrence within \( T_z[i_z \ldots j_z] \). We compare \( z \) to \( q_R \). If \( z \geq q_R + 1 \), we compute \( \text{count}_1 \) and return it (Claim 98). If \( z = q_R \) we compute \( \text{count}_2 \) and return it (Claim 99). If \( z < q_R \) we simply return 0 (Claim 100). We execute at most a single range query in the process of reporting the number of \((l,r)\)-extendable occurrences within \( T_z[i_z \ldots j_z] \). The time complexity is therefore \( O(\log^3(n)) \).

**Case 2:** \( l \leq E_t \) and \( r > E_r \). This case is corresponding to Lemma 99 and is handled symmetrically to case 1.
Case 3: \( l > E_l \) and \( r > E_r \). Again, we start by counting the \((l, r)\)-extendable occurrences in \( T_z \) for \( z \in [r_1 + 1 \ldots r_1 - 1] \). Note that in this case, according to Fact 14, the only acceptable rank for an \((l, r)\)-extendable occurrence is \( q_E \), and the only acceptable size for a cluster containing an \((l, r)\) extendable occurrence is \( q_E + q_E + 1 \). If \( q_E \in [r_1 + 1 \ldots r_1] \), we query:

\[
\text{count} = \text{COUNT}(q_E + q_E + 1 \ldots q_E + q_E + 1) \times \ldots \times \left[ l - q_E \cdot p \ldots \infty \right] \times \left[ r - q_E \cdot p \ldots \infty \right] \times \ldots
\]

According to Fact 14, these are all the \((l, r)\)-extendable occurrences of \( w \). Since \( T_qE \) is completely contained within \([i \ldots j]\), all of these occurrences are in \([i \ldots j]\) and we output \( \text{count} \).

If \( q_E \in [r_1, r_2] \), we have \( T_qE \) only partially contained in \([r_1 \ldots j]\). We need to count \((l, r)\)-extendable occurrences within a certain interval \( T_{qEz}[i_2 \ldots j_2] \) with \( C_{i_2} \) and \( C_{j_2} \) the clusters implying the occurrences starting in \( T_qE \) at \([i \ldots j] \) and in \( T_qE \) at \([j] \) respectively. We query:

\[
\text{count} = \text{COUNT}(q_E + q_E + 1 \ldots q_E + q_E + 1) \times \ldots \times \left[ l - q_E \cdot p \ldots \infty \right] \times \left[ r - q_E \cdot p \ldots \infty \right] \times \ldots
\]

\( \text{count} \) counts only the \((l, r)\)-extendable occurrences within \( T_qEz \) at \([i_2 \ldots j_2] \) since the occurrences in \( T_qEz \) are sorted by the lexicographic order of their clusters’ tails.

The following theorem concludes the result of this subsection:

**Theorem 101.** The periodic occurrences representation \( \text{POR}_w \) of a word \( w = S[s \ldots e] \) with \( \text{per}(w) = p \) can be preprocessed in time \( \tilde{O}(\|\text{POR}_w\|) \) to answer Extension Restricting Range Counting queries of the form \((l, r, i, j)\) in \( O(\log^4(n)) \) time, provided that \( \text{per}(S[s - l \ldots e + r]) \neq p \).

**Proof:** We preprocess \( \text{POR}_w \) for look-ups in \( A_w[i] \) using Theorem 78. We process every cluster \( C \) to obtain \( |C|, r_C(C), \tau_C(C), \text{lcs}(\text{Head}(C), \text{Head}(C_w)) \) and \( \text{lcp}(\text{Tail}(C), \text{Tail}(C_w)) \). We sort the tails of the clusters to obtain \( tr(C) \) for every cluster \( C \). Now we have the required data to construct the point \( PC \) for every cluster. We insert subsets of size at most \( 4 \) of the coordinates of \( PC \) to several 4-dimensional range queries data structures corresponding to the coordinates with non-trivial intervals in the queries described in this section. We also evaluate \( E_r \) and \( E_l \) the extensions of the run with period \( p \) to the right and to the left of \( w \), respectively.

Upon a query \((l, r, i, j)\), we classify the query into one of the 3 cases described in this section. We execute a look-up query to \( I[i] \) and \( I[j] \) to obtain \( T_{r_l}, T_{r_r}, i_r, j_r, C_i \) and \( C_j \). We now have all the information required to execute the range queries.

**Preprocessing time:** The time for constructing a data structure for \( I[i] \) look ups is \( \tilde{O}(\|\text{POR}_w\|) \). Every cluster in \( \text{POR}_w \) is processed with a constant amount of \( LCE \) queries and basic arithmetic operations, which are all polylogarithmic operations. Sorting the tails of the clusters takes \( \tilde{O}(\|\text{POR}_w\|) \). Finally, inserting all the (partial) points to 4-dimensional range queries data structures takes polylogarithmic time per point. The overall complexity is \( \tilde{O}(\|\text{POR}_w\|) \).

**Query time:** The query time consists of a constant amount of comparison operations to classify the query, two look-up queries for \( I \) and a constant amount of 4-dimensional range queries. The complexity is dominated by \( O(\log^4(n)) \).

Recall that a similar data structure needs to be constructed for the subarray \( D \) of \( A_w \) containing occurrences from decreasing clusters.

Putting it together with Theorem 91, we get the following:

**Theorem 102.** The periodic occurrences representation \( \text{POR}_w \) of a word \( w = S[s \ldots e] \) can be preprocessed in time \( \tilde{O}(\|\text{POR}_w\|) \) to answer Extension Restricting Range Counting queries in \( O(\log^4(n)) \) time.

**Proof:** given a query \((l, r, i, j)\), we can check if \( l \leq E_l \) and \( r \leq E_r \). If both are true, we employ Theorem 91. Otherwise, we employ Theorem 101. The classification test takes additional constant time.

We can use the above result to finally prove Theorem 28.

**Proof:** We preprocess \( \text{POR}_w \) for \( A_w \) look-up queries and for Extension Restricting Range Counting queries with theorems 78 and 102.

Upon a query for \( A_w[l \ldots r][i] \), we execute a binary search for the index \( i' \) such that \( A_w[i'] = A_w[l \ldots r][i] \) as follows: start by executing the Extension Restricting Range Counting query \((l, r, 1, [\frac{A_w}{4}], L)\). Denote the output of the query as \( L \). If \( L \geq i \) we deduce that \( A_w[i'] = A_w[l \ldots r][i] \) is in \( A_w[1 \ldots \frac{A_w}{4}] \). Otherwise, we deduce that \( A_w[i'] \) is in \( A_w[\frac{A_w}{4} + 1 \ldots A_w] \). We shrink our search range and proceed recursively until our search interval is of size 1. The sole index in the final interval is \( i' \). We conclude by returning \( A_w[i'] \).

The preprocessing time and the space consumed by the data structure are both bounded by \( \tilde{O}(\|\text{POR}_w\|) \) directly from Theorem 78 and Theorem 102. The query time consists of \( \log(n) \) binary search iterations.
\(|A_w| \leq n\). In every iteration of the binary search we execute an Extensions Restricting Range Counting query costing \(O(\log^4(n))\) time each. We conclude with a single look-up query in \(A_w\). The time complexity is dominated by \(O(\log^4(n))\).

8 Maintaining Indexed Integers Under Batched Updates

In this section, we provide proofs for Claim 33 and Theorem 2.

8.1 Proof of Claim 33

Claim 33. There is a data structure for maintaining an indexed set of integers \(R\) to support both applying an interval increment update to \(R\) and look-up queries \(R[i]\) in time \(O(\log(n))\).

Claim 33 may be considered as folklore and can be solved via the following well known, simple construction. \(C\) can be represented as a full binary tree with \(n\) leaves and weighted edges. The \(k\)'th leaf from the right represents \(C[k]\) and the sum of the edges from the root to that leaf equals to \(C[k]\). Upon update \(i, j, x\), a set of \(O(\log(n))\) edges can be identified such that every leaf \(l\) in position \([i \ldots j]\) from the left has exactly one of these edges in the route from \(l\) to the root, and every other leaf does not have any of these edges in its route to the root. Therefore, increasing the weights of these edges by \(x\) will have the desired effect.

This set of edges can be identified by constructing the paths from \(i\) and from \(j\) to the root. The lowest common ancestor \(lca(i, j)\) of the leaves \(i\) and \(j\) is the lowest node to appear in both paths. For every node \(v\) in the sub-path from \(lca(i, j)\) to \(i\) (not including \(lca(i, j)\)), if the right edge emerging from \(v\) is not part of the path - add it to the set. For every node \(v\) in the path from \(lca(i, j)\) to \(j\) - if the left node emerging from \(v\) is not in the path - add it to the set. In both cases - include the edge entering the leaf in the set. It can be easily verified that this construction yields a set with the required property and can be found in \(O(\log(n))\) time.

8.2 Proof of Theorem 2

Theorem 2. There is a data structure for maintaining an indexed set of integers \(R\) to support both applying a stairs update to \(R\) and look-up queries \(R[i]\) in \(O(\log^3(U))\) time, provided that all the updates \((i, j, p)\) have the same value of \(p\). \(U\) is the overall amount of updates. The data structure consumes \(O(U \log^2(U))\) space. Initializing the data structure (all counters are set to 0) takes \(O(1)\) time.

We denote the set of maintained counters as \(C[1 \ldots n]\). We assume that all the counters are initially set to \(C[i] = 0\). Any other initialization of \(C\) can be easily supported by keeping the initial values in a static array \(A\) and a stairs updates data structure \(D\) with initialized values \(D[i] = 0\). Upon a lookup query for \(C[i]\), we report \(C[i] = A[i] + D[i]\).

Our data structure for restricted decreasing stairs updates works as follows: We maintain two range queries data structure. A 2-dimensional data structure \(O\) and a 3-dimensional data structure \(R\). Note that \(p\) is fixed among all the updates, so it does not need to be specified in the query.

Upon a query for applying a decreasing stairs update \((i, j)\) we insert the point \(p_R = (i, j)\) with the value \(v(p_R) = \left[\frac{x}{p}\right]\) to \(O\), and the point \(p_R = (i, j, (j \mod p))\) (with an arbitrary value assigned to \(v(p_R)\)) to \(R\). If \((j \mod p) = 0\) we set the third coordinate of \(p_R\) to \(p\) instead.

Upon a query for reporting the value of \(C[x]\), we evaluate the unique positive integers \(q_x\) and \(r_x\) such that \(x = q_x \cdot p + r_x\) with \(r_x < p\). We execute the following range queries:

1. Query \(O\) for \(S = \text{SUM}([1 \ldots x] \times [x \ldots n])\)
2. Query \(O\) for \(C = \text{COUNT}([1 \ldots x] \times [x \ldots n])\)
3. Query \(R\) for \(D = \text{COUNT}([1 \ldots x] \times [x \ldots n] \times [1 \ldots r_x - 1])\). If \(r_x = 0\), we set \(D = 0\)

We return \(S - C \cdot q_x - D\).

Lemma 103. \(C[x] = S - C \cdot q_x - D\)

Proof: We denote \(con_j(k)\) to be the value that a decreasing stairs update \((i, j, p)\) adds to index \(k\), assuming that \(k \in [i \ldots j]\). If \(k > j\), the value of \(con_j(k)\) is undefined and allowed to be an arbitrary number. Note that for \(k \in [i \ldots j]\), the value of \(con_j(k)\) is completely independent from \(i\). Therefore,
it can be equivalently defined as the value that the update \((1, j, p)\) adds to index \(k \leq j\). Consider an update \((i, j, p)\) that was applied to \(C\) before the query for the value \(C[x]\). Let \(r_{ij}, q_{ij}\) be the two unique non-negative integers such that \(j = q_{ij} \cdot p + r_{ij}\) and \(r_{ij} \in [1 \ldots p]\).

Note that if \(r_{ij} \leq r_{ij}, con_{ij}(r_{ij}) = \lceil \frac{x}{p} \rceil\), and otherwise, \(con_{ij}(r_{ij}) = \lceil \frac{x}{p} \rceil - 1\). That is due to the fact that the leftmost step in the update \((1, j, p)\) is the \(\lceil \frac{x}{p} \rceil\)th step and it is of size \(r_{ij}\). Another property of \(con_{ij}(k)\) is that for every two integers \(q, k\) we have \(con_{ij}(q \cdot p + k) = con_{ij}(k) - q\). Putting the above observations together, we get \(con_{ij}(x) = con_{ij}(q_{ij} \cdot p + r_{ij}) = con_{ij}(r_{ij}) - q_{ij}\). For \(r_{ij} \leq r_{ij}, \) that is \(con_{ij}(x) = \lceil \frac{x}{p} \rceil - q_{ij}\) and for \(r_{ij} > r_{ij}, \) that is \(con_{ij}(x) = \lceil \frac{x}{p} \rceil - q_{ij} - 1\).

Let \(U\) be the set of updates \((i, j, p)\) with \(x \in [i \ldots j]\) that were applied to \(C\) prior to the query. The definition of \(con_{ij}(x)\) directly implies that \(C[x] = \Sigma_{(i,j,p) \in U} con_{ij}(x)\). Let \(U_1 = \{(i, j, p) \in U \text{ and } r_{ij} \leq r_{ij}\}\) and \(U_2 = U \setminus U_1\). So we have

\[
C[x] = \Sigma_{(i,j,p) \in U_1} con_{ij}(x) + \Sigma_{(i,j,p) \in U_2} con_{ij}(x) = \Sigma_{(i,j,p) \in U_1} \lceil \frac{x}{p} \rceil - q_{ij}\]

\[
+ \Sigma_{(i,j,p) \in U_2} \lceil \frac{x}{p} \rceil - \Sigma_{(i,j,p) \in U_2} q_{ij} = \Sigma_{(i,j,p) \in U} \lceil \frac{x}{p} \rceil - |U| \cdot q_{ij} - |U_2|.
\]

One can easily confirm that \(S = \Sigma_{(i,j,p) \in U} \lceil \frac{x}{p} \rceil,\) \(C = |U|\) and \(D = |U_2|\).

With that, the proof of Theorem 8.2 is completed. Applying a decreasing stairs update and querying for the value of \(C[x]\) is done using a constant number of 2-dimensional and 3-dimensional range queries, so the time complexity for both query and update is \(O(\log^3(U))\). The space complexity is \(O(U \cdot \log^2(U))\) with \(U\) being the amount of updates. The initialization of the data structure requires initializing two empty dynamic range queries data structures.

### 8.3 Variants

The following symmetric variant of decreasing stairs updates can also be supported by a construction symmetric to ours:

**Definition 104**. An increasing stairs update is given as \(U = (i, j, p)\). Applying \(U\) to a set of indexed integers \(C[1 \ldots n]\) increases \(C[a]\) for \(a \in [i+(k-1) \cdot p \ldots \min(i+k \cdot p-1, j)]\) by \(k\) for every \(1 \leq k \leq \lceil \frac{j-i+1}{p} \rceil\).

Increasing stairs updates are equivalent to decreasing stairs updates with inverted indexes. Namely, let \(C'[1 \ldots n]\) be an array with \(C'[x] = C[n-x-1]\). Applying an increasing stairs update \(u = (i, j, p)\) to \(C[i \ldots j]\) is equivalent to applying a decreasing stairs update \((n-j-1, n-i-1, p)\) to \(C'\). Trivially, both types of queries can be supported by maintaining two data structures, \(I\) and \(D\) for increasing and decreasing stairs updates respectively, and reporting \(C[i] = D[i] + I[i]\).

One can also consider a negative stairs update.

**Definition 105**. An increasing (resp. decreasing) negative stairs update is given as \(U = (i, j, p)\). Applying \(U\) to a set of indexed integers \(C[1 \ldots n]\) decreases \(C[a]\) for \(a \in [i+(k-1) \cdot p \ldots \min(i+k \cdot p-1, j)]\) (resp. \(a \in [\max(j-k \cdot p+1, i) \ldots j-(k-1) \cdot p)]\) ) by \(k\) for every \(1 \leq k \leq \lceil \frac{j-i+1}{p} \rceil\).

In words, a negative stairs updates decreases the values of indexes on the \(k\)th step by \(k\) rather than increasing it by \(k\). This kind of updates can also be supported by maintaining two stairs data structures, \(P\) and \(N\), for stairs updates and negative stairs updates respectively. Positive updates are applied to \(P\), and negative updates are applied to \(N\) as if they were positive. Upon a query index \(i\), we output \(C[i] = P[i] - N[i]\).

### 8.4 Updates reduction

We say that a set of updates \(U = u_1, u_2 \ldots u_t\) can be reduced to another set of updates \(U'\) if for every array \(C\), applying all the updates in \(U\) on an array \(C\) will result in the same modified array \(C'\) as applying all the updates in \(U'\) on \(C\).

In some settings, we can reduce a sequence of interval increment updates to a small set of stairs updates and interval increment updates.

**Lemma 106**. Let \(p\) be an integer and \(U = u_1, u_2 \ldots u_{|U|-1}\) be a sequence of interval increment updates with \(u_1 = (i_1, j_1, 1)\). If both \(i_1\) and \(j_1\) are either fixed, or an arithmetic progression with difference \(p\), the sequence \(U\) can be reduced to a set \(U'\) of interval increment updates and stairs updates with stair width \(|p|\) such that \(|U'| \in O(1)\). \(U'\) can be evaluated in constant time.

**Proof**: we distinguish between the following cases:

1. Both \(i_1\) and \(j_1\) are fixed. In this case, \(U\) can be reduced to a single interval update \(u' = (i, j, |U|)\).
2. \( i_t = j_0 + t \cdot p \) with \( p > 0 \). In this case, \( U \) can be reduced to two updates \( U' = \{u', s\} \) with \( s \) being a decreasing stairs update \( s = (j_0, j_{|U|} - 1, p) \) and \( u' \) the interval increment update \( u' = (i, j_0, |U|) \). The case in which \( p < 0 \) can be handled the same way by considering \( j_t \) in a reversed order.

3. \( i_t = i_0 + t \cdot p \) with \( p > 0 \) and \( j_t \) is a constant. \( U \) can be reduced to two updates \( U' = \{u', s\} \) with \( s \) being an increasing stairs update \( s = (i_0, i_{|U|} - 1, p) \) and the interval update \( u' = (i_{|U|} - 1, j, |U|) \). The case where \( p < 0 \) can be handled the same way by considering \( i_t \) in a reversed order.

4. \( i_t = i_0 + t \cdot p \) and \( j_t = j_0 + t \cdot p \) be two arithmetic progressions with \( p > 0 \). \( U \) can be reduced to a set of 3 updates \( U' = \{s_1, s_2, u'\} \) with an increasing stairs update \( s_1 = (i_0, i_{|U|} - 1, p) \), an increasing negative stairs update \( s_2 = (j_0, j_{|U|} - 1, p) \) and an interval increment update \( u'_1 = (i_{|U|} - 1, j_{|U|} - 1, |U|) \).

It can be easily verified that in every case, applying the updates in \( U' \) to a set of indexed integers is equivalent to applying \( U \) to the same set. For a visualization, see Figures 13 and 14. In every case, the set \( U' \) contains at most three updates and can be evaluated in constant time in a straightforward way given the representation of \( i_t \) and \( j_t \) as arithmetic progressions (or fixed values, depending on the case).

![Figure 13](image13.png)

Figure 13: An illustration of how applying all the interval increment updates \( U \) with \( i_t \) being an arithmetic progression and a fixed \( j_t \) has the same effect as applying a stairs update (and a ‘reminder’ interval increment update)

![Figure 14](image14.png)

Figure 14: An illustration of how applying a sequence of interval updates with both ends being an arithmetic progression with difference \( p \) is equivalent to two stairs updates and an interval increment update. This set of updates can be viewed as a ‘sliding window’. The application of all the interval updates can be represented as an increasing stairs update with the stairs corresponding to the starting (left) indexes of the ‘sliding window’ and a negative increasing stairs update with the stairs corresponding to the ending indexes of the windows.

Acknowledgements

We warmly thank Eylon Yogev for assistance with the presentation and the organization of the results presented in this manuscript.
References

[1] Adjeroh, D., Bell, T., and Mukherjee, A. The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching. Springer-Verlag, 2008. ISBN 00387789081.

[2] Amir, A., Benson, G., and Farach, M. Let sleeping files lie: Pattern matching in z-compressed files. Journal of Computer and System Sciences 52, 2 (1996), 299–307.

[3] Amir, A., and Boneh, I. Dynamic palindrome detection. CoRR abs/1906.09732 (2019).

[4] Amir, A., and Boneh, I. Update query time trade-off for dynamic suffix arrays. In Proc. 31st International Symposium on Algorithms and Computation (ISAAC) (2020). to appear.

[5] Amir, A., Boneh, I., Charalampopoulos, P., and Kondratovsky, E. Repetition detection in a dynamic string. In Proc. 27th Annual European Symposium on Algorithms (ESA) (September 2019), LIPIcs, pp. 5:1–5:18.

[6] Amir, A., Boneh, I., Charalampopoulos, P., and Kondratovsky, E. Repetition Detection in a Dynamic String. In 27th Annual European Symposium on Algorithms (ESA 2019) (Dagstuhl, Germany, 2019), M. A. Bender, O. Svensson, and G. Herman, Eds., vol. 144 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 5:1–5:18.

[7] Apostolico, A. Fast gapped variants for lempel–ziv–welch compression. Information and Computation 205, 7 (2007), 1012–1026.

[8] Apostolico, A., Breslauer, D., and Galil, Z. Optimal parallel algorithms for periods, palindromes and squares. In Proc. 19th International Colloquium on Automata, Languages, and Programming (ICALP) (1992), vol. 623 of LNCS, Springer, pp. 296–307.

[9] Apostolico, A., Lewenstein, M., and Erdös, P. Parameterized matching with mismatches. Journal of Discrete Algorithms 5, 1 (2007), 135–140.

[10] Baker, B. S. Parameterized pattern matching: Algorithms and applications. Journal of Computer and System Sciences 52, 1 (1996), 28–42.

[11] Baker, B. S. Parameterized duplication in strings: Algorithms and an application to software maintenance. SIAM Journal on Computing 26, 5 (1997), 1343–1362.

[12] Balkenhol, B., and Kurtz, S. Universal data compression based on the burrows-wheeler transformation: Theorey and practice. IEEE Trans. on Computers 49, 10 (2000), 1043–1053.

[13] Benson, G. Tandem repeats finder: a program to analyze dna sequence. Nucleic Acids Research 27, 2 (1999), 573–580.

[14] Bentley, J. L., and Friedman, J. H. Data structures for range searching. ACM Computing Surveys (CSUR) 11, 4 (1979), 397–409.

[15] Charalampopoulos, P., Gawrychowski, P., and Pokorski, K. Dynamic longest common substring in polylogarithmic time. In Proc. 47th International Colloquium on Automata, Languages and Programming (ICALP) (2020), vol. 168 of LIPIcs, pp. 27:1–27:19.

[16] Charalampopoulos, P., Kociumaka, T., and Mozes, S. Dynamic String Alignment. In 31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020) (Dagstuhl, Germany, 2020), I. L. Gortz and O. Weimann, Eds., vol. 161 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 9:1–9:13.

[17] Charalampopoulos, P., Kociumaka, T., and Wellnitz, P. Faster approximate pattern matching: A unified approach, 2020.

[18] Domanić, N. O., and Preparata, F. P. A novel approach to the detection of genomic approximate tandem repeats in the levenshtein metric. Journal of Computational Biology 14, 7 (2007), 873–891.
[19] Farach, M., and Thorup, M. String matching in lempel-ziv compressed strings. *Proc. 27th Annual ACM Symposium on the Theory of Computing* (1995), 703–712.

[20] Fine, N. J., and Wilf, H. S. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society* 16, 1 (1965), 109–114.

[21] Fischer, J. Inducing the lcp-array, 2011.

[22] Funakoshi, M., Nakashima, Y., Inenaga, S., Bannai, H., and Takeda, M. Faster queries for longest substring palindrome after block edit. In *30th Combinatorial Pattern Matching conference CPM* (2019), vol. 128 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 27:1–27:13.

[23] Gawrychowski, P., and Janczewski, W. Fully dynamic approximation of LIS in polylogarithmic time. In *STOC ’21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021* (2021), S. Khuller and V. V. Williams, Eds., ACM, pp. 654–667.

[24] Gawrychowski, P., Karaczmarz, A., Kociumaka, T., Lacki, J., and Sankowski, P. Optimal dynamic strings. In *Proceedings, 29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA* (2018), pp. 1509–1528.

[25] Hazay, C., Lewenstein, M., and Sokol, D. Approximate parameterized matching. In *Proc. 12th Annual European Symposium on Algorithms (ESA 2004)* (2004), pp. 414–425.

[26] Holmberg, C., and Reutenauer, C. Lyndon words, permutations and trees. *Theoretical Computer Science* 307, 1 (2003), 173–178.

[27] Kaplan, H., Porat, E., and Shafrir, N. Finding the position of the k-mismatch and approximate tandem repeats. In *Scandinavian Workshop on Algorithm Theory (SWAT06)* (2006), pp. 90–101.

[28] Kociumaka, T., and Seddighin, S. Improved dynamic algorithms for longest increasing subsequence, 2021.

[29] Kucherov, G., and Sokol, D. Approximate tandem repeats. In *Encyclopedia of Algorithms*. 2016, pp. 106–109.

[30] Landau, G., and Schmidt, J. An algorithm for approximate tandem repeats. In *Proc. 4th Annual Symp. on Combinatorial Pattern Matching, Lecture Notes in Computer Science*, vol. 648. Springer-Verlag, 1993, pp. 120–133.

[31] Landau, G. M., Schmidt, J. P., and Sokol, D. An algorithm for approximate tandem repeats. *Journal of Computational Biology* 8, 1 (2001), 1–18.

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

[33] Landau, G. M., and Vishkin, U. Fast parallel and serial approximate string matching. *Journal of Algorithms* 10, 2 (1989), 157–169.

[34] Lee, T., Na, J., and Park, K. On-line construction of parameterized suffix trees. *Information Processing Letters* 111 (2011), 201–207.

[35] Loving, J., Scaduto, J. P., and Benson, G. An SIMD algorithm for wraparound tandem alignment. In *Proc. 13th International Symposium on Bioinformatics Research and Applications (ISBRA)* (2017), pp. 140–149.

[36] Manber, U., and Myers, G. Suffix array: A new method for on-line string searches. In *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms (SODA)* (1990), pp. 319–327.

[37] Navarro, G., and Raffinot, M. A general practical approach to pattern matching over ziv-lempel compressed text. In *Proc. 10th Annual Symposium on Combinatorial Pattern Matching (CPM 99)* (1999), pp. 14–36. LNCS 1645.

[38] Nishimoto, T., I, T., Inenaga, S., Bannai, H., and Takeda, M. Fully Dynamic Data Structure for LCE Queries in Compressed Space. In *41st Intl. Symp. Mathematical Foundations of Computer Science (MFCS)* (2016), vol. 58 of LIPIcs, pp. 72:1–72:15.
[39] Nong, G., Zhang, S., and Chan, W. H. Linear suffix array construction by almost pure induced-sorting. In 2009 Data Compression Conference (2009), pp. 193–202.

[40] Sadakane, K. A fast algorithm for making suffix arrays and for burrows-wheeler transformation. In Proc. Data Compression Conference (DCC) (1998), pp. 129–138.

[41] Sokol, D., Benson, G., and Tojeira, J. Tandem repeats over the edit distance. Bioinformatics 23, 2 (2007), 30–35.

[42] Weiner, P. Linear pattern matching algorithm. Proc. 14 IEEE Symposium on Switching and Automata Theory (1973), 1–11.

[43] Welch, T. A. A technique for high-performance data compression. IEEE Computer 17 (1984), 8–19.

[44] Wexler, Y., Yakhini, Z., Kashi, Y., and Geiger, D. Finding approximate tandem repeats in genomic sequences. Journal of Computational Biology 12, 7 (2005), 928–942.

[45] Ziv, J., and Lempel, A. A universal algorithm for sequential data compression. IEEE Trans. on Information Theory IT-23, 3 (1977), 337–343.

[46] Ziv, J., and Lempel, A. Compression of individual sequences via variable rate coding. IEEE Trans. on Information Theory IT-24 (1978), 530–536.