# Near-Optimal Quantum Algorithms for Bounded Edit Distance and Lempel–Ziv Factorization*

Daniel Gibney† Ce Jin‡ Tomasz Kociumaka§ Sharma V. Thankachan¶

**Abstract**

Measuring sequence similarity and compressing texts are among the most fundamental tasks in string algorithms. In this work, we develop near-optimal *quantum* algorithms for the central problems in these two areas: computing the edit distance of two strings [Levenshtein, 1965] and building the Lempel–Ziv factorization of a string [Ziv & Lempel, 1977], respectively.

Classically, the edit distance of two length-$n$ strings can be computed in $\mathcal{O}(n^2)$ time and there is little hope for a significantly faster algorithm: an $\mathcal{O}(n^{2-\epsilon})$-time procedure would falsify the Strong Exponential Time Hypothesis. Quantum computers might circumvent this lower bound, but even 3-approximation of edit distance is not known to admit an $\mathcal{O}(n^{2-\epsilon})$-time quantum algorithm. In the *bounded* setting, where the complexity is parameterized by the value $k$ of the edit distance, there is an $\mathcal{O}(n+k^2)$-time classical algorithm [Myers, 1986; Landau & Vishkin, 1988], which is optimal (up to sub-polynomial factors and conditioned on SETH) as a function of $n$ and $k$. Our first main contribution is a quantum $\widetilde{\mathcal{O}}(\sqrt{nk}+k^2)$-time algorithm that uses $\widetilde{\mathcal{O}}(\sqrt{nk})$ queries, where the $\widetilde{\mathcal{O}}(\cdot)$ notation hides polylogarithmic factors. This query complexity is unconditionally optimal, and any significant improvement in the time complexity would break the quadratic barrier for the unbounded setting. Interestingly, our divide-and-conquer quantum algorithm reduces the bounded edit distance problem to the special case where the two input strings have small Lempel–Ziv factorizations. Then, it combines our quantum LZ compression algorithm with a classical subroutine computing edit distance between compressed strings. The LZ factorization problem can be classically solved in $\mathcal{O}(n)$ time, which is unconditionally optimal in the quantum setting (even for computing just the size $z$ of the factorization). We can, however, hope for a quantum speedup if we parameterize the complexity in terms of $z$. Already a generic oracle identification algorithm [Kothari 2014] yields the optimal query complexity of $\widetilde{\mathcal{O}}(\sqrt{nz})$ at the price of exponential running time. Our second main contribution is a quantum algorithm that also achieves the optimal time complexity of $\widetilde{\mathcal{O}}(\sqrt{nz})$. The key insight is the introduction of a novel LZ-like factorization of size $\mathcal{O}(z\log^2 n)$, which allows us to efficiently compute each new factor through a combination of classical and quantum algorithmic techniques. From this, we obtain the desired LZ factorization. Using existing results [Kempa & Kociumaka, 2020], we can then obtain the string's run-length encoded Burrows-Wheeler Transform (BWT)—another classical compressor [Burrows & Wheeler, 1994], and a structure for longest common extensions (LCE) queries in $\mathcal{O}(z)$ extra time [I, 2017; Nishimoto et al., 2016].

Lastly, we obtain efficient indexes of size $\widetilde{\mathcal{O}}(z)$ for counting and reporting the occurrences of a given pattern and for supporting more general suffix array and inverse suffix array queries, based on the recent *r-index* [Gagie, Navarro, and Prezza, 2020]. These indexes can be constructed in $\widetilde{\mathcal{O}}(\sqrt{nz})$ quantum time, which allows us to solve many fundamental problems, like longest common substring, maximal unique matches, and Lyndon factorization, in time $\widetilde{\mathcal{O}}(\sqrt{nz})$.

## 1 Introduction

String processing constitutes one of the oldest fields within theoretical computer science. Fifty years after the discovery of some of its foundations, such as the suffix tree [Wei73] and the linear-time exact pattern matching algorithm [MP70], it remains a lively research area. Developments have been motivated both by the practical challenges of handling the rapidly growing volume of sequential data, especially in bioinformatics and data compression, and by the theoretical interest in demanding open questions.

More recently, the rapid progress in quantum computing brought increased attention to the development of quantum algorithms for fundamental string processing problems. As a precursor of this line of work, a paper of Hariharan and Vinay [HV03] demonstrated that a clever application of Grover search [Gro96] yields

*The full version of the paper can be accessed at `https://arxiv.org/abs/2311.01793`

†University of Texas at Dallas, Richardson, TX, U.S.; daniel.j.gibney@gmail.com

‡Massachusetts Institute of Technology, Cambridge, MA, U.S.; cejin@mit.edu

§Max Planck Institute for Informatics, SIC, Saarbrücken, Germany; tomasz.kociumaka@mpi-inf.mpg.de

¶North Carolina State University, Raleigh, NC, U.S.; sharma.thankachan@gmail.com

an $\widetilde{\mathcal{O}}(\sqrt{n})$-time[1] quantum algorithm for exact pattern matching within a length-$n$ text; this time complexity is unconditionally optimal. In the last few years, a series of works [GS22, WY20, AJ22, JN23] resulted in a nearly-optimal quantum algorithm for the Longest Common Factor problem (also known as Longest Common Substring, it was the original motivation for the suffix trees) as well as other classic problems such as Lexicographically Minimal String Rotation; see also [BEG+21, AGS19, ABI+20, CKK+22] for quantum algorithms for some other string problems. In this work, we develop quantum algorithms for two fundamental problems in string processing—Edit Distance and Lempel–Ziv (LZ77 [ZL77]) Factorization—which are the central computational tasks in string similarity and text compression, respectively.

The edit distance (also known as the Levenshtein distance [Lev65]) between two strings is defined as the minimum number of character insertions, deletions, and substitutions (collectively called *edits*) needed to transform one string into the other. Along with the Hamming distance (which allows substitutions only), it constitutes one of the two main measures of sequence (dis)similarity. The edit distance of two strings of length at most $n$ can be classically computed in $\mathcal{O}(n^2)$ time using a textbook dynamic-programming algorithm [Vin68, NW70, Sel74, WF74]. One of the celebrated results of fine-grained complexity [BI18] is that any truly-subquadratic-time algorithm (working in $\mathcal{O}(n^{2-\epsilon})$ time for some constant $\epsilon > 0$) would violate the Strong Exponential Time Hypothesis (SETH) [IP01]. A quantum counterpart of SETH only excludes $\mathcal{O}(n^{1.5-\epsilon})$-time quantum algorithms [BPS21], but no quantum speed-up is known for edit distance, and bridging the $\Omega(n^{1.5-\epsilon})$ lower bound with the $\mathcal{O}(n^2)$ upper bound remains a major open question [Rub19].

One of the earliest ways to circumvent the quadratic complexity of Edit Distance is to parameterize the running time in terms of the value $k$ of the edit distance. A series of works from 1980s [Ukk85, Mye86, LV88] resulted in an $\mathcal{O}(n + k^2)$-time classical algorithm for this *Bounded* Edit Distance problem. This running time is optimal, up to subpolynomial factors and conditioned on SETH, as a function of $n$ and $k$. More precisely, if the $\mathcal{O}(k^2)$ term can be reduced to $\mathcal{O}(k^{2-\epsilon})$ for some $k = \Theta(n^\kappa)$, with $\frac{1}{2} < \kappa \leq 1$, then a straightforward reduction translates this to an $\mathcal{O}(n^{2-\epsilon})$-time algorithm for unbounded Edit Distance. The Bounded Edit Distance problem has been extensively studied: there are efficient sketching & streaming algorithms [BZ16, JNW21, KPS21, BK23], algorithms for compressed strings [GKLS22], approximation algorithms [GKS19, KS20, BCFN22a], and algorithms for preprocessed data [GRS20, BCFN22b], to name just a few settings. Many of these results are based on the general scheme of [LV88], whose central component is an efficient implementation of longest common extension (LCE) queries[2]. Thus, when Jin and Nogler [JN23] obtained the optimal quantum trade-off for LCE queries, they asked whether their result could be applied to Bounded Edit Distance. The best one could reasonably hope for would be $\widetilde{\mathcal{O}}(\sqrt{kn})$ query complexity and $\widetilde{\mathcal{O}}(\sqrt{kn} + k^2)$ time complexity. This is because already computing the number of 1s in a length-$n$ binary string (which is its edit distance with $0^n$) requires $\Omega(\sqrt{kn})$ queries [Pat92, BBC+01] and, through the aforementioned reduction, any improvement upon the $\widetilde{\mathcal{O}}(k^2)$ term (whenever it dominates, that is, for $k = \Theta(n^\kappa)$ with $\frac{1}{3} < \kappa \leq 1$) would improve upon the $\mathcal{O}(n^2)$-time quantum algorithm for unbounded Edit Distance. The first main contribution of this work is a quantum algorithm with the desired query and time complexities:

THEOREM 1.1. (BOUNDED EDIT DISTANCE) *There is a quantum algorithm that, given quantum oracle access to strings $X, Y \in \Sigma^{\leq n}$, computes their edit distance $k := \mathsf{ed}(X, Y)$, along with a sequence of $k$ edits transforming $X$ into $Y$, in query complexity $\widetilde{\mathcal{O}}(\sqrt{n + nk})$ and time complexity $\widetilde{\mathcal{O}}(\sqrt{n + nk} + k^2)$.*

Surprisingly, our algorithm uses neither quantum LCE queries nor the underlying technique of quantum string synchronizing sets [JN23]; that approach seems to get stuck at the query complexity of $\tilde{\Theta}(k\sqrt{n})$. Instead of an LCE-based dynamic-programming procedure, we design a novel divide-and-conquer algorithm that crucially uses *compressibility*. If the input strings are compressible, we can retrieve their compressed representations and then solve the problem classically (the folklore implementation combines the Landau–Vishkin algorithm [LV88] with LCE queries on compressed strings [I17]; see [GKLS22]). Otherwise, we exploit the locality of edit distance: in order to optimally align a given character, it suffices to compute the edit distance locally, on the largest *compressible* context of that character (reusing the procedure mentioned above). Once we fix the alignment of a single character, the instance naturally decomposes into two *independent* sub-instances asking to compute the edit distance to the left and the right of the aligned characters. In the aforementioned statements, the *compressibility*

[1] The $\widetilde{\mathcal{O}}(\cdot)$ notations hides factors polylogarithmic in the input size $n$. In particular $\widetilde{\mathcal{O}}(1) = \mathcal{O}(\log^{\mathcal{O}(1)} n)$.
[2] The LCE of two positions is (the length of) the longest substring that occurs at both positions.

is quantified in terms of an upper bound $\widetilde{\mathcal{O}}(k)$ on the size of the LZ77 factorization. Concurrently to this work, a similar divide-and-conquer strategy has been applied in a classical algorithm for *weighted* bounded edit distance [CKW23], where the LCE-based DP is incorrect [DGH+23]. Prior to that, compression has been used for computing edit distance only in the sketching algorithms of [KPS21, BK23], where it comes more naturally because the sketches need to squeeze the input strings into $\widetilde{\mathcal{O}}(k^2)$ bits each so that their edit distance can be recovered.

The efficiency of our edit-distance algorithm thus depends on the construction of the LZ77 factorization [ZL77]. The factorization is defined through a left-to-right scan of the text such that each new factor is either the leftmost occurrence of a symbol or an occurrence of the longest substring that also occurs earlier in the text; see Section 2 for a formal definition and an example.

Finding the LZ factorization of a text is a fundamental problem on its own. In particular, it forms the basis of many practical compression algorithms, such as `zip`, `p7zip`, `gzip`, and `arj`. This problem admits a classic linear-time solution [RPE81], and it has been considered in a variety of other settings, including the external memory setting [KKP14], the dynamic setting [NII+20], and the packed setting [BP16, MNN17]. Our second main contribution is a novel quantum algorithm for LZ factorization, whose time complexity is optimal up to logarithmic factors.

THEOREM 1.2. (LZ77 FACTORIZATION) *There is a quantum algorithm that, given a quantum oracle access to an unknown string $X \in \Sigma^n$, computes the LZ factorization $\mathsf{LZ}(X)$ using $\widetilde{\mathcal{O}}(\sqrt{zn})$ query and time complexity, where $z := |\mathsf{LZ}(X)|$ is the size of the factorization.*

The size $z$ of the LZ77 factorization is known to be within a polylogarithmic factor away from numerous other compressibility measures [Nav21]. This includes the sizes of the smallest context-free grammar [CLL+05], the smallest bidirectional macro scheme [SS82], and the smallest string attractor [KP18], all of which are, however, NP-hard to compute exactly. Recently, the size $r$ of the run-length-encoded Burrows–Wheeler transform (BWT) [BW94] has also been shown to satisfy $r = \mathcal{O}(z \log^2 n)$ [KK22] and $r = \Omega(z/\log n)$ [GNP18a]. This measure is classically computable in linear time, and the underlying compressed string representation constitutes the basis of the practical `bzip2` algorithm. The run-length-encoded BWT can be constructed in $\widetilde{\mathcal{O}}(r)$ time from the LZ77 factorization [KK22], so Theorem 1.2 immediately yields the following important corollary:

COROLLARY 1.3. (RUN-LENGTH-ENCODED BWT) *There is a quantum algorithm that, given a quantum oracle to an unknown string $X \in \Sigma^n$, computes the run-length-encoded Burrows–Wheeler transform of $X$ using $\widetilde{\mathcal{O}}(\sqrt{rn})$ query and time complexity, where $r$ is the number of runs in the BWT.*

For most applications, compressing the data is only half of the battle. We also need to be able to perform computation over this data quickly, which is the motivation behind compressed text indexing. Traditional indexes such as suffix arrays and suffix trees require linear-time preprocessing and, once constructed, occupy linear space. A major achievement in compressed text indexing within the last two decades was the development of space-efficient representations of suffix trees/arrays in space close to "optimal" in terms of (higher-order) statistical entropy [FM05, GV05, NM07, Sad07]. A recent breakthrough by Gagie, Navarro, and Prezza [GNP18b], known as the *r-index*, takes $\mathcal{O}(r)$ space and can answer pattern-matching queries (both counting and reporting the occurrences) in near-optimal time; also see the improvements in [NT21, NKT22]. Its $\mathcal{O}(r \log \frac{n}{r})$-space version can support more general operations such as suffix array and inverse suffix array queries in $\mathcal{O}(\log \frac{n}{r})$ time [GNP20]. We show how to construct these indexes fast, as specified in the following result.

THEOREM 1.4. (COMPRESSED INDEX) *There is an $\widetilde{\mathcal{O}}(\sqrt{rn})$-time quantum algorithm that, given a quantum oracle to an unknown string $X \in \Sigma^n$, constructs*

- *an $\mathcal{O}(r)$-space index that can count the occurrences of any length-$m$ pattern in $\widetilde{\mathcal{O}}(m)$ time and report these occurrences in time $\widetilde{\mathcal{O}}(m + \mathsf{occ})$, where $\mathsf{occ}$ is the number of occurrences;*
- *an $\widetilde{\mathcal{O}}(r)$-space index for suffix array and inverse suffix array queries in $\widetilde{\mathcal{O}}(1)$ time.*

To handle LCE queries, we can use the $\mathcal{O}(z \log \frac{n}{z})$ space data structure with query time $\mathcal{O}(\log n)$ by I [I17], which can be constructed from LZ factorization in $\widetilde{\mathcal{O}}(z)$ time. This structure, combined with Theorem 1.4, enables us to solve numerous other classical string problems. A few examples provided in this work include:

- Finding the longest common substring between two strings of total length $n$ in $\widetilde{\mathcal{O}}(\sqrt{zn})$ time, where $z$ is the number of factors in the LZ77 parse of their concatenation. For highly compressible strings, this beats the best known $\widetilde{\mathcal{O}}(n^{2/3})$ time quantum algorithm [AJ22, GS22]. Similar time bounds can be obtained for finding the set of maximal unique matches (MUMs); the longest repeating substring/shortest unique substring of a given string.
- Obtaining the Lyndon factorization of a string in $\widetilde{\mathcal{O}}(\sqrt{\ell n})$ time, where $\ell = \widetilde{\mathcal{O}}(z)$ is the number of its Lyndon factors.
- Determining the frequencies of all distinct substrings of length $q$ ($q$-grams) in time $\widetilde{\mathcal{O}}(\sqrt{zn} + d_q)$, where $d_q = \mathcal{O}(zq)$ is the number of distinct $q$-grams.

## 2 Preliminaries

A string is a finite sequence of characters from the alphabet $\Sigma$, which we assume to be of the form $[0\,..\,\sigma)$ for an integer $\sigma = n^{\mathcal{O}(1)}$, where $n$ is the input size. We denote the length of a string $X$ as $|X|$. For any $i \in [1\,..\,|X|]$, the $i$th character of $X$ is $X[i]$. For $0 \le i \le j \le |X|$, a string of the form $X[i+1]\cdots X[j-1]X[j]$ is a *substring* of $X$. Its *occurrence* in $X$ ending at position $j$ is called a *fragment* of $X$ and denoted with $X(i\,..\,j]$; this fragment can also be referred to as $X[i+1\,..\,j]$, $X[i+1\,..\,j+1)$, or $X(i\,..\,j+1)$. Prefixes and suffixes are fragments of the form $X[1\,..\,i]$ and $X(i\,..\,|X|]$, respectively. The string $X[|X|]\cdots X[2]X[1]$, called the *reverse* of $X$, is denoted by $\overline{X}$.

**Quantum algorithms** We assume the input string $X \in \Sigma^n$ is accessed via a quantum oracle $O_X\colon |i,b\rangle \mapsto |i, b \oplus X[i]\rangle$, for any index $i \in [n]$ and any $b \in [0\,..\,2^{\lceil \log \sigma \rceil})$, where $\oplus$ denotes the XOR operation. This quantum query model [Amb04, BdW02] is standard in the literature of quantum algorithms. The *query complexity* of a quantum algorithm (with success probability at least $2/3$) is the number of quantum queries it makes to the input oracles.

More specifically, it suffices for us to have a computational model that supports the following:

- We have quantum query access to the input oracle (as described above).
- We can run quantum subroutines on $\mathcal{O}(\log n)$ qubits.
- We have a classical working memory with random access (classical-read and classical-write).

The *time complexity* of our algorithm counts the number of quantum queries, the number of elementary gates that implement the quantum subroutines, and the number of classical random-access operations. Note that we do not need to assume QRAM for working memory, which was required in previous quantum algorithms for some other string problems [GS22, AJ22, JN23] in order to obtain good time complexity.

The key quantum subroutine that we use is the Grover search algorithm.

THEOREM 2.1. (GROVER SEARCH [GRO96]) *There is a quantum algorithm that, given quantum access to a function $f\colon [1\,..\,n] \to \{0,1\}$, finds an index $i \in [1\,..\,n]$ such that $f(i) = 1$ or reports that no such $i$ exists. The algorithm has $2/3$ success probability, $\mathcal{O}(\sqrt{n})$ query complexity, $\widetilde{\mathcal{O}}(\sqrt{n})$ time complexity, and uses only $\mathcal{O}(\log n)$ qubits.*

A bounded-error algorithm can be boosted to have success probability $1-1/n^c$, for arbitrarily large constant $c$, by $\mathcal{O}(\log n)$ repetitions. In this paper, we do not optimize the $\operatorname{poly}\log(n)$ factors in the quantum query complexity (and time complexity) of our algorithms.

We can use Theorem 2.1 to test the equality of two length-$\ell$ substrings of the input string(s) in $\widetilde{\mathcal{O}}(\sqrt{\ell})$ time.[3] Combined with a binary search, this allows us to find the length of their longest common prefix (resp., suffix) in $\widetilde{\mathcal{O}}(\sqrt{\ell})$ time. We can then determine their leftmost (resp, rightmost) position corresponding to a mismatch (if it exists) and hence their lexicographic (resp., co-lexicographic[4]) order in constant time.

**Edit Distance and Alignments** The *edit distance* (also known as *Levenshtein distance* [Lev65]) between two strings $X$ and $Y$, denoted by $\mathsf{ed}(X,Y)$, is the minimum number of character insertions, deletions, and substitutions required to transform $X$ into $Y$. For a formal definition, we first rely on the notion of an *alignment* between fragments of strings.

[3] For substrings $X'$ and $Y'$, define the function $f(i) = 1$ if $X'[i] \neq Y'[i]$ and $f(i) = 0$ otherwise.

[4] Co-lexicographic order refers to the lexicographic order of the reversed strings.

DEFINITION 2.2. (SEE [KPS21]) *A sequence* $\mathcal{A} = (x_i, y_i)_{i=0}^{m}$ *is an* alignment *of* $X(x\,..\,x']$ *onto* $Y(y\,..\,y']$, *denoted by* $\mathcal{A} : X(x\,..\,x'] \rightsquigarrow Y(y\,..\,y']$, *if it satisfies* $(x_0, y_0) = (x, y)$, $(x_i, y_i) \in \{(x_{i-1}+1, y_{i-1}+1), (x_{i-1}+1, y_{i-1}), (x_{i-1}, y_{i-1}+1)\}$ *for* $i \in [1\,..\,m]$, *and* $(x_m, y_m) = (x', y')$.

- *If* $(x_i, y_i) = (x_{i-1}+1, y_{i-1})$, *we say that* $\mathcal{A}$ deletes $X[x_i]$,
- *If* $(x_i, y_i) = (x_{i-1}, y_{i-1}+1)$, *we say that* $\mathcal{A}$ inserts $Y[y_i]$,
- *If* $(x_i, y_i) = (x_{i-1}+1, y_{i-1}+1)$, *we say that* $\mathcal{A}$ aligns $X[x_i]$ *and* $Y[y_i]$. *If additionally* $X[x_i] = Y[y_i]$, *we say that* $\mathcal{A}$ matches $X[x_i]$ *and* $Y[y_i]$; *otherwise,* $\mathcal{A}$ substitutes $Y[y_i]$ *for* $X[x_i]$.

The *cost* of an alignment $\mathcal{A}$ of $X(x\,..\,x']$ onto $Y(y\,..\,y']$, denoted by $\mathsf{ed}_{\mathcal{A}}(X(x\,..\,x'], Y(y\,..\,y'])$, is the total number of characters that $\mathcal{A}$ inserts, deletes, or substitutes. Now, we define the edit distance $\mathsf{ed}(X, Y)$ as the minimum cost of an alignment of $X(0\,..\,|X|]$ onto $Y(0\,..\,|Y|]$. An alignment of $X$ onto $Y$ is *optimal* if its cost is equal to $\mathsf{ed}(X, Y)$.

An alignment $\mathcal{A}'' : X(x\,..\,x'] \rightsquigarrow Z(z\,..\,z']$ is a *product* of alignments $\mathcal{A} : X(x\,..\,x'] \rightsquigarrow Y(y\,..\,y']$ and $\mathcal{A}' : Y(y\,..\,y'] \rightsquigarrow Z(z\,..\,z']$ if, for every $(\bar{x}, \bar{z}) \in \mathcal{A}''$, there is $\bar{y} \in [y\,..\,y']$ such that $(\bar{x}, \bar{y}) \in \mathcal{A}$ and $(\bar{y}, \bar{z}) \in \mathcal{A}'$. Note that such an alignment always exists and satisfies $\mathsf{ed}_{\mathcal{A}''}(X(x\,..\,x'], Z(z\,..\,z']) \le \mathsf{ed}_{\mathcal{A}}(X(x\,..\,x'], Y(y\,..\,y']) + \mathsf{ed}_{\mathcal{A}'}(Y(y\,..\,y'], Z(z\,..\,z'])$. For an alignment $\mathcal{A} : X(x\,..\,x'] \rightsquigarrow Y(y\,..\,y']$ with $\mathcal{A} = (x_i, y_i)_{i=0}^{m}$, we define the *inverse alignment* $\mathcal{A}^{-1} : Y(y\,..\,y'] \rightsquigarrow X(x\,..\,x']$ as $\mathcal{A}^{-1} \coloneqq (y_i, x_i)_{i=0}^{m}$. Note that $\mathsf{ed}_{\mathcal{A}^{-1}}(Y(y\,..\,y'], X(x\,..\,x']) = \mathsf{ed}_{\mathcal{A}}(X(x\,..\,x'], Y(y\,..\,y'])$.

**Lempel–Ziv Factorization** We say that a fragment $X[i\,..\,i+\ell)$ is a *previous factor* if it has an earlier occurrence in $X$, i.e., $X[i\,..\,i+\ell) = X[i'\,..\,i'+\ell)$ holds for some $i' \in [1\,..\,i)$. An *LZ77-like factorization* of $X$ is a factorization $X = F_1 \cdots F_f$ into non-empty *phrases* such that each phrase $F_j$ with $|F_j| > 1$ is a previous factor. In the underlying *LZ77-like representation*, every phrase $F_j = X[i\,..\,i+\ell)$ that is a previous factor is encoded as $(i', \ell)$, where $i' \in [1\,..\,i)$ satisfies $X[i\,..\,i+\ell) = X[i'\,..\,i'+\ell)$ (and is chosen arbitrarily in case of multiple possibilities); if $F_j = X[i]$ is not a previous factor, it is encoded as $(X[i], 0)$; see Fig. 1 for an example.

| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $X$ | $a$ | $b$ | $a$ | $c$ | $a$ | $b$ | $c$ | $a$ | $b$ | $c$ | $a$ | $a$ | $a$ | $a$ | $b$ |

Figure 1: The LZ77 factorization of a string $X = abacabcabcaaaab$ of length $n = 15$. The resulting encoding has $z = 8$ elements: $(a, 0)$, $(b, 0)$, $(1, 1)$, $(c, 0)$, $(1, 2)$, $(4, 5)$, $(11, 3)$, $(9, 1)$.

The LZ77 factorization [ZL77] (or the LZ77 parsing) of a string $X$, denoted $\mathsf{LZ}(X)$ is then just an LZ77-like factorization constructed by greedily parsing $X$ from left to right into longest possible phrases. More precisely, the $j$th phrase $F_j = X[i\,..\,i+\ell)$ is the longest previous factor starting at position $i$; if no previous factor starts there, then $F_j$ consists of a single character. This greedy approach is known to produce the shortest possible LZ77-like factorization.

The size $|\mathsf{LZ}(X)|$ of the LZ77 factorization of $X$ is closely related to other compressibility measures; see [Nav21] for a survey. This includes *substring complexity* $\delta(X)$, which is defined as $\max_{q=1}^{|X|} \frac{d_q(X)}{q}$, where $d_q(X)$ is the number of distinct length-$q$ substrings ($q$-grams) in $X$. It has been implicitly introduced in [RRRS13] and thoroughly studied in [KNP23]. The substring complexity enjoys many desirable features such as invariance under string reversal, monotonicity with respect to taking substring (in comparison, $|\mathsf{LZ}(X)|$ is only monotone with respect to taking prefixes), sub-additivity with respect to concatenations (shared with $|\mathsf{LZ}(X)|$), and stability with respect to edits (that is, $|\delta(X) - \delta(X')| \le \mathsf{ed}(X, X')$). Due to the relation $\delta(X) \le |\mathsf{LZ}(X)| = \mathcal{O}\big(\delta(X) \log \frac{|X|}{\delta(X)}\big)$ proved in [RRRS13], we derive the following fact about the LZ77 factorization size:

FACT 2.3. *Strings* $X, Y$ *of length at most* $n$ *satisfy* $|\mathsf{LZ}(\overline{X})| = \mathcal{O}(|\mathsf{LZ}(X)| \log n)$, $|\mathsf{LZ}(XY)| = \mathcal{O}((|\mathsf{LZ}(X)| + |\mathsf{LZ}(Y)|) \log n)$, *and* $|\mathsf{LZ}(Y)| = \mathcal{O}((|\mathsf{LZ}(X)| + \mathsf{ed}(X, Y)) \log n)$.

LZ-End was introduced by Kreft and Narvarro [KN13] to speed up the extraction of substrings relative to traditional LZ77. Unlike LZ77, LZ-End forces any new phrase that is not a leftmost occurrence of a symbol to match an occurrence ending at a previous phrase boundary, i.e., phrase $X[i\,..\,i+\ell)$ is taken as the longest fragment that is a suffix of $X[1\,..\,j)$, where $j$ is the start of a previous phrase. Like LZ77, LZ-End can be computed in linear time [KK17a, KK17b]. Moreover, the LZ-End encoding size is close to the size of LZ77 encoding, as shown in the following result:

THEOREM 2.4. (KEMPA & SAHA [KS22]) *For any string $X[1\,..\,n]$ with LZ77 factorization size $z$ and LZ-End factorization size $z_e$, we have $z_e = \mathcal{O}(z \log^2 \frac{n}{z})$.*

**Suffix Trees, Suffix Arrays, and the Burrow Wheeler Transform** We assume that the last symbol in $X[1\,..\,n]$ is a special \$ symbol that occurs only once and is lexicographically smaller than the other symbols in $X$. The suffix tree of a string $X$ is a compact trie constructed from all suffixes of $X$. The tree leaves are labeled with the starting indices of the corresponding suffixes and are sorted by the lexicographic order of the suffixes. These values in this order define the suffix array SA, i.e., $\mathrm{SA}[i]$ is such $X[\mathrm{SA}[i]\,..\,n]$ is the $i$th smallest suffix lexicographically. The inverse suffix array ISA, is defined as $\mathrm{ISA}[\mathrm{SA}[i]] = i$; equivalently, $\mathrm{ISA}[i]$ is the lexicographic rank of the suffix $X[i\,..\,n]$. The Burrows–Wheeler Transform (BWT) of a text $X$ is a permutation of the symbols of $X$ such that $\mathrm{BWT}[i] = X[\mathrm{SA}[i]-1]$ if $\mathrm{SA}[i] \neq 1$ and is \$ otherwise. The longest common extension of two suffixes $X[i\,..\,n]$, $X[j\,..\,n]$, denoted as $\mathrm{LCE}(i,j)$, is equal to the length of their longest common prefix. The suffix tree, suffix array, and the Burrows–Wheeler Transform can all be built in linear time for polynomially-sized integer alphabets [FFM00]. While suffix trees and arrays require space $\mathcal{O}(n)$ space (equivalently, $\mathcal{O}(n \log n)$ bits), the BWT requires only $n\lceil \log \sigma \rceil$ bits. Further, we can apply run-length encoding to achieve $\mathcal{O}(r)$ space.

**FM-index and Repetition-Aware Suffix Trees** The FM-index provides the ability to count and locate occurrences of a given pattern efficiently. It is constructed based on the BWT described previously and uses the *LF-mapping* to perform pattern matching. The LF-mapping is defined as $\mathrm{LF}[i] = \mathrm{ISA}[\mathrm{SA}[i]-1]$ if $\mathrm{SA}[i] \neq 1$ and is 1 otherwise. The FM-index was developed by Ferragina and Manzini [FM05] to be more space efficient than traditional suffix trees and suffix arrays. However, supporting location queries utilized sampling SA in evenly spaced intervals, in a way independent of the runs in the BWT of the text, preventing an $\mathcal{O}(r)$ data space structure with optimal (or near optimal) query time.

The r-index and subsequent fully functional text indexes were developed to utilize only $\mathcal{O}(r)$ or $\mathcal{O}(r \log \frac{n}{r})$ space. The r-index developed by Gagie, Navarro, and Prezza [GNP18b] was designed to occupy $\mathcal{O}(r)$ space and support counting and locating queries in near-optimal time. It was based on the observation that suffix array samples are necessary only for the run boundaries of the BWT and subsequent non-boundary suffix array values can be obtained in polylogarithmic time. The fully functional indexes [GNP20] use $\mathcal{O}(r \log \frac{n}{r})$ space and provide most of the capabilities of a suffix tree. The data structure allows one to determine in $\mathcal{O}(\log \frac{n}{r})$ time arbitrary SA, ISA, and LCE values, which in turn lets one determine properties of arbitrary nodes in suffix tree, such as subtree size.

## 3 Technical Overview

### 3.1 Edit Distance

Our algorithm for computing the edit distance between $X, Y \in \Sigma^{\leq n}$ uses a divide-and-conquer approach: letting $x := \lceil |X|/2 \rceil$, we would like to *optimally align* the middle character $X[x]$ to some character $Y[y]$. More formally, we would like to find $y \in [0\,..\,|Y|]$ such that $\mathsf{ed}(X,Y) = \mathsf{ed}(X[1\,..\,x), Y[1\,..\,y)) + \mathsf{ed}(X[x\,..\,|X|], Y[y\,..\,|Y|])$. Such a pair $(x,y)$, called here an *edit anchor*, allows us to decompose the problem of computing $\mathsf{ed}(X,Y)$ into two independent subproblems to be solved recursively. Therefore, a crucial component of our divide-and-conquer algorithm is to efficiently find an edit anchor $(x,y)$ for $X, Y$, given the promise that $\mathsf{ed}(X,Y) \leq k$.

**Edit anchor and LZ compression** Our plan for computing the edit anchor $(x,y)$ is to pick it from a *locally optimal* alignment $\mathcal{A}'$, which is hopefully faster to compute than a globally optimal alignment $\mathcal{A}$. More specifically, we select a suitable window $(i\,..\,j]$ that contains $x$ and define $\mathcal{A}'$ as an optimal alignment between the fragments $X' := X(i\,..\,j]$ and $Y' := Y(i\,..\,j + |Y| - |X|]$ (which satisfy $\mathsf{ed}(X',Y') \leq \mathsf{ed}(X,Y)$). To ensure correctness, this window $(i\,..\,j]$ should be long enough to eliminate ambiguity: any $(x,y) \in \mathcal{A}'$ must be guaranteed to be a (global) edit anchor provided that $\mathsf{ed}(X,Y) \leq k$. On the other hand, for efficiency's sake, this window should not be too long.

Interestingly, our criteria for selecting the window $(i\,..\,j]$ crucially rely on *string compressibility*. Specifically, we define its right boundary as the largest $j \in [x\,..\,|X|]$ such that $|\mathsf{LZ}(X(x\,..\,j])| \leq c \cdot k$ (for a sufficiently large constant $c$), and we define the left boundary $i$ symmetrically. Intuitively, $X' = X(i\,..\,j]$ is the maximal compressible context of $X[x]$. Using our LZ-compression algorithm (Theorem 1.2), the window boundaries $i, j$ can be found in $\widetilde{\mathcal{O}}(\sqrt{kn})$ quantum time by binary search. After we retrieve the LZ compression of the fragments $X'$ and $Y'$ with $\widetilde{\mathcal{O}}(k)$ phrases, we compute an anchor $(x,y)$ contained in an optimal alignment $\mathcal{A}' : X' \rightsquigarrow Y'$. Using the classical Landau–Vishkin algorithm [LV88] and appropriate LCE query implementation [I17] for compressed

strings, this requires $\widetilde{\mathcal{O}}(k^2)$ additional time complexity:

THEOREM 3.1. (SEE ALSO [GKLS22]) *There exists an algorithm that, for any two strings $X, Y \in \Sigma^*$, given $\mathsf{LZ}(X)$ and $\mathsf{LZ}(Y)$, computes $k := \mathsf{ed}(X,Y)$, along with a sequence of $k$ edits transforming $X$ into $Y$, in $\widetilde{\mathcal{O}}(|\mathsf{LZ}(X)| + |\mathsf{LZ}(Y)| + k^2)$ time.*

It remains to explain why the anchor $(x, y)$ derived from the optimal local alignment $\mathcal{A}' : X' \rightsquigarrow Y'$ is globally optimal for $X, Y$. This would follow from the following key claim: any optimal global alignment $\mathcal{A}\colon X \rightsquigarrow Y$ must intersect the alignment $\mathcal{A}'\colon X' \rightsquigarrow Y'$ at two points $(x_\ell, y_\ell), (x_r, y_r)$ such that $(x_\ell, y_\ell) \preceq (x, y) \preceq (x_r, y_r)$. Indeed, this claim implies that we can replace the part of $\mathcal{A}$ between these two points with the corresponding part of $\mathcal{A}'$, which contains the anchor $(x, y)$, without increasing the cost of the alignment. So $(x, y)$ is an edit anchor for $X, Y$.

To see why the claim above is true, here we focus on the intersection $(x_r, y_r)$ to the right of $(x, y)$, and without loss of generality assume the right boundary $j$ of the window achieves the equality $|\mathsf{LZ}(X(x\,..\,j])| = c \cdot k$. If $\mathcal{A}'$ does not intersect $\mathcal{A}$ at any point to the right of of $(x, y)$, then we can restrict both of them to the fragment $Y'_r := Y(y\,..\,j + |Y| - |X|]$ and obtain two *disjoint* $\mathcal{O}(k)$-cost alignments: $\mathcal{A}'_r\colon X(x\,..\,j] \rightsquigarrow Y'_r$ and $\mathcal{A}_r\colon X(\tilde{x}\,..\,\tilde{j}] \rightsquigarrow Y'_r$ for some $\tilde{x} = x \pm \mathcal{O}(k)$ and $\tilde{j} = j \pm \mathcal{O}(k)$. Without loss of generality, assume $\tilde{x} < x$. Then, the product $\mathcal{A}_r^{-1} \circ \mathcal{A}'_r\colon X(x\,..\,j] \rightsquigarrow X(\tilde{x}\,..\,\tilde{j}]$ is an $\mathcal{O}(k)$-cost alignment that, due to disjointness and $\tilde{x} < x$, matches each unedited character $X[t]$ to an earlier character $X[\tilde{t}]$ with $\tilde{t} < t$. This gives an LZ-like factorization of $X(x\,..\,j]$ into $\mathcal{O}(k)$ phrases, which contradicts the assumption that $|\mathsf{LZ}(X(x\,..\,j])| = c \cdot k$ (the constant $c$ is large enough).

We remark that a similar strategy, albeit with a different compressibility measure called *self-edit distance*, has been concurrently applied to efficiently solve the bounded *weighted* edit distance problem [CKW23]. A related compression argument appeared in [KPS21, Lemma III.10] in a different context of sketching edit distance. However, it was only applied to masked strings (with matched characters replaced by #s) and achieved a weaker $\mathcal{O}(k^2)$ bound that does not suffice here.

**Ideal analysis of divide and conquer** So far, we have described a quantum algorithm which, given strings $X, Y$ with promise $\mathsf{ed}(X,Y) \le k$, finds an edit anchor $(x, y)$ in $T_0(|X|, |Y|, k) := \widetilde{\mathcal{O}}(\sqrt{k(|X| + |Y|)})$ query complexity and $\widetilde{\mathcal{O}}(\sqrt{k(|X| + |Y|)} + k^2)$ time complexity. Let us try to analyze the query complexity of our divide-and-conquer approach based on this anchor-finding subroutine (the time complexity has a similar analysis, which we omit from this overview).

Suppose that the anchor $(x, y)$ (with $x = \lceil |X|/2 \rceil$) decomposes the input strings into $X = X_1X_2$ and $Y = Y_1Y_2$, resulting in two subproblems $\mathsf{ed}(X_1, Y_1)$ and $\mathsf{ed}(X_2, Y_2)$. Suppose that, before recursively solving these subproblems, we can somehow obtain upper bounds $k_1 \ge \mathsf{ed}(X_1, Y_1)$ and $k_2 \ge \mathsf{ed}(X_2, Y_2)$ such that $k_1 + k_2 = k$. In this ideal scenario, the overall query complexity of is

$$T(|X|, |Y|, k) = T(|X_1|, |Y_1|, k_1) + T(|X_2|, |Y_2|, k_2) + T_0(|X|, |Y|, k) \le \mathcal{O}(T_0(|X|, |Y|, k) \cdot \log |X|),$$

which can be shown by applying the Cauchy–Schwarz inequality to all the subproblems $(X_i, Y_i, k_i)$ at each of the $\lceil \log |X| \rceil$ levels of recursion (for vectors $(\sqrt{|X_i| + |Y_i|})_i$ and $(\sqrt{k_i})_i$).

This query complexity meets our target of $\widetilde{\mathcal{O}}(\sqrt{kn})$ but relies on the unrealistic assumption about knowing the upper bounds of $\mathsf{ed}(X_1, Y_1)$ and $\mathsf{ed}(X_2, Y_2)$. To remove this assumption, we face the following situation: when solving each subproblem $(X, Y)$ in the recursion tree, we a priori do not know an upper bound on $\mathsf{ed}(X, Y)$, but we still want the query complexity spent on this subproblem to be bounded in terms of the true edit distance $k = \mathsf{ed}(X, Y)$.

**Reducing the overhead of exponential search** A first attempt to resolve this issue is to estimate the distance $\mathsf{ed}(X, Y)$ with exponential search: We iteratively try a sequence of gradually increasing thresholds $k_1, k_2, \ldots$ (where the usual choice is $k_i = 2^i$) and, in the $i$-th iteration, pretend $k_i \ge \mathsf{ed}(X, Y)$. We apply the aforementioned anchor-finding subroutine in $T_0(|X|, |Y|, k_i)$ quantum query complexity, and then recurse on the subproblems defined by this anchor. In the first few iterations, where $k_i < \mathsf{ed}(X, Y)$, the found anchor might be incorrect, of course, causing the whole recursive call to eventually fail. But hopefully the total cost can be still bounded in terms of the cost of the successful iteration (i.e., the first one where $k_i \ge \mathsf{ed}(X, Y)$ holds).

Unfortunately, this standard exponential search idea no longer works in our recursive scenario: at each level, the wasted work incurs at least a constant-factor overhead, which accumulates multiplicatively across the $\lceil \log |X| \rceil$ levels of recursion, resulting in at least a polynomial-factor overall overhead. One possible solution is

to decrease the recursion depth by enlarging the branching factor, and hence decrease the overall overhead to a subpolynomial factor. Instead of this generic idea, we use a more problem-specific approach to carefully implement the exponential search, so that a lot of redundant work can be avoided, and the total overhead is decreased to only a polylogarithmic factor!

As before, we iteratively try a sequence of increasing thresholds, where each iteration leads to a recursion based on the anchor found using the corresponding threshold $k_i$. Our goal is to reduce the cost of the wasted computations so that it becomes almost negligible compared to the correct recursive calls (i.e., based on the true edit-distance anchor).

The key insight here is that, in order to tell whether an anchor $(x_i, y_i)$, computed under a promise $\mathsf{ed}(X, Y) \le k_i$, is a correct anchor, we do not actually need to wait until its entire recursion finishes. Instead, we can pause this recursion after a certain amount of time and proceed to the larger threshold $k_{i+1}$. A procedure similar to the aforementioned anchor-finding subroutine can tell us whether the earlier anchor $(x_i, y_i)$ is still correct under a weakened promise $\mathsf{ed}(X, Y) \le k_{i+1}$. If it is, then we can continue running the earlier paused recursive call using $(x_i, y_i)$ (instead of starting from scratch using a new anchor); otherwise, we can abort that call, because we already know it is useless, and start a new recursive call using a new anchor $(x_{i+1}, y_{i+1})$ instead.

This strategy allows us to control the total complexity contributed by recursive calls generated by incorrect anchors. We want their contribution to be at most a $1/\operatorname{polylog} n$ fraction of the complexity of the correct calls and, for this reason, we adjust the threshold sequence of the exponential search to $k_i = (\log n)^{2i}$ instead of $k_i = 2^i$. More details on implementing this strategy are given in Section 4.1.

Let us remark that the related divide-and-conquer procedure for bounded weighted edit distance [CKW23] faces an analogous issue. However, since its target complexity is $\widetilde{\mathcal{O}}(k\sqrt{kn})$, recursive calls can afford measuring compressibility with respect to the global budget. As a result, all anchors are verified under the global promise of $\mathsf{ed}(X, Y) \le k$, and the failed recursive calls are avoided.

**3.2 LZ77 Factorization** Let us derive our quantum LZ77 factorization algorithm by attempting a straightforward approach and seeing why it fails. Recall that the LZ77 factorization of a text $X$ can be found by processing text from left to right. Assume we are trying to determine $i$-th factor in the LZ77 factorization. We use $s_i$ to denote the starting location of this factor and suppose the factorization of the prefix $X[1\,..\,s_i)$ has been already computed. We next need to determine the largest $\ell_i \ge 0$ such that $X[s_i\,..\,s_i + \ell_i)$ has an occurrence starting at some position $p_i < s_i$. To do this efficiently, we wish to maintain a data structure over the previously processed text. This data structure should (i) occupy $\widetilde{\mathcal{O}}(\sqrt{zn})$ space, (ii) allow us to determine the largest $\ell_i$ as described above efficiently, and (iii) support efficient insertions once we find the new factor. The target time complexity for adding a factor of length $\ell_i$ is $\widetilde{\mathcal{O}}(\sqrt{\ell_i})$. If we can achieve this, then the entire LZ77 factorization will be found in time polylogarithmic factors from $\sum_{i=1}^{z} \sqrt{\ell_i} \le \sqrt{zn}$, where we used that $\sum_{i=1}^{z} \ell_i = n$.

**Suffix-Array Approach** As an initial attempt, consider maintaining the suffix array of the reversed prefix $\overline{X[1\,..\,s_i)}$. Doing so allows us to determine the $k$-th co-lexicographically largest prefix of $X[1\,..\,s_i)$ in constant time. Based on this, we can try to find the *non-overlapping* LZ77 factorization, a variation of LZ77 that requires $p_i + \ell_i \le s_i$ and results in a factorization size within a logarithmic factor away from the optimum. To determine the largest $\ell_i$ such that $X[s_i\,..\,s_i + \ell_i)$ matches a substring of $X[1\,..\,s_i)$, we combine exponential search on $\ell_i$ and binary search on this sorted set of prefixes. To check, for a particular $\ell_i$, whether $X[s_i\,..\,s_i + \ell_i)$ has an earlier occurrence, we start with the median of the sorted prefixes and apply Grover search (Theorem 2.1) in $\widetilde{\mathcal{O}}(\sqrt{\ell_i})$ time to determine if a mismatch exists between $X[s_i\,..\,s_i+\ell_i)$ and the median prefix. If a mismatch exists, we determine whether $X[s_i\,..\,s_i + \ell)$ is co-lexicographically larger or smaller, and continue the binary search accordingly. This technique allows us to determine the $i$-th factor in $\widetilde{\mathcal{O}}(\sqrt{\ell_i})$ time. Unfortunately, the time for updating the suffix array is $\mathcal{O}(\ell_i)$, resulting in a linear time overall.

**LZ-End Approach** A natural approach to overcome this is to use the LZ-End factorization rather than non-overlapping LZ77 factorization. Recall that LZ-End differs in that each new factor is either the first occurrence of a symbol or the longest substring whose earlier occurrence ends at the end of a previous factor. It was recently shown that the number of factors in the LZ-End factorization satisfies $z_e = \mathcal{O}(z \log^2 n)$ [KS22]. As discussed more in Section 5.3, once the LZ-End factorization is computed, we can obtain the actual LZ77 factorization in $\widetilde{\mathcal{O}}(z)$ time. The reason we consider LZ-End is that, since each (non-trivial) factor has an occurrence ending at the end of a previous factor, we only need to maintain the co-lexicographically sorted order of the prefixes $X[1\,..\,s_{i'})$ for

$i' \le i$. For a given $\ell$, this order suffices for checking in $\widetilde{\mathcal{O}}(\sqrt{\ell})$ time if $X[s_i \,..\, s_i + \ell)$ has a previous occurrence ending at the end of a previous factor. However, this idea alone does not work because exponential search on $\ell$ may fail due to the lack of monotonicity. In other words, $X[s_i \,..\, s_i + \ell)$ may have an earlier occurrence ending at a previous factor while $X[s_i \,..\, s_i + \ell - 1)$ does not have one.

**LZ-End+$\tau$ Approach** To facilitate using exponential search for finding the next factor length, we introduce a variation on LZ-End that we call LZ-End+$\tau$. We define LZ-End+$\tau$ by making each new factor $X[s_i \,..\, s_i + \ell_i)$ either the first occurrence of a new symbol or the longest substring whose earlier occurrence is of the form $X[p_i \,..\, q_i)$, where $q_i \le s_i$ satisfies $q_i \bmod \tau = 0$ or $q_i = s_{i'}$ for $i' \le i$. We let $z_{e+\tau}$ denote the number of LZ-End+$\tau$ factors. A crucial observation is that we still have $z_{e+\tau} = \mathcal{O}(z \log^2 n)$. Additionally, the number of prefixes that have to be checked for finding LZ-End+$\tau$ factors is $\widetilde{\mathcal{O}}(z_{e+\tau} + n/\tau)$. Suppose we are trying to find the next factor starting at index $s_i$. We say the *$\tau$-far property* holds for an index $j \ge s_i$ if there exists $h \in [\max(s_i, j - \tau) \,..\, j]$ such that $X[s_i \,..\, h)$ has an earlier occurrence of the form $X[p \,..\, q)$ with $q \le s_i$ satisfying $q \bmod \tau = 0$ or $q = s_{i'}$ for $i' \le i$. By definition, the $\tau$-far property is monotone, in that if it holds for $j > s_i$, then it also holds for $j - 1$. Hence, the problem reduces to testing the $\tau$-far property for a given $j$ and maintaining the order of prefixes $X[1 \,..\, q)$ for $q \le s_i$ satisfying $q \bmod \tau = 0$ or $q = s_{i'}$ for $i' \le i$.

We first consider the problem of checking whether the $\tau$-far property holds for a given $j \ge s_i$. Our algorithm utilizes the dynamic LCE data structure of Nishimoto et al. [NII+16]. Starting from an initially empty string, it supports insertions of individual characters and arbitrary substrings of the existing text in $\widetilde{\mathcal{O}}(1)$ time. In addition, it answers LCE queries between two arbitrary indices in $\widetilde{\mathcal{O}}(1)$ time. To describe the process of finding the next factor $X[s_i \,..\, s_i + \ell_i)$, we assume that we have the co-lexicographic ordering of the selected prefixes $X[1 \,..\, q)$ (across all $q \le s_i$ such that $q \bmod \tau = 0$ or $q = s_{i'}$ for $i' \le i$). We denote this set of prefixes as $\mathcal{P}_{i-1}$. We also assume that the dynamic LCE data structure has been constructed for $\overline{X[1 \,..\, s_i)}$. Our solution first creates at most $\tau + 1$ ranges of indices into $\mathcal{P}_{i-1}$, each corresponding to the prefixes in $\mathcal{P}_{i-1}$ that have $X[\max(s_i, j - \tau) \,..\, h)$ as a suffix. This is accomplished in $\widetilde{\mathcal{O}}(\tau)$ time by using the LCE data structure. Next, we observe that, for arbitrary $k$, we can find the $k$-th co-lexicographically largest prefix contained in these ranges in $\widetilde{\mathcal{O}}(\tau)$ time. Based on this observation, we can then apply binary search and the Grover search algorithm to determine if the $\tau$-far property holds for $j$, similar to the LZ-End case. Combining with exponential search on $j$, we can find the next factor of length $\ell_i$ in $\widetilde{\mathcal{O}}(\tau + \sqrt{\ell_i})$ time.

To update the sorted list of prefixes $\mathcal{P}_{i-1}$ to $\mathcal{P}_i$, we again utilize the dynamic LCE data structure. Assume we just determined the new factor $X[s_i \,..\, s_{i+1})$. Then, this new factor is either a previously occurring substring with a location determined in the previous step or the first occurrence of a symbol. As mentioned, the LCE data structure supports appending such a substring in $\widetilde{\mathcal{O}}(1)$ time. To insert into the sorted order the new prefixes $X[1 \,..\, q)$ with $q \in (s_i \,..\, s_{i+1}]$ such that $q \bmod \tau = 0$ or $q = s_{i+1}$, we apply LCE queries to compare $X[1 \,..\, q)$ to the prefixes in the currently sorted list as needed, resulting in $\mathcal{O}(\log n)$ queries and $\widetilde{\mathcal{O}}(1)$ time per inserted prefix.

Overall, the algorithm takes $\widetilde{\mathcal{O}}(\frac{n}{\tau} + \sqrt{z_{e+\tau} n} + z_{e+\tau}\tau)$ time, which is $\widetilde{\mathcal{O}}(\sqrt{z_{e+\tau} n}) = \mathcal{O}(\sqrt{zn})$ for optimal $\tau$. To complete the proof of Theorem 1.2, we convert the LZ-End+$\tau$ factorization to an LZ77 factorization in $\widetilde{\mathcal{O}}(z_{e+\tau}) = \widetilde{\mathcal{O}}(z)$ time using data structure presented in [KK22], which allows us to determine the leftmost occurrence of any substring in $\widetilde{\mathcal{O}}(1)$ time.

### 3.3 Compressed Text Indexing and Applications

We next outline the construction of the indexes from Theorem 1.4. Let $\tau = \Theta(\sqrt{n/r})$. The first step is to obtain a (less efficient) index in time $\widetilde{\mathcal{O}}(n/\tau + \tau r) = \widetilde{\mathcal{O}}(\sqrt{nr})$ that can support SA and ISA queries in time $\widetilde{\mathcal{O}}(\tau)$. This is done by preprocessing the RL-BWT encoding, obtained using Corollary 1.3, so that we can determine in $\widetilde{\mathcal{O}}(1)$ time the result of applying the LF mapping a total of $\tau$ times starting at position $i$, i.e., $\mathrm{LF}^{\tau}[i]$. The computation utilizes prefix doubling and alphabet replacement techniques. Thanks to the prefix doubling, we only need to perform $\mathcal{O}(\log \tau)$ alphabet replacement steps, making the overall computation time $\widetilde{\mathcal{O}}(\tau r)$. We next take $n/\tau$ SA samples, evenly spaced by text position, in $\widetilde{\mathcal{O}}(n/\tau)$ time. These samples make any SA value computable in $\widetilde{\mathcal{O}}(\tau)$ time. Combined with the LCE data structure, this supports ISA queries in $\widetilde{\mathcal{O}}(\tau)$ time.

The $\mathcal{O}(r)$-space index for pattern matching can be constructed in $\widetilde{\mathcal{O}}(r)$ time, given the text positions corresponding to BWT-run boundaries [GNP18b]. The later can be achieved via $\mathcal{O}(r)$ SA queries using the index described above. Next, we demonstrate that the suffix array index described in [GNP20] can also be

constructed using $\widetilde{\mathcal{O}}(r)$ queries. The main technical challenge lies in efficiently determining, for a given range of $[s\,..\,e]$ of indices in RL-BWT, the smallest $k \geq 0$ such that $\mathrm{LF}^k([s\,..\,e])$ contains a BWT run-boundary as well as the interval $\mathrm{LF}^k([s\,..\,e])$ itself. We outline how to accomplish this using $\widetilde{\mathcal{O}}(1)$ queries on our less-efficient index. Thus, in both cases, the construction time is $\widetilde{\mathcal{O}}(\sqrt{rn})$. We also maintain the $\mathcal{O}(z \log \frac{n}{z})$ space LCE structure [I17] so that ISA queries can be supported in $\widetilde{\mathcal{O}}(1)$ time using binary search on SA values. These indexes allow us to solve several fundamental problems efficiently, as described in Section 6.3.

## 4 Quantum Algorithm for Bounded Edit Distance

**4.1 Recursive algorithm** Our quantum algorithm for computing edit distance can be viewed as a deterministic classical algorithm that makes the following oracle calls: (1) compute the LZ77 factorization of a substring of the input strings; (2) check the equality of two substrings of the input strings. These two tasks can be efficiently solved using quantum subroutines, by Theorem 1.2 and Grover search (Theorem 2.1). By applying error reduction (via a logarithmic number of repetitions) and a global union bound, we assume that all these quantum subroutines work correctly, so in the following, we can ignore the analysis of failure probabilities.

We need the following notion.

Definition 4.1. (Edit anchor) *We say that $(x, y) \in [i\,..\,j] \times [i'\,..\,j']$ is an* edit anchor *of fragments $X(i\,..\,j]$ and $Y(i'\,..\,j']$ if $\mathsf{ed}(X(i\,..\,j], Y(i'\,..\,j']) = \mathsf{ed}(X(i\,..\,x], Y(i'\,..\,y]) + \mathsf{ed}(X(x\,..\,j], Y(y\,..\,j'])$, that is, $(x, y) \in \mathcal{A}$ for some optimal alignment $\mathcal{A} : X(i\,..\,j] \rightsquigarrow Y(i'\,..\,j']$.*

*Moreover, for an integer $k \geq 0$, we say that $(x, y)$ is a $k$-edit anchor of $X(i\,..\,j]$ and $Y(i'\,..\,j']$ if $(x, y)$ is their edit anchor or $\mathsf{ed}(X(i\,..\,j], Y(i'\,..\,j']) > k$.*

We will prove the following two lemmas in Section 4.2.

Lemma 4.2. (Finding an anchor: $\texttt{FindAnchor}(X, Y, k, x)$) *There exists a quantum algorithm that, given strings $X, Y \in \Sigma^{\leq n}$, an integer $k \geq 1$, and a position $x \in [0\,..\,|X|]$, finds a position $y \in [0\,..\,|Y|]$ such that $(x, y)$ is a $k$-edit anchor of $X, Y$. The algorithm has query complexity $\widetilde{\mathcal{O}}(\sqrt{kn})$ and time complexity $\widetilde{\mathcal{O}}(\sqrt{kn} + k^2)$.*

Lemma 4.3. (Testing an anchor: $\texttt{IsAnchor}(X, Y, k, (x, y))$) *There exists a quantum algorithm that, given strings $X, Y \in \Sigma^{\leq n}$, an integer $k \geq 1$, and a pair $(x, y) \in [0\,..\,|X|] \times [0\,..\,|Y|]$,*

- *if $\mathsf{ed}(X, Y) \leq k$, decides whether $(x, y)$ is an edit anchor of $X, Y$;*
- *otherwise, returns an arbitrary answer.*

*The algorithm has query complexity $\widetilde{\mathcal{O}}(\sqrt{kn})$ and time complexity $\widetilde{\mathcal{O}}(\sqrt{kn} + k^2)$.*

The recursive algorithm for computing $\mathsf{ed}(X, Y)$ is given in Algorithm 1. The outermost function call is $\mathsf{Solve}(X, Y)$ with preconditions $X \neq Y$ and $|X| > 0$; the cases of $X = Y$ and $|X| = 0$ can be handled at the very beginning. As discussed earlier in the technical overview, Algorithm 1 may pause the recursive calls it makes after they have spent certain amount of time or queries. In order to cleanly formalize these conditions, we consider two types of *tokens*, called q-tokens and t-tokens, respectively, that our algorithm *burns* in Lines 2 and 9. Recursive calls can be paused (or terminated) if they exceed certain quotas for the number of burnt tokens; see Line 16.

Let $n = \max\{|X|, |Y|, 2\}$ be the global input length, and define a global parameter $r = \lceil 5 \log n \rceil$. We will use the following function

$$T_q(|X|, |Y|, d) := 10\sqrt{d \cdot (|X| + |Y|)} \cdot r^3 \cdot \left(\tfrac{r+2}{r}\right)^{\lceil \log |X| \rceil}$$

to measure the number q-tokens burnt by Algorithm 1 and its recursive calls. Analogously, we measure the number of t-tokens burnt using the following function:

$$T_t(|X|, |Y|, d) := 10 \cdot d^2 \cdot r^9 \cdot \left(\tfrac{r+2}{r}\right)^{\lceil \log |X| \rceil}.$$

Our main claim is the following:

Lemma 4.4. *Given strings $X, Y \in \Sigma^{\leq n}$ satisfying $|X| > 0$ and $\mathsf{ed}(X, Y) = d \geq 1$, the procedure $\mathsf{Solve}(X, Y)$ correctly returns $d$, and (including recursive calls) burns at most $T_q(|X|, |Y|, d)$ q-tokens and at most $T_t(|X|, |Y|, d)$ t-tokens.*

**Algorithm 1:** $\mathsf{Solve}(X, Y)$ (preconditions: $X \neq Y$ and $|X| > 0$)

1 **if** $|X| = 1$ **then**
2  Burn $(|Y| + 1)$ q-tokens and $(|Y| + 1)$ t-tokens
3  **for** $y \leftarrow 1$ **to** $|Y|$ **do**
4  **if** $X[1] = Y[y]$ **then return** $|Y| - 1$
5  **return** $\max(1, |Y|)$
6 Initialize anchor $a \leftarrow (\perp, \perp)$
7 Initialize program $A \leftarrow \perp$
8 **for** $i \leftarrow 0, 1, 2 \ldots$ **do**
9  Burn $\sqrt{r^{2i+2}(|X| + |Y|)}$ q-tokens and $(r^{2i+2})^2$ t-tokens
10  **if not** $\texttt{IsAnchor}(X, Y, r^{2i+2}, a)$ **then** // Lemma 4.3
11  $a \leftarrow \texttt{FindAnchor}(X, Y, r^{2i+2}, \lceil |X|/2 \rceil)$ // Lemma 4.2
12  Check whether $X(0\,..\,a_x] = Y(0\,..\,a_y]$ // Theorem 2.1
13  Check whether $X(a_x\,..\,|X|] = Y(a_y\,..\,|Y|]$ // Theorem 2.1
14  Define program $A_i := \big[\textbf{return } \mathsf{Solve}(X(0\,..\,a_x], Y(0\,..\,a_y]) + \mathsf{Solve}(X(a_x\,..\,|X|], Y(a_y\,..\,|Y|])\big]$, skipping the corresponding recursive call if equality is found in Line 12 or 13
15  Terminate current $A$, and redefine $A \leftarrow A_i$
16  Resume $A$ and run it until it attempts to burn more than $T_q(|X|, |Y|, r^{2i})$ q-tokens or more than $T_t(|X|, |Y|, r^{2i})$ t-tokens since its beginning
17  **if** $A$ has already finished, with return value $d$ **then**
18  **if** $d < r^{2i+2}$ **then return** $d$

*Proof.* We first prove the correctness of Algorithm 1. Suppose $|X| \geq 2$ (otherwise, the correctness is clear; see Lines 4 and 5) and denote $j = \lfloor \log_{r^2} d \rfloor$. Observe that Algorithm 1 always returns the cost of a valid alignment between $X$ and $Y$. We need to show it indeed returns the cost of an optimal alignment. Due to the check at Line 18, the algorithm can only return in iteration $i \geq j$. Starting from iteration $j$, the program $A$ is based on an $r^{2j+2}$-edit anchor $a$ (due to Lines 10 and 11), which belongs to an optimal alignment of $X, Y$ by Definition 4.1 since $d < r^{2j+2}$. Hence, once program $A$ terminates, it indeed returns the correct answer $d = \mathsf{ed}(X, Y)$ (assuming its recursive calls are correct, by induction).

Next, we shall prove that the algorithm does not burn too many tokens. If $|X| = 1$, then $|Y| + 1 \leq 3d$ because $d \geq \max(1, |Y| - 1)$. Thus, the number of burnt q-tokens satisfies

$$|Y| + 1 \leq \sqrt{3d \cdot (|Y| + 1)} < 10\sqrt{d \cdot (|X| + |Y|)} < T_q(|X|, |Y|, d).$$

Similarly, the number of burnt t-tokens is

$$|Y| + 1 \leq 3d < 10d^2 < T_t(|X|, |Y|, d).$$

Henceforth, we assume that $|X| \geq 2$ and analyze the number of burnt tokens in three parts.

- We first consider the program $A$ that is running during iteration $j$. Note that later iterations $i > j$ will not kill program $A$: due to $d < r^{2j+2}$, the procedure $\texttt{IsAnchor}(X, Y, r^{2i+2}, a)$ will not discard the $r^{2j+2}$-edit anchor $a$. So $A$ is the program that eventually returns on Line 18. The program $A$ consists of two recursive calls, $\mathsf{Solve}(X(0\,..\,a_x], Y(0\,..\,a_y])$ and $\mathsf{Solve}(X(a_x\,..\,|X|], Y(a_y\,..\,|Y|])$ and, by induction, they return edit distances $d_1$ and $d_2$, respectively, with $d_1 + d_2 = d$. If either one of $d_1, d_2$ is zero, then the corresponding recursive call is skipped and does not burn any tokens. Let $n_1 = |X(0\,..\,a_x]|$ and $n_2 = |X(a_x\,..\,|X|]|$, which satisfy $n_1 = \lceil \frac{1}{2}|X| \rceil \geq \lfloor \frac{1}{2}|X| \rfloor = n_2$ by definition of $a$ at Line 11. Let $m_1 = |Y(0\,..\,a_y]|$ and $m_2 = |Y(a_y\,..\,|Y|]|$. By the inductive hypothesis, and due to the Cauchy–Schwarz inequality, the number of q-tokens burnt by the two recursive calls is at most

$$T_q(n_1, m_1, d_1) + T_q(n_2, m_2, d_2) \leq 10\left(\sqrt{d_1(n_1 + m_1)} + \sqrt{d_2(n_2 + m_2)}\right) \cdot r^3 \cdot \left(\tfrac{r+2}{r}\right)^{\lceil \log n_1 \rceil}$$

$$\leq 10\sqrt{d\cdot(|X|+|Y|)}\cdot r^3\cdot\left(\tfrac{r+2}{r}\right)^{\lceil\log|X|\rceil-1}$$
$$=\tfrac{r}{r+2}\cdot T_q(|X|,|Y|,d).$$

The number of burnt t-tokens satisfies

$$T_t(n_1,m_1,d_1)+T_t(n_2,m_2,d_2)\leq 10\cdot(d_1^2+d_2^2)\cdot r^9\cdot\left(\tfrac{r+2}{r}\right)^{\lceil\log n_1\rceil}$$
$$\leq 10\cdot d^2\cdot r^9\cdot\left(\tfrac{r+2}{r}\right)^{\lceil\log|X|\rceil-1}$$
$$=\tfrac{r}{r+2}\cdot T_t(|X|,|Y|,d).$$

Note that these two sums are smaller than $T_q(|X|,|Y|,r^{2j+2})$ and $T_t(|X|,|Y|,r^{2j+2})$ respectively, which means that $A$ will finish running in or before iteration $j+1$, since $A$ will have burnt at most $T_q(|X|,|Y|,r^{2j+2})$ q-tokens and at most $T_t(|X|,|Y|,r^{2j+2})$ t-tokens in iteration $j+1$ at Line 16.

- Now, we analyze the number of tokens burnt on Line 9. For each iteration $i\geq 0$, we burned $\sqrt{r^{2i+2}(|X|+|Y|)}$ q-tokens and $r^{4i+4}$ t-tokens. Since we only performed iterations $i\in[0\,..\,j+1]$, the total number of q-tokens burnt is at most

$$\sum_{i=0}^{j+1}\sqrt{r^{2i+2}(|X|+|Y|)}=\sum_{i=0}^{j+1}r^{i+1}\sqrt{|X|+|Y|}$$
$$\leq\tfrac{1}{r-1}\cdot r^{j+3}\sqrt{|X|+|Y|}$$
$$\leq\tfrac{1}{r-1}\cdot\sqrt{d}\cdot\sqrt{|X|+|Y|}\cdot r^3\cdot\left(\tfrac{r+2}{r}\right)^{\lceil\log|X|\rceil}$$
$$=\tfrac{0.1}{r-1}\cdot T_q(|X|,|Y|,d).$$

The total number of t-tokens burnt, on the other hand, is at most

$$\sum_{i=0}^{j+1}(r^{2i+2})^2=\sum_{i=0}^{j+1}r^{4i+4}$$
$$\leq\tfrac{1}{r^4-1}\cdot r^{4j+12}$$
$$\leq\tfrac{1}{r-1}\cdot r^{4j+9}$$
$$\leq\tfrac{1}{r-1}\cdot d^2\cdot r^9\cdot\left(\tfrac{r+2}{r}\right)^{\lceil\log|X|\rceil}$$
$$=\tfrac{0.1}{r-1}\cdot T_t(|X|,|Y|,d).$$

- Now, we analyze the number of tokens burnt by earlier programs that were terminated. Suppose the correct anchor for program $A$ was computed in iteration $i^*\leq j$. Then, every previous wrong anchor was terminated in some iteration $i\in[1\,..\,i^*]$, where we found it did not pass the check at Line 10, which means that the corresponding wrong program has only burnt at most $T_q(|X|,|Y|,r^{2i-2})$ q-tokens and at most $T_t(|X|,|Y|,r^{2i-2})$ t-tokens before it was paused at Line 16 in iteration $i-1$. Summing over all such possible wrong executions, the number of q-tokens is at most

$$\sum_{i=1}^{i^*}T_q(|X|,|Y|,r^{2i-2})\leq\sum_{i=1}^{j}10\sqrt{r^{2i-2}\cdot(|X|+|Y|)}\cdot r^3\cdot\left(\tfrac{r+2}{r}\right)^{\lceil\log|X|\rceil}$$
$$<\tfrac{10\cdot r^j}{r-1}\cdot\sqrt{|X|+|Y|}\cdot r^3\cdot\left(\tfrac{r+2}{r}\right)^{\lceil\log|X|\rceil}$$
$$\leq\tfrac{10}{r-1}\cdot\sqrt{d(|X|+|Y|)}\cdot r^3\cdot\left(\tfrac{r+2}{r}\right)^{\lceil\log|X|\rceil}$$
$$=\tfrac{1}{r-1}\cdot T_q(|X|,|Y|,d).$$

The total number of t-tokens burnt by these terminated calls is at most

$$\sum_{i=1}^{i^*}T_t(|X|,|Y|,r^{2(i-1)})\leq\sum_{i=1}^{j}10\cdot r^{4(i-1)}\cdot r^9\cdot\left(\tfrac{r+2}{r}\right)^{\lceil\log|X|\rceil}$$

$$\leq \tfrac{10}{r^4-1} \cdot r^{4j} \cdot r^9 \cdot \left(\tfrac{r+2}{r}\right)^{\lceil \log |X| \rceil}$$
$$\leq \tfrac{10}{r-1} \cdot d^2 \cdot r^9 \cdot \left(\tfrac{r+2}{r}\right)^{\lceil \log |X| \rceil}$$
$$= \tfrac{1}{r-1} \cdot T_t(|X|, |Y|, d).$$

Finally, summing up the three parts, the total number of q-tokens burnt by Algorithm 1 is at most

$$T_q(|X|, |Y|, d) \cdot \left(\tfrac{r}{r+2} + \tfrac{0.1}{r-1} + \tfrac{1}{r-1}\right) \leq T_q(|X|, |Y|, d),$$

and similarly the number of burnt t-tokens is at most

$$T_t(|X|, |Y|, d) \cdot \left(\tfrac{r}{r+2} + \tfrac{0.1}{r-1} + \tfrac{1}{r-1}\right) \leq T_t(|X|, |Y|, d),$$

where we used $r \geq 5 > \frac{14}{3}$. □

Next, we describe a (classical) scheduler that is used in the implementation of Algorithm 1 to keep track of the quotas for the number of burnt tokens. Recall that the recursion of Algorithm 1 has $\lceil \log n \rceil$ levels. When we are at a certain node $v$ of the recursion tree, each ancestor node $p$ holds two counters $q_p, t_p$ that keep track of the remaining tokens that the program corresponding to $p$ is allowed to burn. When the current node $v$ attempts to burn $T$ t-tokens and $Q$ q-tokens (Line 9), we check the quotas of all ancestors $p$ of $v$. If $T \leq t_p$ and $Q \leq q_p$ holds for all ancestors $p$ of $v$, then we can safely burn the tokens and decrease the quotas, setting $t_p \leftarrow t_p - T$ and $q_p \leftarrow q_p - Q$ for all ancestors $p$. Otherwise, we choose the nearest ancestor $p$ with $t_p < T$ or $q_p < Q$ and pass control from $v$ to the parent of $p$. We also save a back pointer to $v$ so that we know we should resume at $v$ if the program corresponding to $p$ is resumed with increased quotas. This scheduler incurs an $\mathcal{O}(\log n)$-time additive overhead at Lines 9 and 16.

It remains to analyze the complexity of Algorithm 1. By Lemmas 4.2 and 4.3 and Theorem 2.1, the oracle calls in Lines 10 to 13 make $\widetilde{\mathcal{O}}(\sqrt{r^{2i+2}(|X| + |Y|)})$ quantum queries and take $\widetilde{\mathcal{O}}(\sqrt{r^{2i+2}(|X| + |Y|)} + (r^{2i+2})^2)$ quantum time (including the amplification of success probability). We charge these queries to q-tokens burnt at Line 9, whereas the running time is charged to t-tokens burnt at Line 9. Lines 4 and 5 make $\mathcal{O}(1 + |Y|)$ quantum queries and run in $\mathcal{O}(1 + |Y|)$ quantum time, which we charge to the q-tokens and t-tokens burnt at Line 2. No other subroutines make any quantum queries. The classical control instructions (including the scheduler implementation) take $\mathcal{O}(\log n) = \widetilde{\mathcal{O}}(1)$ time per iteration of the main **for** loop, which we can also charge to the number of tokens burnt. Overall, Lemma 4.4 implies that the total query complexity is $\widetilde{\mathcal{O}}(T_q(|X|, |Y|, d)) = \widetilde{\mathcal{O}}(\sqrt{dn})$, whereas the time complexity is $\widetilde{\mathcal{O}}(T_q(|X|, |Y|, d) + T_t(|X|, |Y|, d)) = \widetilde{\mathcal{O}}(\sqrt{dn} + d^2)$.

It remains to explain how to modify Algorithm 1 so that a witness sequence of edits is reported along with every distance. Each recursive call remembers locations of currently processed strings $X$ and $Y$ within the global inputs so that the edits reported use global position numbering. Technically, each call to $\mathsf{Solve}(X, Y)$, along with the answer $d$, reports a linked list of $d$ edits that allow transforming $X$ into $Y$. If the algorithm terminates at Line 4, we report insertions of $Y[1], \ldots, Y[y-1], Y[y+1], \ldots, Y[|Y|]$. If the algorithm terminates at Line 5 with $|Y| = 0$, we report a deletion of $X[1]$. If the algorithm terminates at Line 5 with $|Y| > 0$, we report a substitution of $X[1]$ for $Y[1]$ and insertions of $Y[2], \ldots, Y[|Y|]$. The program $A_i$ defined in Line 14 not only adds the distances but also concatenates the lists reported by the recursive calls (skipped calls correspond to empty lists). If the algorithm terminates at Line 18, we pass the list returned by $A$ along with the answer $d$. In all cases, the extra time needed to handle edits is proportional to the time complexity of control instructions. This completes the proof of Theorem 1.1.

### 4.2 Finding and Testing Anchors

Similar to [KPS21], we use connections between LZ77 factorization and edit distance alignments.

LEMMA 4.5. (DISJOINT ALIGNMENTS IMPLY COMPRESSION) *Consider strings $X, Y \in \Sigma^*$ and alignments $\mathcal{A} : X(i\,..\,j] \rightsquigarrow Y$ and $\mathcal{A}' : X(i'\,..\,j'] \rightsquigarrow Y$. If $\mathcal{A} \cap \mathcal{A}' = \emptyset$, then*

$$|\mathsf{LZ}(X(\hat{\imath}\,..\,\hat{\jmath}])| \leq |i - i'| + 2\mathsf{ed}_{\mathcal{A}}(X(i\,..\,j], Y) + 2\mathsf{ed}_{\mathcal{A}'}(X(i'\,..\,j'], Y) + 1$$

*holds for every fragment $X(\hat{\imath}\,..\,\hat{\jmath}]$ of $X$ with $\min\{i, i'\} \leq \hat{\imath} \leq \hat{\jmath} \leq \max\{j, j'\}$.*

*Proof.* Let $\mathcal{B} = (\mathcal{A}')^{-1} \circ \mathcal{A} : X(i\,..\,j] \rightsquigarrow\!\!\!\!\!\rightarrow X(i'\,..\,j']$ be an alignment obtained as a product of $(\mathcal{A}')^{-1}$ and $\mathcal{A}$. Note that $\mathsf{ed}_{\mathcal{B}}(X(i\,..\,j], X(i'\,..\,j']) \le \mathsf{ed}_{\mathcal{A}}(X(i\,..\,j], Y) + \mathsf{ed}_{\mathcal{A}'}(X(i'\,..\,j'], Y)$ and, for every $(x, x') \in \mathcal{B}$, there is $y \in [0\,..\,|Y|]$ such that $(x, y) \in \mathcal{A}$ and $(x', y) \in \mathcal{A}'$. Since $\mathcal{A}, \mathcal{A}'$ are disjoint, we must have $(x, y) \notin \mathcal{A}'$, and hence $x \neq x'$ for all $(x, x') \in \mathcal{B}$. By symmetry, we assume without loss of generality that $i < i'$; then, $x < x'$ holds for all $(x, x') \in \mathcal{B}$ and, in particular, $j < j'$. We consider two cases:

- If $\hat{\imath} \ge j$, then $|\mathsf{LZ}(X(\hat{\imath}\,..\,\hat{\jmath}])| \le \hat{\jmath} - \hat{\imath} \le j' - j \le i' - i + \mathsf{ed}_{\mathcal{B}}(X(i\,..\,j], X(i'\,..\,j']) \le (i' - i) + \mathsf{ed}_{\mathcal{A}}(X(i\,..\,j], Y) + \mathsf{ed}_{\mathcal{A}'}(X(i'\,..\,j'], Y)$.
- Otherwise, there a position $\hat{\imath}' \in [i'\,..\,j']$ such that $(\hat{\imath}, \hat{\imath}') \in \mathcal{B}$. The alignment $\mathcal{B}$ induces a decomposition of $X(\hat{\imath}'\,..\,j'] = f'_1 \cdots f'_z$ into $z \le 2 \cdot \mathsf{ed}_{\mathcal{B}}(X(\hat{\imath}\,..\,j], X(\hat{\imath}'\,..\,j']) + 1$ phrases, each of which is either a single character inserted or substituted under $\mathcal{B}$ or a fragment $X(x'\,..\,x'+s]$ such that $X(x\,..\,x+s] \simeq_{\mathcal{B}} X(x'\,..\,x'+s]$ for some $(x, x') \in \mathcal{B}$. Since $x < x'$ for all $(x, x') \in \mathcal{B}$, this implies an LZ-like factorization $X[\hat{\imath}+1] \cdots X[\hat{\imath}'] \cdot f'_1 \cdots f'_z$ of $X(\hat{\imath}\,..\,j']$. Hence, $|\mathsf{LZ}(X(\hat{\imath}\,..\,\hat{\jmath}])| \le |\mathsf{LZ}(X(\hat{\imath}\,..\,j'])| \le (\hat{\imath}' - \hat{\imath}) + z \le (i' - i) + \mathsf{ed}_{\mathcal{B}}(X(i\,..\,\hat{\imath}], X(i'\,..\,\hat{\imath}']) + 2 \cdot \mathsf{ed}_{\mathcal{B}}(X(\hat{\imath}\,..\,j], X(\hat{\imath}'\,..\,j']) + 1 \le (i' - i) + 2\mathsf{ed}_{\mathcal{B}}(X(i\,..\,j], X(i'\,..\,j']) + 1 \le (i' - i) + 2\mathsf{ed}_{\mathcal{A}}(X(i\,..\,j], Y) + 2\mathsf{ed}_{\mathcal{A}'}(X(i'\,..\,j'], Y) + 1$. ∎

LEMMA 4.6. *Consider strings $X, Y \in \Sigma^*$, an integer $k \ge 0$, and a pair $(x, y) \in [0\,..\,|X|] \times [0\,..\,|Y|]$. If $\mathsf{ed}(X, Y) \le k$, then $(x, y)$ is an edit anchor of $X$ and $Y$ if and only if it is an edit anchor of fragments $X' = X(i\,..\,j]$ and $Y' = Y(i\,..\,j + |Y| - |X|]$ defined in terms of the minimum $i \in [0\,..\,x]$ such that $|\mathsf{LZ}(\overline{X(i\,..\,x]})| \le 6k + 2$ and the maximum $j \in [x\,..\,|X|]$ such that $|\mathsf{LZ}(X(x\,..\,j])| \le 6k + 2$.*

*Proof.* Consider optimal alignments $\mathcal{A} : X \rightsquigarrow\!\!\!\!\!\rightarrow Y$ and $\mathcal{A}' : X' \rightsquigarrow\!\!\!\!\!\rightarrow Y'$ such that $(x, y) \in \mathcal{A} \cup \mathcal{A}'$. The monotonicity of edit distance guarantees $\mathsf{ed}_{\mathcal{A}'}(X', Y') \le \mathsf{ed}_{\mathcal{A}}(X, Y) \le k$.

We will prove the existence of $(x_\ell, y_\ell) \in \mathcal{A} \cap \mathcal{A}'$ such that $x_\ell \le x$ and $y_\ell \le y$. We proceed with a proof by contradiction and bound $|\mathsf{LZ}(\overline{X(i\,..\,x]})|$ by considering the following two cases:

- $(x, y) \in \mathcal{A}$. Suppose that $\mathcal{A}'$ aligns $X(i\,..\,x']$ with $Y(i\,..\,y]$, whereas $\mathcal{A}$ aligns $X(i'\,..\,x]$ with $Y(i\,..\,y]$. These alignments are disjoint; otherwise, their intersection point $(x_\ell, y_\ell) \in \mathcal{A} \cap \mathcal{A}'$ satisfies $(x_\ell, y_\ell) \le (x, y)$. By Lemma 4.5 applied to the alignment of $\overline{X(i\,..\,x']}$ and $\overline{Y(i\,..\,y]}$ and the alignment of $\overline{X(i'\,..\,x]}$ and $\overline{Y(i\,..\,y]}$, we have
$$|\mathsf{LZ}(\overline{X(i\,..\,x]})| \le |x - x'| + 2\mathsf{ed}_{\mathcal{A}}(X(i'\,..\,x], Y(i\,..\,y]) + 2\mathsf{ed}_{\mathcal{A}'}(X(i\,..\,x'], Y(i\,..\,y]) + 1.$$
- $(x, y) \in \mathcal{A}'$. Suppose that $\mathcal{A}'$ aligns $X(i\,..\,x]$ with $Y(i\,..\,y]$, whereas $\mathcal{A}$ aligns $X(i'\,..\,x']$ with $Y(i\,..\,y]$. These alignments are disjoint; otherwise, their intersection point $(x_\ell, y_\ell) \in \mathcal{A} \cap \mathcal{A}'$ satisfies $(x_\ell, y_\ell) \le (x, y)$. By Lemma 4.5 applied to the alignment of $\overline{X(i'\,..\,x']}$ and $\overline{Y(i\,..\,y]}$ and the alignment of $\overline{X(i\,..\,x]}$ and $\overline{Y(i\,..\,y]}$, we have
$$|\mathsf{LZ}(\overline{X(i\,..\,x]})| \le |x - x'| + 2\mathsf{ed}_{\mathcal{A}}(X(i'\,..\,x'], Y(i\,..\,y]) + 2\mathsf{ed}_{\mathcal{A}'}(X(i\,..\,x], Y(i\,..\,y]) + 1.$$

In both cases, we obtained
$$|\mathsf{LZ}(\overline{X(i\,..\,x]})| \le |x - x'| + 2\mathsf{ed}_{\mathcal{A}}(X, Y) + 2\mathsf{ed}_{\mathcal{A}'}(X', Y') + 1 \le 6k + 1,$$
where the last inequality follows from $|x - x'| \le |x - y| + |y - x'| \le \mathsf{ed}_{\mathcal{A}}(X, Y) + \mathsf{ed}_{\mathcal{A}'}(X', Y') \le 2k$. If $i > 0$, this implies $|\mathsf{LZ}(\overline{X(i-1\,..\,x]})| \le |\mathsf{LZ}(\overline{X(i\,..\,x]})| + 1 \le (6k + 1) + 1 = 6k + 2$, which contradicts the definition of $i$. Consequently, we may assume that $i = 0$. In that case, however, $(0, 0) \in \mathcal{A} \cap \mathcal{A}'$ is an intersection point satisfying $0 \le x$ and $0 \le y$. This completes the existence proof of $(x_\ell, y_\ell) \in \mathcal{A} \cap \mathcal{A}'$ such that $x_\ell \le x$ and $y_\ell \le y$. A symmetric argument yields $(x_r, y_r) \in \mathcal{A} \cap \mathcal{A}'$ such that $x_r \ge x$ and $y_r \ge y$.

If $(x, y) \in \mathcal{A}'$, then we can replace the part of $\mathcal{A}$ between $(x_\ell, y_\ell)$ and $(x_r, y_r)$ by the corresponding part in $\mathcal{A}'$ and obtain an optimal alignment of $X, Y$ that goes through $(x, y)$. Hence, if $(x, y)$ is an edit anchor of $X'$ and $Y'$, then it is an edit anchor of $X$ and $Y$. Symmetrically, if $(x, y) \in \mathcal{A}$, then we can replace the part of $\mathcal{A}'$ between $(x_\ell, y_\ell)$ and $(x_r, y_r)$ by the corresponding part in $\mathcal{A}$ and obtain an optimal alignment of $X', Y'$ that goes through $(x, y)$. Hence, if $(x, y)$ is an edit anchor of $X$ and $Y$, then it is an edit anchor of $X'$ and $Y'$. ∎

Now we prove Lemmas 4.2 and 4.3.

LEMMA 4.2. (FINDING AN ANCHOR: $\texttt{FindAnchor}(X, Y, k, x)$) *There exists a quantum algorithm that, given strings $X, Y \in \Sigma^{\leq n}$, an integer $k \geq 1$, and a position $x \in [0\,..\,|X|]$, finds a position $y \in [0\,..\,|Y|]$ such that $(x, y)$ is a $k$-edit anchor of $X, Y$. The algorithm has query complexity $\widetilde{\mathcal{O}}(\sqrt{kn})$ and time complexity $\widetilde{\mathcal{O}}(\sqrt{kn}+k^2)$.*

*Proof.* Define fragments $X' = X(i\,..\,j]$ and $Y' = Y(i\,..\,j+|Y|-|X|]$ as in Lemma 4.6. By monotonicity of $|\mathsf{LZ}(\cdot)|$ with respect to prefixes, we can compute positions $i$ and $j$ using binary search, with Theorem 1.2 employed to implement the $|\mathsf{LZ}(\cdot)| \leq 6k+2$ test on substrings of $X$ and $\overline{X}$. Overall, this step requires $\widetilde{\mathcal{O}}(\sqrt{kn})$ query complexity and time complexity. By Fact 2.3, we have

$$|\mathsf{LZ}(X')| \leq |\mathsf{LZ}(X(i\,..\,x])| + |\mathsf{LZ}(X(x\,..\,j])| \leq |\mathsf{LZ}(\overline{X(i\,..\,x]})| \cdot \mathcal{O}(\log n) + |\mathsf{LZ}(X(x\,..\,j])| \leq \mathcal{O}(k \log n).$$

Thus, using Theorem 1.2, we can compute $\mathsf{LZ}(X')$ in $\widetilde{\mathcal{O}}(\sqrt{kn})$ query complexity and time complexity. If $\mathsf{ed}(X', Y') \leq k$, then Fact 2.3 yields

$$|\mathsf{LZ}(Y')| \leq (|\mathsf{LZ}(X')| + k) \cdot \mathcal{O}(\log n) \leq \mathcal{O}(k \log^2 n).$$

Consequently, we can use Theorem 1.2 in $\widetilde{\mathcal{O}}(\sqrt{kn})$ query complexity and time complexity to compute $\mathsf{LZ}(Y')$ or report that $|\mathsf{LZ}(Y')|$ exceeds the $\mathcal{O}(k \log^2 n)$ threshold derived above. In the latter case, we conclude that $\mathsf{ed}(X, Y) \geq \mathsf{ed}(X', Y') > k$, so $(x, 0)$ trivially satisfies the definition of a $k$-edit anchor. Otherwise, we use Theorem 3.1 to check whether $\mathsf{ed}(X', Y') \leq k$ and, if so, retrieve an optimal sequence of edits transforming $X'$ into $Y'$. Since we already know the LZ-factorizations of $X'$ and $Y'$, this step takes $\widetilde{\mathcal{O}}(k^2)$ additional time complexity and zero query complexity. If $\mathsf{ed}(X', Y') > k$, then we return 0 again. Otherwise, our algorithm scans the list of edits transforming $X'$ into $Y'$ to derive and return $y \in [0\,..\,|Y|]$ such that $(x, y)$ belongs to the underlying optimal alignment $\mathcal{A}' : X' \rightsquigarrow Y'$. By Lemma 4.6, if $\mathsf{ed}(X, Y) \leq k$, then $(x, y)$ must be an edit anchor of $X$ and $Y$. ∎

LEMMA 4.3. (TESTING AN ANCHOR: $\texttt{IsAnchor}(X, Y, k, (x, y))$) *There exists a quantum algorithm that, given strings $X, Y \in \Sigma^{\leq n}$, an integer $k \geq 1$, and a pair $(x, y) \in [0\,..\,|X|] \times [0\,..\,|Y|]$,*

- *if $\mathsf{ed}(X, Y) \leq k$, decides whether $(x, y)$ is an edit anchor of $X, Y$;*
- *otherwise, returns an arbitrary answer.*

*The algorithm has query complexity $\widetilde{\mathcal{O}}(\sqrt{kn})$ and time complexity $\widetilde{\mathcal{O}}(\sqrt{kn}+k^2)$.*

*Proof.* The proof is similar to that of Lemma 4.2. First, we find the positions $i, j$ defined in Lemma 4.6. Next, we retrieve $\mathsf{LZ}(X(i\,..\,x]), \mathsf{LZ}(X(x\,..\,j])$, and $\mathsf{LZ}(X') = \mathsf{LZ}(X(i\,..\,j])$, as well as $\mathsf{LZ}(Y(i\,..\,y]), \mathsf{LZ}(Y(y\,..\,j+|Y|-|X|])$, and $\mathsf{LZ}(Y') = \mathsf{LZ}(Y(i\,..\,j+|Y|-|X|])$. If $\mathsf{ed}(X', Y') \leq k$, then the sizes of all these LZ factorizations are in $\mathcal{O}(k \log^2 n)$. Consequently, we can use Theorem 1.2 in $\widetilde{\mathcal{O}}(\sqrt{kn})$ query complexity and time complexity to either compute all these LZ factorizations or conclude that $\mathsf{LZ}(X', Y') > k$ (in that case, we return **false**). If $\mathsf{ed}(X', Y') \leq k$, then $(x, y)$ is an edit anchor for $(X', Y')$ if and only if $\mathsf{ed}(X(i\,..\,x], Y(i\,..\,y]) + \mathsf{ed}(X(x\,..\,j], Y(y\,..\,j+|Y|-|X|]) = \mathsf{ed}(X', Y') \leq k$, and our goal is to return **true** if and only if this condition holds. Consequently, we apply Theorem 3.1 in $\widetilde{\mathcal{O}}(k^2)$ additional time complexity (and zero query complexity) to evaluate the three edit distances involved in our test or discover that some of these distances exceed $k$.

It remains to prove the correctness of our algorithm. Either output is valid if $\mathsf{ed}(X, Y) > k$. Hence, we assume $\mathsf{ed}(X, Y) \leq k$ in the following. In this case, we must have $\mathsf{ed}(X', Y') \leq k$ by monotonicity of edit distance. Moreover, by Lemma 4.6, $(x, y)$ is an edit anchor of $X, Y$ if and only if it is an edit anchor of $X'$ and $Y'$. Thus, the algorithm correctly decides whether $(x, y)$ is an edit anchor of $X$ and $Y$. ∎

## 5 Quantum Algorithms for Lempel–Ziv Factorization

**5.1 Algorithms with Near-Optimal Query Complexity** This section provides two preliminary solutions with optimal and near-optimal query times. The first has optimal $\mathcal{O}(\sqrt{zn})$ query complexity but requires exponential time. The second has a near-optimal $\widetilde{\mathcal{O}}(\sqrt{zn})$ query complexity but requires $\widetilde{\mathcal{O}}(n)$ time. The second algorithm introduces ideas that will be expanded on in Section 5.2 for our main algorithm with $\widetilde{\mathcal{O}}(\sqrt{zn})$ query and time complexity.

**5.1.1 Achieving Optimal-Query Complexity in Exponential Time** A naive approach is to first obtain the input string from the oracle (in the worst case using $n$ oracle queries). Then, any compressed representation can be computed without further input queries. The first approach discussed here shows how to find the input string using fewer queries, specifically $\mathcal{O}(\sqrt{zn})$ queries for binary strings. We will prove this query complexity is optimal in Section 7. This algorithm is based on a solution for the problem of identifying an oracle (in our case, an input string) in the minimum number of oracle queries by Kothari [Kot14]. Kothari's solution builds on a previous 'halving' algorithm by Littlestone [Lit87].

We next describe the basic halving algorithm as applied to our problem. Assuming that $z$ is known, we enumerate all binary strings of length $n$ with at most $z$ LZ77 factors. Call this set $\mathcal{S}$. Since an encoding with $z$ factors requires at most $2z \log n$ bits, there are at most $\sum_{i=0}^{2z \log n} 2^i = 2^{2z \log n+1} - 1 = 2n^{2z} - 1$ such strings in $\mathcal{S}$. We construct a string $M$ of length $n$ from $\mathcal{S}$, where $M[i] = 1$ if at least half of strings in $\mathcal{S}$ are 1 at the $i^{th}$ position, and $M[i] = 0$ otherwise. Note that the construction of $M$ requires time exponential in $z$ but does not require any oracle queries. Grover's search is then used to find a mismatch if one exists between $M$ and the oracle string with $\mathcal{O}(\sqrt{n})$ queries. If a mismatch occurs at position $i$, we can then eliminate at least half of the potential strings in $\mathcal{S}$. We repeat this process until no mismatches are found, at which point we have completely recovered the oracle (input string). Known algorithms can then obtain all compressed forms of text.

Naively applying this approach would result in an algorithm with $\mathcal{O}(\sqrt{n} \log |\mathcal{S}|) = \mathcal{O}(z\sqrt{n} \log n)$ query complexity. Kothari's improvements on this basic halving algorithm give us a quantum algorithm that uses $\mathcal{O}(\sqrt{n \log |\mathcal{S}|/ \log n}) = \mathcal{O}(\sqrt{zn})$ input queries. We can avoid assuming the knowledge of $z$ by progressively trying different powers of 2 as our guess of $z$, still resulting in $\sum_{i=0}^{\log z} \mathcal{O}(\sqrt{2^i n}) = \mathcal{O}(\sqrt{zn})$ queries overall. As noted above, this approach is not time-efficient.

**5.1.2 Achieving Near-Optimal Query Complexity in Near-Linear Time** An algorithm with a similar query complexity and far improved time complexity is possible by using a more specialized approach. Specifically, one can obtain the non-overlapping LZ77 factorization. For non-overlapping LZ77, every factor, say $X[s_i \,..\, s_i+\ell_i)$, that is not a new symbol must reference a previous occurrence completely contained in $X[1 \,..\, s_i)$. This only increases the size of this factorization by at most a logarithmic factor. That is, if $z_{no}$ is the number of factors for the non-overlapping LZ77 factorization, then $z \leq z_{no} \leq \mathcal{O}(z \log n)$ [Nav21]. This factorization can be converted into other compressed forms in near-linear time, as described in Section 5.3.

We obtain the factorization by processing $X$ from left to right as follows: Suppose inductively that we have determined the factors $F_1, F_2, \ldots, F_{i-1}$, and we want to obtain the $i^{th}$ factor. Let $s_i$ denote the starting index of the $i^{th}$ factor and $\ell_i$ its length. Assume that we have the prefixes $X[1 \,..\, k]$, for $k \in [1 \,..\, s_i)$, sorted in co-lexicographic order. To find the next factor $X[s_i \,..\, j]$, we apply exponential search[5] on $j$. To evaluate a given $j$ we use binary search on the sorted set of prefixes. To compare a prefix $X[1 \,..\, k]$, we find the rightmost mismatch of the substrings $X[s_i \,..\, j]$ and $X[k-(j-s_i) \,..\, k]$. If no rightmost mismatch is found, then $X[s_i \,..\, j]$ has occurred previously as a substring, and we continue the exponential search on $j$. Otherwise, we compare the symbol at the rightmost mismatch to identify which half of the sorted set of prefixes to continue the binary search. If $\ell_i$ is the length of $i^{th}$ factor found, this requires $\mathcal{O}(\log^2 n \cdot \sqrt{\ell_i})$ queries and time.

To proceed to the $(i+1)^{th}$ factor, we now must obtain the co-lexicographically sorted order of the $\ell_i$ new prefixes. This can be done using a standard linear-time suffix tree construction algorithm. Specifically, if we consider the suffix tree of the reversed text $\overline{X}$, we are prepending $\ell_i$ symbols to a suffix of $\overline{X}$. These are accessed from either the oracle directly only in the case the new factor is a new symbol, and otherwise from the previously obtained string. Since we are prepending to $\overline{X}$, a right-to-left suffix construction algorithm such as McCreight's [McC76] can be used.

The query complexity is $\sum_{i=1}^{z_{no}} \sqrt{\ell_i} \cdot \log^2 n = \widetilde{\mathcal{O}}(\sum_{i=1}^{z_{no}} \sqrt{\ell_i})$. At the same time, we have $\sum_{i=1}^{z_{no}} \ell_i = n$, so the sum is maximized when each $\ell_i = \frac{n}{z_{no}}$ making $\sum_{i=1}^{z_{no}} \sqrt{\ell_i} \leq \sqrt{z_{no} n}$. Hence, the query complexity is $\widetilde{\mathcal{O}}(\sqrt{zn})$. The time complexity is $\mathcal{O}(n + \log^3 n \cdot \sqrt{z_{no} n})$, which is $\widetilde{\mathcal{O}}(n)$. We will focus for the rest of this section on developing these ideas and utilizing more complex data structures to obtain a sublinear-time algorithm.

[5] Recall that exponential search checks ascending powers of 2 until an interval $[2^{x-1}, 2^x]$ for some $x \geq 1$ containing the solution is found, at which point binary search is applied to the interval.

**5.2 Main Algorithm: Optimal Query and Time Complexity** On a high level, the algorithm will proceed very much like the near-linear-time algorithm from Section 5.1.2. It proceeds from left to right finding the next factor and utilizes a co-lexicographically sorted set of prefixes of $X$. After the next factor is found, a set of new prefixes of $X$ is added to this sorted set. However, we face two major obstacles: (i) we cannot afford to explicitly maintain a sorted order of all prefixes needed to check all possible previous substrings efficiently; (ii) if we utilize a factorization other than LZ77, like LZ77-End, where fewer potential positions have to be checked, then the monotonicity of being a next factor is lost, i.e., for LZ77-End, $X[s_i\,..\,j]$ may have occurred as a substring ending at a previous factor, but $X[s_i\,..\,j)$ may not have occurred as a substring ending at a previous factor.

To overcome these problems, we introduce a new factorization scheme that extends the LZ-End factorization scheme discussed in Section 2. It allows for more potential places ending locations for each new factor obtained by the algorithm.

**5.2.1 LZ-End+$\tau$ Factorization** Let $\tau \geq 1$ be an integer parameter. The LZ-End+$\tau$ factorization of the string $X$ is constructed from left to right. Initially, $i \leftarrow 1$. For $i \geq 1$, if $X[i]$ does not occur in $X[1\,..\,i)$, then we make $X[i]$ a new factor and set $i \leftarrow i+1$. Otherwise, let $j$ be the largest index such that $X[i\,..\,j]$ has an occurrence ending at either the last position of an earlier factor or at a position $k < i$ such that $k \equiv 1 \pmod{\tau}$. Let $z_{e+\tau}$ denote the number of factors created by the LZ-End+$\tau$ factorization.

Note that there exist strings where $z_e < z_{e+\tau}$. The smallest binary string example where this is true is 00010011011, which has an LZ-End factorization with seven factors 0, 0, 0, 1, 001, 1, 011 and an LZ-End+$\tau$ for $\tau = 2$ with eight factors 0, 0, 0, 1, 001, 10, 1, 1. Loosely speaking, the LZ77-End+$\tau$ algorithm can be 'tricked' into taking a longer factor earlier on; in this case, the factor '10' which is possible for LZ77-End+$\tau$ but not LZ77-End, and limits future choices. Fortunately, the same bounds in terms of $z$ established by Kempa and Saha [KS22] for $z_e$ also hold for $z_{e+\tau}$.

LEMMA 5.1. *Let $z_{e+\tau}$ (resp., $z$) denote the number of factors in the LZ-End+$\tau$ (resp., LZ77) factorization of a given text $X[1\,..\,n]$. Then, $z_{e+\tau} = \mathcal{O}(z \log^2 n)$.*

*Proof.* We outline Kempa and Saha's proof of the bound for LZ-End and why it continues to hold for LZ-End+$\tau$. We refer the reader to [KS22] for more details. In the proof, a factor is considered *special* if its length is at least half the length of the previous factor. Every special factor is assigned a set of substrings of $X$. In particular, if one of these special factors is of length $\ell$, it is assigned $\ell$ substrings of length $2^k$ for $k \in [0\,..\,\lceil \log n \rceil + 4]$. The bound then follows by showing that: (i) each distinct substring of length $2^k$ is assigned to at most two factors, and (ii) all substrings assigned to a factor are distinct. Both (i) and (ii) use a proof by contradiction and work because a longer factor is possible, i.e., a longer substring occurs ending at a factor end.

For (i), if a substring is assigned to three or more factors, with $X[s_i\,..\,s_i+\ell_i)$ being the leftmost and $X[s_j\,..\,s_j+\ell_j)$ the rightmost, then it is shown that, for some $\delta > 0$, there exists a substring $X[s_{j-1}\,..\,s_{j-1}+\ell_{j-1}+\delta)$ that also ends at a previous factor, contradicting that $X[s_{j-1}\,..\,s_{j-1}+\ell_{j-1})$ was chosen as a factor. This argument is based on the lengths of the substrings and factor $X[s_j\,..\,s_j+\ell_j)$ being special. These properties continue to hold for LZ-End+$\tau$. Moreover, because our LZ-End+$\tau$ is also greedy and takes the largest factor possible, it could have used $X[s_{j-1}\,..\,s_{j-1}+\ell_{j-1}+\delta)$ as a factor instead of $X[s_{j-1}\,..\,s_{j-1}+\ell_{j-1})$. Hence, we arrive at the same contradiction.

For (ii), the contradiction is achieved by showing that, if some substring is assigned to $X[s_j\,..\,s_j+\ell_j)$ two or more times, then an instance of $X[s_j\,..\,s_j+\ell_j+\delta)$ for some $\delta > 0$ exists ending at a previous factor. This argument is based on the lengths of the substrings and the repeated substring causing periodicity. It continues to hold for LZ-End+$\tau$. Again, because LZ-End+$\tau$ is also greedy and could use $X[s_j\,..\,s_j+\ell_j+\delta)$ as a factor instead of $X[s_j\,..\,s_j+\ell_j)$, we arrive at the same contradiction. ∎

Next, we describe how new LZ77-End+$\tau$ factors of $X$ are obtained by using the concept of the $\tau$-*far property* and a dynamic longest common extension (LCE) data structure. Following this, we describe how the co-lexicographically sorted prefixes required by the algorithm are maintained.

**5.2.2 Maintaining the Colexicographic Ordering of Prefixes** The first factor $F_1$ is always $X[1]$. Assume inductively that the factors $F_1, F_2, \ldots, F_{i-1}$ have already been determined. Recall that, for factors $F_j$ of the form $(\alpha)$ with $\alpha \in \Sigma$, we also store $(s_j, 1)$, where $s_j$ is the starting position of the $j^{th}$ factor in $X$. We assume

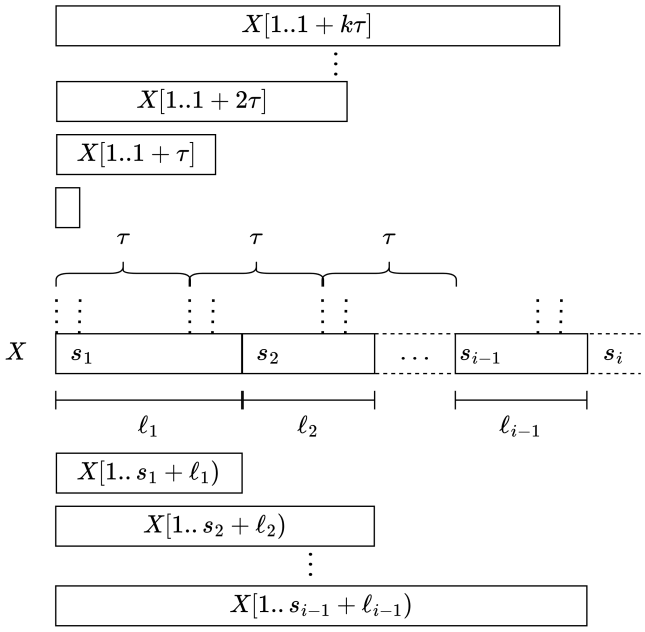


Figure 2: The colexicographic order of the prefixes $X[1\,..\,s_1+\ell_1)$, $X[1\,..\,s_2+\ell_2)$, …, $X[1\,..\,s_{i-1}+\ell_{i-1})$ (shown below $X$) and $X[1\,..\,1], X[1\,..\,\tau+1], \ldots, X[1\,..\,k\tau+1]$ where $k$ is the largest natural number such that $1+k\tau < s_i$ (shown above $X$) are known prior to iteration $i$.

inductively that we have the colexicographically sorted order of prefixes of

$$\begin{aligned}\mathcal{P}_{i-1} \coloneqq \{&X[1\,..\,s_j+\ell_j-1] \mid (s_j,\ell_j) = F_j, 1 \le j \le i-1\}\\ &\cup \{X[1\,..\,j] \mid 1 \le j \le s_{i-1}+\ell_{i-1}-1,\quad j \equiv 1 \pmod{\tau}\}.\end{aligned}$$

See Figure 2 for an illustration of the prefixes contained in $\mathcal{P}_i$. For each of them, we store the ending position of the prefix.

The following section shows how to obtain the factor $F_i$. For now, suppose we just determined the $i^{th}$ factor starting position $s_i$. After the factor length $\ell_i$ is found, we need to determine where to insert the prefixes $X[1\,..\,s_i+\ell_i-1]$ and $X[1\,..\,j]$ for $j \in [s_i\,..\,s_i+\ell_1-1]$ such that $j \equiv 1 \pmod{\tau}$, in the colexicographically sorted order of $\mathcal{P}_{i-1}$ to create $\mathcal{P}_i$. To do this, we use the dynamic longest common extension (LCE) data structure of Nishimoto et al. [NII+16] (see Lemma 5.2).

Lemma 5.2. (Dynamic LCE data structure [NII+16]) *An LCE query on a text $S[1\,..\,m]$ consists of two indices $i$ and $j$ and returns the largest $\ell$ such that $S[i\,..\,i+\ell) = S[j\,..\,j+\ell)$. There exists a data structure that requires $\mathcal{O}(m)$ time to construct, supports LCE queries in $\widetilde{\mathcal{O}}(1)$ time, and supports insertion of either a substring of $S$ or a single character into $S$ at an arbitrary position in $\widetilde{\mathcal{O}}(1)$ time*[6].

The main idea is to use the above dynamic LCE structure over the reverse of the prefix of $X$ found thus far. We initialize the dynamic LCE data structure with the first LZ-End+$\tau$ factor of $X$, which is a single character. For every factor found after that, we prepend the reversed factor to the current reversed prefix and update the data structure, all in $\widetilde{\mathcal{O}}(1)$ time. In particular, if the $i^{th}$ factor of $X$ found is a new character, we prepend that character to our dynamic LCE structure for $S \coloneqq \overline{X[1\,..\,s_i)}$. If the $i^{th}$ factor found is $X[s_i\,..\,s_i+\ell_i-1] = X[x\,..\,y]$, for $x, y \in [1\,..\,s_i)$, then we prepend the substring $\overline{X[x\,..\,y]} = S[s_i-y\,..\,s_i-x]$ to string representation of our dynamic LCE structure. Once the reversed $i^{th}$ factor is prepended to the reversed prefix in the dynamic LCE structure, to compare the colexicographic order of the new prefixes in $\mathcal{P}_i$, we find the LCE of the two reversed prefixes being

[6]Polylogarithmic factors here are with respect to the final string length after all insertions.

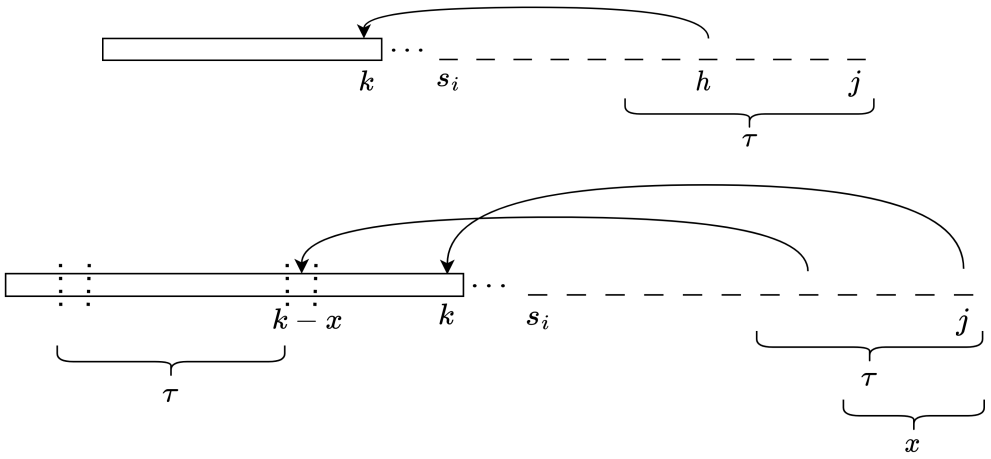


Figure 3: The two cases given the proof of Lemma 5.3. On the top is Case 1, where $X[s_i\,..\,j]$ is not a potential factor. On the bottom is Case 2, where $X[s_i\,..\,j]$ is a potential factor. Here, we are implying $k - x \equiv 1 \pmod{\tau}$.

compared and compare the symbol in the position after their furthest match. Applying this comparison technique and binary search on $\mathcal{P}_{i-1}$, we determine where each prefix in $\mathcal{P}_i \setminus \mathcal{P}_{i-1}$ should be inserted in the sorted order in polylogarithmic time.

**5.2.3 Finding the Next LZ-End+$\tau$ Factor** We now show how to obtain the new factor $F_i = (s_i, \ell_i)$. Firstly, $s_i = s_{i-1} + \ell_{i-1}$. We say $X[s_i\,..\,h]$ is a *potential factor* if either $h = s_i$ and $X[s_i\,..\,h] \in \Sigma$ is the leftmost occurrence of a symbol in $X$ or $X[s_i\,..\,h] = X[x\,..\,y]$, where $y < s_i$ and $y$ is the end of a previous factor or $y \equiv 1 \pmod{\tau}$. We say the *$\tau$-far property* holds for an index $j \geq s_i$ if there exists $h$ such that $j - \tau \leq h \leq j$ and $X[s_i\,..\,h]$ is a potential factor.

Lemma 5.3. (Monotonicity of $\tau$-far property) *When finding a new factor starting at position $s_i$, if the $\tau$-far property holds for $j > s_i$, then it holds for $j - 1$.*

*Proof.* There are two cases; see Figure 3. Case 1: If $X[s_i\,..\,j]$ is not a factor, since the $\tau$-far property holds for $j$, there exists an $h \in [j - \tau\,..\,j)$ and $k < s_i$ such that $X[s_i\,..\,h] = X[k - (h - s_i)\,..\,k]$ is a potential factor. Then, this $h$ demonstrates that the $\tau$-far property holds for $j - 1$. Case 2: Suppose instead that $X[s_i\,..\,j]$ is a potential factor and matches some $X[k - (j - s_i)\,..\,k]$, where $k$ is the last position in a previous factor or $k < s_i$ and $k \equiv 1 \pmod{\tau}$. If $j - s_i + 1 > \tau$, then there exists some $x \in [1\,..\,\tau]$ such that $k - (j - s_i) \leq k - x < k$ and $k - x \equiv 1 \pmod{\tau}$; hence, $X[s_i\,..\,j - x] = X[k - (j - s_i)\,..\,k - x]$, making $X[s_i\,..\,j - x]$ a potential factor. Since $j - 1 - \tau \leq j - x \leq j - 1$, the $\tau$-far property holds for $j - 1$. If instead $j - s_i + 1 \leq \tau$, then $j - \tau \leq s_i \leq j$ and $X[s_i\,..\,s_i]$ is always a potential factor since either it is the first occurrence of a symbol or we can refer to the factor created by the first occurrence of $X[s_i]$. This proves that the property still holds for $j - 1$. ∎

By Lemma 5.3, monotonicity holds for the $\tau$-far property when trying to find the next factor starting at position $s_i$. Thus, to find the largest $j$ such that the $\tau$-far property holds, we can now use exponential search. At its core, we need to determine whether the $\tau$-far property holds for a given $j > s_i$. Once this largest $j$ is determined, the largest $h \in [\max(s_i, j - \tau)\,..\,j]$ such that $X[s_i, h]$ is a potential factor must be determined as well.

We show a progression of algorithms to accomplish the above task. Firstly, we make some straightforward, yet crucial, observations. Let $S$ be any string. Since $\mathcal{P}_{i-1}$ is colexicographically sorted, all prefixes that have the same string (say $S$) as a suffix can be represented as a range of indices. This range is empty when $S$ is not a suffix of any prefix in $\mathcal{P}_{i-1}$. Moreover, this range can always be computed in $\widetilde{\mathcal{O}}(|S|)$ time using binary search. However, if $S$ has an occurrence within the $X[1\,..\,s_i)$ (i.e., the prefix seen thus far) and is specified by the start and end position of that occurrence, we can use LCE queries and improve the time for finding the range to $\widetilde{\mathcal{O}}(1)$.

**Next factor in $\widetilde{\mathcal{O}}(\tau + \ell_i)$ time:** For $h \in [\max(s_i, j - \tau)\,..\,j]$, let $k_h \in [0\,..\,h - s_i + 1]$ be the largest value such that $X[h - k_h + 1\,..\,h]$ is a suffix of a prefix in $\mathcal{P}_{i-1}$. We initialize $h = j$. Since the prefixes in $\mathcal{P}_{i-1}$ are

co-lexicographically sorted, we can find $k_h$ in $\widetilde{\mathcal{O}}(k_h)$ time by using binary search on $\mathcal{P}_{i-1}$. To do so, symbols are prepended one by one and binary search is used to check if the corresponding sorted index range of $\mathcal{P}_{i-1}$ is non-empty.

Next, we compute $k_h$ for $h \in [\max(s_i, j-\tau)\,..\,j)$ in the descending order $h$. We keep track of $h' := \arg\min_{y\in[h+1\,..\,j]}(y - k_y)$. If $h' - k_{h'} + 1 \leq h$, then $X[h' - k_{h'} + 1\,..\,h]$ has an occurrence in $X[1\,..\,s_i)$, and we can now use LCE queries to determine the range of $X[h' - k_{h'} + 1\,..\,h]$ in $\mathcal{P}_{i-1}$. If this range is empty, we conclude that $k_h < k_{h'} - (h' - h)$ and LCE can be used to find $k_h$ in $\widetilde{\mathcal{O}}(1)$ time. Otherwise, we proceed by prepending symbols one by one until $k_h$ is found.

The time per $h \in [\max(s_i, j-\tau)\,..\,j]$ is $\widetilde{\mathcal{O}}(1)$ for LCE queries, in addition to $\widetilde{\mathcal{O}}(x_h)$ where $x_h$ is the number of symbols we prepended for $h$. Since we always use the smallest $h' - k_{h'}$ value seen thus far, $\sum_{h=j-\tau}^{j} x_h \leq j - s_i = \mathcal{O}(\ell_i)$. This makes it so checking if the $\tau$-far property holds for $j$ takes $\widetilde{\mathcal{O}}(\ell_i)$ time. The algorithm also identifies the rightmost $h \in [\max(s_i, j-\tau)\,..\,j]$ such that $X[s_i\,..\,h]$ is a potential factor (if one exists). This only provides at best a near-linear time algorithm.

**Next factor in $\widetilde{\mathcal{O}}(\sqrt{\tau\ell_i})$ time:** Instead of prepending characters individually and using binary search after exhausting the reach of the LCE queries, we can instead find the rightmost mismatch and then use binary search on $\mathcal{P}_{i-1}$. Specifically, suppose that, for a given $h \in [\max(s_i, j-\tau)\,..\,j]$, we apply the LCE query and identify a non-empty range of prefixes in $\mathcal{P}_{i-1}$ with $X[w\,..\,h]$ as a suffix. On this set of prefixes, we continue the search from $w-1$ downward using exponential search and identifying whether a mismatch occurs with the right-most mismatch algorithm.

For $h \in [j-\tau\,..\,j]$, let $x_h$ now be the number of characters searched using exponential search and the right-most mismatch algorithm. As before $\sum_{h=j-\tau}^{j} x_h = \widetilde{\mathcal{O}}(j - s_i)$. The total time required for this is logarithmic factors from $\sum_{h=j-\tau}^{j} \sqrt{x_j} = \widetilde{\mathcal{O}}(\sqrt{\tau\ell_i})$. This will give us a sub-linear time algorithm if we choose $\tau$ appropriately; however, it will not be sufficient to obtain our goal.

**Next factor in $\widetilde{\mathcal{O}}(\tau + \sqrt{\ell_i})$ time:** Here we do not apply the rightmost-mismatch search for every $h \in [\max(s_i, j-\tau)\,..\,j]$. Instead, for each $h$, we identify a set of prefixes in $\mathcal{P}_{i-1}$ such that $X[s_i\,..\,h]$ shares a suffix of length at least $d_h = h - \max(s_i, j-\tau) + 1$. This set is represented by the range of indices, $[s_h, e_h]$, in the sorted $\mathcal{P}_{i-1}$ corresponding to prefixes sharing this suffix of length $d_h = h - \max(s_i, j-\tau) + 1$. By using the same LCE technique and prepending and stopping at index $\max(s_i, j-\tau)$, this can be accomplished in $\widetilde{\mathcal{O}}(\tau)$ time. After this, we have a set of ranges in $\mathcal{P}_{i-1}$. Note that for a given $h$, if we delete the last $d_h$ characters in each prefix represented in $[s_h\,..\,e_h]$, they remain co-lexicographically sorted. We want to search for $X[s_i\,..\,j-\tau)$ as a suffix on these ranges, each with their appropriate suffix removed. Since the rightmost-mismatch algorithm is costly, we can first merge these ranges (each with their appropriate suffix removed), and then use binary search. However, merging these sorted ranges would be too costly. Instead, we can take advantage of the following lemma to avoid this cost.

LEMMA 5.4. ([FIL20]) *Given $\tau$ sorted arrays $A_1, \ldots, A_\tau$ of $n$ elements in total, the $x^{th}$ largest element in the array formed by merging them can be found using $\mathcal{O}(\tau \log \tau \cdot \log(n/\tau))$ comparisons.*

Using the LCE data structure to compare any to prefixes, the $x^{th}$ largest element in the merged array can be found in $\widetilde{\mathcal{O}}(\tau)$ time. Using Lemma 5.4, we can find whichever rank prefix in the subset of $\mathcal{P}_{i-1}$ we are concerned with, then find the rightmost mismatch and compare it to $X[s_i\,..\,j-\tau)$. Doing so, the total time needed for obtaining the next factor is $\widetilde{\mathcal{O}}(\tau + \sqrt{\ell_i})$.

### 5.2.4 Time and Query Complexity

Taken over the entire string, the time complexity of finding the factors and updating the sorted order of the newly added prefixes is up to logarithmic factors bound by

$$\tfrac{n}{\tau} + z_{e+\tau} + \sum_{i=1}^{z_{e+\tau}} (\tau + \sqrt{\ell_i}) \leq \tfrac{n}{\tau} + z_{e+\tau} + \tau z_{e+\tau} + \sqrt{z_{e+\tau} n} = \mathcal{O}(\tfrac{n}{\tau} + \tau z_{e+\tau} + \sqrt{z_{e+\tau} n}),$$

where the inequality follows from $\sum \ell_i = n$. Combined with Lemma 5.1, which bounds $z_{e+\tau}$ to be logarithmic factors from $z$, and a logarithmic number of repetitions of each call to Grover's algorithm, the total time complexity is $\widetilde{\mathcal{O}}(\sqrt{zn} + \tau z + \frac{n}{\tau})$. To minimize the time complexity, we should set $\tau = \sqrt{n/z}$, bringing the total time to the desired $\widetilde{\mathcal{O}}(\sqrt{zn})$.

Note that we do not know $z$ in advance to set $\tau$. However, the desired time complexity can be obtained by increasing our guess of $z$ as follows: Let $z_{guess}$ be initially 1, set $\tau_{guess} = \lceil\sqrt{n/z_{guess}}\rceil$, and run the above algorithm until either the entire factorization of the string $X$ is obtained or the number of factors encountered is greater than $z_{guess}$. For a given $z_{guess}$, the time complexity is bound by $\widetilde{\mathcal{O}}(\sqrt{z_{guess}n} + \tau_{guess}z_{guess} + \frac{n}{\tau_{guess}})$, which is $\widetilde{\mathcal{O}}(\sqrt{z_{guess}n})$. If a complete factorization of $X$ is not obtained, we set $z_{guess} \leftarrow 2 \cdot z_{guess}$, similarly update $\tau_{guess}$, and repeat our algorithm for the new $\tau_{guess}$. The total time taken over all guesses is logarithmic factors from $\sqrt{n}\sum_{i=1}^{\lceil \log z \rceil}(2^i)^{\frac{1}{2}}$ which, again, is $\widetilde{\mathcal{O}}(\sqrt{zn})$.

The following lemma summarizes our result on LZ-End+$\tau$ factorization.

LEMMA 5.5. *Given a text $X$ of length $n$ having $z$ LZ77 factors, there exists a quantum algorithm that obtains the LZ-End+$\tau$ factorization of $X$ in $\widetilde{\mathcal{O}}(\sqrt{zn})$ time and input queries.*

**5.3 Obtaining the LZ77, SLP, RL-BWT Encodings** To obtain the other compressed encodings, we utilize the following result by Kempa and Kociumaka, stated here as Lemma 5.6. We need the following definitions: a factor $F_i = (s_i, \ell_i)$ is called *previous factor* if $X[s_i \,..\, s_i+\ell_i) = X[j \,..\, j+\ell_i)$ for some $j < s_i$. We say a factorization $X = F_1, \ldots, F_f$ of a string is *LZ77-like* if each factor $F_i$ is non-empty and $|F_i| > 1$ implies $F_i$ is a previous factor. Note that LZ-End+$\tau$ is LZ77-like with $f = \mathcal{O}(z \log^2 n)$ as shown in Lemma 5.1.

LEMMA 5.6. ([KK22] THM. 6.11) *Given an LZ77-like factorization of a string $X[1 \,..\, n]$ into $f$ factors, we can in $\mathcal{O}(f \log^4 n)$ time construct a data structure that, for any pattern $P$ represented by its arbitrary occurrence in $X$, returns the leftmost occurrence of $P$ in $X$ in $\mathcal{O}(\log^3 n)$ time.*

Starting with the LZ-End+$\tau$ factorization obtained in Section 5.2, we construct the data structure from Lemma 5.6. To obtain the LZ77 factorization, we again work from left to right and apply exponential search to obtain the next factor. In particular, if the start of our $i^{th}$ factor is $s_i$ and $X[s_i \,..\, s_i + \ell_i)$ if the leftmost occurrence of the substring is at position $j < s_i$, then we continue the search by increasing $\ell$. Since $\mathcal{O}(\log^3 n)$ time is used per query, we get that $\mathcal{O}(\log^4 n)$ time is used to obtain each new factor. Therefore, once the data structure from Lemma 5.6 is constructed, the required time to obtain the LZ77 factorization is $\mathcal{O}(z \log^4 n)$. The total time complexity of constructing all LZ77 factorization starting from the oracle for $X$ is $\widetilde{\mathcal{O}}(\sqrt{z_{e+\tau}n} + z)$, which is $\widetilde{\mathcal{O}}(\sqrt{zn})$, as summarized below.

THEOREM 1.2. (LZ77 FACTORIZATION) *There is a quantum algorithm that, given a quantum oracle access to an unknown string $X \in \Sigma^n$, computes the LZ factorization $\mathsf{LZ}(X)$ using $\widetilde{\mathcal{O}}(\sqrt{zn})$ query and time complexity, where $z := |\mathsf{LZ}(X)|$ is the size of the factorization.*

To obtain the RL-BWT of the text we directly apply an algorithm by Kempa and Kociumaka. In particular, they provide a Las-Vegas randomized algorithm that, given the LZ77 factorization of a text $X$ of length $n$, computes its RL-BWT in $\mathcal{O}(z \log^8 n)$ time (see Theorem 5.35 in [KK22]). Combined with $r = \mathcal{O}(z \log^2 n)$ [KK22], we obtain the following result.

COROLLARY 1.3. (RUN-LENGTH-ENCODED BWT) *There is a quantum algorithm that, given a quantum oracle to an unknown string $X \in \Sigma^n$, computes the run-length-encoded Burrows–Wheeler transform of $X$ using $\widetilde{\mathcal{O}}(\sqrt{rn})$ query and time complexity, where $r$ is the number of runs in the BWT.*

Obtaining a balanced CFG of size $\widetilde{\mathcal{O}}(z)$ is similarly an application of previous results, and can be derived by using either the original LZ77 to balanced grammar conversion algorithm of Rytter [Ryt03], or more recent results for converting LZ77 encodings to grammars [KL21], and even balanced run-length straight-line programs [KK22].

## 6 Compressed Text Indexing and Applications

We start this section having obtained the RL-BWT and the $\widetilde{\mathcal{O}}(z)$-space LCE data structure for the input text. There are two main stages to the remaining algorithm for obtaining a suffix array index. The first is to obtain a less efficient index with fast construction in time $\widetilde{\mathcal{O}}(\sqrt{rn})$, which supports SA and ISA queries in time $\widetilde{\mathcal{O}}(\sqrt{n/r})$. We accomplish this by applying a form of prefix doubling and alphabet replacement. These techniques allow us to 'shortcut' the LF-mapping described in Section 2. Using this shortcutted LF-mapping, we then sample the suffix array values every $\tau$ text indices apart, where $\tau = \sqrt{n/r}$, similar to the construction of the original FM-index. Once this less efficient index construction is complete, we move on to build the other indexes in [GNP20].

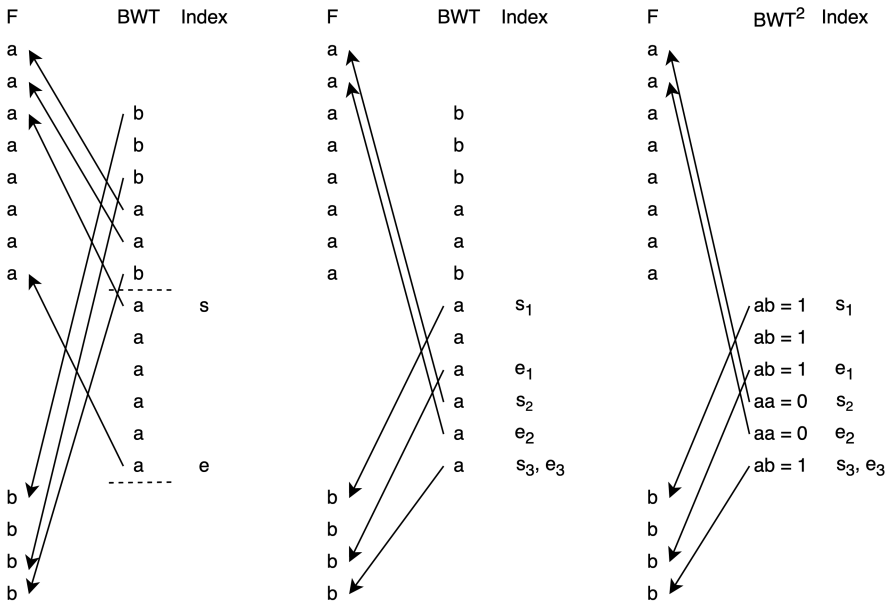


Figure 4: (Left) The initial LF-mapping from the interval $[s\,..\,e]$. (Middle) The intervals created for $[s\,..\,e]$ by the $\mathrm{LF}^2$ pull-back. (Right) Alphabet replacement is applied to each interval to create $\mathrm{BWT}^2$. Note that the contents of the F-column are not important and are not used.

**6.1 Computing $\mathrm{LF}^\tau$ and Suffix Array Samples** Recall that the LF-mapping of an index $i$ of the BWT is defined as $\mathrm{LF}[i] = \mathrm{ISA}[\mathrm{SA}[i]-1]$. The RL-BWT can be equipped rank-and-select structures in $\widetilde{\mathcal{O}}(r)$ time to support computation of the LF-mapping of a given index in $\widetilde{\mathcal{O}}(1)$ time.

For a given BWT run corresponding to the interval $[s\,..\,e]$, we have for $i \in [s\,..\,e)$ that $\mathrm{LF}[i+1] - \mathrm{LF}[i] = 1$, i.e., intervals contained in BWT runs are mapped on to intervals by the LF-mapping. If we applied the LF-mapping again to each $i \in [\mathrm{LF}[s]\,..\,\mathrm{LF}[e]]$, the BWT-runs occurring $[\mathrm{LF}[s]\,..\,\mathrm{LF}[e]]$ may split the $[\mathrm{LF}[s]\,..\,\mathrm{LF}[e]]$ interval. We define the *pull-back* of a mapping $\mathrm{LF}^2$ as $[s\,..\,e] \mapsto [s_1\,..\,e_1], [s_2\,..\,e_2], \ldots, [s_k\,..\,e_k]$ that satisfies $s_1 = s$, $s_{i+1} = e_i + 1$ for $i \in [1\,..\,k)$, $e_k = e$, $\mathrm{BWT}[\mathrm{LF}[s_{i+1}]] \neq \mathrm{BWT}[\mathrm{LF}[e_i]]$, and $j \in [s_i\,..\,e_i)$, $i \in [1\,..\,k]$ implies $\mathrm{BWT}[\mathrm{LF}[j]] = \mathrm{BWT}[\mathrm{LF}[j+1]]$. We also assign to each interval $[s_i\,..\,e_i]$ created by the $\mathrm{LF}^2$ pull-back: (i) a string of length two (specifically, $[s_i\,..\,e_i]$ is assigned the string $\mathrm{BWT}[\mathrm{LF}[s_i]] \circ \mathrm{BWT}[s_i]$) (ii) the indices that $s_i$ and $e_i$ map to, that is $\mathrm{LF}^2[s_i]$, $\mathrm{LF}^2[e_i]$; see Figure 4.

Observe that the LF-mapping maps distinct BWT-runs onto disjoint intervals. Hence, each BWT run boundary appears in exactly one interval $[\mathrm{LF}[s]\,..\,\mathrm{LF}[e]]$, where $[s\,..\,e]$ is a BWT run. As a consequence, if we apply $\mathrm{LF}^2$ pull-back to all BWT run intervals and split each BWT run interval according to its $\mathrm{LF}^2$ pull-back, the number of intervals at most doubles.

For a given $\tau$ that is a power of two, we next describe how to apply this pull-back technique and alphabet replacement to precompute mappings for each BWT run. These precomputed mappings make it so that, given any $i \in [1\,..\,n]$, we can compute $\mathrm{LF}^\tau[i]$ in $\widetilde{\mathcal{O}}(1)$ time. The time and space needed to precompute these mappings are $\widetilde{\mathcal{O}}(\tau r)$.

We start as above and compute the $\mathrm{LF}^2$ pull-back for every BWT-run interval. We then replace each distinct string of length two with a new symbol. This could be accomplished, for example, by sorting and replacing each string by its rank. However, it should be noted that the order of these new symbols is not important. This process assigns each index in $[1\,..\,n]$ a new symbol. We denote this assignment as $\mathrm{BWT}^2$ and observe that the run-length encoded $\mathrm{BWT}^2$ is found by iterating through each $\mathrm{LF}^2$ pull-back.

We now repeat this entire process for the runs in $\mathrm{BWT}^2[j]$. Doing so gives us, for each interval $[s\,..\,e]$ corresponding to a $\mathrm{BWT}^2$ run, a set of intervals $[s\,..\,e] \mapsto [s_1\,..\,e_1], [s_2\,..\,e_2], \ldots, [s_k\,..\,e_k]$ that satisfies $s_1 = s$, $s_{i+1} = e_i + 1$ for $i \in [1\,..\,k)$, $e_k = e$, $\mathrm{BWT}[\mathrm{LF}^2[s_{i+1}]] \neq \mathrm{BWT}[\mathrm{LF}^2[e_i]]$, and $j \in [s_i\,..\,e_i)$, $i \in [1\,..\,k]$ implies $\mathrm{BWT}[\mathrm{LF}^2[j]] = \mathrm{BWT}[\mathrm{LF}^2[j+1]]$. We call this the $\mathrm{LF}^4$ pull-back. Next, we again apply alphabet replacement

(that is, replacing a pair of symbols with a single symbol in $[1\,..\,n]$), defining $\mathrm{BWT}^4$ accordingly. The next iteration will compute the $\mathrm{LF}^8$ pull-back and $\mathrm{BWT}^8$. Repeating $\log\tau$ times, we get a set of intervals corresponding to $\mathrm{LF}^\tau$ pull-back.

Recall that each pull-back step at most doubles the number of intervals. Hence, by continuing this process $\log\tau$ times, the number of intervals created by corresponding $\mathrm{LF}^\tau$ pull-back is $2^{\log\tau}r = \tau r$. For a given index $i$, to compute $\mathrm{LF}^\tau[i]$ we look at the $\mathrm{LF}^\tau$ pull-back. Suppose that $i \in [s'\,..\,e']$, where $[s'\,..\,e']$ is an interval computed in the $\mathrm{LF}^\tau$ pull-back. We look at the mapped onto interval $[\mathrm{LF}^\tau[s']\,..\,\mathrm{LF}^\tau[e']]$, which we have stored as well, and take $\mathrm{LF}^\tau[i] = \mathrm{LF}^\tau[s'] + (i - s')$.

We are now ready to obtain our suffix array samples. We start with the position for the lexicographically smallest suffix (which, by concatenating a special symbol \$ to $X$, we can assume is the rightmost suffix). Utilizing $\mathrm{LF}^\tau$, we compute and store $\frac{n}{\tau}$ suffix array values evenly spaced by text position and their corresponding positions in the RL-BWT. This makes the SA value of any position in the BWT obtainable in $\tau$ applications of $\mathrm{LF}^1$ and computable in $\widetilde{\mathcal{O}}(\tau)$ time. Inverse suffix array, ISA, queries can be supported with additional logarithmic factor overhead by using the LCE data structure (simply binary search over SA values). In summary, we have the following.

LEMMA 6.1. *In time $\widetilde{\mathcal{O}}(\sqrt{rn})$, we can obtain an index that answers* SA *and* ISA *queries in time $\widetilde{\mathcal{O}}(\sqrt{n/r})$ time.*

For a given range $[s\,..\,e]$, the longest common prefix, LCP, of all suffixes $X[\mathrm{SA}[i]\,..\,n]$, $i \in [s\,..\,e]$ is $\mathrm{LCE}(\mathrm{SA}[s], \mathrm{SA}[e])$. Therefore, such queries can also be supported in time $\widetilde{\mathcal{O}}(\tau)$.

**6.2 Constructing the Index for Suffix Array and Inverse Suffix Array Queries** The r-index ($\mathcal{O}(r)$ space version) for locating and counting pattern occurrences can be constructed by sampling suffix array values at the boundaries of BWT runs [GNP18b]. Requiring $\widetilde{\mathcal{O}}(r)$ queries on the structure in Lemma 6.1, this takes $\widetilde{\mathcal{O}}(\sqrt{rn})$ time. Next, we describe how to construct the suffix array index in [GNP20] also in $\widetilde{\mathcal{O}}(\sqrt{rn})$ time. We start with the notion of the differential suffix array ($\mathrm{DSA}[i] = \mathrm{SA}[i] - \mathrm{SA}[i-1]$ for all $i > 1$) and a related lemma:

LEMMA 6.2. ([GNP20]) *Let $[i-1\,..\,i+s]$ be within a BWT run, for some $1 < i \leq n$ and $0 \leq s \leq n-i$. Then, there exists $q \neq i$ such that $\mathrm{DSA}[q\,..\,q+s] = \mathrm{DSA}[i\,..\,i+s]$ and $[q-1\,..\,q+s]$ contains the first position of a BWT run.*

Lemma 6.2 implies that the LF-mapping applied to a portion of the DSA completely contained within a BWT run preserves the DSA values, making the DSA highly compressible. With this observation, Gagie et al. obtained their index, which consists of $\mathcal{O}(\log\frac{n}{r})$ levels, with $\mathcal{O}(r)$ nodes each. The nodes on a given level maintain pointers to nodes on the next level based on the $q$ values from Lemma 6.2. Along with DSA values, SA values, and 'offsets' kept for each node, these pointers are sufficient for efficiently recovering any suffix array value. We refer the reader to [GNP20] for further details.

A key operation to construct this data structure is finding these pointers, or $q$ values, from Lemma 6.2. Following this, the remaining values needed per node are easily obtained from SA and ISA queries using Lemma 6.1. As the next lemma demonstrates, for an arbitrary range $[s\,..\,e]$ we can obtain such a pointer in $\widetilde{\mathcal{O}}(\tau)$ time using the previously computed values from Section 6.1.

LEMMA 6.3. *Given that* $\mathrm{SA}[i]$ *for arbitrary $i$ can be computed in $\widetilde{\mathcal{O}}(\tau)$ time and (reversed) LCE queries in $\widetilde{\mathcal{O}}(1)$ time, for a given range $[s\,..\,e]$, we can find $k$, $s'$, $e'$ such that $k \geq 0$ is the smallest value where $\mathrm{LF}^k([s\,..\,e]) = [s'\,..\,e']$ and $[s'\,..\,e']$ contains the start of BWT-run.*

*Proof.* We assume $k \geq 1$; otherwise, we determine from the RL-BWT that $[s\,..\,e]$ contains a run. We use exponential search on $k$. For a given $k$, we compute $s' = \mathrm{ISA}[\mathrm{SA}[s] - k]$ and $e' = \mathrm{ISA}[\mathrm{SA}[e] - k]$. We check whether $e' - s' = e - s$ and whether $\mathrm{LCP}([s'\,..\,e']) \geq k$. If both of these conditions hold, then no run boundary has been encountered yet. The largest $k$, $s'$, and $e'$ for which these conditions hold is returned. Utilizing Lemma 6.1, this takes $\widetilde{\mathcal{O}}(\tau)$ time per SA or ISA query, resulting in $\widetilde{\mathcal{O}}(\tau)$ time overall. ∎

In summary, the construction of the suffix array index requires a total of $\widetilde{\mathcal{O}}(r)$ queries on the structure in Lemma 6.1 and calls to the algorithm described in the proof of Lemma 6.3. Inverse suffix array queries are supported with the help of LCE data structure as before. This completes the proof of Theorem 1.4.

THEOREM 1.4. (COMPRESSED INDEX) *There is an $\widetilde{\mathcal{O}}(\sqrt{rn})$-time quantum algorithm that, given a quantum oracle to an unknown string $X \in \Sigma^n$, constructs*

- *an $\mathcal{O}(r)$-space index that can count the occurrences of any length-$m$ pattern in $\widetilde{\mathcal{O}}(m)$ time and report these occurrences in time $\widetilde{\mathcal{O}}(m + \mathsf{occ})$, where $\mathsf{occ}$ is the number of occurrences;*
- *an $\widetilde{\mathcal{O}}(r)$-space index for suffix array and inverse suffix array queries in $\widetilde{\mathcal{O}}(1)$ time.*

**6.3 Applications** Having constructed suffix array index for $X$, we can solve a number of other problems in sub-linear time:

**Longest Common Substring:** Given two strings $S_1$ and $S_2$, we let $z_{1,2}$ be the number of LZ77 factors of $S_1\$S_2$. In $\widetilde{\mathcal{O}}(\sqrt{z_{1,2}(|S_1| + |S_2|)})$ time, we construct the LZ77 parse and LCE-data structure for the $S_1\$S_2$.

LEMMA 6.4. *If $X$ is the longest common substring of $S_1$ and $S_2$, then there exists $i_1$,$j_1$, $i_2$, $j_2$ such that $S_1[i_1\,..\,j_1] = X$ and $S_2[i_2\,..\,j_2] = X$. For the* BWT *and* ISA *of $S_1\$S_2$, we have $|\,\mathrm{ISA}[i_1] - \mathrm{ISA}[|S_1| + 1 + i_2]| = 1$. Moreover, for this instance, $\mathrm{BWT}[\mathrm{ISA}[i_1]] \neq \mathrm{BWT}[\mathrm{ISA}[i_2]]$.*

*Proof.* Suppose all occurrences of $X$ satisfy $|\,\mathrm{ISA}[i_1] - \mathrm{ISA}[|S_1| + 1 + i_2]| > 1$, and let $i_1$ and $i_2$ be such that $|\,\mathrm{ISA}[i_1] - \mathrm{ISA}[|S_1| + 1 + i_2]|$ is minimized. Suppose w.l.o.g. that $\mathrm{ISA}[i_1] < \mathrm{ISA}[|S_1| + 1 + i_2]$. Then, there must exist indices $x$ and $y$ such that $\mathrm{ISA}[i_1] \leq x \leq y \leq \mathrm{ISA}[|S_1| + 1 + i_2]$ such that $y - x = 1$ and $\mathrm{SA}[x], \mathrm{SA}[y]$ are indices into ranges for different strings, i.e., $\mathrm{SA}[x] \in [1\,..\,|S_1|]$ and $\mathrm{SA}[y] \in [|S_1| + 2\,..\,|S_1| + |S_2| + 1]$ or $\mathrm{SA}[x] \in [|S_1| + 2\,..\,|S_1| + |S_2| + 1]$ and $\mathrm{SA}[y] \in [1\,..\,|S_1|]$, and $\mathrm{LCE}(\mathrm{SA}[x], \mathrm{SA}[y]) \geq \mathrm{LCE}(\mathrm{SA}[i_1], \mathrm{SA}[i_2])$. Hence, the prefix $X$ is shared by $(S_1\$S_2)[x\,..]$ and $(S_1\$S_2)[y\,..]$, a contradiction. At the same time, if $\mathrm{BWT}[i_1] = \mathrm{BWT}[|S_1| + 1 + i_2]$, then the longest common substring could be extended to the left, a contradiction. ∎

Based on Lemma 6.4, we can find the longest common string of $S_1$ and $S_2$ by checking the runs in BWT of $S_1\$S_2$, checking if adjacent SA values correspond to suffixes of different strings, and through the LCE queries, taking the pair with the longest shared prefix.

Note that $\Omega(\sqrt{|S_1| + |S_2|})$ time is necessary for the problem when $z = \Theta(1)$, as determining whether $S_1 = 0^n$ or whether $S_1$ contains a single 1 requires $\Omega(\sqrt{n})$ time. The reduction simply sets $S_2 =$ '1'.

**Maximal Unique Matches:** Given two strings $S_1$ and $S_2$, we can identify all maximal unique matches in $\widetilde{\mathcal{O}}(r)$ additional time after constructing the RL-BWT and our index for $S_1\$S_2$. To do so we iterate through all run boundaries in the RL-BWT. We wish to identify all indices $i$ where:

- $\mathrm{BWT}[i] \neq \mathrm{BWT}[i + 1]$;
- $\mathrm{SA}[i] \in [1\,..\,|S_1|]$ and $\mathrm{SA}[i + 1] \in [|S_1| + 2\,..\,|S_1| + |S_2|]$, or $\mathrm{SA}[i] \in [|S_1| + 2\,..\,|S_1| + |S_2|]$ and $\mathrm{SA}[i + 1] \in [1\,..\,|S_1|]$;
- and either
  - $i = 1$ and $\mathrm{LCE}(\mathrm{SA}[i], \mathrm{SA}[i + 1]) > \mathrm{LCE}(\mathrm{SA}[i + 1], \mathrm{SA}[i + 2])$,
  - $i > 1$ and $i + 1 < n$ and
$$\mathrm{LCE}(\mathrm{SA}[i], \mathrm{SA}[i + 1]) > \max(\mathrm{LCE}(\mathrm{SA}[i - 1], \mathrm{SA}[i]), \mathrm{LCE}(\mathrm{SA}[i + 1], \mathrm{SA}[i + 2])),$$
    or
  - $i > 1$ and $i + 1 = n$ and $\mathrm{LCE}(\mathrm{SA}[i], \mathrm{SA}[i + 1]) > \mathrm{LCE}(\mathrm{SA}[i - 1], \mathrm{SA}[i])$.

**Lyndon Factorization:** The Lyndon factors of a string are determined by the values in ISA that are smaller than any previous value, i.e., $i \in [1\,..\,n]$ s.t. $\mathrm{ISA}[i] < \mathrm{ISA}[j]$ for all $j < i$. After constructing the SA index above in $\widetilde{\mathcal{O}}(\sqrt{zn})$, each ISA value can be queried in $\widetilde{\mathcal{O}}(1)$ time as well. Let $f$ denote the total number of Lyndon factors and let $i_j$ be the index where the $j^{th}$ Lyndon factor starts. To find all Lyndon factors, we proceed from left to right keeping track of the minimum ISA value encountered thus far. Initially, this is $\mathrm{ISA}[i_1] = \mathrm{ISA}[1]$. We use exponential search on the rightmost boundary of the subarray being searched and Grover's search to identify the left-most index $x \in [i_j + 1\,..\,n]$ such that $\mathrm{ISA}[x] < \mathrm{ISA}[i_j]$. If one is encountered, we set $i_{j+1} = x$ and continue. Thanks to the exponential search, the total time taken is logarithmic factors from

$$\sum_{j=1}^{f} \sqrt{i_{j+1} - i_j} \leq \sqrt{fn}.$$

At the same time, the number of Lyndon factors $f$ is always bound by $\widetilde{\mathcal{O}}(z)$ [KKN+17, UNI+19]. This yields the desired time complexity of $\widetilde{\mathcal{O}}(\sqrt{zn})$.

**Q-gram Frequencies:** We need to identify the nodes $v$ of the suffix tree at string depth $q$ and the number of leaves in their respective subtrees. A string matching the root-to-$v$ path, where $v$ has string depth $q$, is a q-gram and the size of the subtree its frequency. To do this, we start with $i = 1$. Our goal is to find the largest $j$ such that $\text{LCE}(\text{SA}[i], \text{SA}[j]) \geq q$. This can be found using exponential search on $j$ starting with $j = i$. Once this largest $j$ is found, the size of the subtree is equal to $j - i + 1$. The q-gram itself is $X[\text{SA}[i] \,..\, \text{SA}[i] + q)$. We then set $i = j + 1$ and repeat this process until the $j$ found through exponential search equals $n$. The overall time after index construction is $\widetilde{\mathcal{O}}(occ)$, where $occ$ is the number of q-grams.

The $\widetilde{\mathcal{O}}(\sqrt{zn})$ time complexity is near optimal as a straightforward reduction from the Threshold problem discussed in Section 7 with $q = 1$ and binary strings indicates that $\Omega(\sqrt{zn})$ input queries are required.

These are likely only a small sample of the problems that can be solved using the proposed techniques. It is also worth restating that the compressed forms of the text can be obtained in $\mathcal{O}(\sqrt{zn})$ input queries, hence all string problems can be solved with that many input queries, albeit, perhaps with greater time needed.

## 7 Lower Bounds

**7.1 Hardness for Extreme $z$ and $r$** We first consider the case of $z, r = \Theta(1)$. Let $X$ be a binary string and $S$ be the set of indices $i$ in $X$ such that $X[i] = 1$. One can easily show, through an application of the adversarial method of Ambainis, that $\Omega(\sqrt{n})$ queries are required to determine if $|S| = 1$, even given the promise that $|S| = 0$ or $|S| = 1$ [Amb00]. In the case where $|S| = 0$, we have $z = 2$ (and $r = 1$), and in the case where $|S| = 1$, we have $3 \leq z \leq 5$ and $2 \leq r \leq 3$. Note also that, even if we only obtained a suffix array index and not the compressed encodings, this would allow us to determine if there exists a 1 in $X$ with a single additional query.

To show hardness for $z = r = n$ (and a large alphabet), we consider the problem of determining the $\left(\frac{n}{2}\right)^{th}$ largest value output from an oracle $f$ that outputs distinct values for each index. This unordered searching problem has a known $\Omega(n)$ lower bound [NW99]. For the string representation $X = \bigcirc_{i=1}^{n} f(i)$, if we could obtain an LZ77 encoding of $X$ using $o(z)$, or equivalently $o(n)$, oracle queries, then we could decompress the result and return the median element to solve the unordered searching problem. Similarly, if we could obtain the RL-BWT of $X$ in $o(r)$ queries, we could decompress it to obtain the median element in $o(n)$ queries. Note also that even if we only had a suffix array index and not the compressed encoding, with a single additional query we could find the median element.

The above observations yield the following results.

THEOREM 7.1. *Obtaining the LZ77 factorization of a text $X[1..n]$ with $z$ LZ77 factors, or RL-BWT with $r$ runs, requires $\Omega(\sqrt{zn})$ queries ($\Omega(\sqrt{rn})$ queries resp.) when $z, r = \Theta(1)$ or $z, r = \Theta(n)$.*

THEOREM 7.2. *Constructing a data structure that supports $o(\sqrt{zn})$ time suffix array queries of a text $X[1..n]$ with $z$ LZ77 factors, or RL-BWT with $r$ runs, requires $\Omega(\sqrt{zn})$ ($\Omega(\sqrt{rn})$ resp.) input queries when $z, r = \Theta(1)$ or $z, r = \Theta(n)$.*

**7.2 Parameterized Hardness of Obtaining the LZ77 Factorization and RL-BWT** In this section, we aim to show that $\Omega(\sqrt{zn})$ ($\Omega(\sqrt{rn})$) input oracle queries are still required to obtain the LZ77 (RL-BWT resp.) encoding, even when $z$ ($r$ resp.) is restricted to a small range of possible values that are not necessarily constant or close to $n$. We provide a reduction from the Threshold Problem, which is defined as follows:

PROBLEM 1. (THRESHOLD PROBLEM) *Given an oracle $f : [1..n] \to \{0, 1\}$ and integer $t \geq 0$, determine if there exist at least $t$ inputs $i$ such that $f(i) = 1$, i.e., if $S = |\{i \in [1..n] \mid f(i) = 1\}|$ satisfies $|S| \leq t$.*

Known lower bounds on the quantum query complexity state that, for $0 \leq t < \frac{n}{2}$, at least $\Omega(\sqrt{(t+1)n})$ queries to the input oracle are required to solve the Threshold Problem [HS05, Pat92]. Here, we treat $t$ as a polynomial function of $n$, i.e., $t = n^{\xi}$ where $\xi \in (0, 1)$ is a constant.

One obstacle in using the Threshold problem to establish hardness results is that we cannot make assumptions concerning the size of the set $|S|$, and we do not assume knowledge of $z$ for the problem of finding the LZ77 factorization. The following Lemma helps to relate the size of the set $S$ and the number of LZ77 factors, $z$, in the binary string representation of $f$.

LEMMA 7.3. *If the string representation $X = \bigcirc_{i=1}^{n} f(i)$ has $|S| \geq 0$ ones, then the LZ77 factorization size of $X$ is $z \leq 3|S| + 2$.*

*Proof.* Each run of 0's contributes at most two factors to the factorization and each 1 symbol contributes at most one factor; hence, each $0^x 1^y$ substring, $x \geq 0$, $y \geq 1$ accounts for at most 3 factors. Additional two factors account for a possible suffix of all 0's. ∎

For finding the RL-BWT, we can establish a similar result.

LEMMA 7.4. *If the string representation $X = \bigcirc_{i=1}^{n} f(i)$ has $|S| \geq 0$ ones, then the number of runs in BWT of $X$ is $r \leq 2|S| + 1$.*

*Proof.* Consider any permutation of a binary string with $|S|$ ones and $n - |S|$ zeroes. To maximize the number of runs, we alternate between ones and zeroes. This creates at most two runs per every one. Accounting for the suffix of zeroes, this creates at most $2|S| + 1$ runs. Finally, note that the BWT is a permutation of $X$. ∎

Let the instance of the Threshold Problem with $f : [1\,..\,n] \to \{0, 1\}$ and $t$ be given. Assume for the sake of contradiction that there exists an algorithm $A$ for finding the LZ77 factorization of $X$ in $q(n, z) = o(\sqrt{zn})$ queries for strings satisfying $t^{1-\varepsilon} \leq z \leq t$ for some constant $\varepsilon > 0$. As an example, suppose that there exists an algorithm for finding the LZ77 factorization in $q(n, z)$ time whenever $n^{\frac{1}{2}-\varepsilon} \leq z \leq n^{\frac{1}{2}}$ for some constant $\varepsilon > 0$.

Based on $n$, $t$, and $q(n, t)$, we create a query threshold $\kappa(n, t) \in \omega(q(n, t)) \cap o(\sqrt{(t+1)n})$. In particular, we can take $\kappa(n, t) = \sqrt{q(n, t)} \cdot ((t + 1)n)^{\frac{1}{4}}$, which has $\lim_{n\to\infty} \kappa(n, t)/\sqrt{(t+1)n} = 0$ and $\lim_{n\to\infty} q(n, t)/\kappa(n, t) = 0$. To solve the Threshold Problem with the LZ77 Factorization algorithm, do as follows:

1. Run the $\widetilde{\mathcal{O}}(\sqrt{zn})$-query algorithm from Section 5, but halt if the number of input queries reaches $\kappa(n, t)$. If we obtain a complete encoding without halting, we output the solution; otherwise, we continue to Step 2. This solves the Threshold Problem instance in the case where $z \leq t^{1-\varepsilon}$ in $o(\sqrt{(t+1)n})$ input queries.
2. If we did not obtain an encoding in Step 1, we next run the algorithm $A$ but halt early if the number of input queries exceeds $\kappa(n, t)$. If we halt with a completed encoding, we output the solution based on the encoding of $X$. If we do not halt with a completed encoding, we output that $|S| > t$.

In the case where $|S| \leq t$, we have $z \leq 3|S| + 2 \leq 3t + 2$, and our algorithm solves the Threshold Problem in Step 1 using $\widetilde{\mathcal{O}}(\sqrt{zn})$ queries if $z \leq t^{1-\varepsilon}$, which is $o(\sqrt{(t+1)n})$, or in Step 2 using $q(n, z) = o(\sqrt{(t+1)n})$ queries if $z \in [t^{1-\varepsilon}\,..\,t]$. In the case where $|S| > t$, the number of queries is still bounded by $\kappa(n, t) = o(\sqrt{(t+1)n})$ in Steps 1 and 2, and we output that $|S| > t$. In either case, we solve the Threshold Problem with $o(\sqrt{(t+1)n})$ input queries. As such, the assumption that algorithm $A$ exists contradicts the known lower bounds.

Using Lemma 7.4, a near-identical argument holds for computing the RL-BWT of $X$ with $z$ replaced by $r$, showing that $\Omega(\sqrt{rn})$ queries are required. The above proves the following.

THEOREM 7.5. *No quantum algorithm exists for computing the LZ77 factorization, or RL-BWT, using $o(\sqrt{zn})$ queries ($o(\sqrt{rn})$ queries resp.) for all texts with $z \in [t^{1-\varepsilon}\,..\,t]$ ($r \in [t^{1-\varepsilon}\,..\,t]$ resp.) where $t = n^{\xi}$, $\xi \in (0, 1)$, and $\varepsilon > 0$ is any constant. This holds for alphabets of size at least two.*

**7.3 Parameterized Lower Bounds for Computing the Value $z$** We next turn our attention to the problem of determining the number of factors, $z$, in the LZ77-factorization. This is potentially an easier problem than actually computing the factorization. Unlike the proof for the hardness of computing the actual LZ77 factorization that uses a binary alphabet, this proof uses a larger integer alphabet.

Given inputs $f : [1\,..\,n] \to \{0, 1\}$ and $t \geq 0$ to the Threshold Problem, we construct an input oracle for a string $X$ for the problem of determining the size of the LZ77 factorization. We first define the function $s : \{0, 1\} \times [1\,..\,n] \to [0\,..\,n]$ as:

$$s(f(i), i) = \begin{cases} 0 & \text{if } f(i) = 0, \\ i & \text{if } f(i) = 1. \end{cases}$$

Our reduction will create an oracle for a string $X = 0^{2n} \circ \$ \circ (\bigcirc_{i=1}^{n} 0 \circ s(f(i), i)) \circ 0$. Formally, we define the oracle

$X : [1\,..\,4n+2] \to \{0, \$\} \cup [1\,..\,n]$ where

$$X[i] = \begin{cases} 0 & \text{if } 1 \le i \le 2n \text{ or } i \text{ is even,} \\ \$ & \text{if } i = 2n+1, \\ s\left(f(\frac{i-(2n+1)}{2}), \frac{i-(2n+1)}{2}\right) & \text{if } i > 2n+1 \text{ and } i \text{ is odd.} \end{cases}$$

Observe that every symbol in $X$ can be computed in constant time given access to the oracle $f$. The correctness of the reduction will follow from Lemma 7.6.

LEMMA 7.6. *The LZ77 factorization of $X$ constructed above has $z = 2|S| + 4$.*

*Proof.* The prefix $0^{2n} \circ \$$ always requires exactly 3 factors: 0, $0^{2n-1}$, \$. Any subsequent run of 0's only requires one factor. This is because any run of 0's following this prefix is of length at most $2n$. Next, note that the 0 following \$ is the start of a new factor. Following this 0 symbol, we claim that every $i$, where $f(i) = 1$, contributes exactly two new factors. This is due to a new factor being created for the leftmost occurrence of the symbol $i$ followed by a new factor for the run of 0's beginning after the symbol $i$. Observe that the first 0 following the \$ is necessary for the lemma; otherwise, the cases where $f(1) = 1$ would result in a different number of factors. □

Similar to Section 7.2, assume for the sake of contradiction that we have an algorithm that solves the problem of determining the value $z$ in $q(n, z) = o(\sqrt{zn})$ for $z = [t^{1-\varepsilon}\,..\,t]$. We again define the same query threshold $\kappa(n, t) \in \omega(q(n, t)) \cap o(\sqrt{(t+1)n})$ as given earlier. Then, to solve the instance of the Threshold Problem, the two-step algorithm from Section 7.2 is applied to the oracle for $X$, resulting in a solution using $o(\sqrt{(t+1)n})$ input queries. This demonstrates the following.

THEOREM 7.7. *No quantum algorithm exists for computing the number of LZ77 factors of a text $X[1\,..\,n]$ that uses only $o(\sqrt{zn})$ queries for all texts with $z \in [t^{1-\varepsilon}\,..\,t]$, where $t = n^{\xi}$, $\xi \in (0, 1)$, and $\varepsilon > 0$ is any constant.*

**Acknowledgement.** S. Thankachan is partially supported by the U.S. National Science Foundation (NSF) awards CCF-2315822 and CCF-2316691.

## References

[ABI+20] Andris Ambainis, Kaspars Balodis, Jānis Iraids, Kamil Khadiev, Vladislavs Kļevickis, Krišjānis Prūsis, Yixin Shen, Juris Smotrovs, and Jevgēnijs Vihrovs. Quantum lower and upper bounds for 2D-grid and Dyck language. In *45th International Symposium on Mathematical Foundations of Computer Science, MFCS 2020*, pages 8:1–8:14, 2020. doi:10.4230/LIPIcs.MFCS.2020.8.

[AGS19] Scott Aaronson, Daniel Grier, and Luke Schaeffer. A quantum query complexity trichotomy for regular languages. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019*, pages 942–965, 2019. doi:10.1109/FOCS.2019.00061.

[AJ22] Shyan Akmal and Ce Jin. Near-optimal quantum algorithms for string problems. In *33rd ACM-SIAM Symposium on Discrete Algorithms, SODA 2022*, pages 2791–2832. SIAM, 2022. doi:10.1137/1.9781611977073.109.

[Amb00] Andris Ambainis. Quantum lower bounds by quantum arguments. In *32nd Annual ACM Symposium on Theory of Computing, STOC 2000*, pages 636–643. ACM, 2000. doi:10.1145/335305.335394.

[Amb04] Andris Ambainis. Quantum query algorithms and lower bounds. In *Classical and New Paradigms of Computation and their Complexity Hierarchies*, pages 15–32. Springer, 2004. doi:10.1007/978-1-4020-2776-5_2.

[BBC+01] Robert Beals, Harry Buhrman, Richard Cleve, Michele Mosca, and Ronald de Wolf. Quantum lower bounds by polynomials. *Journal of the ACM*, 48(4):778–797, 2001. doi:10.1145/502090.502097.

[BCFN22a] Karl Bringmann, Alejandro Cassis, Nick Fischer, and Vasileios Nakos. Almost-optimal sublinear-time edit distance in the low distance regime. In *54th Annual ACM SIGACT Symposium on Theory of Computing STOC 2022*, pages 1102–1115. ACM, 2022. doi:10.1145/3519935.3519990.

[BCFN22b] Karl Bringmann, Alejandro Cassis, Nick Fischer, and Vasileios Nakos. Improved sublinear-time edit distance for preprocessed strings. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022*, volume 229 of *LIPIcs*, pages 32:1–32:20. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ICALP.2022.32.

[BdW02] Harry Buhrman and Ronald de Wolf. Complexity measures and decision tree complexity: a survey. *Theoretical Computer Science*, 288(1):21–43, 2002. doi:10.1016/S0304-3975(01)00144-X.

[BEG+21] Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, MohammadTaghi HajiAghayi, and Saeed Seddighin. Approximating edit distance in truly subquadratic time: Quantum and MapReduce. *Journal of the ACM*, 68(3):1–41, 2021. doi:10.1145/3456807.

[BI18] Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM Journal on Computing*, 47(3):1087–1097, 2018. doi:10.1137/15M1053128.

[BK23] Sudatta Bhattacharya and Michal Koucký. Locally consistent decomposition of strings with applications to edit distance sketching. In *55th Annual ACM Symposium on Theory of Computing, STOC 2023*, pages 219–232. ACM, 2023. doi:10.1145/3564246.3585239.

[BP16] Djamal Belazzougui and Simon J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, pages 2053–2071. SIAM, 2016. doi:10.1137/1.9781611974331.ch143.

[BPS21] Harry Buhrman, Subhasree Patro, and Florian Speelman. A framework of quantum strong exponential-time hypotheses. In *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021*, volume 187 of *LIPIcs*, pages 19:1–19:19. Schloss Dagstuhl– Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.STACS.2021.19.

[BW94] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[BZ16] Djamal Belazzougui and Qin Zhang. Edit distance: Sketching, streaming, and document exchange. In *57th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2016*, pages 51–60. IEEE Computer Society, 2016. doi:10.1109/FOCS.2016.15.

[CKK+22] Andrew M. Childs, Robin Kothari, Matt Kovacs-Deak, Aarthi Sundaram, and Daochen Wang. Quantum divide and conquer, 2022. arXiv:2210.06419.

[CKW23] Alejandro Cassis, Tomasz Kociumaka, and Philip Wellnitz. Optimal algorithms for bounded weighted edit distance. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023*. IEEE, 2023. arXiv:2305.06659.

[CLL+05] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116.

[DGH+23] Debarati Das, Jacob Gilbert, MohammadTaghi Hajiaghayi, Tomasz Kociumaka, and Barna Saha. Weighted edit distance computation: Strings, trees, and Dyck. In *55th Annual ACM Symposium on Theory of Computing, STOC 2023*, pages 377–390. ACM, 2023. doi:10.1145/3564246.3585178.

[FFM00] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.

[Fil20] Yuval Filmus. To find median of $k$ sorted arrays of $n$ elements each in less than $O(nk \log k)$, Nov 2020. URL: https://cs.stackexchange.com/questions/87695/to-find-median-of-k-sorted-arrays-of-n-elements-each-in-less-than-onk-log/156925#156925.

[FM05] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

[GKLS22] Arun Ganesh, Tomasz Kociumaka, Andrea Lincoln, and Barna Saha. How compression and approximation affect efficiency in string distance measures. In *33rd ACM-SIAM Symposium on Discrete Algorithms, SODA 2022*, pages 2867–2919. SIAM, 2022. doi:10.1137/1.9781611977073.112.

[GKS19] Elazar Goldenberg, Robert Krauthgamer, and Barna Saha. Sublinear algorithms for gap edit distance. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019*, pages 1101–1120. IEEE, 2019. doi:10.1109/FOCS.2019.00070.

[GNP18a] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. On the approximation ratio of lempel-ziv parsing. In *13th Latin American Symposium on Theoretical Informatics, LATIN 2018*, volume 10807 of *LNCS*, pages 490–503. Springer, 2018. doi:10.1007/978-3-319-77404-6_36.

[GNP18b] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in bwt-runs bounded space. In *29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 1459–1477. SIAM, 2018. doi:10.1137/1.9781611975031.96.

[GNP20] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1):2:1–2:54, 2020. doi:10.1145/3375890.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *28th Annual ACM Symposium on the Theory of Computing, STOC 1996*, pages 212–219, 1996. doi:10.1145/237814.237866.

[GRS20] Elazar Goldenberg, Aviad Rubinstein, and Barna Saha. Does preprocessing help in fast sequence comparisons? In *52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 657–670. ACM, 2020. doi:10.1145/3357713.3384300.

[GS22] François Le Gall and Saeed Seddighin. Quantum meets fine-grained complexity: Sublinear time quantum

algorithms for string problems. In Mark Braverman, editor, *13th Innovations in Theoretical Computer Science Conference, ITCS 2022*, volume 215 of *LIPIcs*, pages 97:1–97:23. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ITCS.2022.97.

[GV05] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.

[HS05] Peter Høyer and Robert Spalek. Lower bounds on quantum query complexity. *Bulletin of EATCS*, 87:78–103, 2005. arXiv:quant-ph/0509153.

[HV03] Ramesh Hariharan and V. Vinay. String matching in $\tilde{O}(\sqrt{n}+\sqrt{m})$ quantum time. *Journal of Discrete Algorithms*, 1(1):103–110, 2003. doi:10.1016/S1570-8667(03)00010-8.

[I17] Tomohiro I. Longest common extensions with recompression. In *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78 of *LIPIcs*, pages 18:1–18:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.18.

[IP01] Russell Impagliazzo and Ramamohan Paturi. On the complexity of $k$-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.

[JN23] Ce Jin and Jakob Nogler. Quantum speed-ups for string synchronizing sets, longest common substring, and $k$-mismatch matching. In *34th ACM-SIAM Symposium on Discrete Algorithms, SODA 2023*, pages 5090–5121. SIAM, 2023. doi:10.1137/1.9781611977554.ch186.

[JNW21] Ce Jin, Jelani Nelson, and Kewen Wu. An improved sketching algorithm for edit distance. In *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021*, volume 187 of *LIPIcs*, pages 45:1–45:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.STACS.2021.45.

[KK17a] Dominik Kempa and Dmitry Kosolobov. LZ-end parsing in compressed space. In *2017 Data Compression Conference, DCC 2017*, pages 350–359. IEEE, 2017. doi:10.1109/DCC.2017.73.

[KK17b] Dominik Kempa and Dmitry Kosolobov. LZ-end parsing in linear time. In *25th Annual European Symposium on Algorithms, ESA 2017*, volume 87 of *LIPIcs*, pages 53:1–53:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ESA.2017.53.

[KK22] Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. *Communications of the ACM*, 65(6):91–98, 2022. doi:10.1145/3531445.

[KKN+17] Juha Kärkkäinen, Dominik Kempa, Yuto Nakashima, Simon J. Puglisi, and Arseny M. Shur. On the size of Lempel-Ziv and Lyndon factorizations. In *34th Symposium on Theoretical Aspects of Computer Science, STACS 2017*, volume 66 of *LIPIcs*, pages 45:1–45:13. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.STACS.2017.45.

[KKP14] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lempel-Ziv parsing in external memory. In *Data Compression Conference, DCC 2014*, pages 153–162. IEEE, 2014. doi:10.1109/DCC.2014.78.

[KL21] Dominik Kempa and Ben Langmead. Fast and space-efficient construction of AVL grammars from the LZ77 parsing. In *29th Annual European Symposium on Algorithms, ESA 2021*, volume 204 of *LIPIcs*, pages 56:1–56:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ESA.2021.56.

[KN13] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013. doi:10.1016/j.tcs.2012.02.006.

[KNP23] Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Towards a definitive compressibility measure for repetitive sequences. *IEEE Transactions on Information Theory*, 69(4):2074–20, 2023. doi:10.1109/TIT.2022.3224382.

[Kot14] Robin Kothari. An optimal quantum algorithm for the oracle identification problem. In *31st International Symposium on Theoretical Aspects of Computer Science, STACS 2014*, volume 25 of *LIPIcs*, pages 482–493. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPIcs.STACS.2014.482.

[KP18] Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 827–840. ACM, 2018. doi:10.1145/3188745.3188814.

[KPS21] Tomasz Kociumaka, Ely Porat, and Tatiana Starikovskaya. Small-space and streaming pattern matching with $k$ edits. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021*, pages 885–896. IEEE, 2021. doi:10.1109/FOCS52979.2021.00090.

[KS20] Tomasz Kociumaka and Barna Saha. Sublinear-time algorithms for computing & embedding gap edit distance. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*, pages 1168–1179. IEEE, 2020. doi:10.1109/FOCS46700.2020.00112.

[KS22] Dominik Kempa and Barna Saha. An upper bound and linear-space queries on the LZ-end parsing. In *33rd ACM-SIAM Symposium on Discrete Algorithms, SODA 2022*, pages 2847–2866. SIAM, 2022. doi:10.1137/1.9781611977073.111.

[Lev65] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Doklady Akademii*

*Nauk SSSR*, 163(4):845–848, 1965. URL: http://mi.mathnet.ru/eng/dan31411.

[Lit87] Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, 1987. doi:10.1007/BF00116827.

[LV88] Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988. doi:10.1016/0022-0000(88)90045-1.

[McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.

[MNN17] J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *28th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017*, pages 408–424. SIAM, 2017. doi:10.1137/1.9781611974782.26.

[MP70] James H. Morris, Jr. and Vaughan R. Pratt. A linear pattern-matching algorithm. Technical Report 40, Department of Computer Science, University of California, Berkeley, 1970.

[Mye86] Eugene W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986. doi:10.1007/BF01840446.

[Nav21] Gonzalo Navarro. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Computing Surveys*, 54(2):29:1–29:31, 2021. doi:10.1145/3434399.

[NII+16] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016*, volume 58 of *LIPIcs*, pages 72:1–72:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.MFCS.2016.72.

[NII+20] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index and LZ factorization in compressed space. *Discrete Applied Mathematics*, 274:116–129, 2020. doi:10.1016/j.dam.2019.01.014.

[NKT22] Takaaki Nishimoto, Shunsuke Kanda, and Yasuo Tabei. An optimal-time RLBWT construction in BWT-runs bounded space. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022*, volume 229 of *LIPIcs*, pages 99:1–99:20. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ICALP.2022.99.

[NM07] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2, 2007. doi:10.1145/1216370.1216372.

[NT21] Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on BWT-runs compressed indexes. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021*, volume 198 of *LIPIcs*, pages 101:1–101:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ICALP.2021.101.

[NW70] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970. doi:10.1016/0022-2836(70)90057-4.

[NW99] Ashwin Nayak and Felix Wu. The quantum query complexity of approximating the median and related statistics. In *31st Annual ACM Symposium on Theory of Computing, STOC 1999*, pages 384–393. ACM, 1999. doi:10.1145/301250.301349.

[Pat92] Ramamohan Paturi. On the degree of polynomials that approximate symmetric boolean functions (preliminary version). In *24th Annual ACM Symposium on Theory of Computing, STOC 1992*, pages 468–474. ACM, 1992. doi:10.1145/129712.129758.

[RPE81] Michael Rodeh, Vaughan R. Pratt, and Shimon Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28(1):16–24, 1981. doi:10.1145/322234.322237.

[RRRS13] Sofya Raskhodnikova, Dana Ron, Ronitt Rubinfeld, and Adam D. Smith. Sublinear algorithms for approximating string compressibility. *Algorithmica*, 65(3):685–709, 2013. doi:10.1007/s00453-012-9618-6.

[Rub19] Aviad Rubinstein. Quantum dna sequencing and the ultimate hardness hypothesis, 2019. URL: https://theorydish.blog/2019/12/09/quantum-dna-sequencing-the-ultimate-hardness-hypothesis/.

[Ryt03] Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.

[Sad07] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007. doi:10.1007/s00224-006-1198-x.

[Sel74] Peter H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793, 1974. doi:10.1137/0126070.

[SS82] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982. doi:10.1145/322344.322346.

[Ukk85] Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1-3):100–118, 1985. doi:10.1016/S0019-9958(85)80046-2.

[UNI+19] Yuki Urabe, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. On the size of overlapping Lempel-Ziv and Lyndon factorizations. In *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019*, volume 128 of *LIPIcs*, pages 29:1–29:11. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.CPM.2019.29.

[Vin68] Taras Klymovych Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4(1):52–57, 1968. doi:10.1007/BF01074755.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, SWAT 1973*, pages 1–11. IEEE, 1973. doi:10.1109/SWAT.1973.13.

[WF74] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974. doi:10.1145/321796.321811.

[WY20] Qisheng Wang and Mingsheng Ying. Quantum algorithm for lexicographically minimal string rotation, 2020. arXiv:2012.09376.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.